Discovering Functional Dependencies through Hitting Set Enumeration

TOBIAS BLEIFUSS, Hasso Plattner Institute, University of Potsdam, Germany
THORSTEN PAPENBROCK, Philipps University of Marburg, Germany
THOMAS BLÄSIUS, Karlsruhe Institute of Technology, Germany
MARTIN SCHIRNECK, University of Vienna, Austria
FELIX NAUMANN, Hasso Plattner Institute, University of Potsdam, Germany

Functional dependencies (FDs) are among the most important integrity constraints in databases. They serve to normalize datasets and thus resolve redundancies, they contribute to query optimization, and they are frequently used to guide data cleaning efforts. Because the FDs of a particular dataset are usually unknown, automatic profiling algorithms are needed to discover them. These algorithms have made considerable advances in the past few years, but they still require a significant amount of time and memory to process datasets of practically relevant sizes.

We present FDHITS, a novel FD discovery algorithm that finds all valid, minimal FDs in a given relational dataset. FDHITS is based on several discovery optimizations that include a hybrid validation approach, effective hitting set enumeration techniques, one-pass candidate validations, and parallelization. Our experiments show that FDHITS, even without parallel execution, has a median speedup of 8.1 compared to state-of-the-art FD discovery algorithms while using significantly less memory. This allows the discovery of all FDs even on datasets that could not be processed by the current state-of-the-art.

CCS Concepts: • **Information systems** → **Information integration**; • **Theory of computation** → **Data structures and algorithms for data management**.

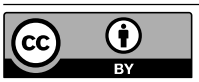
Additional Key Words and Phrases: data profiling; metadata; integrity constraints; data cleaning; normalization

ACM Reference Format:
Tobias Bleifuß, Thorsten Papenbrock, Thomas Bläsius, Martin Schirneck, and Felix Naumann. 2024. Discovering Functional Dependencies through Hitting Set Enumeration. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 43 (February 2024), 24 pages. https://doi.org/10.1145/3639298

1 MODERN FD DISCOVERY

Data profiling is the process of improving the understandability and usability of datasets by automatically capturing a variety of metadata that describes their structure and interrelationships [1]. This metadata is often expressed in terms of dependencies, and one of the most widely recognized kinds of dependencies are functional dependencies (FDs). An FD $S \rightarrow A$ expresses a relation between the attribute set S and the attribute A: All record pairs that share the same values for all S also need to hold the same value for A. If $S \rightarrow A$ is valid for S but not for any subset of S, then it is a valid and minimal FD. Knowing the FDs of a dataset is particularly important for schema

Authors' addresses: Tobias Bleifuß, tobias.bleifuss@hpi.de, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany; Thorsten Papenbrock, papenbrock@informatik.uni-marburg.de, Philipps University of Marburg, Marburg, Germany; Thomas Bläsius, thomas.blaesius@kit.edu, Karlsruhe Institute of Technology, Karlsruhe, Germany; Martin Schirneck, martin.schirneck@univie.ac.at, University of Vienna, Faculty of Computer Science, Vienna, Austria; Felix Naumann, felix.naumann@hpi.de, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany.



ACM 2836-6573/2024/2-ART43
https://doi.org/10.1145/3639298

normalization [12, 13, 34], but FDs also support query optimization [22, 35], data integration [23, 28], and data translation [8, 10] activities. Furthermore, when used as integrity constraints, FDs improve data cleaning applications to detect and resolve data inconsistencies [9, 20, 25]. While for smaller schemata and datasets, it might be viable to find (or specify) the dependencies manually, this quickly becomes infeasible for larger schemata. Despite the size of *complete* metadata, i.e., dependency sets, many use cases expect the dependencies to be complete to improve their impact [22] or utilization efficiency [34]. For this reason, most use cases of FDs take the set of valid FDs as input, which requires an efficient automatic discovery algorithm.

A field in which we generally still see a lot of leeway besides the already mentioned applications of FDs is dynamic data. For example, Explain-Da-V is a recent system to explain changes between dataset versions [37]. This system requires FDs between consecutive table versions, which rules out incremental approaches. To make such systems scale beyond the relatively small data sets with few versions used in their experiments, there is a need for faster FD discovery algorithms such as the FDHits approach of this paper.

Both the search space of FD candidates and the potential number of valid, minimal FDs grow exponentially with the number of attributes in a relational instance [15]; FD discovery is, in fact, known to be not only NP- but also W[2]-complete with respect to the solution size [7]. However, many real-world datasets are rather narrow and long, which is, they have only few attributes but many records. Due to this shape and modern search and pruning techniques, the candidate checking (together with necessary preprocessing) often dominates the overall runtime. An effective FD discovery algorithm consequently needs to optimize for both dimensions.

In the past thirty years, various approaches have been developed to make the discovery of functional dependencies more efficient. We discuss the most important ones in Section 3 and provide an overview in Table 1. Traditionally, there are essentially two classes of algorithms: one that is good at dealing with long, narrow datasets and one that is good at dealing with short, wide datasets. The prior approach HyFD [33] bridges this gap with a hybrid approach that combines ideas from both classes together with an efficient sampling phase. To date, HyFD is the most effective FD discovery algorithm in related work and shall, therefore, serve as our baseline.

Despite many important advances in the field of FD profiling, the discovery of all minimal FDs still takes a significant amount of resources when processing datasets of practically relevant (multiple gigabyte) size: First, the profiling times of most algorithms are unnecessarily high, because the algorithms perform expensive minimality pruning and redundant checks and/or comparisons. This happens, for example, when a level-wise candidate search, that enumerates candidates from small to large, misses pruning potential, the number of record comparisons is too high, and no intermediate results are considered for candidate validations. Second, all known profiling solutions consume a considerable amount of memory, because they require to keep either all or a large portion of the validated FD candidates in main-memory for search space pruning. Due to the exponential search space size, this often results in many millions or even billions of in-memory candidates. Especially for datasets with large solution sizes, these approaches cause an extremely high memory load. State-of-the-art answers to this memory issue propose to either give up the completeness of the results, which is quite unsatisfactory, or swap memory to disk, impacting runtime.

We propose the FD discovery algorithm FDHits, which improves the runtime performance and memory consumption of the profiling process by modeling the FD discovery task as a *hitting set problem*. Making use of established profiling strategies from HyFD [33] (hybrid sampling and validation elements) and HPIValid [5] (search for unique column combinations via hitting set enumeration), FDHits makes the following main contributions:

(1) Efficient FD discovery: We integrate several optimizations for effective FD discovery, namely hybrid validation, hitting set enumeration, one-pass candidate validations, and parallelization, into a single profiling algorithm: FDHITS.

(2) Hitting set enumeration: We apply and adapt the idea of hitting set enumeration under incomplete information to the problem of functional dependency discovery.

(3) One-pass candidate validation: We develop a new enumeration scheme for FD candidates that handles all dependent attributes jointly to save validation operations.

(4) Exhaustive evaluation: We evaluate FDHITS and its optimizations in various experiments that show significant improvements over related work for both runtime and memory requirement.

We first discuss the foundations of FD discovery in Section 2 and related work in Section 3. We then review the HPIVALID algorithm in Section 4 to the extent necessary to describe our approach. Section 5 presents our algorithm in its two variants $\text{FDHITS}_{\text{sep}}$ and $\text{FDHITS}_{\text{joint}}$. Section 6 reports on our extensive comparative evaluation and shows on average a many-fold superiority of our approach in terms of runtime and memory consumption on more than 40 datasets. Finally, we conclude in Section 7.

2 FOUNDATIONS OF FD DISCOVERY

Functional dependencies are defined on relational attributes and need to be validated against concrete relational instances. Throughout the paper, we use the following notation:

- upper-case letters A, B, C, D for individual attributes;
- upper-case letters S, T, V, W, X, Y for attribute sets;
- upper-case letter R for a relational schema;
- lower-case letter r for a relational instance;
- lower-case letter t for tuples/records in such an instance.

For a record t, we write $t[A]$ or $t[S]$ to denote the projection of t on attribute A or set S, respectively. For any schema R, we denote the number of attributes as $|R|$ and, for any relational instance r, we denote the number of records as $|r|$. With this notation, functional dependencies are defined as follows.

Definition 2.1. Given a relational instance r of schema R, an attribute set $S \subseteq R$, and an attribute $A \in R$, the **functional dependency** $S \rightarrow A$ is *valid* on r, if and only if no two records $t_1, t_2 \in r$ exist, such that $t_1[S] = t_2[S]$, but $t_1[A] \neq t_2[A]$.

A valid functional dependency $S \rightarrow A$ is *minimal* if there are no valid dependencies $S' \rightarrow A$ with $S' \subsetneq S$. If the FD $S \rightarrow A$ is valid, so is every $T \rightarrow A$ with $T \supseteq S$. It thus suffices to discover the minimal FDs. A functional dependency $S \rightarrow A$ is *non-trivial* if $A \notin S$. Only non-trivial FDs need to be discovered, trivial FDs are always valid. For an FD $S \rightarrow A$, we call S the *determinant* attributes and A the *dependent* attribute. With this, we can define the problem of *functional dependency discovery*. Given a relational instance r, output all valid, minimal, and non-trivial FDs for r exactly once.

Most FD discovery algorithms, including our FDHITS approach, model the search space of FD candidates with either a single or multiple powerset lattice(s) of attribute sets. Figure 1 shows an example of a model that represents the search space of each dependent attribute as a powerset lattice of determinant attribute sets. In this way, every node represents an FD candidate and the edges model specialization/generalization implications between the FD candidates. With an increasing number of attributes, this search space (and the potential number of minimal FDs within it) grows exponentially. The challenge of FD discovery is therefore to traverse this candidate space effectively (not necessarily modeled as a lattice in other approaches) without materializing major parts of it

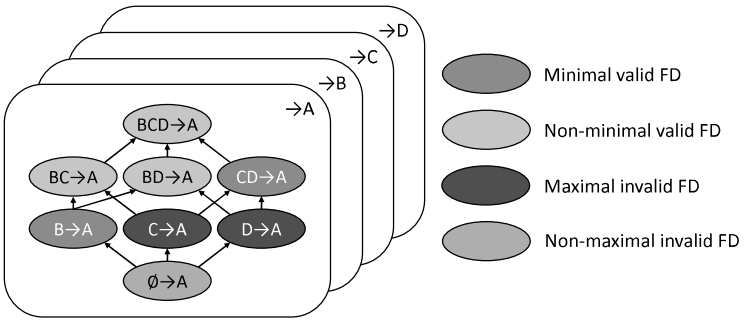


Fig. 1. The FD search space modeled as power set lattices.

and by maximizing the pruning of candidates. The next section gives an overview of how previous works tackled this challenge.

3 RELATED WORK

Because the field of research on functional dependency discovery is broad, we refer to Abedjan et al. [1] for an in-depth overview of the many existing algorithms and discovery approaches. For a systematic evaluation of the most popular FD discovery algorithms, we recommend [32]. In this section, we focus on works that contributed important techniques to our algorithm. We provide a summary of their most pertinent properties in Table 1.

Table 1. Comparison of related work approaches in four rated dimensions. The first two dimensions typically correlate with the scalability in the number of attributes, and the last two dimensions with scalability in the number of records.

	TANE [21]	**FDEP [18]**	**Dep-Miner [24]**	**FastFDs [43]**	**DFD [2]**	**HyFD [33]**	**FDHITS** (ours)
Active search space	− Level-wise traversal	+ Compact tree representation	− Level-wise traversal	++ Depth-first traversal	−− Entire lattice	+ Compact tree representation	++ Depth-first traversal
Minimality check	+ Apriori-gen	~ Tree-based lookup	+ Apriori-gen	− Subset checking	− Subset checking	~ Tree-based lookup	++ Critical edges (MMCS)
Validation speed / Tuple comparisons	++ Incremental PLI-based validation	−− Compares all tuple pairs	− Avoids some comparisons	− Avoids some comparisons	+ PLI-based validation	++ Sample & hash-based validation	++ Sample & incremental PLI-based
Validation memory usage	− Level-wise PLIs	++ No validation	++ No validation	++ No validation	~ LRU cache	+ On-demand hash-based	+ Current path of tree

TANE [21] is one of the first and most popular FD profiling algorithms. It traverses the search space in an Apriori-style, bottom-up fashion [3], which increases the set of determinant attributes incrementally. To make the search efficient, TANE relies on minimality and key pruning rules that can rule out certain FD candidates before checking them. For the actual checking of FD candidates, TANE introduces the concept of stripped partitions. As do many other FD discovery algorithms, we refer to these partitions as position list indices (PLIs) and adopt the PLI-based validations from TANE. Section 5.3 explains PLIs in detail and how they are implemented in FDHITS. In comparison

to our depth-first approach, TANE's level-wise approach must hold large portions of the search space in memory and scales poorly with the schema size.

The **fdep** algorithm [18] discovers FDs by mining all so-called *difference sets* and, then, systematically inferring all valid, minimal FDs from them. We also make use difference sets and give more details in Section 4. The authors of fdep propose a novel data structure, the FD-tree, to store all difference sets and, then, efficiently infer all valid FDs. The inference-based approach is especially efficient for datasets with many attributes and only few records. However, its runtime suffers under datasets with many records because it needs to compare all record pairs to calculate the difference sets. In contrast, FDHITS does not require a complete set of difference sets as input and uses a very different inference approach, which at any time materializes only small parts of the search space.

Dep-Miner [24] deduces FDs based on the concept of *agree sets*, which are the counterparts of difference sets. First, the algorithm calculates all agree sets based on PLIs. In subsequent stages, it computes maximal agree sets, which indicate maximal invalid FDs, to, then, derive the valid FDs from the agree set complements. In a last stage, the algorithm searches these complements in a level-wise, bottom-up Apriori-style to generate all valid and minimal FDs. FastFDs [43] builds on the ideas of Dep-Miner, but instead of the level-wise approach, it uses a depth-first search strategy to find the minimal, valid FDs. The proposed depth-first search requires extensive subset checking to ensure the minimality property of the discovered FDs, which FDHITS overcomes by using a different enumeration approach that relies on critical edges [29] improving efficiency. In contrast to our algorithm, both Dep-Miner and FastFDs calculate the complete sets of agree or difference sets, respectively, which does not scale well with the number of records.

The **DFD** [2] algorithm solves the FD discovery problem by splitting the search space in one lattice per dependent attribute. On each of these lattices, DFD performs random-walks in a depth-first approach. The advantage of this approach is that it can adapt well to datasets with either small or large results. It can prune large parts of the search space whenever it finds valid or invalid FDs instead of having to rely on a level-wise approach, which works only well either for small or large results (depending on the search direction). Like other lattice-based approaches, the performance of DFD still suffers when applied to wide datasets [32]. Our approach and DFD have in common that they both perform depth-first search (depending on the variant on subspaces or on a joint search space).

To unite the strengths of lattice-based and record comparison-based FD discovery algorithms, **HyFD** [33] suggests a new sampling-based approach. It combines the two different strategies, such that the result works well for wide and long datasets. HyFD starts by comparing a sample of record pairs and calculating a set of FDs that are violated by those record pairs. The algorithm then uses FD-Trees as suggested by Flach et al. [18] to induce all valid FDs that hold on the sample. A validator component then checks whether these FDs hold on the complete dataset and outputs those valid FDs. The validation proceeds level-wise through the candidate lattice, generating new candidates similar to the FUN algorithm [30]. If the number of invalid FD candidates grows larger than a certain threshold, HyFD returns to the sampling phase. For FDHITS, we adopt the idea of combining sampling and validation strategies, but introduce a new hybridization scheme and a novel reasoning component (Section 5.2). The latter is based on hitting set enumeration, which can be solved much more efficiently than FD-Tree-based inference. Both HyFD and our FDHITS propose strategies for parallelization. Meanwhile, a few adaptations of the HyFD algorithm have been proposed, such as DHyFD [40], which contribute minor optimizations. In this study, we have chosen to compare against the original algorithm, because the significance of our improvement also clearly surpasses the improvements of all variants of HyFD.

A closely related problem to FD discovery is the problem of finding unique column combinations (UCCs). These are attribute sets whose projection yields unique records and, hence, are key candidates. Every UCC functionally determines all other attributes in the dataset, which shows their connection to the FD discovery problem. It is, therefore, not surprising that the algorithms for both problems are similar and mutually inspiring. HPIValid [5] is the state-of-the-art UCC discovery algorithm. It enumerates hitting sets with partial information without having to materialize the result set. In this paper, we explore how these effective ideas can be transferred from UCC discovery to the problem of FD discovery.

Related to our research, but also orthogonal in their primary goal, are *distributed* FD discovery approaches. Tu and Huang present a distributed FD discovery algorithm that tries to minimize communication cost [38]. Further, Zhu et al. propose SmartFD [44] and Wu and Mao propose DistTFD [42], which are both distributed versions of HyFD on Apache Spark. For a more universal contribution to the field of distributed FD discovery, Saxena et al. identified general FD profiling primitives and introduced distributed implementations of these primitives for Apache Spark; with these primitives, they distributed various FD discovery algorithms [36].

As an orthogonal line of research, there also exist FD discovery algorithms for relaxed functional dependencies [11], such as approximate [6], conditional [17], or embedded functional dependencies [39]. As relaxation strategies usually extend exact algorithms, extending our own approach is left as promising future work, as discussed in Section 7.

4 HPIVALID IN A NUTSHELL

The overall structure of our FDHits algorithm for functional dependency discovery is based on the HPIValid algorithm [5] for unique column combinations. We first briefly explain the concepts of HPIValid that are necessary to understand our contribution.

Let r be a relation with schema R. A set of attributes $S \subseteq R$ is a *unique column combination* if, for any two different tuples $t_1, t_2 \in r$, there is an attribute $B \in S$ such that $t_1[B] \neq t_2[B]$. Intuitively, it is enough to know the values in S to distinguish all records. There is a well-known alternative characterization of UCCs in terms of so-called hitting sets of a hypergraph, see [26]. A *hypergraph* is a generalized graph in which edges can have more (or fewer) than 2 vertices. To connect this to databases, we use the following definition.

Definition 4.1. Given two different records $t_1, t_2 \in r$, their **difference set** is the set of all attributes $A \in R$, with $t_1[A] \neq t_2[A]$.

We take the schema R as the vertices of a hypergraph, and the difference sets for all pairs of records as the (hyper-)edges $\mathcal{D}$. The hypergraph is denoted $\mathcal{H} = (R, \mathcal{D})$. With this setup, some set $S \subseteq R$ of attributes is a unique column combination if and only if S *hits* every hyperedge, i.e., if $S \cap E \neq \emptyset$ for every $E \in \mathcal{D}$. If so, S is a *hitting set* of $\mathcal{H}$. Discovering all minimal unique column combinations of r is equivalent to enumerating all minimal hitting sets of $\mathcal{H}$.

Example. *For the example dataset in Table 2, the set* {`Time`, `Course`} *is a unique column combination. The only two records that have the same* `Time` *are* t_3 *and* t_4*. They can be distinguished by the attribute* `Course` *since it hits their difference set* {`Room_Nr`, `Course`, `Lecturer`}*. Conversely,* `Time` *can also not be omitted from the UCC as, for example,* t_1 *and* t_3 *have the same* `Course`*. The UCC is minimal.*

The above observations lead to the following two-step algorithm for UCC discovery. First, create the hypergraph of difference sets $\mathcal{H} = (R, \mathcal{D})$. Second, enumerate all minimal hitting sets of $\mathcal{H}$. However, both steps have a major caveat. Creating the hypergraph by computing the difference set for every pair of tuples $t_1, t_2 \in r$ takes quadratic time in $|r|$, which is not acceptable for many instances. Moreover, enumerating all minimal hitting sets of a hypergraph is a hard problem. In

Table 2. Example dataset

	Room_Nr	Time	Course	Lecturer
t_1	101	Wed 10:00 am	Programming	Miako
t_2	101	Wed 02:00 pm	Databases	Daniel
t_3	102	Fri 02:00 pm	Programming	Miako
t_4	101	Fri 02:00 pm	Databases	Saurabh

principle, the output can have exponential size in the number of attributes. Moreover, even if the output has reasonable size, it is a major open question whether the minimal hitting sets can be enumerated in output-polynomial time.

HPIvalid resolves these issues as follows. For the second step of enumerating hitting sets, the MMCS algorithm [29] is used. Though it has no theoretical performance guarantees, it is known to perform well in practice if there are not too many minimal hitting sets [19]. MMCS is a tree search that branches on the decision which edge to hit next and, after fixing the edge, tries out all possible options. Its efficiency comes from cleverly choosing the branching edges and from pruning the search space when it is safe to do so without loosing solutions. We describe MMCS in more detail in Section 5.1.

The other issue of having too many tuples to compute the difference sets for every pair is resolved as follows by HPIvalid. Not all difference sets are actually relevant. If we have two difference sets X and Y with $X \subseteq Y$, then fulfilling the requirement of X to select one attribute from X automatically fulfills the same requirement for Y. Thus, Y can be omitted from $\mathcal{D}$. Also, if we find the same difference set multiple times, it is clearly sufficient to keep only one copy. Thus, instead of the hypergraph of difference sets, its *minimization* is computed, containing only those difference sets that are not a subset of another. Whenever we speak of the hypergraph $\mathcal{H}$ of differences sets below, we mean its minimization.

The main difficulty here is that we do not know a priori which pairs of tuples give the *minimal* difference sets. Overcoming this is the core contribution of HPIvalid. It starts by randomly sampling some difference sets, yielding a tentative hypergraph $\mathcal{H}'$. With this partial information, MMCS is started to enumerate some minimal hitting sets. Whenever this search finds a hitting set S of $\mathcal{H}'$, there could be additional unseen difference sets that are not hit by S. Thus, the candidate solution S has to be validated by checking in the database r whether there are tuples that are still not distinguishable with just attributes from S. This can be done efficiently using *position list indices* (PLIs), which are described in more detail in Section 5.3. The validation has two possible outcomes, either S is indeed a UCC, or there are clusters c of tuples in the PLI such that any two tuples $t_1, t_2 \in c$ coincide on S. In the former case, HPIvalid can output S (it is then known to not only be a UCC but to also be minimal). In the latter case, the difference set of t_1 and t_2 is a new hyperedge that was not previously present in the tentative hypergraph $\mathcal{H}'$. HPIvalid simply adds the new difference set to the hypergraph and lets MMCS continue the enumeration where it left off. It was shown in [5] that, despite starting with only partial information $\mathcal{H}'$, MMCS does not miss any minimal hitting set of the true hypergraph $\mathcal{H}$.

In summary, HPIvalid consists of the following components.

Initial sampling. Sample pairs of tuples and compute their difference sets to form the initial tentative hypergraph $\mathcal{H}'$.

Tree search. Enumerate the minimal hitting sets of the current tentative hypergraph $\mathcal{H}'$ using the MMCS algorithm.

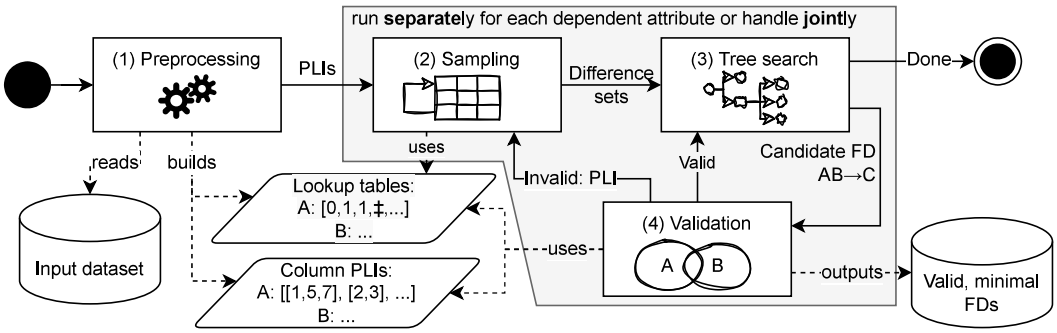


Fig. 2. Overview of the general discovery process of FDHITS.

Validation. Check whether a minimum hitting set S of $\mathcal{H}'$ is actually a UCC using PLIs. If not, the PLIs provide clusters of tuples that coincide on S.

Subsequent sampling. Sample additional pairs of tuples from within the clusters. The resulting difference sets are guaranteed to not be hit by S and thereby witness that S is not a UCC. They are added as new hyperedges to the tentative hypergraph $\mathcal{H}'$.

Hypergraph minimization. Whenever new hyperedges are added, $\mathcal{H}'$ is minimized, i.e., hyperedges that are supersets of smaller hyperedges are removed.

5 FUNCTIONAL DEPENDENCIES

The characterization of data dependencies via hitting sets extends also to functional dependencies, see [27]. As before, let r be a relation with schema R. Let $S \subseteq R$ be a set of attributes and A a single attribute. Then, the functional dependency $S \to A$ is valid in r if and only if any two tuples of r that differ in A also differ in at least one of the attributes of S. In this case, the attributes in S may not distinguish all record pairs, but knowing the values of $t[S]$ also determines the value $t[A]$. In terms of the hypergraph $\mathcal{H} = (R, \mathcal{D})$ of difference sets, $S \to A$ being valid is equivalent to S hitting every hyperedge of $\mathcal{H}$ that contains A. This motivates the definition of the subhypergraph $\mathcal{H}_A = (R, \mathcal{D}_A)$ *induced by* A that contains all difference sets that include A, i.e., $\mathcal{D}_A = \{e \in \mathcal{D} \mid A \in e\}$. With this, discovering all functional dependencies with dependent attribute A side is equivalent to enumerating all hitting sets of $\mathcal{H}_A$.

Example. *In Table 2, there are three difference sets that contain* Room_Nr. *These difference sets are* {Room_Nr, Time} *(from tuples* t_1 *and* t_3*),* {Room_Nr, Course, Lecturer} *(*t_3, t_4*), and their superset* {Room_Nr, Time, Course, Lecturer} *(*t_2, t_3*). The combination* {Time, Course} *is a minimal hitting set and therefore* {Time, Course} → Room_Nr *is a valid, minimal, and non-trivial functional dependency.*

This relation to the hitting set problem allows us to enumerate FDs with an algorithm that has a similar overall structure as HPIvalid. Our algorithm, which we call FDHITS, is illustrated in Figure 2. Although the structure is similar as for UCCs, the individual components need to be adapted to the enumeration of FDs instead. In the following, we discuss how we do this for the individual components. We start with the most interesting component, the tree search (3). For this, we provide approaches for separate and joint handling of the dependent attributes in Section 5.1, yielding two variants of our algorithm, $\text{FDHITS}_{\text{sep}}$ and $\text{FDHITS}_{\text{joint}}$. Although the joint handling is usually superior, it comes with the difficulty that minimizing the hypergraph of difference sets is no longer feasible. We discuss this issue and a method for selecting one of the two variants in

Section 5.2. Afterwards, we discuss the validation (4) and preprocessing (1) in Section 5.3, followed by the sampling (2) in Section 5.4.

5.1 Tree search

The straightforward generalization of the ideas in HPIvalid [5] to FD discovery is to treat the subhypergraphs $\mathcal{H}_A = (R, \mathcal{D}_A)$ separately for each attribute $A \in R$. We use this idea in the FDHITS$_{\text{sep}}$ variant of our algorithm. It runs a tree search (using MMCS) for each possible dependent attribute A with a tentative hypergraph $\mathcal{H}'_A$ that only accounts for those difference sets containing A.

We observe in our experiments in Section 6 that already FDHITS$_{\text{sep}}$ is more efficient than the previous state of the art on many datasets. However, since the subhypergraphs for different attributes usually have a large overlap, treating them as independent is bound to re-do the same or very similar computations multiple times, creating inefficiencies. One example for such duplicated work are the validations for candidate FDs that are executed alongside the tree search, see Section 5.3. We see in Section 6.5 that this makes up for a large portion of the computation time, especially for long datasets. Thus, avoiding recomputations can improve performance.

Handling multiple dependent attributes together can create synergies also in the tree search itself. As an illustration, consider the ideal case of two attributes A and B such that every difference set that contains A also contains B, which is equivalent to the FD $B \to A$ being valid. For their induced subhypergraphs $\mathcal{H}_A = (R, \mathcal{D}_A)$ and $\mathcal{H}_B = (R, \mathcal{D}_B)$ this means $\mathcal{D}_A \subseteq \mathcal{D}_B$, whence any hitting set S of $\mathcal{H}_B$ is also a hitting set of $\mathcal{H}_A$. In this case, one can, in principle, first discover all FDs with dependent attribute A, which only considers difference sets that are also relevant for B. From there it remains to additionally cover the difference sets relevant for B but not for A to also get the FDs with dependent attribute B.

Example. *In Table 2, the difference sets containing* `Course` *are* $\mathcal{D}_{\texttt{Course}} = \{\{\texttt{Time}, \texttt{Course}, \texttt{Lecturer}\}, \{\texttt{Room\_Nr}, \texttt{Course}, \texttt{Lecturer}\}\}$ *(ignoring the difference set containing all attributes). For* `Lecturer` *we have* $\mathcal{D}_{\texttt{Lecturer}} = \mathcal{D}_{\texttt{Course}} \cup \{\{\texttt{Time}, \texttt{Lecturer}\}\}$*, so* $\mathcal{D}_{\texttt{Course}} \subseteq \mathcal{D}_{\texttt{Lecturer}}$*. The minimal hitting sets of the hypergraph* $\mathcal{H}_{\texttt{Course}}$ *are* $T_1 = \{\texttt{Time}, \texttt{Room\_Nr}\}$*,* $T_2 = \{\texttt{Course}\}$*, and* $T_3 = \{\texttt{Lecturer}\}$*, yielding the minimal FDs* $T_1 \to \texttt{Course}$*,* $T_2 \to \texttt{Course}$*,*[1] *and* $T_3 \to \texttt{Course}$*. Regarding the FDs for the dependent attribute* `Lecturer`*,* T_1 *and* T_3 *also hit the additional difference set* $\{\texttt{Time}, \texttt{Lecturer}\}$ *of the hypergraph* $\mathcal{H}_{\texttt{Lecturer}}$*. Thus,* $T_1 \to \texttt{Lecturer}$ *and* $T_3 \to \texttt{Lecturer}$ *are also minimal FDs. For* T_2*,* `Time` *is added to also hit* $\{\texttt{Time}, \texttt{Lecturer}\}$*, yielding the additional FD* $\{\texttt{Course}, \texttt{Time}\} \to \texttt{Lecturer}$*.*

Of course, the above illustration is an idealized setting. Nevertheless, even if $B \to A$ is not valid but A and B still share many difference sets, handling them together can speed up finding new hitting sets for both hypergraphs simultaneously.

We therefore propose FDHITS$_{\text{joint}}$ as our main algorithmic contribution in this work. It discovers all functional dependencies in a single tree search. While this holds potential for performance improvements over FDHITS$_{\text{sep}}$, the management of multiple dependent attributes also leads to new difficulties. In order to describe our new search strategy and how to solve those difficulties, we review the original approach of MMCS along the way.

Branching. MMCS is a search algorithm for hitting sets that branches on edges. That means, each node of the search tree maintains a set of *selected vertices* $S \subseteq R$. In each step, the algorithm picks one edge $E \in \mathcal{D}$ that is not yet hit by S. As E needs to be hit by any solution, the algorithm must select at least one vertex from E. To not miss any solutions, the algorithm considers all options, i.e.,

[1] For the sake of this example, the trivial FDs $\{\texttt{Course}\} \to \texttt{Course}$ and $\{\texttt{Lecturer}\} \to \texttt{Lecturer}$ are still included. We explain later how to avoid them in the tree search.

Algorithm 1: The tree search of FDHITS$_{\text{joint}}$.

```
1  function treeSearch(S, V, W):
       ▷ Pruning
2      for (C, A) ∈ S × W do
3          if critical_S(C, A) = ∅ then
4              W ← W \ {A}
5      for B ∈ V do
6          if ∀A ∈ W ∃C ∈ S ∀E ∈ critical_S(C, A): B ∈ E then
7              V ← V \ {B}
8      if W = ∅ then
9          return
       ▷ Validation at the leaves
10     if uncov(S, W) = ∅ ∧ validate(S → W) then
11         output S → W
12         return
       ▷ Branching
13     E ← edge in uncov(S, W) minimizing |E ∩ V| + |W \ E|
14     {B_1, ..., B_k} ← E ∩ V
15     treeSearch(S, V, W \ E)                                   ▷ μ_0
16     for i ∈ {1, ..., k} do
17         treeSearch(S ∪ {B_i}, V \ {B_1, ..., B_i}, W ∩ E)      ▷ μ_i
```

for each vertex in $B \in E$ it creates a new branch from the current node that corresponds to adding B to the selected vertices in S.

Without some additional care, this may have the effect that the same hitting set is enumerated twice: Some edge $E = \{B, C\}$ causes two branches, the first for B and the second for C. However, it is possible that C is later selected also in the first branch, and A is selected in the second branch, resulting in the same selected vertices. To prevent this, an additional set $V \subseteq R \backslash S$ of *candidate vertices* is maintained in each node of the search tree. The interpretation is that only vertices from V are allowed to be added to the selection S.

Our tree search in FDHITS$_{\text{joint}}$ is based on MMCS and uses a similar branching. To handle multiple dependent attributes simultaneously, we additionally maintain a third set $W \subseteq R$ of *possible dependent attributes*. To properly describe the extended branching and to argue for its correctness, we need to introduce a bit more notation. As before, let $\mathcal{H} = (R, \mathcal{D})$ be the considered hypergraph and for every $A \in R$, let $\mathcal{H}_A = (R, \mathcal{D}_A)$ be the subhypergraph induced by A. Let $\mu = (S, V, W)$ be a node of the search tree with selected vertices S, candidate vertices V, and possible dependent attributes W. For the subtree below μ, the goal is to enumerate all FDs $T \to A$ with $A \in W$ and $S \subseteq T \subseteq S \cup V$. Equivalently, in terms of the hypergraph, the goal is to enumerate for every $A \in W$ all minimal hitting sets T of $\mathcal{H}_A$ with $S \subseteq T \subseteq S \cup V$. Starting the search in the root $(S = \emptyset, V = R, W = R)$ discovers all minimal FDs.

Algorithm 1 shows pseudocode for the tree search. A call to the function `treeSearch`(S, V, W) corresponds to the node (S, V, W). Thus, the search is started by calling `treeSearch`$(\emptyset, R, R)$.

To define the branching as well as to discuss the base case, i.e., the leaves where we output the solutions, let $\text{uncov}(S, W)$ be the set of hyperedges of $\mathcal{H}$ that contain a vertex from W but are not yet hit by S. If $\text{uncov}(S, W)$ is empty for the node $\mu = (S, V, W)$ (lines 10–12 in Algorithm 1), then μ is a leaf in that $S \rightarrow W$ is a valid FD. In more detail, it is an FD candidate that is either output after successful validation on the relational instance r, or for which the subsequent sampling finds a new difference set belonging to $\text{uncov}(S, W)$. The FD might not be minimal, but checking for minimality is not difficult. This will become clearer below when we discuss pruning.

Otherwise, $\text{uncov}(S, W)$ is not empty, either because it was not empty to begin with or because the validation procedure added a new difference set to it. Then we branch on an uncovered edge (lines 13–17 in Algorithm 1). As a heuristic, we select some edge $E \in \text{uncov}(S, W)$ for which $|E \cap V| + |W \backslash E|$ is minimum. Let $E \cap V = \{B_1, \ldots, B_k\}$ be the candidate vertices in E. We branch on E by creating $k+1$ child nodes $\mu_0, \ldots, \mu_k$. The child $\mu_0 = (S, V, W \backslash E)$ accounts for the fact that hitting E is only relevant for dependent attributes that are also contained in E. Thus, for all dependent attributes in $W \backslash E$, we do not have to hit E.

This is a new branching option that stems from handling multiple dependent attributes together. For the dependent attributes in $W \cap E$, we branch on which vertex from E to add to the selected vertices. This yields the children $\mu_1, \ldots, \mu_k$ with $\mu_i = (S \cup \{B_i\}, V \backslash \{B_1, \ldots, B_i\}, W \cap E)$.

Example. *Let us assume* FDHits$_{\text{joint}}$ *started to work on Table 2, so we have* $S = \emptyset$ *and* $V = W = \{\texttt{Room\_Nr}, \texttt{Time}, \texttt{Course}, \texttt{Lecturer}\}$. *Say the selected difference set stems from comparing* t_1 *and* t_2*, which is* $E = \{\texttt{Time}, \texttt{Course}, \texttt{Lecturer}\}$. *$E$ is ignored in child* μ_0*, the search continues with* $S = \emptyset$*, a single dependent attribute* $W = \{\texttt{Room\_Nr}\}$ *and candidate vertices* $V = \{\texttt{Room\_Nr}, \texttt{Time}, \texttt{Course}, \texttt{Lecturer}\}$. *In the subtree rooted at* μ_0*, only FDs* $T \rightarrow \texttt{Room\_Nr}$ *are found, where T can be any set of attributes. In the child node* μ_1 *that hits edge E via the attribute* `Time`*, we get* $S = \{\texttt{Time}\}$ *and* $W = \{\texttt{Course}, \texttt{Lecturer}\}$*, V is updated to* $\{\texttt{Room\_Nr}, \texttt{Course}, \texttt{Lecturer}\}$. *The child nodes* μ_2 *and* μ_3 *for the attributes* `Course` *and* `Lecturer` *are analogous.*

Correctness of this branching follows from a simple inductive argument. The induction hypothesis is the goal mentioned above, which is restated in the following lemma.

Lemma 5.1. *In the subtree below a node $\mu = (S, V, W)$, the tree search finds each minimal valid functional dependencies $T \rightarrow A$ with $A \in W$ and $S \subseteq T \subseteq S \cup V$ exactly once.*

Proof. The base case is given by the leaves. If $\text{uncov}(S, W)$ is empty, then S is a hitting set for $\mathcal{H}_A$ for $A \in W$. Minimality is tested explicitly before any output, so those FDs $S \rightarrow A$ with $A \in W$ that are indeed output are exactly the desired ones.

For the induction step, we assume that, after branching, the induction hypothesis holds for the children $\mu_0, \ldots, \mu_k$. Let $T \rightarrow A$ with $A \in W$ and $S \subseteq T \subseteq S \cup V$ be one of the minimal FDs that needs to be found below the subtree of μ. If $A \notin E$, then $T \rightarrow A$ will be found below $\mu_0 = (S, V, W \backslash E)$ and below none of the other μ_i. Otherwise, we have $A \in W \cap E$. Recall that $E \cap V = \{B_1, \ldots, B_k\}$ are the candidate vertices added to S. Let i be the smallest index such that $B_i \in T$. Then $T \rightarrow A$ will be found in $\mu_i = (S \cup \{B_i\}, V \backslash \{B_1, \ldots, B_i\}, W \cap E)$. Moreover, it will not be found below any other μ_j with $j \neq i$ as for $j > i$, B_i is explicitly excluded from the set of candidates V, and for $j < i$, $B_j \in T$ and thus i cannot be the smallest index such that $B_i \in T$. □

Pruning trivial and non-minimal FDs. As mentioned above, we find all minimal solutions at the leaves of the search tree but some leaves might correspond to non-minimal solutions. Here we describe how to recognize these cases. This also leads to a rule for early pruning of branches that are guaranteed to not contain a solution. As for the branching, we first briefly describe how this is done in MMCS, which directly applies to FDHits$_{\text{sep}}$. Afterwards, we show how something similar can be achieved for FDHits$_{\text{joint}}$ when simultaneously handling multiple dependent attributes.

It is a well-known fact, see e.g. [4], that a hitting set T is minimal if and only if each vertex $C \in T$ has a *critical edge* (with respect to T) that contains C but no other vertex from T. If the set of currently selected vertices S already contains a vertex that has no critical edge (with respect to S), then S cannot be extended to a minimal hitting set and thus the tree search can be pruned at the current node. This observation can additionally be used to eliminate vertices from the set of candidates V. Specifically, a candidate vertex $B \in V$ is a *violator* if adding B to S caused this type of pruning. A violator can be safely removed from the set of candidates V.

For jointly handling multiple dependent attributes in FDHits$_{\text{joint}}$, more care must be taken when pruning the search tree. In Algorithm 1, pruning happens in lines 2–9. Let $\mu = (S, V, W)$ be the current node and let $A \in W$ be one of the dependent attributes. If a vertex from S has no critical edge in the subhypergraph $\mathcal{H}_A$, then there is no minimal FD $T \to A$ with $S \subseteq T$. As there are other dependent attributes, we cannot simply prune the search at μ. However, we can safely remove A from W without violating the property stated in Lemma 5.1. If W runs empty, we can prune the tree at the current node (lines 8–9). To implement this, we maintain for every $C \in S$ and every $A \in W$ the set $\text{critical}_S(C, A)$ that contains the critical edges for C (with respect to S) in the subhypergraph $\mathcal{H}_A$. The above pruning then means that if $\text{critical}_S(C, A)$ is empty, then A can be removed from W (lines 2–4). Moreover, to adapt the concept of violators, we remove a candidate vertex $B \in V$ from V if adding B to S had the effect that *every* dependent attribute $A \in W$ would be removed from W due to the pruning described above. In terms of $\text{critical}_S(C, A)$, this is the case if for every dependent attribute $A \in W$ there is an already selected vertex $C \in S$ such that all critical edges $\text{critical}_S(C, A)$ of C also contain B (lines 8–9). This is correct as in this case there is no minimal valid FD $T \to A$ with $S \cup \{C\} \subseteq T$ and $A \in W$, i.e., the property stated in Lemma 5.1 remains true.

The result above also tells us how to avoid trivial FDs $S \to A$ with $A \in S$. While they are valid, we do not need to discover them. By Lemma 5.1, it is enough to keep S and W disjoint. When creating the child node μ_i, in which attribute $B_i \in E \cap V$ is added to S, we remove B_i from W in the child (if previously present).

5.2 Minimization and strategy selection

Although FDHits$_{\text{joint}}$ is superior to FDHits$_{\text{sep}}$ in that it saves PLI intersections by validating FDs candidates for different dependent attributes simultaneously, it has one major downside. Recall from Section 4 that HPIvalid minimizes the hypergraph of difference sets, i.e., for two difference sets X and Y with $X \subseteq Y$, Y can be omitted from the hypergraph. This is still true for FDHits$_{\text{sep}}$, as the subset X still poses a stronger requirement than Y (assuming both contain the current dependent attribute A and are thus relevant for A). When considering multiple dependent attributes, however, the difference set Y can still be relevant for dependent attributes that are in $Y \setminus X$. As finding the difference sets that are irrelevant for all dependent attributes is computationally too expensive, FDHits$_{\text{joint}}$ lacks the minimization of the input hypergraph. This may result in the input for the tree search of FDHits$_{\text{joint}}$ becoming substantially larger than that for FDHits$_{\text{sep}}$, which slows down the computation.

To mitigate this effect, we designed FDHits as a hybrid system that invokes either FDHits$_{\text{sep}}$ or FDHits$_{\text{joint}}$ depending on which strategy likely performs best. We propose to make this decision via a simple, but effective heuristic. The initial sampling of record pairs is common to both variants. After this phase, we compare the number of distinct difference sets in the initial tentative hypergraph, to the number of record pairs sampled. We calculate $\frac{\#\text{difference sets}}{\#\text{record pairs}}$, which is the ratio of comparisons that actually contributed a new edge. If the resulting hypergraph is large, i.e., this fraction is above some threshold, we use FDHits$_{\text{sep}}$; otherwise, we use FDHits$_{\text{joint}}$. Our evaluation shows that a threshold of 0.5 is a robust choice.

5.3 Preprocessing and validation

A common data structure used to represent the input data for UCC or FD discovery are so-called *Position List Indices (PLIs)* (or partitions) [14] that can be used to check whether a functional dependency is valid for a relational instance [21]. The main idea is as follows. The PLI π_S for a subset of attributes $S \subseteq R$ partitions the database into *clusters* such that two records (represented by their ID) are in the same cluster if and only if they agree on S. To see how PLIs are a useful concept for validating candidate solutions, note that S is a UCC if and only if each cluster of π_S contains only one record. Similarly, $S \rightarrow A$ is a functional dependency if and only if for each cluster of π_S, all records in this cluster coincide on A. For the validation of FDs, clusters of size one are not relevant and can be removed. Such reduced lists are also called *stripped partitions* [21].

Example 5.2. *In Table 2, the PLI* $\pi_{\{\texttt{Lecturer}\}}$ *has the three clusters* $[1, 3]$, $[2]$, *and* $[4]$. *For the candidate functional dependency* $\{\texttt{Lecturer}\} \rightarrow \texttt{Course}$, *we can observe that* t_1 *and* t_3 *coincide on* `Course`. *Thus, this is a valid FD. Moreover,* $\{\texttt{Lecturer}\} \rightarrow \texttt{Room\_Nr}$ *is invalid as* t_1 *and* t_3 *have different room numbers.*

In contrast to previous works, FDHits can significantly shrink PLIs, by removing whole clusters that are not relevant for the current dependent attribute A in the case of $\text{FDHits}_{\text{sep}}$ or any of the attributes in W for $\text{FDHits}_{\text{joint}}$. A cluster is relevant for a dependent attribute A, if the tuples contained in it do not coincide on A, which is a prerequisite for violating any FD $S \rightarrow A$. Because of this pruning, every cluster in a PLI π_S computed for the right-hand sides W must contain at least one pair of tuples that violate $S \rightarrow A$ for at least one $A \in W$. Hence, if this PLI is not empty, at least one of the current FD candidates is invalid. This optimization can lead to speed-ups of more than an order of magnitude on some datasets, as we show in Section 6.5.

As PLIs are a common data structure in UCC or FD discovery, we now only briefly discuss implementation details and refer to the literature for more elaborate descriptions [2, 21, 33]. In the preprocessing, we compute the PLI for each individual attribute. To this end, FDHits reads the input dataset record by record and, for each attribute, it constructs a hashmap that maps each value to a list of record IDs that it appears in. The values of each completed hashmap represent the PLI of the respective attribute; the hashmaps' keys are dropped, because they are no longer needed, and also singleton ID sets are stripped from the PLIs. This construction step takes linear time in the input size $|r| \cdot |R|$, if we assume expected constant access to the hashmaps.

The calculation of PLIs for larger attribute sets is performed through an operation called PLI *intersection* (or *partition refinement*) [21]. Given the PLI π_S for some attribute set $S \subseteq R$ and an attribute A ($A \notin S$), we want to efficiently compute $\pi_{S \cup \{A\}}$ based on previously computed PLIs. To support this operation, we need a *lookup table* (constructed together with the PLIs for single-attributes) that maps for each attribute from the record IDs to the cluster containing the respective record. The clusters are integer-encoded, with a special marker (†) for clusters of size 1 hat have been stripped.

Example. *The lookup table for* `Lecturer` *looks like this:* $[1, \dagger, 1, \dagger]$. *Assume that we want to intersect* $\pi_{\{\texttt{Course}\}} = [[1, 3], [2, 4]]$ *with* `Lecturer` *to get the PLI* $\pi_{\{\texttt{Course,Lecturer}\}}$. *The cluster* $[1, 3]$ *stays untouched, because both indices have the same value in the lookup table. The cluster* $[2, 4]$ *turns into two singleton clusters, which are removed (stripped). Hence, the result is* $\pi_{\{\texttt{Course,Lecturer}\}} = [[1, 3]]$.

5.4 Sampling of difference sets

For the sampling of difference sets, we follow the approach of HPIvalid [5] with some adjustments that are necessary due to the differences between UCCs and FDs. Sampling is always done for a given PLI π_S where $S \subseteq R$ is an attribute set. We sample uniformly at random among the pairs of

records that are in the same cluster of π_S. This is repeated c_p^ε times where c_p is the number of such record pairs, which can be quadratic in the number of records. In our experiments, we show that $\varepsilon = 0.3$ is a good choice for the sampling exponent (which was also proven to be a good choice for UCC discovery in HPIvalid).

The initial sampling is done once with the PLI π_A for each attribute $A \in R$. In the case of FDHits$_{\text{sep}}$, the tentative hypergraph that results from processing one dependent attribute is reused for the remaining ones as well.

In the following description, we use the notation $S \to W$ for a generalized functional dependency that is valid if and only if all FDs $S \to A$ with $A \in W$ are valid. The subsequent sampling happens whenever the verification of a candidate solution $S \to W$ fails. In this case, we sample with respect to the PLI π_S. Note that this samples difference sets that coincide on S, thus yielding hyperedges that are not yet hit by S.

Example. *In Example 5.2, we observe that* {Lecturer} → Room_Nr *is invalid as* t_1 *and* t_3 *are in the same cluster* $[1, 3]$ *but have different room numbers. Sampling in* $[1, 3]$ *yields the difference set of* t_1 *and* t_3*, which is* {Room_Nr, Time}*. This difference set is a witness for the fact that* {Lecturer} → Room_Nr *is no FD and it thus extends the current tentative hypergraph of difference sets.*

For the subsequent sampling, this is, however, not the full story. For UCC discovery every difference set sampled this way is not yet hit and thus yields new information. For the FD discovery, this is not true. If $S \to W$ is found to be invalid, we sample new difference sets not hit by S. However, they are not guaranteed to contain attributes from W, in which case they are not relevant for the current dependent attribute. To ensure that we make progress, we first deterministically add one new difference set that comes from a pair of records in one cluster of π_S that differ in the dependent attributes W. This is guaranteed to exist, as the validation would have been successful otherwise. More specifically, because of the filtering explained in Section 5.3, the FD is valid if the PLI is empty. If it is not empty, it is sufficient to inspect any of the clusters that it contains to find a violation. The filtering is also helpful for the sampling in general, because it guarantees that every cluster contains at least one tuple pair that yields new information.

6 EVALUATION

We evaluate the hybrid FDHits as well as its two variants FDHits$_{\text{sep}}$ and FDHits$_{\text{joint}}$ comparatively demonstrating that all variants improve upon the state-of-the-art FD discovery algorithm HyFD by large factors. We first present the setup and methodology before we report and analyze the runtime and memory results on various datasets. To analyze FDHits' runtime behavior on datasets of increasing size, we then measure the FD discovery times while gradually scaling either the number of records or the number of attributes in the input relation. Finally, we explore some special properties of FDHits in a set of in-depth experiments.

6.1 Setup

We run all experiments on a Dell® R620 server with two Intel® Xeon® E5-2650 and 256 GB of DDR3-1600 RAM. The server runs Ubuntu 20.04.4 LTS, Rust 1.59 and OpenJDK 11.0.15. For our competitor algorithms HyFD, FDEP and TANE, we use the Java-implementations of the Metanome project [31]. We execute all algorithms with a heap limit of 128 GB; for FDHits, we use a Rust-implementation that was compiled with the `lto` flag in `release` mode. Our experiments use the same datasets as the evaluation of HPIvalid [5]. For repeatability, we publicly provide both code and datasets[2].

[2]https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html

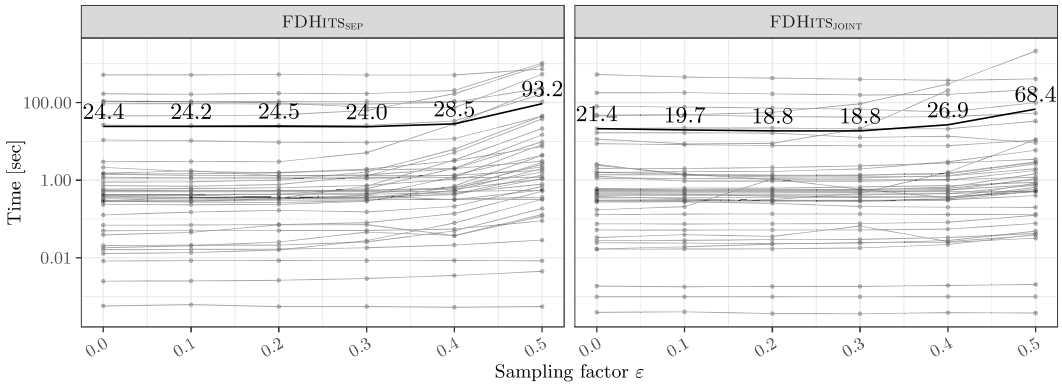


Fig. 3. Influence of the sampling factor ε on the runtime of both variants of FDHits. The thin lines represent individual datasets, and the bold line is the average over all datasets.

For HyFD, we opted for the already well-optimized original implementation and granted it additional benefits: To keep the differences between Java and Rust as small as possible, we proceed similarly to the experiments of HPIvalid in the comparison with HyUCC [5]: We start the JVM with the server flag and deduct the times spent at checkpoints from the runtimes. For runtime measurements, we use the runtimes reported by the algorithm to exclude all startup times of the Java virtual machines. For memory measurements, we report the peak memory usage as returned by `time -f'%M'`. Ideally, we would have a native implementation of HyFD to compare against. However, re-implementing the existing algorithm in Rust is difficult, and carries the risk of implementation flaws. Although we consider several aspects by the measures described above, other overheads such as object creation remain unconsidered, so the absolute numbers need to be taken with a grain of salt. Because the differences in the comparative results of HyFD and FDHits are so large, they are however far beyond what could be explained with programming language and implementation differences.

In relational databases, NULL is a special value that represents missing or inapplicable values. There are various ways to treat this value in FD discovery, each having a different influence on the validity of FDs that contain NULL fields. We configured both algorithms to use NULL-equals-NULL semantics, which is the most common default semantics for FD discovery [32]. For a broader discussion on NULL semantics for FDs, we refer to [1] and [41].

6.2 Parameter choice

FDHits is largely parameter-free, but there is one parameter ε that determines the number of samples per sampling phase, as described in Section 5.4. The larger this parameter is chosen, the more time the algorithm spends in the sampling phase. Thereby, it has the chance to complete the hypergraph faster, although it becomes more and more difficult to discover new edges, as the input graph becomes increasingly complete. A more complete input graph leaves FDHits with fewer invalid FDs to test and therefore less resampling phases, which potentially enables a faster tree-search (because it needs to enumerate fewer candidates). However, a smaller value for ε greatly reduces the number of record comparisons and, hence, allows searching for new edges in a much more targeted way.

To find a robust sampling factor ε, Figure 3 shows the influence of different settings on the runtime of all datasets, as well as the average runtime over all datasets. Overall, the influence of ε is negligible on most datasets, if a rather small value is chosen. $\text{FDHits}_{\text{joint}}$ benefits a bit from a

Table 3. Performance results on various datasets for our FDHITS$_{sep}$ and FDHITS$_{joint}$ and the state-of-the-art FD discovery algorithms TANE, FDEP, and HyFD. For each dataset/algorithm combination, we report the average over 5 runs. TL denotes runs for which the algorithm did not finish within our 1h time limit, and ML for which the memory limit of 128GB did not suffice. Best runtimes and those within a 5% range of them are highlighted in bold. The runtimes of the variant that our heuristic chooses are underlined, and we report the speedup of this variant over HyFD, on average the fastest competitor.

				TANE		FDEP		HyFD		FDHITS$_{sep}$		FDHITS$_{joint}$		
Dataset	$\|R\|$	$\|r\|$	#FDs	Time [s]	Mem.	Time [s]	Mem.	Time [s]	Mem.	Time [s]	Mem.	Time [s]	Mem.	Speedup
Iris	5	147	4	0.144	91.5 MB	0.040	71.9 MB	0.052	61.8 MB	0.000	27.2 MB	0.000	27.3 MB	163.6
T-Bioc-Metadata	56	4	2575	0.161	99.5 MB	0.041	68.2 MB	0.064	62.5 MB	0.007	27.3 MB	0.001	27.3 MB	56.6
Echocardiogram	13	132	527	0.209	100 MB	0.069	80.4 MB	0.095	68.2 MB	0.003	27.3 MB	0.002	27.3 MB	55.7
T-Bioc-Measurementsorfacts	24	3.11 k	449	0.488	198 MB	3.655	566 MB	0.267	128 MB	0.023	27.3 MB	0.012	27.3 MB	21.5
T-Bioc-Specimenunit-Mark	12	8.98 k	84	0.565	191 MB	16.901	642 MB	0.360	147 MB	0.024	27.3 MB	0.021	27.3 MB	17.0
Hepatitis	20	155	8250	6.145	859 MB	0.295	121 MB	0.384	235 MB	0.017	27.3 MB	0.026	27.3 MB	22.9
Nursery	9	13 k	1	1.452	426 MB	26.578	640 MB	0.357	124 MB	0.036	27.3 MB	0.032	27.3 MB	11.1
Sg-Taxon-Name	3	106 k	2	0.577	258 MB	738.985	2.36 GB	0.592	309 MB	0.050	27.3 MB	0.050	27.3 MB	11.8
T-Bioc-Multimediaobject	15	18.8 k	133	0.941	338 MB	135.130	694 MB	0.628	255 MB	0.077	27.2 MB	0.070	27.3 MB	8.9
Chess	7	28.1 k	1	0.999	334 MB	84.967	660 MB	0.366	158 MB	0.054	27.3 MB	0.071	27.3 MB	5.1
Amalgam1-Denormalized	87	50	450,020	32.719	2.17 GB	1.605	335 MB	0.950	436 MB	0.982	27.3 MB	0.172	27.2 MB	5.5
Spstock	7	122 k	56	1.486	669 MB	1871.975	2.5 GB	1.405	607 MB	0.299	27.3 MB	0.221	27.3 MB	6.4
T-Bioc-Gath-Agent	18	72.7 k	186	4.662	1.57 GB	2604.715	885 MB	1.479	613 MB	0.271	34.9 MB	0.244	34.8 MB	6.1
Sg-Bioentry-Ref-Assoc	5	358 k	5	2.141	809 MB	TL	-	2.526	647 MB	0.328	46.1 MB	0.269	46 MB	9.4
T-Bioc-Unit	14	91.3 k	69	4.922	1.51 GB	3412.231	878 MB	1.478	612 MB	0.284	46 MB	0.273	46 MB	5.4
T-Bioc-Id-Highertaxon	3	563 k	1	1.903	726 MB	TL	-	2.366	627 MB	0.356	43.8 MB	0.289	45.8 MB	8.2
T-Bioc-Preparation	21	81.8 k	363	2.645	924 MB	3066.389	1.51 GB	1.523	628 MB	0.376	38.2 MB	0.300	38.2 MB	5.1
Hospital	15	115 k	83	54.188	21.6 GB	TL	-	2.759	652 MB	0.339	27.3 MB	0.340	27.8 MB	8.1
Sg-Bioentry	9	184 k	19	1.562	570 MB	TL	-	1.720	613 MB	0.341	68.2 MB	0.343	68.2 MB	5.0
T-Bioc-Gath-Namedareas	11	138 k	59	3.485	1.43 GB	TL	-	2.738	649 MB	0.441	46.3 MB	0.384	46.3 MB	7.1
Entytysrcgen	46	26.1 k	1454	-	ML	1448.888	2.07 GB	25.923	1.24 GB	0.620	27.3 MB	0.401	27.3 MB	64.7
T-Bioc-Gath-Sitecoordinates	25	91.3 k	467	6.226	1.98 GB	TL	-	2.072	648 MB	0.556	43.9 MB	0.402	43.9 MB	5.2
Sg-Biosequence	6	184 k	9	2.235	783 MB	TL	-	2.179	757 MB	0.421	114 MB	0.428	114 MB	5.1
Sg-Reference	6	129 k	13	1.576	496 MB	2237.348	2.38 GB	1.380	585 MB	0.500	105 MB	0.501	105 MB	2.8
Sg-Dbxref	4	618 k	4	1.746	823 MB	TL	-	1.842	795 MB	0.530	135 MB	0.526	135 MB	3.5
Sg-Seqfeature-Qual-Assoc	4	825 k	3	2.222	1.12 GB	TL	-	2.600	1.04 GB	0.533	97.8 MB	0.548	97.8 MB	4.7
Letter	17	18.7 k	61	234.215	55.8 GB	138.529	761 MB	2.315	593 MB	0.133	27.3 MB	0.618	27.3 MB	17.4
T-Bioc-Gath	35	91 k	925	17.464	5.65 GB	TL	-	4.413	1.15 GB	1.154	47.2 MB	0.643	47.1 MB	6.9
Horse	29	300	128,727	TL	-	6.659	394 MB	4.888	1.52 GB	0.290	27.3 MB	0.657	27.3 MB	16.8
T-Bioc-Id	38	91.8 k	972	43.765	13.6 GB	TL	-	4.934	1.21 GB	1.479	68 MB	0.850	68.1 MB	5.8
Sg-Seqfeature	6	1.02 M	7	3.383	1.38 GB	TL	-	4.636	1.36 GB	1.063	182 MB	0.917	182 MB	5.1
Sg-Bioentry-Dbxref-Assoc	3	1.85 M	2	4.026	1.55 GB	TL	-	6.333	1.41 GB	1.227	125 MB	0.971	138 MB	6.5
Sg-Bioentry-Qual-Assoc	4	1.82 M	2	8.268	2.01 GB	TL	-	7.460	1.32 GB	1.090	137 MB	1.145	167 MB	6.5
Sg-Location	8	1.02 M	11	4.918	1.53 GB	TL	-	4.955	1.64 GB	1.302	281 MB	1.274	281 MB	3.9
Plista	63	996	178,152	TL	-	27.742	1.79 GB	18.721	4.46 GB	0.652	27.2 MB	2.062	27.3 MB	28.7
Flight	109	1 k	982,631	-	ML	210.770	3.2 GB	44.781	13.3 GB	1.732	27.3 MB	2.508	27.3 MB	17.9
Tax	15	1 M	263	1006.375	79.9 GB	TL	-	58.741	2.54 GB	8.931	190 MB	7.374	218 MB	8.0
Ditag-Feature	13	3.96 M	58	1979.883	125 GB	TL	-	623.954	8.81 GB	25.270	1.19 GB	19.294	1.19 GB	32.3
Fd-Reduced-30	30	250 k	89,571	34.923	4.24 GB	TL	-	289.790	8.63 GB	104.308	139 MB	19.889	139 MB	14.6
Census	42	196 k	41,861	-	ML	TL	-	TL	-	4.511	95.7 MB	23.183	163 MB	>798
Struct-Sheet-Range	32	664 k	9150	-	ML	TL	-	407.227	20.9 GB	91.496	404 MB	41.729	404 MB	9.8
Pdbx-Poly-Seq-Scheme	13	17.3 M	68	TL	-	TL	-	262.810	21.4 GB	74.853	3.44 GB	58.350	3.44 GB	4.5
Musicbrainz-Denormalized	100	79.6 k	1,678,277	TL	-	TL	-	TL	-	51.690	85.1 MB	82.428	260 MB	>69
Ncvoter	19	8.06 M	822	TL	-	TL	-	2459.640	24.5 GB	159.354	3.56 GB	156.788	3.56 GB	15.7
Lineitem	16	6 M	3984	TL	-	TL	-	1965.967	25.1 GB	499.043	1.7 GB	416.412	1.72 GB	4.7

slightly larger sample, but for both algorithm variants, the minimum average runtime was achieved with a sampling factor ε of 0.3. One reason that FDHITS$_{sep}$ benefits less from the larger sample could be that the algorithm can carry over most edges from previous iterations, and therefore does not discover as many new edges through sampling in later iterations. Hence, one could justify a more conservative choice with a smaller ε, but we chose $\varepsilon = 0.3$ for all our experiments.

6.3 Performance

Table 3 gives an overview of the results regarding runtime and memory requirements on various datasets of the two algorithm variants in comparison to TANE, FDEP and HyFD. The table also contains some metadata about the FDs, namely the size of the table, i.e., the number of records $|r|$ and attributes $|R|$, and the number of valid minimum FDs: #FDs. For each dataset and variant,

we repeated the experiments five times and report the average runtime and memory requirement. Although all algorithms contain random elements, the variance between runs is quite small: Except for a few datasets per algorithm, the relative standard variation ($\frac{\sigma}{\mu}$) of runtime and memory consumption is well below 10%.

Comparing the two variants shows that FDHITS$_{\text{joint}}$ is faster than FDHITS$_{\text{sep}}$ on most datasets. The difference is particularly large when PLI intersections account for a large part of the runtime, which tends to be the case when datasets are quite long, i.e., have many records. Especially, the measurements for the datasets *struct_sheet_range*, *lineitem*, and *fd_reduced* confirm this observation. However, there are also datasets where FDHITS$_{\text{joint}}$ is slower than FDHITS$_{\text{sep}}$. Two datasets in particular catch the eye here: *census* and *musicbrainz_denormalized*. FDHITS$_{\text{sep}}$ is faster for *census*, because this dataset generates an exceptionally large number of hyperedges; because these hyperedges can no longer be minimized in FDHITS$_{\text{joint}}$, managing the large graph becomes expensive. On *musicbrainz_denormalized*, however, most time is still spent on the PLI intersections. Moreover, with the heuristics we propose, it is possible to select the better variant in almost all cases. There are only two datasets where there would still be potential for optimization through a better choice of variant, namely *chess* and *flight*. But even for these two datasets, the price for the worse choice is in the sub-second range.

The memory consumption is more or less the same for both FDHITS variants on almost all datasets with the exception of those two datasets, where FDHITS$_{\text{joint}}$ performs worse, for which FDHITS$_{\text{joint}}$ also requires significantly more memory. Overall, the memory requirements of both FDHITS variants are very low compared to HYFD (and other FD algorithms), so that all tested datasets could be handled easily on a modern laptop, which is, without the need of a high-performance server.

In the comparison against related work, we focus on HYFD as it is clearly superior on almost all datasets in comparison to its competitors TANE and FDEP. In the median case, FDHITS$_{\text{sep}}$ is 6.6 times and FDHITS$_{\text{joint}}$ is 7.4 times faster than HYFD. With the heuristic, FDHITS has a median speedup of 8.1 over HYFD. There is a single dataset, *amalgam1_denormalized*, for which FDHITS$_{\text{sep}}$ is slower than HYFD, but FDHITS$_{\text{joint}}$ is always faster than HYFD with the lowest speedup on *SG_REFERENCE* with 1.76x. On the datasets *census* and *musicbrainz_denormalized*, HyFD was unable to complete the discovery within the 1-hour time limit; given that both FDHITS variants complete both datasets in a few seconds, the speedups here are at least 798x and 70x for FDHITS$_{\text{sep}}$ and 155x and 44x for FDHITS$_{\text{joint}}$, respectively. Because of these outliers, the mean speedup is significantly larger than the median speedup that we report above. In terms of memory requirements, FDHITS uses on average about one order of magnitude less memory. Especially for datasets with a high number of results, such as *plista* and *flight*, HYFD uses up to 480x more memory. Due to the fact that FDHITS performs a depth-first search, it generally needs to keep only a very small portion of the search space in memory at any time.

Both FDHITS$_{\text{sep}}$ and HYFD can easily be parallelized: FDHITS$_{\text{sep}}$ runs the discovery for each dependent attribute individually and HYFD parallelizes the validation of FD candidates. To compare the parallelization gains of both approaches, Table 4 lists the runtimes of parallel executions of the two algorithms, which we denote as FDHITS$_{\text{parallel}}$ and HYFD$_{\text{parallel}}$, respectively. We ran both algorithms with 16 threads on our 16-core server. The speedup-factors for both algorithms over their single-threaded versions are comparable both in absolute range (1.6x to 8.9x for FDHITS$_{\text{parallel}}$ and 1.0x to 10.1x for HYFD$_{\text{parallel}}$) and w.r.t. the same datasets. The maximum speedup that HYFD$_{\text{parallel}}$ can achieve is limited by the number of attributes and the time that is needed to process each individual dependent attribute. The memory requirements for both algorithms increase in their parallel versions, because they need to hold more datasets in memory (multiple trees and more PLIs).

Table 4. Results for parallel versions of FDHITS$_{sep}$ (FDHITS$_{parallel}$) and HYFD$_{parallel}$

Dataset	$\|R\|$	$\|r\|$	#FDs	HYFD$_{parallel}$ Time [s]	HYFD$_{parallel}$ Mem.	FDHITS$_{parallel}$ Time [s]	FDHITS$_{parallel}$ Mem.	Speedup
Musicbrainz-Denormalized	100	79.6 k	1,678,277	2423.902	40.2 GB	5.790	491 MB	418.6
Ditag-Feature	13	3.96 M	58	608.025	9.71 GB	12.861	1.78 GB	47.3
Struct-Sheet-Range	32	664 k	9150	65.869	44.6 GB	14.427	1.08 GB	4.6
Fd-Reduced-30	30	250 k	89,571	28.495	34.3 GB	21.635	299 MB	1.3
Pdbx-Poly-Seq-Scheme	13	17.3 M	68	214.921	24.3 GB	44.848	7.81 GB	4.8
Ncvoter	19	8.06 M	822	1516.107	51.7 GB	51.960	7.67 GB	29.2
Lineitem	16	6 M	3984	512.368	41.1 GB	98.327	6.57 GB	5.2
ncvoter_allc	94	7.50 M	1,197,767,282	-	-	11,971.066	204.45 GB	-
pdb-atom-site	31	219 M	9052	-	-	2214.245	253.71 GB	-

To test the limits of FDHITS, we ran the algorithm again on a more powerful server with 64 cores and 512 GB of RAM. On this machine, FDHITS$_{parallel}$ was able to find all 1,197,767,282 valid FDs of the *ncvoter_allc* dataset, for which the full set of FDs was previously unknown, in 200 minutes. Another even bigger dataset with more than 200 million rows is *pdb-atom-site*, which takes up more than 40 GB on disk as a CSV file. For this file, all 9052 FDs were enumerated in about 37 minutes. Thus, and as the scalability experiments below show, the number of rows is usually not the limiting factor, especially because the number of valid FDs is not overly dependent on the number of rows. On the other hand, even supposedly small datasets, but with many columns, can quickly reach the limits of the algorithm. For example, the *uniprot* dataset with only 539,166 rows, but 223 columns can still not be fully processed even on our large server. The reason for this effect is the large number of valid FDs, which is typical for a dataset with this many columns. The algorithm runs out of memory after a few minutes, but at this point it has already enumerated more than a billion FDs. The even smaller (31 MB on disk) *isolet* dataset, which has only 1000 rows but 618 columns, exhibits a similar behavior. In summary, datasets that generate very large results, which is commonly seen for datasets with more than 200 columns, remain difficult for FD discovery. However, it also raises the question whether a complete set of results for these datasets with billions of results is really useful. In the following, for a deeper analysis of the limits, we take a closer look at the scaling behavior in the two table dimensions.

6.4 Scaling behavior

Figure 4 shows the record scaling behavior on different datasets. Both variants of FDHITS and HYFD show linear growth with the number of records. In theory, however, both variants of FDHITS have at least a quadratic runtime behavior in the worst-case, because there can be datasets that produce a quadratic number of (minimal) difference sets. The theoretical bounds for FDHITS$_{sep}$ are even higher (cubic in the number of records), because it employs minimality pruning which iterates over all previously found difference sets and takes linear time. The size of the result sets vary only slightly over the number of records, which is why the enumeration of the results takes similar amounts of time for different subsets of records of the same dataset. For longer datasets, however, both reading the input files and validating the individual results take more time. Because *lineitem* and *ncvoter* both have only few results, the difference between FDHITS$_{sep}$ and FDHITS$_{joint}$ is marginal (in the case of *ncvoter*, the two lines even overlap almost completely). However, compared to HyFD, both show a much slower growth with an increasing number of records. On the third dataset, which is *fdreduced*, the impact of the pruned candidate validations is particularly evident.

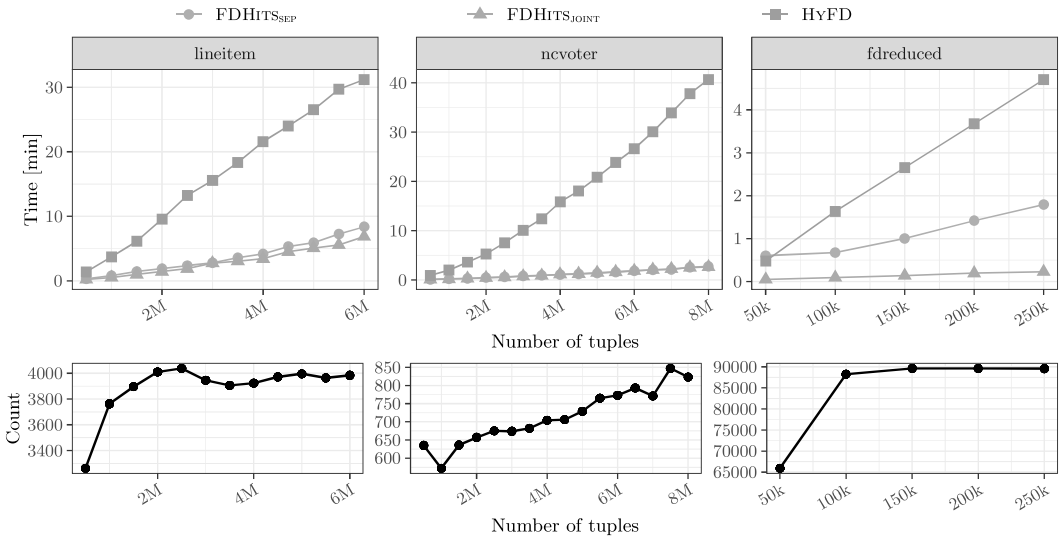


Fig. 4. Record scaling experiment with runtime and result size on three datasets.

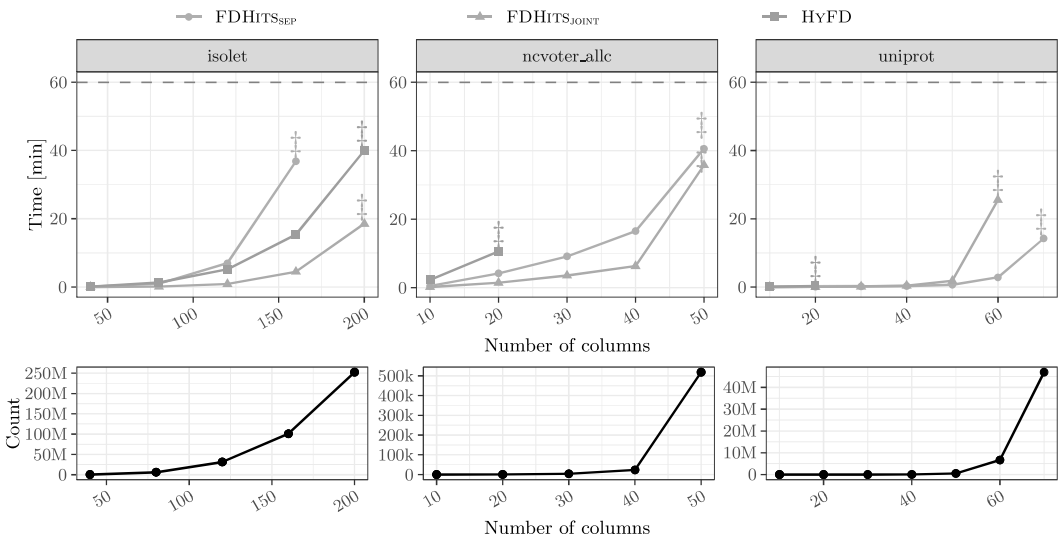


Fig. 5. Attribute scaling experiment with the runtime and result size on three datasets with a time limit of 60 minutes. The measurements annotated with ‡ are the last successful runs before the time limit occurred.

While the runtime of FDHITS$_{sep}$ already grows substantially slower than that of HyFD, the impact of the additional records is marginal for FDHITS$_{joint}$.

Figure 5 (top) shows the attribute scaling behavior of FDHITS and HyFD on different datasets. Overall, all algorithms can exhibit an exponential scaling behavior with the number of attributes (depending on the dataset). This is not surprising, because the number of results can also grow exponentially with the number of attributes (shown in the lower charts of Figure 5). While Table 3 shows that there are datasets for which the result size is relatively small in comparison to the

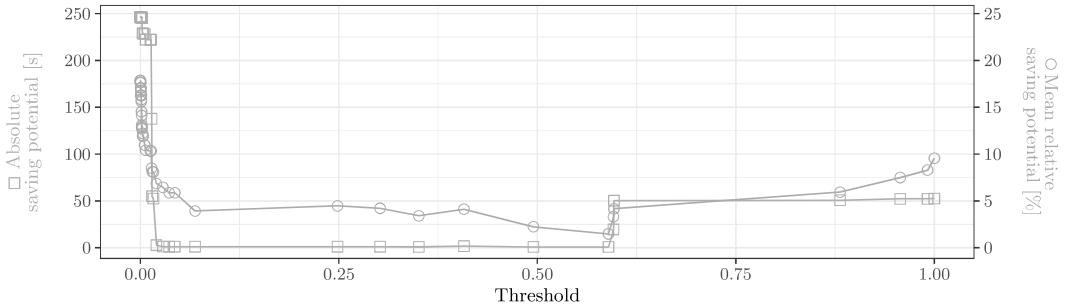


Fig. 6. Left saving potential after applying our heuristic with different thresholds.

number of columns, it remains an open question whether a hitting set-enumeration algorithm exists with a runtime that is polynomial in the number of hitting sets [5, 16]. There is no known runtime bound for MMCS that is sub-exponential [5], and FDHITS inherits this property because it uses MMCS for hitting set enumeration.

Because the behavior of the algorithms differs on the three datasets, we discuss them individually. The *isolet* dataset has some uncommon characteristics: It consists mainly of numeric attributes, most of which have four decimal places. Thus, small combinations of attributes are often sufficient to determine most of the other attributes. In fact, for most attributes, more than half of all combinations of three other attributes determine them (minimally). This, in turn, results in a large overlap between the determinant attributes, which means that FDHITS$_{\text{sep}}$ must perform many of the PLI intersections multiple times. For this reason, FDHITS$_{\text{sep}}$ shows the worst performance on this dataset. For *ncvoter_allc* and *uniprot*, both variants of FDHITS scale much further than HYFD before they hit the time limit of 1 hour. While on *ncvoter_allc* the joint enumeration pays off and FDHITS$_{\text{joint}}$ is constantly faster, this is not the case for *uniprot*. However, in the last successful run of FDHITS$_{\text{joint}}$ on *uniprot* with 60 attributes, more than 54% of the record comparisons generated new difference sets. Therefore, our heuristic switches to FDHITS$_{\text{sep}}$ at this point.

6.5 Detailed analysis

Figure 6 shows how well our heuristic chooses between the two variants FDHITS$_{\text{sep}}$ and FDHITS$_{\text{joint}}$. On the Y-axis, the plot shows how much savings potential remains after applying the heuristic through a better choice of strategy. We distinguish once between the absolute savings and the relative savings per data set (in %) to give weight to both large and small data sets. It can be seen that there is a relatively large area where the penalty for making a wrong decision is relatively small. In the entire range between about 0.05 and 0.6, the heuristic comes very close to the absolute minimum. We propose a default threshold of 0.5, because it performs very well in both relative and absolute performance.

To further investigate the effects of large input graphs, we disabled minimization in FDHITS$_{\text{sep}}$. This allows us to compare how this algorithm behaves on datasets with larger input graphs but otherwise the same characteristics. We observed the largest changes in graph size on the *Census* and *Musicbrainz-Denormalized* datasets, where the input graphs grew from about 2000 and 4500 edges, respectively, to over 300,000 edges. As a result, the algorithm takes about twice as long for both datasets and thus only finishes after 9.5 and 161 seconds, respectively. These are also two datasets for which FDHITS$_{\text{joint}}$, which has no minimization, takes particularly long in comparison, once again showing that the criterion is useful.

We also examined how much additional work is generated by the validation of invalid FDs. With the FDHITS$_{\text{joint}}$ approach, the number of PLIs that need to be calculated can be even lower than the number of valid FDs (because one PLI can be used to validate the same determinant attributes for multiple dependent attributes). In fact, we observe this behavior on numerous datasets and the minimum is at even only 3% of the result size. If the number of validations exceeds the number of FDs, then this happens basically only on data sets with small result sets (<10), for which a few additional FDs are checked. In FDHITS$_{\text{sep}}$, the number of validations cannot be less than the number of valid FDs, but except for very small result sets, there are at most as many invalid FDs as there are valid FDs that are checked.

To better understand under which circumstances FDHITS$_{\text{joint}}$ can save particularly much time, we inspected the tree sizes, the number PLI operations, and other dimensions to see how well they correlate with the time savings. For many of the datasets, FDHITS$_{\text{joint}}$ generated a smaller tree than the sum of sizes of the individual trees of FDHITS$_{\text{sep}}$ in the tree search. For example, for *Fd-Reduced-30*, the individual trees have a total size of on average 97,743 nodes, while the FDHITS$_{\text{joint}}$ tree contains only 8170 nodes. A similar decrease is observable for *T-Bioc-Gath*, with 1088 nodes versus 163 nodes. This decrease is mostly because the tree can keep the dependent attributes together for large parts of the tree. In fact, for both datasets, we observe that many nodes contain more than 20 dependent attributes. This also saves on PLI intersections, because the same operations can be used to validate many FDs.

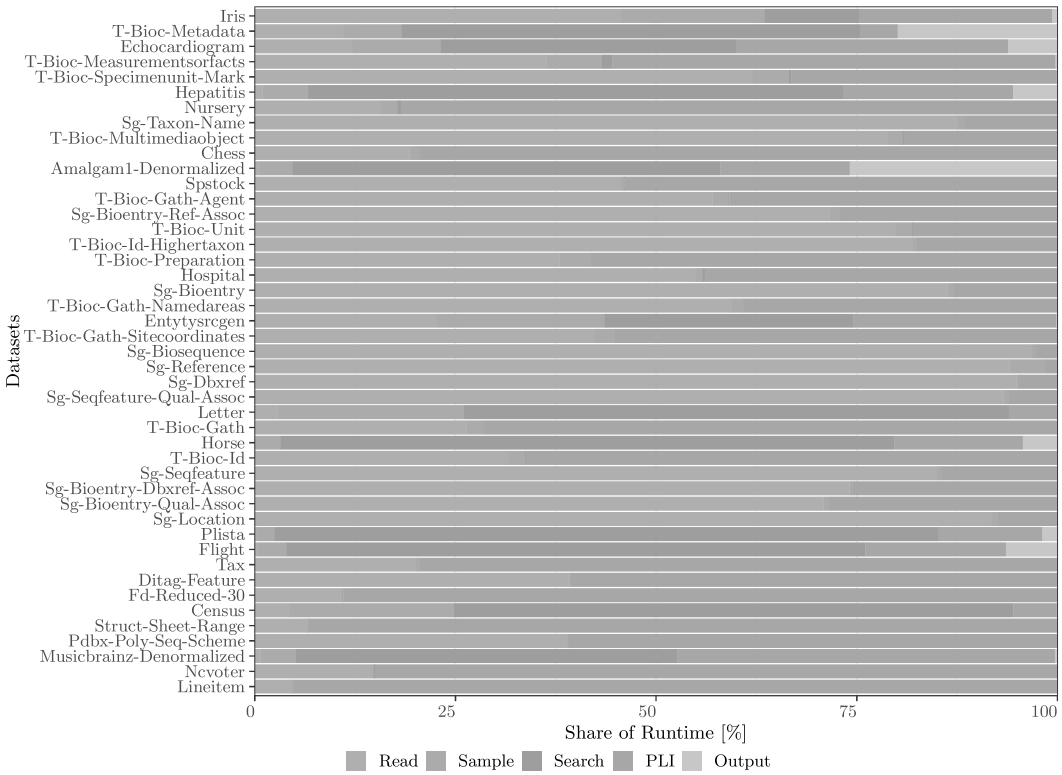


Fig. 7. Proportional runtime for the components of FDHITS$_{\text{joint}}$.

But even though we already compute only the required PLIs, PLI intersections still take up a large portion of the runtime for many datasets, as can be seen in Figure 7. In this figure, we measure the relative runtimes of reading the dataset and constructing the index structures (Read), initial and validation sampling (Sample), tree search (Search), PLI intersections (PLI), and outputting the results (Output). Especially for long datasets (toward the bottom of the figure), these costs dominate the overall runtime. At the same time, it can be seen that for some datasets the potential for optimization is already quite exhausted. In cases where reading and output take up nearly the entire time, there is nothing left to save. We therefore believe that further gains can only be made by improving the validation (through better PLI intersection or entirely new methods).

Moreover, it is critical to filter out clusters in the PLIs that cannot contain violations of the current FD candidate(s), as the following experiment shows: We disabled this filtering and compared the runtimes for both variants of the algorithm. The impact varies depending on the dataset, but we observed differences in one order of magnitude on two datasets. The *Census* dataset is processed without filtering in about 65s by FDHITS$_{\text{sep}}$ and 72s by FDHITS$_{\text{joint}}$, which is a slowdown of a factor of 15 for FDHITS$_{\text{sep}}$ and a factor of 3 for FDHITS$_{\text{joint}}$. However, the difference is even more extreme with *Musicbrainz-Denormalized*. Here, the runtime for FDHITS$_{\text{sep}}$ grows from 52s to over 38 minutes, which corresponds to a factor of 44. The difference is not quite as extreme for the joint variant, but that runtime also increases from 82s to 30 minutes, which corresponds to a factor of 22. On our other evaluation datasets, the difference is not as severe, but a slowdown of about a factor 3-5x is still observable.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented FDHITS, a new FD discovery algorithm based on hitting set enumeration. FDHITS automatically chooses between two discovery variants, which are FDHITS$_{\text{sep}}$ and FDHITS$_{\text{joint}}$. Both variants outperform related work on most datasets by many factors regarding runtime and required memory.

For future work, we investigate how FDHITS can be extended to the discovery of partial FDs, because real-world datasets often contain errors that invalidate genuine FDs. We are confident that FDHits can be adapted for this purpose with the following main modifications: (i) The sampling would need to track the number of violations, e.g., by introducing an edge weight; tracking evidence record pairs could make sure to not count the same record pair twice. (ii) The validation would need to check for n violations instead of only one violation. (iii) The tree search would need to adapt criticality checks and branching. While the first two modifications should be easy, but potentially costly, the implementation of the third adaptation is less obvious.

We also suggest applying a similar approach to other dependencies, such as denial constraints, whose validation is much costlier than the validation of UCCs or FDs; we might be able to reuse previous validation results here.

REFERENCES

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

[2] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. 2014. DFD: Efficient Functional Dependency Discovery. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 949–958.

[3] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 487–499.

[4] Claude Berge. 1989. *Hypergraphs - Combinatorics of Finite Sets*. North-Holland Mathematical Library, Vol. 45. North-Holland Publishing Company, Amsterdam, Netherlands.

[5] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. 2020. Hitting set enumeration with partial information for unique column combination discovery. *PVLDB* 13, 12 (2020),

2270–2283.
[6] Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2016. Approximate discovery of functional dependencies for large datasets. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 1803–1812.
[7] Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. 2022. The Complexity of Dependency Detection and Discovery in Relational Databases. *Theoretical Computer Science* 900 (2022), 79–96.
[8] Alex Bogatu, Norman W Paton, and Alvaro AA Fernandes. 2017. Towards automatic data format transformations: Data wrangling at scale. In *British International Conference on Databases*. 36–48.
[9] Philip Bohannon, Wenfei Fan, and Floris Geerts. 2007. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 746–755.
[10] C. Robert Carlson, Adarsh K. Arora, and Miroslava Milosavljevic Carlson. 1982. The Application of Functional Dependency Theory to Relational Databases. *Computer Journal* 25, 1 (1982), 68–73.
[11] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2020. Mining relaxed functional dependencies from data. *Data Mining and Knowledge Discovery* 34, 2 (2020), 443–477.
[12] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.
[13] Edgar F Codd. 1972. Further normalization of the data base relational model. *Data base systems* 6 (1972), 33–64.
[14] Stavros S Cosmadakis, Paris C Kanellakis, and Nicolas Spyratos. 1986. Partition semantics for relations. *J. Comput. System Sci.* 33, 2 (1986), 203–233.
[15] Scott Davies and Stuart Russell. 1994. NP-completeness of searches for smallest possible feature sets. In *AAAI Symposium on Intelligent Relevance*. AAAI Press Menlo Park, 37–39.
[16] Thomas Eiter, Kazuhisa Makino, and Georg Gottlob. 2008. Computational aspects of monotone dualization: A brief survey. *Discrete Applied Mathematics* 156, 11 (2008), 2035–2049.
[17] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. 2011. Discovering Conditional Functional Dependencies. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 23, 5 (2011), 683–698.
[18] Peter A Flach and Iztok Savnik. 1999. Database dependency discovery: a machine learning approach. *AI Communications* 12, 3 (1999), 139–160.
[19] Andrew Gainer-Dewar and Paola Vera-Licona. 2017. The Minimal Hitting Set Generation Problem: Algorithms and Computation. *SIAM Journal on Discrete Mathematics* 31 (2017), 63–100.
[20] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2020. Cleaning data with llunatic. *VLDB Journal* 29, 4 (2020), 867–892.
[21] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.* 42, 2 (1999), 100–111.
[22] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB Journal* 31, 1 (2022), 1–22.
[23] Oliver Lehmberg and Christian Bizer. 2017. Stitching web tables for improving matching quality. *PVLDB* 10, 11 (2017), 1502–1513.
[24] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient discovery of functional dependencies and Armstrong relations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 350–364.
[25] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *PVLDB* 13, 12 (2020), 1948–1961.
[26] Heikki Mannila and Kari-Jouko Räihä. 1987. Dependency Inference. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 155–158.
[27] Heikki Mannila and Kari-Jouko Räihä. 1994. Algorithms for inferring functional dependencies from relations. *Data & Knowledge Engineering* 12, 1 (1994), 83–99.
[28] Rene J. Miller, Mauricio A. Hernandez, Laura M. Haas, Ling-Ling Yan, Howard Ho, Ronald Fagin, and Lucian Popa. 2001. The Clio Project: Managing Heterogeneity. *SIGMOD Record* 30, 1 (2001), 78–83.
[29] Keisuke Murakami and Takeaki Uno. 2014. Efficient Algorithms for Dualizing Large-Scale Hypergraphs. *Discrete Applied Mathematics* 170 (2014), 83–94.
[30] Noël Novelli and Rosine Cicchetti. 2001. FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the International Conference on Database Theory (ICDT)*. 189–203.
[31] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome. *PVLDB* 8, 12 (2015), 1860–1871.
[32] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *PVLDB* 8, 10 (2015), 1082–1093.
[33] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 821–833.

[34] Thorsten Papenbrock and Felix Naumann. 2017. Data-driven Schema Normalization. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, Vol. 17. 342–353.
[35] Glenn Norman Paulley. 2000. *Exploiting Functional Dependence in Query Optimization.* Technical Report. University of Waterloo.
[36] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. 2019. Distributed Implementations of Dependency Discovery Algorithms. *PVLDB* 12, 11 (2019), 1624–1636.
[37] Roee Shraga and Renée J Miller. 2023. Explaining Dataset Changes for Semantic Data Versioning with Explain-Da-V. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1587–1600.
[38] S. Tu and M. Huang. 2016. Scalable Functional Dependencies Discovery from Big Data. In *International Conference on Multimedia Big Data (BigMM)*. 426–431.
[39] Ziheng Wei, Sven Hartmann, and Sebastian Link. 2021. Algorithms for the discovery of embedded functional dependencies. *VLDB Journal* 30, 6 (2021), 1069–1093.
[40] Ziheng Wei and Sebastian Link. 2019. Discovery and ranking of functional dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1526–1537.
[41] Ziheng Wei and Sebastian Link. 2019. Embedded Functional Dependencies and Data-Completeness Tailored Database Design. *PVLDB* 12, 11 (2019), 1458–1470.
[42] Wanqing Wu and Wenyu Mao. 2022. An Efficient and Scalable Algorithm to Mine Functional Dependencies from Distributed Big Data. *Sensors* 22, 10 (2022), 3856.
[43] Catharine Wyss, Chris Giannella, and Edward Robertson. 2001. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *Proceedings of the International Conference of Data Warehousing and Knowledge Discovery (DaWaK)*. 101–110.
[44] Guanghui Zhu, Qian Wang, Qiwei Tang, Rong Gu, Chunfeng Yuan, and Yihua Huang. 2019. Efficient and Scalable Functional Dependency Discovery on Distributed Data-Parallel Platforms. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 30, 12 (2019), 2663–2676.

Received July 2023; revised October 2023; accepted November 2023