Dynamic Functional Dependency Discovery with Dynamic Hitting Set Enumeration

Renjie Xiao [1,2] Yong'an Yuan [1,2] Zijing Tan [1,2†] Shuai Ma [3] Wei Wang [1,2]

[1]*Fudan University, China* [2]*Shanghai Key Laboratory of Data Science, China* [3]*SKLSDE Lab, Beihang University, China*

***Abstract*—Functional dependencies (FDs) are widely applied in data management tasks. Since FDs on data are usually unknown, FD discovery techniques are studied for automatically finding hidden FDs from data. In this paper, we develop techniques to dynamically discover FDs in response to changes on data. Formally, given the complete set Σ of minimal and valid FDs on a relational instance r, we aim to find the complete set Σ' of minimal and valid FDs on $r \oplus \triangle r$, where $\triangle r$ is a set of tuple insertions and deletions. Different from the batch approaches that compute Σ' on $r \oplus \triangle r$ from scratch, our dynamic method computes Σ' in response to $\triangle r$ by leveraging the known Σ on r, and avoids processing the whole of r for each update from $\triangle r$. We tackle dynamic FD discovery on $r \oplus \triangle r$ by dynamic *hitting set enumeration* on the *difference-set* of $r \oplus \triangle r$. Specifically, (1) leveraging auxiliary structures built on r, we first present an efficient algorithm to update the difference-set of r to that of $r \oplus \triangle r$. (2) We then compute Σ', by recasting dynamic FD discovery as dynamic hitting set enumeration on the difference-set of $r \oplus \triangle r$ and developing novel techniques for dynamic hitting set enumeration. (3) We finally experimentally verify the effectiveness and efficiency of our approaches, using real-life and synthetic data. The results show that our dynamic FD discovery method outperforms the batch counterparts on most tested data, even when $\triangle r$ is up to 30% of r.**

***Index Terms*—Data profiling; Data dependency; Functional dependency**

I. Introduction

Functional dependencies (FDs) [5] are extensively studied, for their usefulness in schema design [26], [35], query optimization [8], [30] and data quality management [9], [12], among others. In practice, FDs on data are usually unknown and too expensive to be designed manually [1], [2], [31]. Hence, discovery techniques for FDs are actively studied; see, *e.g.,* [4], [11], [13], [14], [17], [18], [20], [25], [33], [34], [36], [37]. They aim at automatically finding hidden FDs from data, to enable FDs in data management tasks and alleviate the burden of users. There are two criteria that are usually considered in FD discoveries, namely *validity* and *minimality*.

Example 1: Consider the relational instance r in Table I. $DE \rightarrow A$ is an FD that will be discovered, since it is *valid* and *minimal*. The validity of $DE \rightarrow A$ can be verified by checking whether any tuple pair from r violates it. To check the minimality of $DE \rightarrow A$, it suffices to consider the validity of $D \rightarrow A$ and $E \rightarrow A$. Since neither of them is valid, $DE \rightarrow A$ is minimal. Intuitively, $D \rightarrow A$ ($E \rightarrow A$) is more *general* than $DE \rightarrow A$, in the sense that if $D \rightarrow A$ ($E \rightarrow A$) is valid on any relational instance, then so is $DE \rightarrow A$. □

† Zijing Tan is the corresponding author.

TABLE I
RELATIONAL INSTANCE r

	A	B	C	D	E
t_1	0	0	0	0	0
t_2	1	1	1	0	1
t_3	2	2	2	1	2
t_4	3	0	3	0	3
t_5	4	2	2	2	2

TABLE II
TUPLE INSERTIONS

	A	B	C	D	E
t_6	5	3	3	2	2
t_7	6	4	4	3	1

Most of the FD discovery methods, *e.g.,* [4], [11], [17], [25], [34], [36], aim to discover *all* minimal valid FDs, *i.e.,* a *complete* set Σ of minimal valid FDs. The FD discovery is costly on instances with large numbers of attributes and tuples, since the size of Σ can be exponential in the number of attributes and value comparisons of tuple pairs are required to check the validity of FDs. Worse, data in practice are frequently updated and the changes in data usually incur changes in the discovered FDs. It suffices to consider tuple insertions and deletions, since each value modification can be modeled as a deletion followed by an insertion. As shown below, the changes of FDs in response to updates can be very intricate.

Example 2: To simplify the discussion, we consider tuple insertions and deletions separately in this example.

(1) Suppose the two tuples in Table II are inserted into r. Note that insertions can only introduce new FD violations. $DE \rightarrow A$ is violated by (t_5, t_6) and (t_6, t_5), and is not a valid FD now. We see $BDE \rightarrow A$ and $CDE \rightarrow A$ are valid and minimal FDs; they are valid but not minimal before the tuple insertions since $DE \rightarrow A$ is valid. Hence, $DE \rightarrow A$ should be removed from while $BDE \rightarrow A$ and $CDE \rightarrow A$ should be added into Σ.

(2) Suppose tuples t_1, t_4 are deleted from r, as shown in Table III. Note that deletions can only remove FD violations. Since all violating tuple pairs are removed, $D \rightarrow A$ becomes a minimal valid FD and should be put into Σ. In contrast, $DE \rightarrow A$ should be removed from Σ; it is still valid but not minimal now due to the validity of $D \rightarrow A$.

To sum up, we see the following. (a) Tuple insertions (resp. deletions) can introduce (resp. remove) FD violations, and hence tuple insertions can make a valid FD invalid and

2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00026

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00026

TABLE III
r AFTER TUPLE DELETIONS

	A	B	C	D	E
t_2	1	1	1	0	1
t_3	2	2	2	1	2
t_5	4	2	2	2	2

deletions can make an invalid FD valid. (b) The minimality of FDs should be considered in computing the changes to Σ. $\square$

Formally, in response to a set $\triangle r$ of updates to an instance r, the set Σ of FDs on r will evolve into the new set Σ' on $r \oplus \triangle r$. We use the notation "$\oplus$" since $\triangle r$ is a set of tuple insertions and deletions. As shown in Example 2, the evolutions are intricate since both $\Sigma' \setminus \Sigma$ and $\Sigma \setminus \Sigma'$ can be not empty. The baseline approach to computing Σ' is to perform FD discovery on $r \oplus \triangle r$ from scratch. However, this is usually inefficient, especially when $\triangle r$ is (much) smaller than r. A better approach is to perform a *dynamic* discovery, by computing Σ' in response to $\triangle r$. The efficiency can be further improved by leveraging auxiliary structures built upon the known Σ and r in the pre-processing step.

Contributions & Organizations. In this paper, we study the problem of dynamic FD discovery. We contend this problem is more relevant to data profiling in the real world. We tackle the problem with the *row-based* strategy, by recasting dynamic FD discovery on $r \oplus \triangle r$ as dynamic *hitting set enumeration* of the *difference-set* of $r \oplus \triangle r$ (Section IV). In this way, our method fully leverages the good scalability of the row-based strategy in the number of attributes. Our method also performs well on instances with a large number of tuples, since it avoids processing the whole of r for each update from $\triangle r$.

(1) To improve the efficiency of dynamic computations, we propose auxiliary structures (including the difference-set) built upon r. We present an efficient algorithm to update the structures in response to $\triangle r$, and show the computation avoids processing the whole of r, for each tuple insertion and deletion from $\triangle r$ (Section V).

(2) We present a method to compute Σ', by developing novel techniques to *dynamically* enumerate hitting sets in response to the changes in the *difference-set* from r to $r \oplus \triangle r$. We contend that the technique for dynamic hitting set enumeration is valuable in itself and can be applied to other domains in which hitting set problems are studied, *e.g.,* computational biology and data mining [10], [15] (Section VI).

(3) We conduct an experimental study to verify our approach, using a host of real-life and synthetic datasets. The results show our dynamic FD discovery method outperforms the batch methods that compute from scratch on most tested data, even when $|\triangle r| = 30\%|r|$, and our method is much faster than the only existing method for dynamic FD discovery (Section VII).

II. Related Work

Dependency discovery techniques are extensively studied; see, *e.g.,* [1], [2], [31] for surveys on this topic. In this section, we investigate works close to ours: FD discoveries and dependency discoveries on dynamic data.

FD discoveries. There are many FD discovery algorithms for finding the complete set of minimal and valid FDs, *e.g.,* [4], [11], [17], [22], [25], [34], [36], and some of them are compared in [24]. They can be roughly categorized into three types. (1) The column-based approaches, *e.g.,* [4], [11], enumerate FD candidates from the space of all candidates following the traversal strategies and then validate them. They typically scale well with the number of tuples. (2) The row-based approaches, *e.g.,* [17], [22], [36], are based on the value comparisons of all tuple pairs. In particular, [36] builds the so-called *difference-set* on the relation instance and then recasts FD discovery as hitting set enumeration of the difference-set. These approaches usually scale well with the number of attributes. (3) The hybrid approaches, *e.g.,* [25], [34], combine row-based strategy on sample data with column-based strategy on the full instance, and are experimentally verified to outperform the row-based and column-based methods in practice.

Different from the above-mentioned works, other FD discovery methods find *approximate* FDs holding on dirty data with some exceptions [13], [14], discover *embedded* FDs from data with missing values [33], or heuristically identify only "interesting" or "reliable" FDs, by using, *e.g.,* entropy-based measures, mutual information or probabilistic graphical models; see, *e.g.,* [18]–[20], [27], [37]. We consider discovering all minimal valid FDs in this paper, along the same setting as [4], [11], [17], [25], [34], [36]. Further post-processing steps can be used to select FDs to meet the specific user requirements, which is beyond the scope of this paper.

The above-mentioned works perform *batch* FD discoveries; they aim to discover the set Σ of FDs on a given instance r. In this paper we study the problem of dynamic FD discovery. We tackle the problem by proposing auxiliary structures on r, presenting methods to update structures in response to $\triangle r$, and computing Σ' on $r \oplus \triangle r$ via dynamic hitting set enumeration on the difference-set of $r \oplus \triangle r$.

Dependency discovery on dynamic data. In contrast to batch discoveries that find dependencies from scratch, dynamic or incremental discoveries are recently studied to handle updates on data. They are investigated for unique column combinations (UCCs) [3], *a.k.a.* candidate keys, FDs [7], [28], pointwise order dependencies [32] and inclusion dependencies [29]. These works differ in the constraint types. UCCs are a special case of FDs. [32] focuses on the efficient processing of the inequality operators in pointwise order dependencies, *i.e.,* "$<, >$", while FDs only concern the equality operator, *i.e.,* "=". Moreover, [3], [28], [29] and this paper consider dynamic settings with both insertions and deletions, while [7], [32] study incremental discoveries only with tuple insertions.

To our best knowledge, [28] is the only work for dynamic FD discovery. This work differs from [28] in the following. We adopt the row-based strategy, by recasting FD discovery as hitting set enumeration, while [28] adopts a column-based strategy. Hence, our method scales much better with the

number of attributes, by leveraging the advantage of the row-based strategy. Moreover, by putting the row-based strategy together with our carefully designed dynamic algorithms, we avoid processing the whole of r for each tuple insertion and deletion, while [28] enjoys this feature only for insertion but not deletion. Our method is verified to be much faster than [28] in the experimental evaluations (Section VII), which well demonstrates the benefit of our approach.

III. Preliminaries

In this section, we review the basic notions of FDs, and the approach to batch FD discovery via hitting set enumeration.

A. Basic notations

$R(A,\ldots)$ denotes a relation schema, r denotes a specific instance of R, t, s denote tuples from r, and $t[A]$ denotes the value of attribute A in a tuple t. Each tuple t is associated with a distinct identifier (*id*). We denote by $|r|$ the number of tuples in r and $|R|$ the number of attributes in R, respectively.

Functional dependency (FD). We consider FDs of the form $X \rightarrow A$, where X (resp. A) is a set of attributes (resp. a single attribute) from R ($A \notin X$). Obviously, FDs with a single right-hand-side (RHS) attribute do not lose generality. An FD γ = $X \rightarrow A$ is valid on r, *i.e.,* γ holds on r, iff for any $t, s \in r$, $t[A] = s[A]$ if $t[X] = s[X]$.

Minimal FD. An FD $X \rightarrow A$ is minimal, iff $X' \rightarrow A$ is invalid for any proper subset X' of X.

Former works, *e.g.,* [4], [11], [17], [25], [34], [36], consider batch FD discoveries, concerning the validity and minimality.

Batch FD discovery. Given a relation instance r of schema R, batch FD discovery is to find the complete set Σ of minimal and valid FDs on r.

B. Batch FD discovery via hitting set enumeration

There is a long history of studies on the relationship between batch FD discovery and hitting set enumeration, *a.k.a.* hypergraph transversal; see, *e.g.,* [17], [21], [22], [36]. In this subsection, we review the notations and techniques of [36].

Difference-set. The *difference-set* of a tuple pair (t, s) is $D(t, s) = \{A \in R \mid t[A] \neq s[A]\}$, *i.e.,* the set of attributes in which t, s have different values. Obviously, $D(t, s) = D(s, t)$. The *difference-set* of r is $D_r = \{D(t, s) \mid t, s \in r, D(t, s) \neq \emptyset\}$.

Example 3: Consider the instance r in Table I. (1) $D(t_1, t_2) = \{ABCE\}$. To simplify the presentation, in the rest of the paper we write $ABCE$ instead of $\{ABCE\}$. Similarly, $D(t_1, t_4) = ACE$ and $D(t_3, t_5) = AD$. (2) Different tuple pairs can have the same difference-set. For example, $D(t_1, t_3) = D(t_1, t_5) = D(t_2, t_3) = D(t_2, t_5) = D(t_3, t_4) = D(t_4, t_5) = ABCDE$. (3) Finally, $D_r = \{ABCE, ACE, AD, ABCDE\}$. □

It can be seen that, for $\gamma = X \rightarrow A$ and $A \in D(t, s)$, t, s do not violate γ iff X contains at least one attribute (not A) from $D(t, s)$. Therefore, γ is valid on r iff X contains at least one attribute (not A) from $D(t, s)$ for all $t, s \in r$.

Hitting set of difference-set. For a given attribute $A \in R$, the difference set of r modulo A is $D_r^A = \{\ U \setminus \{A\} \mid U \in D_r, A \in U\}$. $X \rightarrow A$ ($A \notin X$) is valid iff X intersects with every element of D_r^A. Such X is referred to as a *hitting set* (HS) of D_r^A [15]. In the formal notations, D_r^A is a subset family $\{U_1,\ldots,U_m\}$ defined on $R \setminus \{A\}$. A HS of D_r^A is a subset X of $R \setminus \{A\}$ such that $X \cap U \neq \emptyset$ for every $U \in D_r^A$. X is a *minimal* HS if no proper subset of X is a HS. $X \rightarrow A$ ($A \notin X$) is a minimal valid FD iff X is a *minimal* HS of D_r^A.

Example 4: (Example 3 continued.) (1) $D_r^A = \{BCE, CE, D, BCDE\}$. $D_r^E = \{ABC, AC, ABCD\}$. (2) BDE is a HS of D_r^A, *i.e.,* BDE intersects with every element in D_r^A. $BDE \rightarrow A$ is hence a valid FD, but is not minimal since BDE is not a minimal HS. We see DE is a minimal HS, and hence, $DE \rightarrow A$ is a minimal and valid FD. □

HS enumeration for FD discovery. [36] first builds the *difference-set* of r. For each A from R, it then discovers minimal and valid FDs of the form $X \rightarrow A$ by enumerating minimal hitting sets (HSs) X of D_r^A. By using all attributes from R on the RHS, it discovers all minimal valid FDs. There can be an exponential number of minimal valid FDs, so is the number of minimal HSs. The algorithms for HS enumeration are well studied and employed in not only data dependency discoveries [6], [16], but also in, *e.g.,* data mining and AI. Please refer to [10], [15] for surveys.

Example 5: (Example 4 continued.) We discover FDs of the form $X \rightarrow A$ by enumerating minimal HSs X of $D_r^A = \{BCE, CE, D, BCDE\}$. They are $CD \rightarrow A$ and $DE \rightarrow A$. □

IV. Formulation of Dynamic FD Discovery

In this section, we first formalize the problem of dynamic FD discovery. We then outline our approach to dynamic FD discovery by dynamic HS enumeration.

Dynamic FD discovery. Given the complete set Σ of minimal and valid FDs on r, and a set $\triangle r$ of updates to r, dynamic FD discovery is to find the complete set Σ' of minimal and valid FDs on $r \oplus \triangle r$.

Specifically, let $\triangle r = \triangle r^+ \cup \triangle r^-$, where $\triangle r^+$ (resp. $\triangle r^-$) is a set of tuple insertions (resp. deletions). $r \oplus \triangle r = r \cup \triangle r^+ \setminus \triangle r^-$.

Overview of our approach. We will develop novel and efficient techniques for dynamic FD discovery in the following sections. Our aim is to present a solution that scales well with both $|R|$ and $|r|$. This is achieved by combining the advantage of the row-based strategy in its scalability with $|R|$ and the advantage of our dynamic computation that avoids visiting the whole of r for each update. Roughly speaking, we adapt the row-based approach to the *dynamic* setting as follows.

(1) We update the difference-set D_r to $D_{r \oplus \triangle r}$ of $r \oplus \triangle r$, in response to $\triangle r$ (Section V).

(2) We compute the complete set Σ' of minimal valid FDs on $r \oplus \triangle r$, by performing dynamic HS enumeration on $D_{r \oplus \triangle r}$, in response to the changes from D_r to $D_{r \oplus \triangle r}$ (Section VI).

At both steps, we perform dynamic computations in response to the changes, and leverage auxiliary structures built at the pre-processing step to further improve the efficiency.

V. Updating Difference-set

In this section, we present an efficient algorithm to update the difference-set D_r of r to $D_{r\oplus\triangle r}$ of $r \oplus \triangle r$.

A. Auxiliary structures

To facilitate our approach, we propose to build some simple yet effective auxiliary structures on the known instance r. These structures are built in the pre-processing step.

Position list indexes. The first auxiliary structure, known as position list indexes (Plis), is commonly used in FD discoveries [14], [25], [28]. Each Pli is built for an attribute from R, and we denote the Pli on attribute A by π_A. π_A is a set of *clusters*; each cluster is also a set and tuples with the same value in A are in the same cluster from π_A. We denote by $cl_A(t)$ the cluster from π_A that contains tuple t. With hashing, it takes $O(|r|)$ to build π_A, and $O(1)$ to find $cl_A(t)$.

We build π_A for all $A \in R$. The instance r is only used to create Plis, and no longer needed after that. The space complexity of Plis on r is $O(|r||R|)$. To reduce the memory footprint, only tuple *ids* are stored in Plis.

Example 6: In Table I, π_A = {{t_1},{t_2},{t_3},{t_4},{t_5}}, and π_B = {{t_1,t_4},{t_2},{t_3,t_5}}. We also have $cl_A(t_1)$ = {t_1}, and $cl_B(t_1)$ = {t_1,t_4}. □

Difference-set with weights. The other auxiliary structure we use is the difference-set D_r of r. The original version from [36] is not sufficient to support updates, and a minor modification is required for this purpose. Recall that each element of D_r is a difference-set of tuple pair from r. We additionally associate each element U in D_r with a weight $wt(U)$ denoting the number of tuple pairs (t, s) such that $D(t, s) = U$, *i.e.,* the number of tuple pairs having U as the difference-set. D_r^A is extended similarly. In the sequel, we still use D_r and D_r^A to denote difference-sets with weights. Let $wt(D_r) = \sum_{U\in D_r} wt(U)$. It is easy to see that $wt(D_r) = |r|(|r| - 1)$, *i.e.,* the total number of tuple pairs.

Example 7: (Example 3 and Example 4 continued.) We have D_r = {$ABCE$: 4, ACE: 2, AD: 2, $ABCDE$: 12}, D_r^A = {BCE: 4, CE: 2, D: 2, $BCDE$: 12}, and D_r^E = {ABC: 4, AC: 2, $ABCD$: 12}. □

D_r is obtained by value comparisons of all tuple pairs, and some optimizations are proposed in [36]. We build D_r as a hash table, where the difference-sets of tuple pair are keys and weights are values. The number of elements in D_r has an upper bound of $2^{|R|}$, but is usually much smaller in practice, as experimentally verified in Section VII.

B. Updating the difference-set of r with $\triangle r$

We first update Plis with $\triangle r$, which then facilitates the update of the difference-set D_r. For $\triangle r = \triangle r^+ \cup \triangle r^-$, we process $\triangle r^+$ before $\triangle r^-$, in case $\triangle r^+ \cap \triangle r^- \neq \emptyset$.

Updating Plis. (1) For each tuple $t \in \triangle r^+$ and each attribute $A \in R$, we find the cluster that t should belong to, *i.e.,* the cluster containing the tuples that have the same value as t in A. We still use $cl_A(t)$ to denote the cluster. We add t into $cl_A(t)$ if it exists, otherwise we create $cl_A(t)$ with t and add it into π_A. (2) For each tuple $t \in \triangle r^-$, we find $cl_A(t)$ for each attribute $A \in R$. We remove t from the cluster $cl_A(t)$, and the cluster from π_A if the cluster is empty after the removal of t.

Algorithm 1: Update

```
Input: the difference-set D_r of r, △r = △r+ ∪ △r−, Plis
Output: the difference-set D_{r⊕△r} of r ⊕ △r
1  foreach tuple t in △r+ and then in △r− do
2      map ← an empty hash table;
3      foreach attribute A in R do
4          update π_A with t;
5          foreach tuple s in cl_A(t) do
6              if s is not in map then
7                  insert a key-value pair (s, R \ {A}) into
                     map, where s is the key;
8              else
9                  update the pair (s, X) to (s, X \ {A});
10     update D_r with map;
```

Updating the difference-set D_r. Applying $\triangle r$ to r incurs changes to the difference-set D_r of r. Intuitively, a tuple $t \in \triangle r^+$ leads to new difference-sets that should be combined into D_r, while a tuple $t \in \triangle r^-$ leads to obsolete difference-sets of tuple pair that should be separated from D_r. The observation critical to the efficiency is that, for $t \in \triangle r$, we need to compute $D(t, s)$ only when s has the same values as t in some attributes, and all such s can be efficiently obtained by leveraging Plis. To complement the strategy, we need to adjust the weight $wt(R)$ associated with R in D_r. Note that $D(t, s) = R$ if t, s have different values in all attributes. More details are provided in Algorithm Update.

Algorithm. Update (Algorithm 1) takes as inputs the difference set D_r of r, $\triangle r = \triangle r^+ \cup \triangle r^-$ and Plis on r. It computes the difference set $D_{r\oplus\triangle r}$ of $r \oplus \triangle r$, by updating Plis and D_r.

Update enumerates tuples first from $\triangle r^+$ and then from $\triangle r^-$. For each enumerated tuple t, Update deals with attribute $A \in R$ one by one. It first updates π_A with t (line 4), as stated before. By checking the *id* of t, it can directly stop the processing of t if $t \in \triangle r^+$ already belongs to $cl_A(t)$ or $t \in \triangle r^-$ does not belong to $cl_A(t)$.

Update then computes the new or obsolete difference-sets concerning t, *i.e.,* the changes incurred by t (lines 5-9). Specifically, it computes $D(t, s)$ for s that belongs to some $cl_A(t)$. A tuple s can have the same values as t in several attributes, and a hash table map is used to gradually compute $D(t, s)$ for such s. Each element in map is a key-value pair. The tuple *id* of s is used as the key, which guarantees that s is inserted into map only once (line 6). The difference-set $D(t, s)$ computed so far is used as the value. If s is found in $cl_A(t)$, then A is removed from $D(t, s)$ (lines 7 and 9). After all attributes from R are processed, the final $D(t, s)$ for s can be obtained in map. Note that for $t, s \in \triangle r$, $D(t, s)$ is also computed in this way if at least one of t, s is from $\triangle r^+$.

All changes incurred by t are finally applied to D_r (line 10). We first enumerate each element (s, X) in map that denotes

the new or obsolete difference-set $X = D(t,s) = D(s,t)$, and do the following. (1) If $t \in \triangle r^+$, then we increase $wt(X)$ by 2 if X exists in D_r, otherwise we add X into D_r and set $wt(X) = 2$. (2) If $t \in \triangle r^-$, then in D_r we decrease $wt(X)$ by 2 and remove X from D_r if $wt(X)$ becomes 0.

We then additionally adjust $wt(R)$ in D_r. (1) If $t \in \triangle r^+$, then we increase $wt(R)$ by $2(k-|map|)$, where $|map|$ is the number of elements in map, and k is the number of tuples in r *before* inserting t into r. This is because the addition of t always incurs $2k$ new difference-sets of tuple pair. Besides those stored in map, all the others must equal R, arising from tuples that have different values as t in all attributes. (2) If $t \in \triangle r^-$, then we decrease $wt(R)$ by $2(k-|map|)$, where k is the number of tuples in r *after* removing t.

Example 8: (Example 7 continued.) Recall that $D_r = \{ABCE\text{: 4}, ACE\text{: 2}, AD\text{: 2}, ABCDE\text{: 12}\}$. We show how Plis and D_r are updated for deleting t_1. (1) For attribute A, we have $cl_A(t_1) = \emptyset$ after removing t_1 and hence remove $cl_A(t_1)$ from π_A. (2) For attribute B, $cl_B(t_1) = \{t_4\}$ after removing t_1. We insert $(t_4, ACDE)$ into the hash table. (3) Attribute C is treated in the same way as attribute A. (4) For attribute D, $cl_D(t_1) = \{t_2, t_4\}$ after removing t_1. We insert $(t_2, ABCE)$ and update $(t_4, ACDE)$ to (t_4, ACE) in the hash table. (5) Attribute E in treated similarly as attribute A.

We have $D(t_1,t_2) = ABCE$ and $D(t_1,t_4) = ACE$ in the hash table. After removing them and adjusting $wt(R = ABCDE)$, $D_r = \{ABCE\text{: 2}, AD\text{: 2}, ABCDE\text{: 8}\}$. □

Complexity. Update deals with tuples from $\triangle r$ one by one. For each $t \in \triangle r$, (1) it updates π_A for each $A \in R$; (2) the cost of computing the changes to D_r incurred by t is linear in the number of tuples from $cl_A(t)$ for $A \in R$, *i.e.,* $O(\sum_{A\in R}|cl_A(t)|)$; and (3) the cost of updating D_r is linear in the number of elements in the hash table (the number of distinct tuples from $cl_A(t)$ for $A \in R$), *i.e.,* $O(|\cup_{A\in R} cl_A(t)|)$. To sum up, Update avoids visiting the whole of r for each t.

Remark. Update handles $\triangle r^+$ before $\triangle r^-$, to guarantee the correctness when a tuple is inserted first and then deleted. An alternative method is to first remove tuple insertions in $\triangle r^+$ and deletions in $\triangle r^-$ that cancel each other. After that, it is more efficient to treat $\triangle r^-$ first, since the computation of any difference-set of a deleted tuple and a new tuple is avoided.

VI. Dynamic HS Enumeration for Dynamic FD Discovery

In this section, we compute the set Σ' of minimal valid FDs on $r \oplus \triangle r$, based on the changes from D_r to $D_{r\oplus\triangle r}$. We tackle the problem by solving a more general problem, namely, *dynamic hitting set enumeration*.

A. Problem formulation

To simplify presentation, in this section we use the notations of hitting set (HS) enumeration [6], [23]. A *hypergraph* $\mathcal{F}$ is a subset family $\{F_1,\ldots,F_m\}$ defined on a vertex set V; each element F of $\mathcal{F}$ is called a *hyperedge*. We denote by $|\mathcal{F}|$ the number of hyperedges in $\mathcal{F}$. A subset H of V is called a *vertex* subset. H *covers* hyperedge F if $H \cap F \neq \emptyset$. H is a HS of $\mathcal{F}$ if H covers all hyperedges in $\mathcal{F}$, and is *minimal* if no proper subset of H is a HS. The HS enumeration of $\mathcal{F}$ is the set of all minimal HSs of $\mathcal{F}$, denoted by $Hs(\mathcal{F})$.

TABLE IV
Notations of FD discovery and HS enumeration

FD Discovery	HS Enumeration
attribute set R	vertex set V
the complement set D_r^A	hypergraph $\mathcal{F}$
difference-set of tuple pair $D(t,s)$	hyperedge F
attribute A	vertex v
attribute set X	vertex subset H
minimal valid FDs $X \to A \in \Sigma$	HS enumeration $Hs(\mathcal{F})$

The problem of HS enumeration is well studied in the literature [10], [15]. However, to our best knowledge, the dynamic version of HS enumeration has never been investigated before.

Dynamic HS enumeration. Given $Hs(\mathcal{F})$ of $\mathcal{F}$ and a set $\triangle\mathcal{F}$ of updates to $\mathcal{F}$ (hyperedges that are added into or removed from $\mathcal{F}$), it is to compute the HS enumeration $Hs(\mathcal{F} \oplus \triangle\mathcal{F})$.

From dynamic HS enumeration to dynamic FD discovery. We discover FDs of the form $X \to A$ (all FDs are discovered by using every attribute from R on the RHS), by recasting dynamic FD discovery as dynamic HS enumeration:

Input: (1) $\mathcal{F} = D_r^A$; (2) $Hs(\mathcal{F}) = \{X \mid X \to A \in \Sigma\}$; and (3) $\triangle\mathcal{F}$ is the changes from D_r^A to $D_{r\oplus\triangle r}^A$, *i.e.,* the new (resp. obsolete) difference-sets of tuple pair that (a) are added into (resp. removed from) D_r and (b) include attribute A. $\triangle\mathcal{F}$ is obtained in Algorithm Update (Section V), and the weights of difference-sets are neglected in HS enumeration.

Output: $X \in Hs(\mathcal{F} \oplus \triangle\mathcal{F})$ corresponds to $X \to A \in \Sigma'$, where Σ' is the complete set of minimal valid FDs on $r \oplus \triangle r$.

We summarize the relationship between notations of FD discovery and HS enumeration in Table IV.

B. Techniques of HS enumeration and auxiliary structures

We review some techniques of HS enumeration, and propose a simple auxiliary structure to boost dynamic HS enumeration.

HS enumeration methods aim to generate all minimal combinations of vertices that can cover all hyperedges. Various enumeration strategies and pruning rules are developed in the literature [10], [15]. It is known that MMCS [23] is currently the fastest algorithm, as verified in [10].

Critical hyperedges. The idea of *critical hyperedge* is introduced in MMCS and critical to the efficiency. For a vertex set $H \subseteq V$ and a vertex $v \in H$, a hyperedge $F \in \mathcal{F}$ is a *critical* hyperedge of v if $H \cap F = \{v\}$, *i.e.,* v is the only common vertex from H and F. We denote by $crit_v^H$ the set of all critical hyperedges of v, *i.e.,* $crit_v^H = \{F | F \in \mathcal{F}, H \cap F = \{v\}\}$, and denote by $crit^H$ all the critical hyperedges concerning H, *i.e.,* $crit_v^H$ for all $v \in H$. It is proved in [23] that H is a minimal HS of $\mathcal{F}$ iff (a) H covers all hyperedges in $\mathcal{F}$, and (b) $crit_v^H \neq \emptyset$ for any $v \in H$, *i.e.,* each v is necessary for H to be a HS. Leveraging critical hyperedges, the minimality of a HS is checked locally without considering the other HSs.

Example 9: Let $\mathcal{F} = \{ABC, AD, BD\}$. $H = AB$ is a minimal HS, since (a) it covers all hyperedges from $\mathcal{F}$; and (b) $crit_A^H = \{AD\}$ and $crit_B^H = \{BD\}$. Each hyperedge can serve as a critical hyperedge for at most one vertex by the definition. When the intersection of a hyperedge and a HS contains several vertices, the hyperedge is not a critical hyperedge. For example, ABC is *not* a critical hyperedge of any $v \in H$. □

Minimal hyperedges. Not all of the hyperedges from $\mathcal{F}$ are equally important in HS enumeration. A hyperedge F_i is a *minimal* hyperedge of $\mathcal{F}$ if there does not exist any $F_j \in \mathcal{F}$ such that $F_j \subset F_i$. The *minimization* of $\mathcal{F}$, denoted by $\mathcal{F}_{min}$, is the subset of $\mathcal{F}$ with only minimal hyperedges. It is easy to see that $Hs(\mathcal{F}) = Hs(\mathcal{F}_{min})$ [6], [36]. Hence, it suffices to consider $\mathcal{F}_{min}$ in HS enumeration. Note that $|\mathcal{F}_{min}|$ is usually much smaller than $|\mathcal{F}|$ in practice.

Auxiliary structures. Based on the known input $\mathcal{F}$, we compute $\mathcal{F}_{min}$ in the pre-processing step, to facilitate dynamic HS enumeration. This can be easily adapted to dynamic FD discovery, following the conversions in Table IV. Specifically, for every $A \in R$, we treat D_r^A as hypergraph $\mathcal{F}$ and build $\mathcal{F}_{min}$ accordingly. Note that $\mathcal{F}_{min}$ is computed in terms of every D_r^A instead of D_r. A minimal hyperedge of D_r^A does not necessarily correspond to a minimal hyperedge of D_r. For example, $D_r^C = \{AB\}$ if $D_r = \{A, ABC\}$. We see AB is a minimal hyperedge of D_r^C, but its corresponding hyperedge ABC is not a minimal hyperedge of D_r. Our experimental evaluations (Section VII) show that the accumulated memory usage of $\mathcal{F}_{min}$ for all D_r^A is still very small.

The set $\triangle\mathcal{F}$ of updates may incur changes to $\mathcal{F}_{min}$. We update $\mathcal{F}_{min}$ and compute critical hyperedges for (candidate) HSs *w.r.t.* $\mathcal{F}_{min}$ in our dynamic HS enumeration methods.

C. Updating $Hs(\mathcal{F})$ with $\triangle\mathcal{F}^+$

Let $\triangle\mathcal{F} = \triangle\mathcal{F}^+ \cup \triangle\mathcal{F}^-$, where $\triangle\mathcal{F}^+$ (resp. $\triangle\mathcal{F}^-$) is the set of inserted (resp. removed) hyperedges. In the following two subsections, we present two algorithms to handle $\triangle\mathcal{F}^+$ and $\triangle\mathcal{F}^-$ respectively. They are independent of each other. The order of applying them is irrelevant if $\triangle\mathcal{F}^+ \cap \triangle\mathcal{F}^- = \emptyset$. Otherwise, $\triangle\mathcal{F}^+$ is processed before $\triangle\mathcal{F}^-$.

We first consider $\triangle\mathcal{F}^+$. $\mathcal{F} \oplus \triangle\mathcal{F}^+ = \mathcal{F} \cup \triangle\mathcal{F}^+$. There are two observations that guide our algorithm design. (1) $\triangle\mathcal{F}^+$ does not affect $Hs(\mathcal{F})$ unless $\mathcal{F}_{min}$ is changed after applying $\triangle\mathcal{F}^+$ to $\mathcal{F}$. That is, our computation is guided by the changes in $\mathcal{F}_{min}$, which is usually much smaller than $\triangle\mathcal{F}^+$. (2) A minimal HS of $\mathcal{F} \cup \triangle\mathcal{F}^+$, is either a minimal HS of $\mathcal{F}$, or obtained by adding vertices to a minimal HS of $\mathcal{F}$. The formal result is stated as follows.

Proposition 1: If $H \in Hs(\mathcal{F} \cup \triangle\mathcal{F}^+)$, then there exists $H' \in Hs(\mathcal{F})$ such that $H' \subseteq H$.

Algorithm. Insert (Algorithm 2) takes as inputs $Hs(\mathcal{F})$, $\triangle\mathcal{F}^+$, $\mathcal{F}_{min}$, and outputs $Hs(\mathcal{F} \cup \triangle\mathcal{F}^+)$. Note that $\mathcal{F}$ is not needed. Insert first updates $\mathcal{F}_{min}$ by computing the set of new (resp. obsolete) minimal hyperedges for $\mathcal{F}_{min}$, denoted by $\triangle\mathcal{F}^+_{min}$ (resp. $\triangle\mathcal{F}^-_{min}$) (lines 1-2). Specifically, (1) $\triangle\mathcal{F}^+_{min} \subseteq \triangle\mathcal{F}^+$, and a hyperedge F from $\triangle\mathcal{F}^+$ belongs to $\triangle\mathcal{F}^+_{min}$ if there does not exist $F' \in \triangle\mathcal{F}^+ \cup \mathcal{F}_{min}$, such that $F' \subset F$. (2) $\triangle\mathcal{F}^-_{min} \subseteq \mathcal{F}_{min}$, and a hyperedge F belongs to $\triangle\mathcal{F}^-_{min}$ if there exists $F' \in \triangle\mathcal{F}^+$, such that $F' \subset F$.

Insert then computes critical hyperedges for HSs in $Hs(\mathcal{F})$ and updates $Hs(\mathcal{F})$ (lines 3-7). It enumerates $H \in Hs(\mathcal{F})$ and does the following. (1) For each $F \in \mathcal{F}_{min}$, if H, F have only one common vertex v, then it adds F into $crit_v^H$. (2) If H has any empty $crit_v^H$, then it removes H from $Hs(\mathcal{F})$ because H is not a minimal HS and no minimal HSs can be obtained by adding vertices to H. Insert finally calls WalkDown to compute $Hs(\mathcal{F} \cup \triangle\mathcal{F}^+)$ (lines 10-13). Specifically, for each $H \in Hs(\mathcal{F})$, it saves uncovered hyperedges in $uncov^H$ and candidate vertices in $cand$, respectively, and provides WalkDown with $crit^H$, *i.e.,* $crit_v^H$ for all $v \in H$.

Algorithm 2: Insert

Input: $Hs(\mathcal{F})$, $\triangle\mathcal{F}^+$, $\mathcal{F}_{min}$
Output: $Hs(\mathcal{F} \cup \triangle\mathcal{F}^+)$

1 compute the set of new (resp. obsolete) minimal hyperedges for $\mathcal{F}_{min}$, denoted by $\triangle\mathcal{F}^+_{min}$ (resp. $\triangle\mathcal{F}^-_{min}$);
2 $\mathcal{F}_{min} \leftarrow \mathcal{F}_{min} \cup \triangle\mathcal{F}^+_{min} \setminus \triangle\mathcal{F}^-_{min}$;
3 **foreach** $H \in Hs(\mathcal{F})$ **do**
4 **foreach** $F \in \mathcal{F}_{min}$ **do**
5 **if** $F \cap H = \{v\}$ **then** add F into $crit_v^H$;
6 **if** $crit_v^H = \emptyset$ *for any* $v \in H$ **then**
7 remove H from $Hs(\mathcal{F})$;
8 *tested* $\leftarrow$ an empty hash set;
9 *result* $\leftarrow \emptyset$;
10 **foreach** $H \in Hs(\mathcal{F})$ **do**
11 $uncov^H \leftarrow \{F \in \triangle\mathcal{F}^+_{min} \mid F \cap H = \emptyset\}$;
12 $cand \leftarrow R \setminus H$;
13 WalkDown $(H, crit^H, uncov^H, cand)$;
14 **return** *result*;

Algorithm 3: WalkDown

Input: candidate HS H, critical hyperedges $crit^H$, uncovered hyperedges $uncov^H$, and candidate vertices *cand*

1 **if** $H \in$ *tested* **then return**;
2 *tested* $\leftarrow$ *tested* $\cup\{H\}$;
3 **if** $uncov^H = \emptyset$ **then**
4 add H into *result*;
5 **return**;
6 choose a hyperedge F from $uncov^H$;
7 $U \leftarrow cand \cap F$;
8 $cand \leftarrow cand \setminus U$;
9 **foreach** $u \in U$ **do**
10 $G \leftarrow H \cup \{u\}$;
11 **if** $G \in$ *tested* **then continue**;
12 $crit_u^G \leftarrow \{F \in uncov^H | u \in F\}$;
13 $uncov^G \leftarrow uncov^H \setminus crit_u^G$;
14 $crit^G \leftarrow \{crit_u^G\}$;
15 **foreach** $v \in H$ **do**
16 $crit_v^G \leftarrow \{F \in crit_v^H | u \notin F\}$;
17 $crit^G \leftarrow crit^G \cup \{crit_v^G\}$;
18 **if** $crit_v^G \neq \emptyset$ *for each* $v \in G$ **then**
19 WalkDown $(G, crit^G, uncov^G, cand)$;
20 $cand \leftarrow cand \cup \{u\}$;

WalkDown (Algorithm 3) is an adaption of MMCS [23]. It adds candidate H into the result set if H covers all hyperedges (lines 3-5). Otherwise, it chooses an uncovered hyperedge F, and tries all combinations of vertices from $cand$ that can cover F (lines 6-8). When a vertex u is included in H for a new candidate G, $crit^G$ and $uncov^G$ are computed based on $crit^H$ and $uncov^H$ (lines 12-17), just as in MMCS. Specifically, all hyperedges uncovered by H but covered by G belong to $crit^G_u$ (line 12). For each $v \in H$, a hyperedge F from $crit^H_v$ is included in $crit^G_v$ if F does not contain u (line 16). If G passes the minimality check (line 18), then WalkDown is recursively called following a depth-first-search (DFS) strategy (line 19).

WalkDown differs from MMCS mainly in two aspects. (1) A DFS traversal starts from each $H \in Hs(\mathcal{F})$ in WalkDown, rather than from an empty set in MMCS. That is, we compute $Hs(\mathcal{F} \cup \triangle\mathcal{F}^+)$ based on $Hs(\mathcal{F})$. (2) We use a hash set $tested$ to save all the tested candidate HSs (line 8 in Insert and line 2 in WalkDown). WalkDown may generate the same candidate from different HSs in $Hs(\mathcal{F})$, *e.g.,* ABC from AB and BC. This is necessary to obtain all minimal HSs, since we perform DFS from all HSs of $Hs(\mathcal{F})$ independently. We employ $tested$ to avoid duplicate computations (lines 1, 11 in WalkDown).

Example 10: Let $\mathcal{F} = \{ABE, CDE, ABCE, BCDE\}$. We have $\mathcal{F}_{min} = \{ABE, CDE\}$ and $Hs(\mathcal{F}) = \{AC, AD, BC, BD, E\}$. Now let $\triangle\mathcal{F}^+ = \{AE, BE, CE, EG\}$. (1) We see $\triangle\mathcal{F}^+_{min} = \{AE, BE, CE, EG\}$, $\triangle\mathcal{F}^-_{min} = \{ABE, CDE\}$, and the updated $\mathcal{F}_{min} = \{AE, BE, CE, EG\}$. (2) We compute critical hyperedges for each HS in $Hs(\mathcal{F})$, as shown in Figure 1. Recall that critical hyperedges are computed in terms of $\mathcal{F}_{min}$ instead of $\mathcal{F}$. Leveraging critical hyperedges, we have $Hs(\mathcal{F}) = \{AC, BC, E\}$ after removing the non-minimal HSs. (3) We call WalkDown to search for new minimal HSs based on HSs from $Hs(\mathcal{F})$; the details are provided in Figure 1. Take AC as an example. Two hyperedges BE and EG are not covered by AC. Suppose we first cover BE. We have ABC (resp. ACE) by adding B (resp. E) to AC. ACE is pruned since some vertices from it have no critical hyperedges, while ABC passes the minimality check. We further add vertices to ABC for covering EG, and obtain a minimal HS $ABCG$. (4) When ABC is generated for the first time, it is added into the set $tested$. When we generate ABC based on BC again, we can directly discard ABC since it is found in $tested$. (5) Finally, we have two minimal HSs $ABCG$ and E. □

Correctness. We show the correctness of Insert. (1) Insert may prune candidates from $Hs(\mathcal{F})$ leveraging critical hyperedges. It is easy to see that no minimal HSs can be generated based on these candidates by including more vertices. (2) For each $H \in Hs(\mathcal{F})$, WalkDown guarantees to find all minimal HSs of $Hs(\mathcal{F} \cup \triangle\mathcal{F}^+)$ that include H, according to the correctness of MMCS. Putting this together with Proposition 1, Insert guarantees to find all minimal HSs of $Hs(\mathcal{F} \cup \triangle\mathcal{F}^+)$.

Complexity. It takes $O((|\mathcal{F}_{min}| + |\triangle\mathcal{F}^+|) \cdot |\triangle\mathcal{F}^+|)$ to compute $\triangle\mathcal{F}^+_{min}$ and $\triangle\mathcal{F}^-_{min}$, and $O(|\mathcal{F}_{min} \cup \triangle\mathcal{F}^+_{min} \setminus \triangle\mathcal{F}^-_{min}| \cdot |Hs(\mathcal{F})|)$ to compute critical hyperedges for HSs in $Hs(\mathcal{F})$. For each $H \in Hs(\mathcal{F})$, there are at most $2^{|R \setminus H|}$ candidate HSs based on it. For each candidate, it takes at most $O(|\triangle\mathcal{F}^+_{min}|)$ to check whether the candidate is a minimal HS.

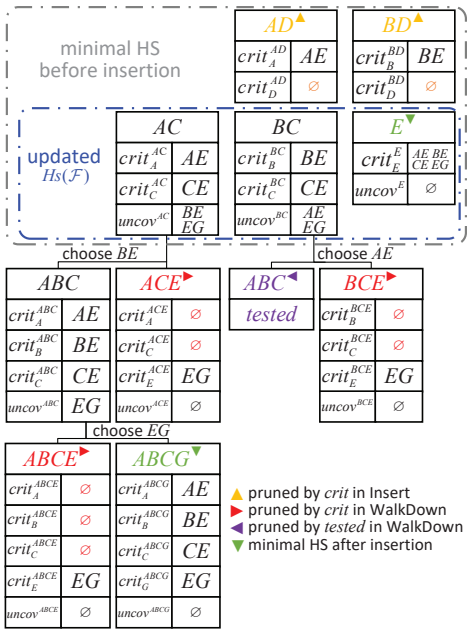

Fig. 1. Example 10 for Algorithms Insert and WalkDown

D. Updating $Hs(\mathcal{F})$ with $\triangle\mathcal{F}^-$

We then consider the subset $\triangle\mathcal{F}^-$ of $\triangle\mathcal{F}$ with only hyperedge deletions. $\mathcal{F} \oplus \triangle\mathcal{F}^- = \mathcal{F} \setminus \triangle\mathcal{F}^-$. It is easy to see that $Hs(\mathcal{F})$ remains unchanged unless $\mathcal{F}_{min}$ changes after applying $\triangle\mathcal{F}^-$ to $\mathcal{F}$, the same as $\triangle\mathcal{F}^+$. However, the following example shows the challenges inherent in dealing with hyperedge deletions, compared with insertions.

Example 11: Consider $\mathcal{F} = \{A, AB, CD\}$. We see $\mathcal{F}_{min} = \{A, CD\}$ and $Hs(\mathcal{F}) = \{AC, AD\}$. With $\triangle\mathcal{F}^- = \{A, CD\}$, *i.e.,* the deletion of hyperedges A and CD, $\mathcal{F}_{min}$ becomes $\{AB\}$ and $Hs(\{AB\})$ is $\{A, B\}$. □

We see the following. (1) The changes to $\mathcal{F}_{min}$ *cannot* be computed with only $\mathcal{F}_{min}$ and $\triangle\mathcal{F}^-$; $\mathcal{F}$ is also needed. This is because non-minimal hyperedges from $\mathcal{F}$, *e.g.,* AB, may become minimal after hyperedges in $\triangle\mathcal{F}^-$ are removed from $\mathcal{F}$. (2) After some hyperedges are removed, HSs in $Hs(\mathcal{F})$, *e.g.,* AC and AD, may become not minimal since they contain "redundant" vertices to cover the removed hyperedges. This implies that we need to consider candidate HSs by removing vertices from HSs in $Hs(\mathcal{F})$. However, we see some new HSs, *e.g.,* B, *cannot* be obtained in this way. Indeed, a carefully designed strategy that combines both addition and removal of vertices is required for generating candidate HSs, to guarantee the correctness and completeness of the results.

Algorithm. Delete (Algorithm 4) computes $Hs(\mathcal{F} \setminus \triangle\mathcal{F}^-)$, based on $Hs(\mathcal{F})$, $\triangle\mathcal{F}^-$, $\mathcal{F}$ and $\mathcal{F}_{min}$.

Delete first updates $\mathcal{F}_{min}$ by computing $\triangle\mathcal{F}^-_{min}$ and $\triangle\mathcal{F}^+_{min}$ (line 1-2). Specifically, (1) $\triangle\mathcal{F}^-_{min} = \mathcal{F}_{min} \cap \triangle\mathcal{F}^-$.

Algorithm 4: Delete

Input: $Hs(\mathcal{F})$, $\triangle\mathcal{F}^-$, $\mathcal{F}$, $\mathcal{F}_{min}$
Output: $Hs(\mathcal{F} \setminus \triangle\mathcal{F}^-)$

1 compute the set of new (resp. obsolete) minimal hyperedges for $\mathcal{F}_{min}$, denoted by $\triangle\mathcal{F}^+_{min}$ (resp. $\triangle\mathcal{F}^-_{min}$);
2 $\mathcal{F}_{min} \leftarrow \mathcal{F}_{min} \cup \triangle\mathcal{F}^+_{min} \setminus \triangle\mathcal{F}^-_{min}$;
3 *affected* $\leftarrow \bigcup_{F \in \triangle\mathcal{F}^-_{min}} F$;
4 *pending* $\leftarrow \{F \in \mathcal{F}_{min} | F \cap$ *affected* $\neq \emptyset\}$;
5 **foreach** $H \in Hs(\mathcal{F})$ **do**
6 $\quad H \leftarrow H \setminus$ *affected*;
7 $\quad$ **foreach** $F \in \mathcal{F}_{min}$ **do**
8 $\quad\quad$ **if** $F \cap H = \{v\}$ **then** add F into $crit^H_v$;
9 *tested* $\leftarrow$ an empty hash set;
10 *result* $\leftarrow \emptyset$;
11 **foreach** $H \in Hs(\mathcal{F})$ **do**
12 $\quad uncov^H \leftarrow \{F \in$ *pending* $\mid F \cap H = \emptyset\}$;
13 $\quad cand \leftarrow R \setminus H$;
14 $\quad$ WalkDown $(H, crit^H, uncov^H, cand)$;
15 **return** *result*;

(2) $\triangle\mathcal{F}^+_{min} \subseteq \mathcal{F} \setminus \triangle\mathcal{F}^-$, and a hyperedge F belongs to $\triangle\mathcal{F}^+_{min}$, if (a) there exists at least one $F' \in \triangle\mathcal{F}^-_{min}$ such that $F' \subset F$, and (b) there does not exist any $F'' \in \mathcal{F} \setminus \triangle\mathcal{F}^-$ such that $F'' \subset F$. That is, F is not a minimal hyperedge of $\mathcal{F}$, but is minimal after applying $\triangle\mathcal{F}^-$ to $\mathcal{F}$.

Delete then computes two sets, referred to as *affected* and *pending* respectively (lines 3-4). Specifically, *affected* is the set of vertices that are contained in hyperedges from $\triangle\mathcal{F}^-_{min}$, and *pending* is the set of hyperedges from the updated $\mathcal{F}_{min}$ that contain vertices in *affected*. Delete computes $Hs(\mathcal{F} \setminus \triangle\mathcal{F}^-)$ by leveraging the two sets. In a nutshell, it first updates HSs in $Hs(\mathcal{F})$ by removing the vertices in *affected*, to "cancel" the effect of the removed hyperedges from $\triangle\mathcal{F}^-_{min}$, and then adds vertices to the updated (candidate) HSs for further covering hyperedges from *pending*.

Specifically, this is done in two steps. (1) Vertices in *affected* are removed from HSs in $Hs(\mathcal{F})$, and the critical hyperedges are computed (lines 5-8). It is worth mentioning that there is no need to check the minimality of HSs here as we do in Insert, since vertices that may have no critical hyperedges are already removed. (2) WalkDown (Algorithm 3) is called to compute $Hs(\mathcal{F} \setminus \triangle\mathcal{F}^-)$ based on the updated HSs from $Hs(\mathcal{F})$ (lines 11-14). The inputs here are different from those in Insert (Algorithm 2). The set $uncov^H$ of hyperedges is a subset of *pending* rather than $\triangle\mathcal{F}^+_{min}$. It is easy to see that $\triangle\mathcal{F}^+_{min}$ is a subset of *pending* by the definition.

Example 12: Let $\mathcal{F} = \{ABC, ACD, BE, BCDE\}$. We see $\mathcal{F}_{min} = \{ABC, ACD, BE\}$ and $Hs(\mathcal{F}) = \{AB, AE, BC, BD, CE\}$. Now let $\triangle\mathcal{F}^- = \{BE\}$. (1) We have $\triangle\mathcal{F}^-_{min} = \{BE\}$, $\triangle\mathcal{F}^+_{min} = \{BCDE\}$, and the updated $\mathcal{F}_{min} = \{ABC, ACD, BCDE\}$. (2) B and E are the two *affected* vertices. We remove them from every HS in $Hs(\mathcal{F})$ and hence have the updated $Hs(\mathcal{F}) = \{A, C, D\}$ and *pending* = $\{ABC, BCDE\}$. We compute the related critical hyperedges, as shown in Figure 2. (3) We call WalkDown to further cover hyperedges from *pending*. The details are presented in Figure 2. We can employ critical hyperedges to prune non-

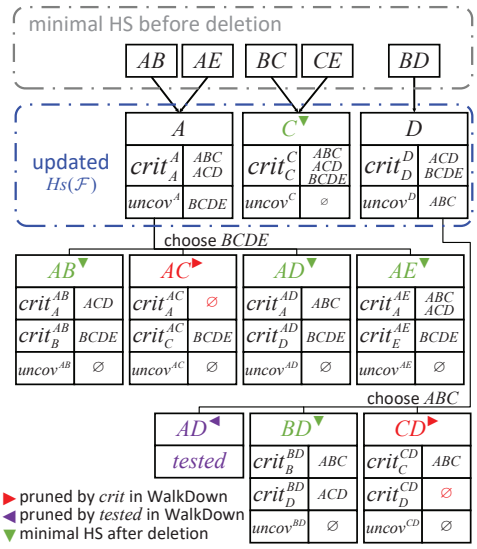

Fig. 2. Example 12 for Algorithms Delete and WalkDown

minimal candidates, *e.g.,* AC, and *tested* to avoid visiting the same candidate again, *e.g.,* AD. We finally have five minimal HSs C, AB, AD, AE and BD. □

We present a result underlying the correctness of Delete. Given $\mathcal{F} = \{F_1, \ldots, F_m\}$ on the vertex set V, and a subset $U \subseteq V$, we denote by $\mathcal{F}^U$ the hyperedges from $\mathcal{F}$ that contain vertices only from U.

Proposition 2: For each $H \in Hs(\mathcal{F}^U)$, we can find $H' \in Hs(\mathcal{F})$ that is a superset of H, such that $H' \setminus H \subseteq V \setminus U$.

Note that hyperedges are partitioned by vertices in Proposition 2; this differs from the setting of Proposition 1. As a corollary of Proposition 2, we have the result below.

Corollary 3: Let $Hs(\mathcal{F}) = \{H'_1, H'_2, \ldots, H'_k\}$. $Hs(\mathcal{F}^U) \subseteq \{H'_1 \setminus (V \setminus U), H'_2 \setminus (V \setminus U), \ldots, H'_k \setminus (V \setminus U)\}$.

Correctness. We show the correctness of Delete. (1) After removing the *affected* vertices, we know the updated $Hs(\mathcal{F})$ contains all minimal HSs of $\mathcal{F}^{V \setminus affected}$. This follows from Corollary 3. (2) WalkDown can find $Hs(\mathcal{F} \setminus \triangle\mathcal{F}^-)$ based on all minimal HSs of $\mathcal{F}^{V \setminus affected}$, since in the set *pending* we have all hyperedges that contain the *affected* vertices. It also guarantees to find only HSs in $Hs(\mathcal{F} \setminus \triangle\mathcal{F}^-)$, according to the correctness of WalkDown.

Complexity. It takes $O(|\mathcal{F}| \cdot |\triangle\mathcal{F}^-_{min}|)$ to update $\mathcal{F}_{min}$ (lines 1-2) and $O(|\mathcal{F}_{min} \cup \triangle\mathcal{F}^+_{min} \setminus \triangle\mathcal{F}^-_{min}| \cdot |Hs(\mathcal{F})|)$ to update $Hs(\mathcal{F})$ by removing vertices (lines 5-8). In the worst case, the computation of WalkDown (lines 11-14) starts from the empty set and has a search space of $2^{|R|}$, if the empty set belongs to the updated $Hs(\mathcal{F})$, *i.e.,* there exists H in the original $Hs(\mathcal{F})$ such that all vertices in H are the *affected* vertices. Since *affected* is the set of vertices contained in hyperedges from $\triangle\mathcal{F}^-_{min}$ instead of $\triangle\mathcal{F}^-$ and $\triangle\mathcal{F}^-_{min}$ is usually much smaller than $\triangle\mathcal{F}^-$, the worst case typically happens only when the ratio of $\triangle\mathcal{F}^-$ to $\mathcal{F}$ is large. Even in the worst case, we can still

TABLE V
DATASETS AND EXECUTION STATISTICS (TIME IN SECOND)

Dataset Properties				Batch			Dynamic ($\|\triangle r\| = 10\%\|r\|$)			Dynamic ($\|\triangle r\| = 20\%\|r\|$)			Dynamic ($\|\triangle r\| = 30\%\|r\|$)		
DataSet	$\|r\|$ (full)	$\|R\|$	$\|\Sigma\|$	Tane	FastFD	HyFD	$\|\triangle\Sigma\|$	DynFD	DHSFD	$\|\triangle\Sigma\|$	DynFD	DHSFD	$\|\triangle\Sigma\|$	DynFD	DHSFD
Balance	625	5	1	0.22	0.12	**0.1**	0	0.14	**0.034**	0	0.14	**0.052**	0	0.21	**0.071**
Iris	150	5	4	0.24	0.04	**0.075**	0	0.17	**0.029**	0	0.19	**0.033**	7	0.2	**0.049**
Claim	20K	11	12	1.1	202.3	**0.8**	0	**0.4**	1.5	2	**0.5**	2.2	2	**0.7**	3.5
Letter	20K	17	61	223.9	491.2	**2.3**	32	12	**1.2**	68	19.4	**2.1**	96	25.1	2.6
Bridges	108	13	142	0.3	**0.1**	**0.1**	61	0.2	**0.001**	71	0.25	**0.005**	96	0.27	**0.007**
Echo	132	13	527	0.3	**0.1**	**0.1**	48	0.2	**0.004**	89	0.23	**0.006**	157	0.26	**0.01**
NCV	1K	19	758	0.6	0.4	**0.2**	182	0.3	**0.03**	307	0.4	**0.07**	611	0.44	**0.09**
Hepa	155	20	8,250	4.9	4.1	**0.28**	2,127	0.4	**0.04**	4,623	0.5	**0.1**	11,008	0.7	**0.23**
FDR	250K	30	89,571	**25.7**	134.3	69	27	118.6	**2.1**	27	183.5	**4.7**	42	247.2	**7.2**
Census	50K	42	103,999	TL	TL	**403**	49,283	TL	**155**	63,728	TL	**243**	83,948	TL	**369**
Flights	250K	31	117,367	TL	TL	**645**	152,546	TL	**194**	57,535	TL	**395**	226,183	TL	**612**
Horse	300	28	128,726	76	60	**4.5**	53,342	29.1	**1.8**	93,449	32.7	**3.9**	129,121	33.4	5.3
Plista	1001	63	173,409	TL	TL	**12.4**	134,983	144.7	**4.2**	192,573	146.3	**7.6**	168,247	149.8	**11.4**
Accident	300K	47	196,455	TL	TL	**3,233**	121,594	TL	**1,102**	169,212	TL	**1,974**	213,324	TL	3,347
HAEM	15.5K	34	272,138	TL	4h31min	**78**	85,695	324.4	**7.5**	152,546	433.5	**16.5**	226,183	574.9	**24.9**
Pitches	250K	40	608,928	TL	TL	**923**	114,007	TL	**302**	225,402	TL	**598**	362,521	TL	**915**
CAB	67.3K	54	3,353,531	TL	TL	**6,614**	624,910	TL	**319.2**	1,368,206	TL	**692**	1,891,256	TL	**1,325**

leverage the updated $\mathcal{F}_{min}$, in contrast to the batch approach computing from scratch based on the updated $\mathcal{F}$.

VII. EXPERIMENTAL EVALUATIONS

In this section, we conduct experiments to verify the effectiveness and efficiency of our dynamic FD discovery and dynamic HS enumeration, and to analyze the properties of our methods in detail. All tested datasets and code are available at https://github.com/RangerShaw/DHSFD.

A. Experimental settings

Datasets. We use a host of real-life and synthetic datasets[1]. We summarize the characteristics of them in Table V. These datasets differ in not only the numbers of tuples and attributes, but also in data distributions. We employ them to give a detailed analysis of the performance of our approach.

Algorithms. All the algorithms are implemented in Java. (1) We compare our dynamic FD discovery method, referred to as DHSFD, against (a) DynFD [28]: the only known dynamic FD discovery method[2], and (b) HyFD [25], Tane [11] and FastFD [36]: batch FD discovery methods. HyFD adopts a hybrid strategy and is known as one of the fastest methods, and Tane and FastFD are representative column-based and row-based methods, respectively[3]. (2) We compare our dynamic HS enumeration methods, Insert and Delete (Section VI), against the known fastest batch method MMCS [23]. We develop a best-effort implementation of MMCS in Java, based on the original version written in C.

[1]Datasets are obtained from https://hpi.de/naumann/projects/repeatability/data-profiling/ and https://www.kaggle.com/datasets.

[2]Code is from https://github.com/HPI-Information-Systems/dynfd.

[3]All the codes are from https://hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling/algorithms.html.

Running environment. We run all experiments on a PC server with an Intel Xeon E-2224 3.4G CPU, 64GB of memory and CentOS 7. We report the average results of 5 runs.

B. Experimental results of dynamic FD discovery

Updates. We generate updates $\triangle r$ randomly, controlled by the size $|\triangle r| = |\triangle r^+| + |\triangle r^-|$ and the ratio λ of $|\triangle r^-|$ to $|\triangle r|$.

Measurement. In the preprocessing step, we compute Σ with HyFD, and build auxiliary structures for dynamic methods. Different from DHSFD, DynFD leverages only Plis (see Exp-5 for details). We compare the times of DHSFD and DynFD that dynamically compute Σ' on $r \oplus \triangle r$, against those of the batch methods that compute Σ' from scratch. The correctness of DHSFD is verified by checking the equivalence of the results. The time of DHSFD consists of the times for updating the difference-set and dynamic HS enumeration (including the time for updating auxiliary structures in the process).

Exp-1: DHSFD against batch and dynamic methods. We report running times of all methods in Table V, by varying $\frac{|\triangle r|}{|r|}$ from 10% to 30% and setting $\lambda = \frac{1}{3}$. Specifically, we randomly select 80% tuples as the instance r, and generate $\triangle r^+$ from the remaining 20% tuples and $\triangle r^-$ from $r \cup \triangle r^+$, respectively. We process $\triangle r^+$ and then $\triangle r^-$ in DHSFD and DynFD. All times are in seconds unless otherwise stated. We denote the time by TL if one algorithm cannot terminate within the time limit of 5 hours. The times of batch methods are obtained on $r \oplus \triangle r$. In this set of experiments, $|r \oplus \triangle r|$ only slightly increases; $|r \oplus \triangle r| = 110\%|r|$ when $|\triangle r| = 30\%|r|$ and $\lambda = \frac{1}{3}$. For space limitation, we report the results of batch methods with $|\triangle r| = 10\%|r|$. This favors batch methods.

We show the number $|\Sigma|$ of FDs on r. Moreover, we denote by $\triangle\Sigma$ the changes from Σ to Σ'. Specifically, $\triangle\Sigma$ is computed as $(\Sigma' \setminus \Sigma) \cup (\Sigma \setminus \Sigma')$, *i.e.,* the FDs that are added

into or removed from Σ. Intuitively, a large $|\triangle\Sigma|$ implies more challenges for the dynamic FD discovery.

(1) We see the following. (a) DHSFD is much faster than the batch methods on almost all tested data when $|\triangle r|$ = $20\%|r|$, and still performs better on most tested data even when $|\triangle r|$ = $30\%|r|$. (b) DHSFD significantly outperforms DynFD on almost all tested data. (c) DHSFD performs well on datasets with a wide range of $|R|$ and $|r|$, and exhibits very good efficiency even when $|\triangle\Sigma|$ is very large.

(2) We compare DHSFD against HyFD in detail. HyFD significantly outperforms other batch methods on almost all datasets, which is consistent with the results reported in [25], [34]. However, we see running HyFD on some datasets with large $|R|$ and $|r|$ is still very costly, *e.g.,* on Accident and CAB. This highlights the quest for dynamic FD discoveries.

DHSFD is much faster than HyFD on most datasets. This happens in datasets with large $|R|$. For example on CAB, HyFD takes 6,614 seconds, while DHSFD only takes 1,325 seconds even when $|\triangle r|$ = $30\%|r|$. This is also seen in datasets with large $|r|$, *e.g.,* FDR and Flights. Note that on instances with large $|r|$, the setting of $|\triangle r|$ = $30\%|r|$ implies large updates. The results show DHSFD well leverages the benefits of the row-based strategy and dynamic computations, and performs well on instances that are costly for batch methods.

(3) We compare DHSFD against DynFD in detail. DynFD is experimentally found to be much slower than DHSFD and HyFD, on almost all datasets. DynFD adopts the column-based strategy, and has to validate all FDs from Σ in response to tuple insertions. We find the validation step does not scale well on datasets with large $|\Sigma|$. Another reason for the inefficiency of DynFD is the inherent difficulty in dealing with tuple deletions. As an example, suppose $ABC{\rightarrow}D$ is in Σ. When tuples are deleted from r, DynFD has to check the validity of $AB{\rightarrow}D$, $AC{\rightarrow}D$ and $BC{\rightarrow}D$; when either of them is valid, $ABC{\rightarrow}D$ is not minimal and should be removed from Σ. Worse, if $AB{\rightarrow}D$ is valid, then $A{\rightarrow}D$ and $B{\rightarrow}D$ have to be validated as well. The validations are very expensive since they have to be conducted on $r \oplus \triangle r$. Although some optimizations are used in DynFD to improve the efficiency, the performance necessarily degrades. DynFD outperforms DHSFD only on Claim. This dataset has small $|\Sigma|$ and $|\triangle\Sigma|$, and in particular, only tuple insertions are applied to Claim by using λ = 0. This setting of Claim is similar to that used in [28]. We will further study the impact of tuple deletions by varying λ in Exp-2.

We find DynFD is suitable to maintain a small set of FDs in response to updates. In contrast, DHSFD aims to efficiently perform dynamic FD discovery when batch methods are expensive, *e.g.,* on the instances with large $|R|$, $|r|$ and $|\Sigma|$.

Exp-2: Scalability of algorithms. In this set of experiments, we study the scalability of DHSFD and the compared methods by varying parameters. For space limitation, we report results on CAB and Pitches; results on other datasets are similar.

(1) We set $|r|$ = 10K, $\frac{|\triangle r|}{|r|} = 20\%$, $|R| = 54$ and $\lambda = \frac{1}{2}$ by default on CAB, and vary one parameter in each experiment. We omit the results of DynFD if they are more than 5 hours.

Varying $|R|$. We report results in Figure 3a by varying $|R|$ from 34 to 54. The time of FD discovery is exponential in $|R|$ and hence very sensitive to $|R|$. We see DHSFD scales better than HyFD. As $|R|$ increases from 34 to 54, the time of DHSFD increases from 6.7s to 30.1s while that of HyFD increases from 11s to 89s. DynFD is much slower, and costs more than 5 hours when $|R|$ = 54.

Varying $|\triangle r|$. Figure 3b shows results by varying $|\triangle r|$ from 2K to 10K (the ratio of $|\triangle r|$ to $|r|$ increases from 20% to 100%). HyFD is indifferent to $|\triangle r|$, since $|r \oplus \triangle r| = |r|$ when $|\triangle r^+| = |\triangle r^-|$ (recall we set $\lambda = \frac{1}{2}$). DHSFD scales very well with $|\triangle r|$: the time increases from 30.1s to 127s. DHSFD still outperforms HyFD even when the ratio of $|\triangle r|$ to $|r|$ is more than 60% on CAB. DynFD cannot terminate within 5 hours.

Varying $|r|$. In Figure 3c, we vary $|r|$ from 10K to 14K ($|\triangle r|$ from 2K to 2.8K). The time of DHSFD increases from 30.1s to 56s while that of HyFD increases from 89s to 476s. We find the number of FDs on CAB almost triples (not shown) as $|r|$ increases from 10K to 14K, which necessarily hinders the efficiency of HyFD. We see DHSFD is very efficient in coping with a large number of FDs.

Varying λ. We report results in Figure 3d by varying λ from 0 to 100%. $\triangle r$ contains only tuple insertions (resp. deletions) if λ = 0 (resp. 100%). As λ increases, $|r \oplus \triangle r|$ decreases and hence the time of HyFD decreases. We see DynFD is very sensitive to the increase of tuple deletions. The time of DynFD increases from 2,100s to 7,023s and then grows beyond the time limit as λ increases. Recall this is because the cost of DynFD depends on $|r|$ for tuple deletions. In contrast, the time of DHSFD only increases from 20s to 57s.

(2) We set $|r|$ = 50K, $\frac{|\triangle r|}{|r|} = 20\%$, $|R| = 40$ and $\lambda = \frac{1}{2}$ by default on Pitches, and vary one parameter in each experiment. Specifically, we vary $|R|$ from 25 to 40 in Figure 3e, $|\triangle r|$ from 2K to 15K (the ratio of $|\triangle r|$ to $|r|$ from 4% to 30%) in Figure 3f, $|r|$ from 10K to 50K ($|\triangle r|$ from 2K to 10K) in Figure 3g, and λ from 0 to 100% in Figure 3h.

The results confirm our observations on CAB and tell us the following. (a) DHSFD scales better with $|R|$ than HyFD. The time of DHSFD increases from 9.2s to 45s, while that of HyFD increases from 4.8s to 82s. (b) DHSFD scales very well with $|\triangle r|$. The time increases from 15.2s to 77s, as $|\triangle r|$ increases from 2K to 15K. (c) As $|r|$ increases, the gap between the performance of HyFD and that of DHSFD becomes larger. (d) Compared with DynFD, DHSFD is less sensitive to the increase of λ. It still beats HyFD even when λ = 100% ($\triangle r$ consists of only tuple deletions).

Exp-3: Time decomposition. In this set of experiments, we decompose the time of DHSFD into the times for (a) updating the difference-set and (b) dynamic HS enumeration.

(1) We use CAB in Figure 3i, in the same setting as Figure 3c. The time for dynamic HS enumeration governs the overall time, due to the huge number of FDs on CAB (recall Table V). The increase of $|r|$ incurs more growth in the time of dynamic HS enumeration than that of updating the difference-set. This is because the number of FDs on CAB almost triples as $|r|$

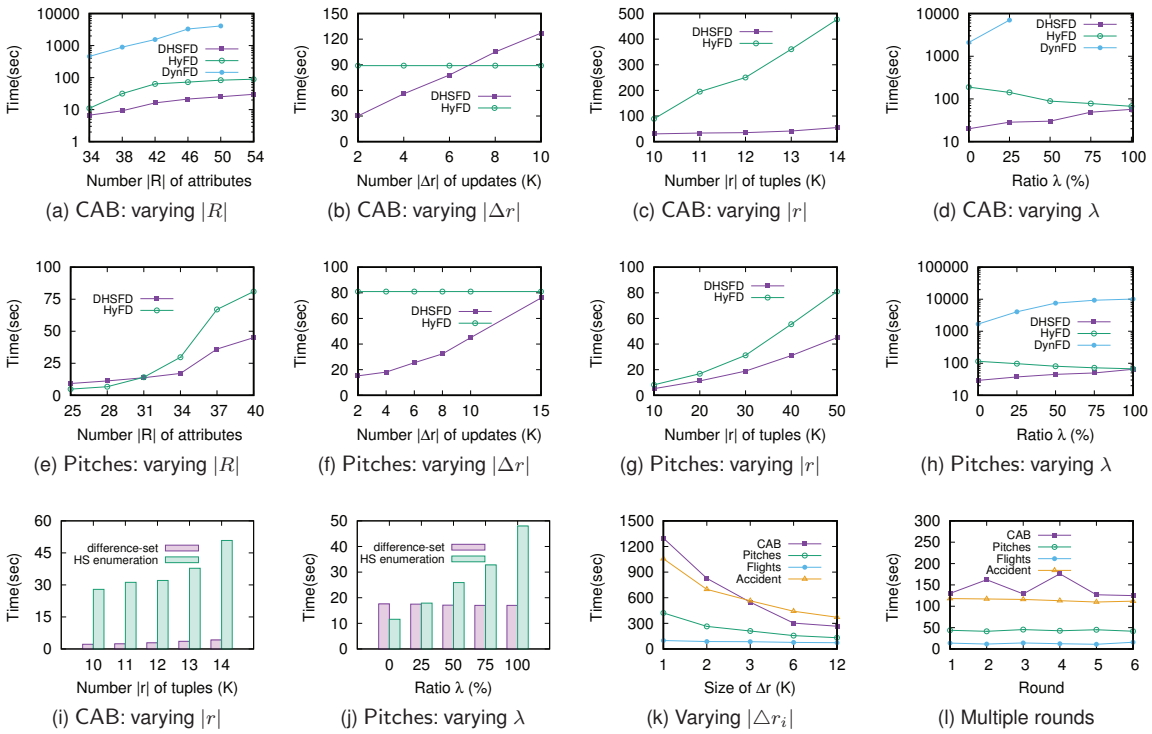

Fig. 3. DHSFD against DynFD and HyFD

increases. Nevertheless, the dynamic HS enumeration method scales very well with the number of FDs.

(2) We report the results on Pitches in Figure 3j, in the same setting as Figure 3h. As λ increases, $\triangle r$ contains more tuple deletions. We see the dynamic HS enumeration part takes more time, since it is necessarily more costly to handle deletions than insertions. In contrast, the time of updating the difference-set slightly decreases, since the sizes of the difference-set and Plis become smaller with more tuple deletions.

Exp-4: A series of updates. In this set of experiments, we employ DHSFD to handle a series of updates. There is no need to rebuild the auxiliary structures in this process, since they are maintained when updating the difference-set and performing dynamic HS enumeration. We handle $\triangle r$ as a series of updates ($\triangle r = \triangle r_1 \cup \ldots \cup \triangle r_k$), where $\triangle r_i$ ($i \in [1, k]$) is of the same size. We set $\triangle r$ = 12K, $\lambda = \frac{1}{3}$, and report the experimental results on datasets with large $|R|$ and $|r|$ from Table V.

(1) In Figure 3k, we vary $|\triangle r_i|$ from 1K to 12K, and report the total time for applying all $\triangle r_i$. Note that a single set of updates is applied to r if $|\triangle r_i|$ = 12K, while 12 sets of updates are applied one by one if $|\triangle r_i|$ = 1K. On tested datasets, the times for handling the series of updates decrease by 20%-70% as $|\triangle r_i|$ increases. This is mainly because the total time for HS enumeration decreases due to fewer rounds of requests.

(2) We fix $|\triangle r_i|$ = 2K in Figure 3l, and show the times for applying $\triangle r_1$ to r, $\triangle r_2$ to $r \oplus \triangle r_1$, etc. The time for different $\triangle r_i$ varies on a dataset. We find the reason is that different $\triangle r_i$ causes (significantly) different changes in the FD set.

Exp-5: Auxiliary structures. In Table VI, we present details of the auxiliary structures. For reference, we provide $|R|$, $|\Sigma|$ and the memory footprint (Data) of each dataset. We show the auxiliary structures of each dataset r: (a) the memory footprint of Plis, (b) the number $|D_r|$ of difference-sets of tuple pair in D_r, and (c) the number $|\mathcal{F}_{min}|$ of hyperedges in $\mathcal{F}_{min}$. Herein, $\mathcal{F}_{min}$ is the accumulated result by treating D_r^A as hypergraph $\mathcal{F}$ for every $A \in R$. Each difference-set of tuple pair in D_r (each hyperedge in $\mathcal{F}_{min}$) is encoded in $|R|$ bits, where each bit denotes whether an attribute from R exists in the difference-set or not. For example, the D_r of Census is the largest in Table VI, and needs less than 80M (bytes). The actual memory usage is a little larger, due to necessary overhead of the implementation.

We see the following. (a) Plis costs less memory than the original dataset (Data), since only tuple *ids* are stored in Plis. (b) $|D_r|$ differs significantly on different datasets, and is much smaller than the upper bound of $2^{|R|}$ on all tested data. A large $|R|$ does not necessarily lead to a large $|D_r|$, since $|D_r|$ also depends on the data distributions. (c) $|\mathcal{F}_{min}|$ is usually much smaller than $|D_r|$. This favors dynamic HS enumeration methods. Recall that Algorithm Insert leverages $\mathcal{F}_{min}$ but does not need $\mathcal{F}$, and that Algorithm Delete is guided by the set of vertices contained in hyperedges from $\triangle\mathcal{F}_{min}^{-}$.

We compare DHSFD against HyFD and DynFD in terms of the memory usage. (a) On each tested dataset, we see the

TABLE VI
AUXILIARY STRUCTURES ON DIFFERENT DATASETS

Dataset Properties				Auxiliary Structures		
DataSet	$\|R\|$	$\|\Sigma\|$	Data	Plis	$\|D_r\|$	$\|\mathcal{F}_{min}\|$
Balance	5	1	46.4K	15.6K	19	14
Iris	5	4	188.8K	58.1K	30	20
Claim	11	12	13.1M	6.2M	511	19
Letter	17	61	17.8M	6.7M	110,389	329
Bridges	13	142	76.4K	33.8K	482	96
Echo	13	527	93.5K	50K	185	204
NCV	19	758	913.6K	785.6K	1,483	330
Hepa	20	8,250	164.2K	52.3K	1,185	630
FDR	30	89,751	409.5M	177M	348	6,391
Census	42	103,999	159.3M	44.6M	14,177,591	1,648
Flights	31	117,367	340.4M	157.1M	88,374	14,947
Horse	28	128,726	439.8K	176.3K	15,909	10,227
Plista	63	173,409	2.4M	1.4M	57,578	14,109
Accident	47	196,455	749.7M	445.9M	10,778,684	16,040
HAEM	34	272,138	118.6M	11.2M	340,554	14,658
Pitches	40	608,928	501.1M	208.8M	179,993	150,693
CAB	54	3,353,531	184.1M	77.5M	2,844,553	57,794

total memory of auxiliary structures is less than or similar to Data. This implies that DHSFD does *not* incur extra memory cost; the datasets are needed by HyFD but only used in the preprocessing step of DHSFD. (b) DHSFD additionally needs D_r and $\mathcal{F}_{min}$, compared with DynFD. $\mathcal{F}_{min}$ facilitates dynamic HS enumeration and always incurs small memory footprint. D_r is necessary for the dynamic row-based approach. It brings much better performance than DynFD with reasonable memory usage, as shown in the experiments.

$|D_r|$ and $|\mathcal{F}_{min}|$ provide valuable hints on the advantage of the row-based dynamic FD discovery method against batch methods, which helps us analyze the experimental results in Table V. (a) The largest $|\mathcal{F}_{min}|$ is seen on Pitches, which negatively affects dynamic HS enumerations. (b) A large $|D_r|$ is found on Accident. Updates to Accident with the same value distributions are very likely to incur large changes to D_r, and in practice each operation on a large D_r may become costly. (c) Census has the largest $|D_r|$, but its $|\mathcal{F}_{min}|$ is very small. The dynamic HS enumerations on Census can be very efficient, which partly balances the cost of updating D_r.

C. Experimental results of dynamic HS enumeration

We experimentally verify the benefit of dynamic HS enumeration, using datasets with large $|D_r|$ from Table VI. On each dataset r, the time of MMCS (resp. dynamic HS enumeration) is the sum of the times of MMCS (resp. dynamic HS enumeration) on every D_r^A for $A \in R$. For dynamic HS enumeration of $\mathcal{F}$, the only auxiliary structure we use is $\mathcal{F}_{min}$.

Exp-6: Dynamic HS enumeration against MMCS. To analyze the performance in detail, we compare Insert and Delete against MMCS, respectively. We set $\triangle\mathcal{F} = \triangle\mathcal{F}^+$ (resp. $\triangle\mathcal{F}^-$) for Insert (resp. Delete), adjust the ratio $\frac{|\triangle\mathcal{F}|}{|\mathcal{F}|}$ in the experiments, and report the speedup ratio, *i.e.,* the ratio of the time of MMCS to that of Insert (or Delete). In Figure 4,

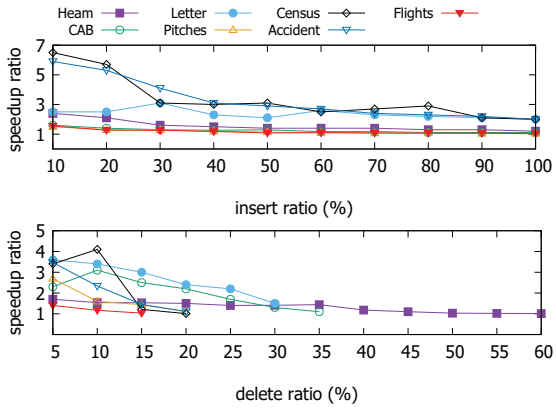

Fig. 4. Dynamic HS enumeration against MMCS (varying $|\triangle\mathcal{F}|$ to $|\mathcal{F}|$)

we show the results of each dataset until the speedup ratio is smaller than 1, *i.e.,* Insert (Delete) is slower than MMCS.

(1) We first compare Insert against MMCS. Insert is much more efficient than MMCS. For example, Insert is almost up to 7 times faster than MMCS on Census when $|\triangle\mathcal{F}| = 10\%|\mathcal{F}|$, and about 3 times faster than MMCS on Accident when $|\triangle\mathcal{F}|$ is up to $50\%|\mathcal{F}|$. The results also show that Insert still outperforms MMCS on all tested data, when $|\triangle\mathcal{F}| = |\mathcal{F}|$.

(2) We then compare Delete against MMCS. It is more challenging to handle hyperedge deletions than insertions in dynamic HS enumeration, and the time of MMCS decreases with more hyperedge deletions. We see the advantage of Delete varies on different datasets. For example, Delete outperforms MMCS, when $|\triangle\mathcal{F}|$ is up to 60% of $|\mathcal{F}|$ on HAEM and up to 15% of $|\mathcal{F}|$ on Flights. The results show that Delete is on average 2.46 (resp. 1.73) times faster than MMCS when $|\triangle\mathcal{F}| = 10\%|\mathcal{F}|$ (resp. $15\%|\mathcal{F}|$).

VIII. CONCLUSION

We have studied the problem of dynamic FD discovery, by recasting it as dynamic HS enumeration of the difference-set. We have presented methods to update the difference-set, and techniques for dynamic HS enumeration. We have conducted experiments to verify the effectiveness and scalability of our dynamic FD discovery and HS enumeration methods.

We intend to explore the possibility of combining our approach with transaction or streaming data processing systems to continuously monitor and maintain FDs, and conduct more studies to verify the effectiveness of dynamic HS enumeration methods on datasets from, *e.g.,* AI and semantic web.

ACKNOWLEDGMENTS

This work is supported by National Key R&D Program of China 2018YFB1700403, 2018YFB1403200 and NSFC 62172102, 61572135, 61925203.

We are really grateful to anonymous reviewers for their valuable comments and suggestions.

REFERENCES

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

[2] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. *Data Profiling*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[3] Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Detecting unique column combinations on dynamic data. In *ICDE*, pages 1036–1047, 2014.

[4] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. DFD: efficient functional dependency discovery. In *CIKM*, pages 949–958, 2014.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. Hitting set enumeration with partial information for unique column combination discovery. *Proc. VLDB Endow.*, 13(11):2270–2283, 2020.

[7] Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, and Giuseppe Polese. Incremental discovery of functional dependencies with a bit-vector algorithm. In *SEBD*, 2019.

[8] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *VLDB*, pages 687–698, 1999.

[9] Wenfei Fan and Floris Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[10] Andrew Gainer-Dewar and Paola Vera-Licona. The minimal hitting set generation problem: Algorithms and computation. *SIAM J. Discret. Math.*, 31(1):63–100, 2017.

[11] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[12] Ihab F. Ilyas and Xu Chu. *Data Cleaning*. ACM, 2019.

[13] Jyrki Kivinen and Heikki Mannila. Approximate dependency inference from relations. In *ICDT*, pages 86–98, 1992.

[14] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *PVLDB*, 11(7):759–772, 2018.

[15] Li Lin and Yunfei Jiang. The computation of hitting sets: Review and new algorithms. *Inf. Process. Lett.*, 86(4):177–184, 2003.

[16] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. Approximate denial constraints. *PVLDB*, 13(10):1682–1695, 2020.

[17] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT*, pages 350–364, 2000.

[18] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. Discovering reliable approximate functional dependencies. In *SIGKDD*, 2017.

[19] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. Discovering reliable dependencies from data: Hardness and improved algorithms. In *ICDM*, 2018.

[20] Panagiotis Mandros, David Kaltenpoth, Mario Boley, and Jilles Vreeken. Discovering functional dependencies from mixed-type data. In *SIGKDD*, 2020.

[21] Heikki Mannila and Kari-Jouko Räihä. Dependency inference. In *VLDB*, pages 155–158, 1987.

[22] Heikki Mannila and Kari-Jouko Räihä. Algorithms for inferring functional dependencies from relations. *Data Knowl. Eng.*, 12(1):83–99, 1994.

[23] Keisuke Murakami and Takeaki Uno. Efficient algorithms for dualizing large-scale hypergraphs. *Discret. Appl. Math.*, 170:83–94, 2014.

[24] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *PVLDB*, 8(10):1082–1093, 2015.

[25] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*, pages 821–833, 2016.

[26] Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In *EDBT*, pages 342–353, 2017.

[27] Frédéric Pennerath, Panagiotis Mandros, and Jilles Vreeken. Discovering approximate functional dependencies using smoothed mutual information. In *SIGKDD*, 2020.

[28] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. Dynfd: Functional dependency discovery in dynamic datasets. In *EDBT*, pages 253–264, 2019.

[29] Nuhad Shaabani and Christoph Meinel. Incrementally updating unary inclusion dependencies in dynamic data. *Distributed Parallel Databases*, 37(1):133–176, 2019.

[30] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, pages 57–67, 1996.

[31] Shaoxu Song, Fei Gao, Ruihong Huang, and Chaokun Wang. Data dependencies extended for variety and veracity: A family tree. *IEEE Trans. Knowl. Data Eng. Accepted for publication. doi: 10.1109/TKDE.2020.3046443*.

[32] Zijing Tan, Ai Ran, Shuai Ma, and Sheng Qin. Fast incremental discovery of pointwise order dependencies. *PVLDB*, 13(10):1669–1681, 2020.

[33] Ziheng Wei, Sven Hartmann, and Sebastian Link. Discovery algorithms for embedded functional dependencies. In *SIGMOD*, pages 833–843, 2020.

[34] Ziheng Wei and Sebastian Link. Discovery and ranking of functional dependencies. In *ICDE*, pages 1526–1537, 2019.

[35] Ziheng Wei and Sebastian Link. Embedded functional dependencies and data-completeness tailored database design. *ACM Trans. Database Syst.*, 46(2):7:1–7:46, 2021.

[36] Catharine M. Wyss, Chris Giannella, and Edward L. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK*, 2001.

[37] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. A statistical perspective on discovering functional dependencies in noisy data. In *SIGMOD*, 2020.