DynFD: Functional Dependency Discovery in Dynamic Datasets

Philipp Schirmer
bakdata GmbH
Berlin, Germany
philipp.schirmer@bakdata.com

Thorsten Papenbrock
Hasso Plattner Institute
University of Potsdam, Germany
thorsten.papenbrock@hpi.de

Sebastian Kruse
Hasso Plattner Institute
University of Potsdam, Germany
sebastian.kruse@hpi.de

Dennis Hempfing
Hasso Plattner Institute
University of Potsdam, Germany
dennis.hempfing@hpi-alumni.de

Torben Meyer
Hasso Plattner Institute
University of Potsdam, Germany
torben.meyer@hpi-alumni.de

Daniel Neuschäfer-Rube
Hasso Plattner Institute
University of Potsdam, Germany
daniel.neuschaefer-rube@hpi-alumni.de

Felix Naumann
Hasso Plattner Institute
University of Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Functional dependencies (FDs) support various tasks for the management of relational data, such as schema normalization, data cleaning, and query optimization. However, while existing FD discovery algorithms regard only static datasets, many real-world datasets are constantly changing – and with them their FDs. Unfortunately, the computational hardness of FD discovery prohibits a continuous re-execution of those existing algorithms with every change of the data.

To this end, we propose DynFD, the first algorithm to discover and maintain functional dependencies in dynamic datasets. Whenever the inspected dataset changes, DynFD *evolves* its FDs rather than recalculating them. For this to work efficiently, we propose indexed data structures along with novel and efficient update operations. Our experiments compare DynFD's incremental mode of operation to the repeated re-execution of existing, static algorithms. They show that DynFD can maintain the FDs of dynamic datasets over an order of magnitude faster than its static counter-parts.

1 FUNCTIONAL DEPENDENCIES

Traditional data profiling algorithms solve the problem of *discovering* all metadata of type X in dataset Y. However, once those algorithms finish, the dataset usually keeps evolving, thereby rendering the discovered metadata outdated. An *incremental* data profiling algorithm, on the contrary, acknowledges the dynamic nature of data by *maintaining* all metadata of some type. It takes as input the data and its (statically profiled) metadata and, then, updates the metadata with every change, i. e., insert, update, and delete of the data. In this paper, we propose such an incremental algorithm for functional dependencies.

For an instance r of a relation R, a functional dependency (FD) $X \rightarrow A$ holds iff all records with the same values for the set of attributes $X \subseteq R$ also share the same value for attribute $A \in R$ [5]. We say that X *functionally determines* A.

Definition 1.1. The *functional dependency* $X \rightarrow A$ with $X \subseteq R$ and $A \in R$ is *valid* for instance r of R, iff $\forall t_i, t_j \in r\colon t_i[X] = t_j[X] \Rightarrow t_i[A] = t_j[A]$. We call X the left-hand side (Lhs) and A the right-hand side (Rhs), respectively.

FDs often arise from real-world relationships. Consider, for example, a relation with information about people including their *ZIP code* and *city* name. In such a dataset, the ZIP codes functionally determine the city names of the records, because ZIP codes are a more fine-grained localization. The presence or absence of certain dependencies, in general, helps to understand complex semantic relationships in data exploration scenarios. Being able to track the validity of certain dependencies over time provides even deeper insights. In a product database, for instance, the FD *num_sales* → *num_shipments* might hold only overnight, because shipments are delayed in daily business. Or the FD *product* → *price* in a pricing database was temporarily violated at the time of a system migration. Apart from data exploration, further applications for functional dependencies include schema normalization [4], query optimization [14], data integration [11], data cleansing [2], and data translation [3]. Given that the functional dependencies are not only known for a snapshot of the data but over a longer period of time, it is for any of these use cases easier to identify *robust* dependencies. Continuous *change patterns* for non-robust dependencies, on the other hand, can be of interest themselves; and *sudden changes* of thus far robust FDs might signal data quality issues, i. e., erroneous updates.

The most interesting FDs for all such applications are *minimal, non-trivial* FDs. An FD $X \rightarrow A$ is non-trivial if $A \notin X$, otherwise it would hold on any instance r of R and, hence, would not characterize r. An FD is minimal if no *generalization*, i. e., no subset of the FD's Lhs also describes a valid FD. More formally, an FD $X' \rightarrow A$ is a generalization of another FD $X \rightarrow A$ if $X' \subset X$. On the other hand, an FD $X'' \rightarrow A$ is a *specialization* of an FD $X \rightarrow A$ if $X \subset X''$, i. e., if its Lhs is a superset of the other FD's Lhs. Any valid, minimal FD implies that all of its specializations are valid as well, which makes them particularly interesting. As a result, given the *complete* set of minimal FDs, all other FDs can be inferred from it. For this reason, it suffices to discover and maintain only minimal, non-trivial FDs.

Despite the restriction to minimal FDs, the discovery of all such dependencies is still expensive: Liu et al. have shown that, when using nested loops for the dependency validations, the complexity of the discovery is in $O\left(n^2 \left(\frac{m}{2}\right)^2 2^m\right)$ for relations with m attributes and n records [10]. More sophisticated, index-based algorithms, such as [8] or [13], avoid the quadratic complexity of the candidate validations (n^2), but the algorithms' complexity w. r. t. the number of attributes stays exponential (2^m). This is inevitable due to the potentially exponential number of discovered FDs. The discovery problem becomes even harder when taking data changes into account. As mentioned above, all state-of-the-art FD discovery algorithms operate only on *static datasets* and one needs to re-execute them after every change of the data to maintain the FDs on *dynamic datasets*. Unfortunately, this is computationally far too expensive in practical situations: It is not unusual for those algorithms to take minutes or even hours to complete.

However, the following two observations suggest that this problem can be solved with a novel, *incremental* algorithm: First, most changes affect only a small subset of records and only these small deltas need to be investigated for causing metadata changes – the majority of records still support the same FDs as before. Second, most changes in the FDs are minor, meaning that a close specialization or generalization of a former minimal FD becomes a new minimal FD.

On the face of this opportunity, we propose DynFD, the first algorithm that maintains the complete and exact set of minimal, non-trivial FDs on dynamic data. The algorithm monitors data changes, i. e., inserts, updates, and deletes, and calculates their effect on the metadata. The changes are grouped into batches of configurable size so that a user can specify timeliness of the metadata (at the cost of performance). So rather than frequently re-computing all FDs, DynFD continuously deduces FD changes from the previous set of FDs and the batch of change operations. In detail, our contributions are the following:

(1) *FD maintenance algorithm.* We present DynFD, an algorithm that incorporates data changes, i. e., inserts, updates, and deletes, as batches into sets of minimal functional dependencies. While updates are simply handled as a combination of insert and delete, the algorithm offers specialized handling strategies for inserts and deletes (Section 2).

(2) *FD maintenance data structures.* To update the dependencies efficiently, our DynFD algorithm needs to maintain several data structures, such as position list indexes, dictionary-encoded records, and FD prefix trees, over time. We explain these data structures and how they need to change w. r. t. newly inserted or deleted tuples. We also propose a novel cover inversion algorithm to deduce an FD negative cover from an FD positive cover (Section 3).

(3) *FD maintenance pruning rules and techniques.* We devise novel pruning rules and techniques for insert and delete operations that allow DynFD to optimize or even skip certain validations. For validations that cannot be skipped, we propose efficient validation methods that exploit the incremental nature of the data changes (Section 4 and Section 5).

(4) *Evaluation.* We provide an exhaustive evaluation of DynFD w. r. t. scalability and speed-up. We investigate the effectiveness of our pruning and maintenance strategies and compare DynFD to HyFD, the state-of-the-art FD discovery algorithm for static data (Section 6).

Table 1: An example relation with four initial tuples. A batch of changes inserts and deletes tuples, as indicated by the "–" and "+" signs, respectively.

ID	firstname	lastname	zip	city
1	Max	Jones	14482	Potsdam
2	Max	Miller	14482	Potsdam
– 3	Max	Jones	10115	Berlin
4	Anna	Scott	13591	Berlin
+ 5	Marie	Scott	14467	Potsdam
+ 6	Marie	Gray	14469	Potsdam

2 OVERVIEW OF DYNFD

Before we discuss implementation details, we give an overview of our proposed algorithm DynFD. This algorithm operates on a single relation, such as the one exemplified in Table 1, which may or may not contain an initial set of tuples (here: tuples 1–4). The relation is then subject to a series of batches of changes. Each batch inserts and/or deletes tuples from the relation. For instance, Table 1 shows a batch that removes tuple 3 and inserts tuples 5 and 6. Note that tuple updates can be expressed by a delete and an insert operation. Also, note that the size of the batches is at the discretion of users and their use cases and allows for a trade-off between granularity and performance of the FD maintenance process.

As explained in Section 1, each batch *can* change the set of minimal FDs in a relation. For instance, while the FD $z \rightarrow c$ continues to be a minimal FD in Table 1 before and after the batch has been applied, $f \rightarrow c$ becomes a new minimal FD and $fc \rightarrow z$ ceases to be a (minimal) FD.

To discover these changes, DynFD comprises three principal components as can be seen in Figure 1: (i) the *data structures*, which are position list indexes and dictionary-encoded records, concisely model all relevant features of the relation to determine the currently valid FDs; (ii) the *positive cover* indexes all minimal FDs and allows to reason on the effect of insert operations; and (iii) the *negative cover* with all maximal non-FDs allows to process delete operations analogously. If the profiled relation contains initial tuples, we employ the static algorithm HyFD [13] to bootstrap the data structures and the positive cover. The negative cover can then be derived from the positive cover via a *cover inversion* as we describe in Section 3.2.

Having initialized all necessary data structures, DynFD begins to monitor changes of the profiled relation. These changes arrive as a stream that is first transformed and then processed in batches, i. e., non-overlapping groups of insert, update, and delete operations. The batches can be, e. g., equally sized groups of change operations or, alternatively, all operations from within a tumbling time window. Each batch is processed according to the following observation: According to Definition 1.1, insert operations can introduce violations to existing FDs, but never remove them. As a result, those FDs become invalid. We can efficiently retrace these changes by operating on the positive cover, i. e., on the existing minimal FDs. Delete operations constitute the opposite case: Exiting violations may be removed, thereby introducing new FDs – or in other words, existing non-FDs may become valid. Here, the negative cover is more appropriate to efficiently reason on delete operations. For this reason, we handle insert and delete operations separately, yet with the same basic principles.

Figure 1: Processing one batch of insert and delete operations with DynFD.

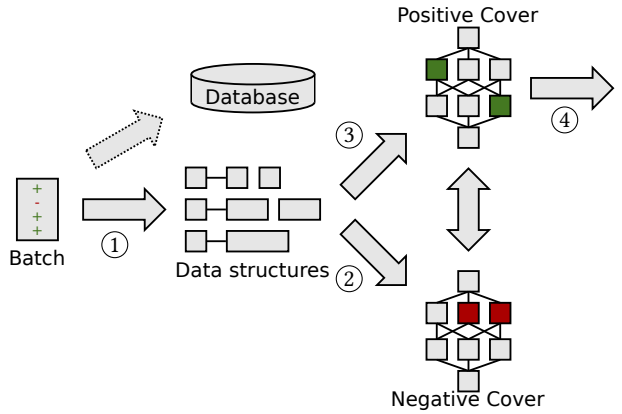


Figure 1 depicts the processing pipeline in more detail. In Step (1) of this process, DynFD efficiently updates its data structures according to the changes in the batch (see Section 3.1). In doing so, DynFD does not need to perform potentially expensive read operations on the database. Not accessing the database is particularly important, also because reads would lead to race conditions with the changes applied by the database itself, i. e., a change that is being processed by DynFD might not have been (fully) applied by the database, yet, or the database may have already applied a subsequent change that DynFD has not yet seen. In Step (2), DynFD processes all deletes in the batch by checking whether they resolve any maximal non-FD in the negative cover (see Section 5); if a non-FD becomes a valid FD, this change is also propagated to the positive cover. Then, in Step (3), the algorithm processes all inserts by checking whether they introduce a violation to any minimal FD in the positive cover (see Section 4); in case they do, the positive cover is updated and the changes are propagated to the negative cover. Hence, due to the change propagation, Steps (2) and (3) may both affect both covers. In Step (4), DynFD finally signals all changed FDs to the user and is then ready to process the next batch.

The attentive reader may have noticed that we choose to process deletes before inserts, although the other way around is also possible. The decision on which type of operation to process first is particularly important for the special but common case of tuple updates, which we split into an insert and a delete. By processing the delete first, we avoid operating on an intermediate relation that contains both the old and the new version of the updated tuple. Such an almost duplicate tuple would violate many dependencies, in particular key dependencies, for the time of its existence. Hence, many FDs would change only to change back when the (almost) duplicate is removed again. So in short, processing deletes first significantly reduces the number of temporarily changing FDs.

3 DATA STRUCTURES

Rather than recalculating the FDs of a relation after each batch of changes, DynFD creates and maintains several data structures from which to derive the FDs. For that matter, we discern two types of data structures, namely those that represent the relation in a compact format that is suitable to efficiently check the validity of FD candidates; and the positive and negative cover, which we use to evolve the set of minimal FDs and maximal non-FDs, respectively. In the following, we briefly describe those data structures and how to update them in incremental scenarios.

3.1 Representing relations compactly

When validating FD candidates against a relation, the actual values within that relation are irrelevant. Instead, we merely need to know, which tuple pairs have identical values for which attributes. Therefore, it is sufficient for DynFD to represent the relation as *compressed records* [13]. Consider the example in Table 2, which represents the initial state of Table 1, i. e., before the change. Essentially, the compressed records replace all values of the original relation with a number that uniquely identifies that value within its column. For instance, the city name "Potsdam" is replaced by the value 0 and the city name "Berlin" is replaced by the value 1. This replacement not only has a small memory footprint but also allows for more efficient equality comparisons. This scheme is further optimized by replacing unique values with the value "−1". If DynFD encounters a "−1", it can skip all comparisons, as by definition all other tuples must have distinct values for the affected column.

Table 2: The dictionary-encoded example of Table 1.

ID	f	l	z	c
1	0	0	0	0
2	0	−1	0	0
3	0	0	−1	1
4	−1	−1	−1	1

While compressed records are well-suited to compare individual tuple pairs to quickly detect FD violations and rule out some FD candidates, they are not suitable to validate FD candidates as a whole. Therefore, DynFD complements the compressed records with *position list indexes* (Plis) [13], also known as stripped partitions [8]. In few words, a Pli lists the *clusters* of tuple IDs that have the same values for a certain attribute. For our example data from Table 1, we therefore obtain the Plis $\pi_f = \{\{1,2,3\},\{4\}\}$ for firstname, $\pi_l = \{\{1,3\},\{2\},\{4\}\}$ for lastname, $\pi_z = \{\{1,2\},\{3\},\{4\}\}$ for zip, and $\pi_c = \{\{1,2\},\{3,4\}\}$ for city. The Pli π_X for a set of attributes X can be computed via Pli intersection, i. e., by intersecting all pairs of overlapping clusters in the Plis π_Y and π_Z with $Y \cup Z = X$. Hence, $\pi_X = \pi_Y \cap \pi_Z$ and, for example, $\pi_{fc} = \pi_f \cap \pi_c = \{\{1,2\},\{3\},\{4\}\}$. A functional dependency $X \rightarrow A$ holds iff $\pi_X \cap \pi_A = \pi_X$, i. e., π_A does not split any cluster in π_X so that all records with equal values in X have also equal values in A [8]. Calculating $\pi_X \cap \pi_A$ can be done efficiently by using the compressed records: The (well-known) FD validation algorithm uses the Pli for some attribute $A \in X$ as an index to sets of tuples in the compressed records that are, then, grouped by same X values and checked against their A values. For more details and optimizations of this validation algorithm, we refer the interested reader to [13].

Compressed records and Plis synergize well in validating FD candidates, which has already been shown for static FD discovery. In a static setup, however, both data structures identify each record by its row number, i. e., position in the relational table. These row numbers, change in the dynamic setting, because the table grows and shrinks. For this reason, we assign a continuous number as a surrogate key to each record to identify it. The main challenge for using compressed records and Plis in the dynamic case is that we need to update these data structures with every batch of changes. For this, we propose the following:

Insert. A newly inserted record adds an entry to both data structures: We first add its identifier to all Plis. For every attribute, we read the attribute's value from the new record and fetch the attribute's Pli from the list of Plis. We then need to find the cluster in the Pli that corresponds to the said value and add the record's identifier to it. To find that cluster, the algorithm needs to remember the value of each cluster. It does so by storing and maintaining an additional *inverted index* on top of the Plis, i. e., the inverted index points each value to the Pli cluster in which it occurs. Following this mapping, it is easy to find the clusters were DynFD needs to add an identifier; if some value has no cluster in a Pli yet, it creates a new cluster. Given the cluster numbers of all attributes for the new record, updating the dictionary-encoded records is done by simply appending the array of these cluster numbers to the list of dictionary-encoded records.

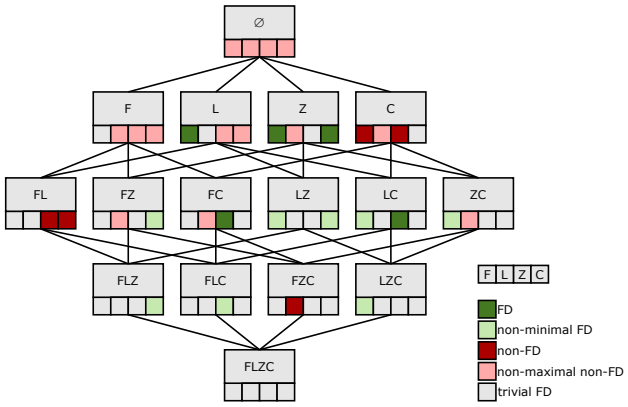
Delete. To delete a record from the data structures, we follow a similar, quick look-up strategy: For every attribute, DynFD first retrieves the Pli cluster that corresponds to the attribute's value in the deleted record. Then, it removes the identifier of the deleted record from these clusters; if a cluster becomes empty, the algorithm deletes the cluster from the Pli entirely. After the Pli update, DynFD also removes the record's compressed representation from the list of dictionary-encoded records. To find the record, we use another additional index, the *hash index*, that points the identifiers to their dictionary-encoded records. Alternatively, one could also find the record via binary search on the list of dictionary-encoded records, but the hash index has an infinitesimally small memory footprint in comparison to the other data structures and offers better performance than binary search. Once the compressed record is determined, DynFD deletes it from the list and the hash index.

3.2 Organizing functional dependencies

Let us now describe how DynFD organizes its discovered FDs and non-FDs. The FD search space is usually modeled as a *powerset lattice*, which is a graph representation of all possible attribute combinations. Due to the partial order of the power set, every two elements have a unique supremum and a unique infimum so that the graph can connect each node $X \subseteq R$ to its direct subsets $X \setminus A$ and direct supersets $X \cup \{B\}$ (with $A \in X$, $B \in R \setminus X$). Every node in the lattice represents one Lhs attribute combination for a set of FD candidates; each such FD candidate is defined by its Rhs attribute, which can bee seen as annotations at every node. In this way, all $2^m \cdot m$ possible FDs are covered by the lattice. During FD discovery, we classify each annotation and, hence, the respective Lhs $\rightarrow$ Rhs candidate as either valid or invalid FD.

Figure 2 depicts the lattice of FD candidates for the initial state of our example relation from Table 1 (tuples 1 to 4). The five minimal FDs $l \rightarrow f$, $z \rightarrow f$, $z \rightarrow c$, $fc \rightarrow z$, and $lc \rightarrow z$ have been discovered with a static profiling algorithm and we can infer all non-minimal and invalid FDs from them. To show how valid and invalid FDs are located in this search space visualization, all annotations have been color coded: Green cells represent valid FDs whereas red cells represent invalid FDs or short *non-FDs*. Stronger colors denote minimality for FDs and maximality for non-FDs. Note that a non-FD is maximal if no specialization of it is also a non-FD. Trivial FDs are shown in grey, because they are not of interest. So for example, we find the valid, minimal FD $fc \rightarrow z$ as the dark green annotation for Rhs attribute Z in Lhs node *FC*.

Figure 2: FD lattice for the data shown in Table 1.



The complete set of minimal FDs is called the *positive cover*, because all non-minimal FDs can be derived from it. The complete set of maximal non-FDs, in contrast, is called *negative cover*, because it defines all existing non-FDs.

DynFD stores both the negative and the positive cover as *FD prefix trees*, which are technically prefix trees with annotations: Each node in the tree represents a Lhs attribute, any path starting from the root node represents a Lhs, and annotations on the nodes indicate valid Rhs attributes for the respective paths [6]. FD prefix trees are not only a compact data structure for storing FDs, they also offer efficient look-up functions for FD generalizations and specializations – functions that are called frequently by DynFD.

The positive cover, the Plis, and the dictionary compressed records represent the initial input for DynFD. By running the static FD discovery algorithm HyFD first, we can simply obtain all three data structures directly from that algorithm; otherwise, if only the set of minimal FDs is given, it is trivial to construct them in a preprocessing step. What is not trivial is the calculation of the negative cover, i. e., all maximal non-FDs, from the given FDs. The process of calculating the positive from the negative cover is known as *cover inversion* [6] or *dependency induction* [13], but its inverse, the calculation of the negative from the positive cover, has not been studied before. With Algorithm 1, we hence present the first inversion algorithm for this step.

We start with an empty FD prefix tree *nonFds* (line 1). For every attribute A of a relation R, the algorithm then adds the most specific non-FD, which states that all other attributes do not functionally determine A (lines 2-4). Initialized in this way, the negative cover invalidates all possible FDs and basically states that there are no FDs in the data. This initialization is probably not true, but serves as a starting point for successive refinement. Hence, the algorithm then checks for every valid FD whether it covers some non-FD by looking up all specializations of that FD in the negative cover (lines 5-6). This look-up is implemented as a simple depth-first search in the FD prefix tree. If a non-FD has been found to be a specialization of a valid FD, it must in fact be valid. For this reason, Algorithm 1 removes it from the negative cover (line 8). Then, we need to check all direct generalizations of the removed non-FD for being maximal non-FDs. The inversion algorithm creates each of these generalizations by removing one Lhs attribute from the current non-FD's Lhs (lines 9-11). If such a created non-FD is maximal, i. e., if it has no specialization in the cover, it is added to the negative cover (lines 12-13); otherwise, if

Algorithm 1: Cover inversion

Data: relation R, positive cover *fds*
Result: negative cover *nonFds*

```
1  nonFds ← ∅;
2  for A ∈ R do
3      initialLhs ← R \ {A};
4      nonFds ← nonFds ∪ {initialLhs → A};
5  for fd ∈ fds do
6      violated ← nonFds.getSpecializations(fd);
7      for nonFd ∈ violated do
8          nonFds ← nonFds \ {nonFd};
9          for l ∈ fd.getLhs() do
10             newLhs ← nonFd.getLhs() \ {l};
11             gen ← (newLhs → nonFd.getRhs());
12             if ¬nonFds.containsSpecialization(gen) then
13                 nonFds ← nonFds ∪ {gen};
14 return nonFds
```

it is not maximal, it is simply discarded. Repeating this for every valid, minimal FD creates the negative cover.

Applied to the example data shown in Table 1, we start with the assumption that $flz \rightarrow c$, $flc \rightarrow z$, $fzc \rightarrow l$, and $lzc \rightarrow f$ are maximal non-FDs. Because the non-FD $lzc \rightarrow f$ is a specialization of the minimal FD $l \rightarrow f$, it needs to be removed as a non-FD. We then generalize it to $zc \rightarrow f$ as a new maximal non-FD candidate. This non-FD is a specialization of the minimal FD $z \rightarrow f$ (that we check next) so we generalize it once more, to $c \rightarrow f$. After also applying the remaining three minimal FDs to the negative cover, the inversion algorithms yields the final maximal non-FDs $fzc \rightarrow l$, $fl \rightarrow z$, $fl \rightarrow c$, $c \rightarrow f$, and $c \rightarrow z$.

4 HANDLING INSERTS

Having discussed DynFD's principal workflow along with its basic data structures, we now present a mechanism to maintain FDs in the light of insert operations (delete operations are discussed in the following section). Inserts can only render previously valid FDs invalid. Because the minimal FDs in the positive cover imply all valid FDs, they are the starting point for our validations. To process a batch of inserts, we validate all known minimal FDs and specialize them in case they became invalid. To ensure minimality for newly created specializations, we validate the minimal FDs from most general to most specific. In this way, we can check for each new specialization, if there is a valid generalization in the positive cover; the specialization would then not be minimal, because its generalization has already been determined to be true (minimality pruning). We discuss this main validation process, which is basically a *lattice traversal* type FD discovery approach, in Section 4.1.

To identify invalid FDs more quickly, we add an optimization to the validation process that, if the process becomes inefficient, progressively searches for FD violations. If a certain amount of FDs has been found invalid, the lattice traversal starts creating and checking many new candidates – most of which are usually invalid. For this situation, related work proposed *dependency induction* and sampling techniques that find most such violations faster than validating all candidates individually [13]. In Section 4.3, we propose an adapted version of these techniques for our dynamic setup.

Algorithm 2: Lattice-based FD validation

Data: relational schema R, positive cover *fds*, negative cover *nonFds*
Result: updated *fds*, *nonFds*

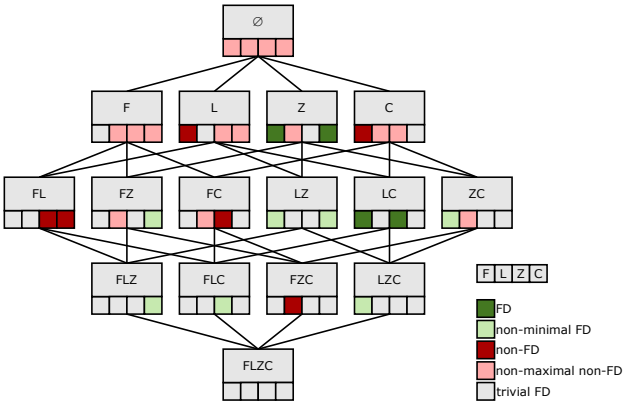
```
1  for level ∈ 0...|R| do
2      invalidFds ← ∅;
3      for fd ∈ fds.getLevel(level) do
4          if ¬isValid(fd) then
5              invalidFds ← invalidFds ∪ {fd};
6      for nonFd ∈ invalidFds do
7          fds ← fds \ {nonFd};
8          nonFds.removeGeneralizations(nonFd);
9          nonFds ← nonFds ∪ {nonFd};
10         rhs ← nonFd.getRhs();
11         for r ∈ R \ (nonFd.getLhs() ∪ {rhs}) do
12             newLhs ← nonFd.getLhs() ∪ {r};
13             spec ← ⟨newLhs → rhs⟩;
14             if ¬fds.containsGeneralization(spec) then
15                 fds ← fds ∪ {spec};
16     if |invalidFds| / |fds.getLevel(level)| > 0.1 then
17         progressiveViolationSearch(fds, nonFds)
18 return fds, nonFds
```

Note that whenever either of the two validation strategies, i. e., the lattice traversal or the dependency induction, discovers a non-FD, this non-FD must also be added to the negative cover. The process for updating the negative cover with a new non-FD covers two simple steps: First, remove all generalizations of the new non-FD from the cover (they are not maximal any more); then, add the new non-FD to the negative cover (without a check for specializations, because the non-FD used to be a valid FD before and is, therefore, inevitably maximal).

4.1 Incremental FD validation process

Algorithm 2 shows the lattice traversal-based FD validation algorithm that is executed for every batch in the dynamic setting. We start with the most general FDs in the positive cover and proceed to ever larger FDs (line 1). On each level of the lattice, the algorithm validates all minimal FDs and stores the invalid ones (lines 2-5). The validation function *isValid()* implements an optimized version of the Pli intersection technique that we touched on in Section 3.1; we discuss the optimization in Section 4.2. Iterating over all found non-FDs (line 6), the algorithm first removes each non-FD from the positive cover (line 7). As stated before, these non-FDs must also be maximal and are, therefore, added to the negative cover (lines 8-9). Afterwards, Algorithm 2 generates and adds all specializations of the current non-FD to the positive cover that are minimal w. r. t. the existing FDs (lines 10-15): To generate the specializations, we first add each attribute that is not already part of the Lhs or Rhs to the new Lhs (lines 11-13); each specialization is checked for generalizations in the positive cover before it is finally added to the cover (lines 14-15). On the next level, Algorithm 2 automatically validates these specializations. However, before moving to the next level, we check which fraction of validations in the current level has led to non-FDs (line 16). If this fraction is greater than 10% (see [13] for why this is a good threshold), then we consider the lattice traversal to be

Figure 3: Lattice after handling inserts in Table 1.



inefficient and start a progressive search for violations (line 17). When the search returns, it might yield updated versions of the positive and negative cover. Algorithm 2 then proceeds to the next lattice level until all minimal FDs have been checked.

Applying this to our example, we have the initial candidates $z \rightarrow c$, $z \rightarrow f$, $l \rightarrow f$, $lc \rightarrow z$, and $fc \rightarrow z$. They are represented by the dark green cells in Figure 2. We start at the top of the lattice and work our way to the bottom. When validating the most general FDs, we find that $l \rightarrow f$ is not valid anymore and, hence, a candidate for a maximal non-FD. The only new candidate is $lc \rightarrow f$, since $lz \rightarrow f$ is not minimal. Validating the now most general candidates shows that $fc \rightarrow z$ is also invalid. There are no new candidates to be added. It also follows that $c \rightarrow z$ is no maximal non-FD anymore and needs to be removed from the negative cover. Because no candidates are left, we are done and found the minimal FDs holding after inserting the new records. The corresponding lattice is shown in Figure 3.

4.2 Cluster pruning

To validate the minimal FDs in the positive cover, we build on the Pli-based validation algorithm presented in [13]: This algorithm uses the single-column Plis and dictionary-encoded records to dynamically calculate the Pli intersection of all Lhs attributes; at the same time, the algorithm checks the resulting clusters against the Rhs attribute clusters. If a check fails, i. e., if it reveals an FD violation, the algorithm terminates the validation process early. Furthermore, it performs this check for all FD candidates with the same Lhs simultaneously.

Our DynFD algorithm enhances this validation strategy: Instead of validating the FD against the entire dataset, we validate it against only the newly added and a few related records. Recall from Definition 1.1 that an FD is invalidated by a pair of records with equal values in the Lhs attributes but different values in the Rhs attribute. Because for inserts we validate only previously valid FDs, all pairs of *old* records still satisfy the FD. Only pairs of records containing at least one new record might introduce violations. Thus, the validation step for inserts needs to check such pairs only.

We integrate this optimization into our validation algorithm as follows: Given a fixed ordering of attributes by their respective Pli sizes, the validation starts by iterating the clusters of the Pli of the first Lhs attribute. For each cluster, the algorithm dynamically calculates the intersection with all other Lhs clusters to check the result against the Rhs clusters. At this point, which is, before calculating the intersections, DynFD first checks if the current cluster of the first Lhs attribute contains an identifier of a new record. This check is very simple, because the identifier numbers are assigned monotonically increasing (see Section 3.1) and the identifiers in each cluster are sorted. Hence, DynFD simply checks whether the last entry in the cluster is less than the identifier of the first insert-record in the current batch: If so, the cluster can be ignored; otherwise, the cluster contains at least one new record and needs to be checked via dynamically intersecting the Lhs attributes and probing the result against the Rhs attribute clusters.

As a result, DynFD's validation function *isValid()* (Algorithm 2 line 4) checks only the delta of the current batch for changing an FD and not the entire dataset. This optimization significantly improves the efficiency of the validation step, as many unnecessary comparisons are saved.

4.3 Violation search

If the lattice traversal becomes inefficient (Algorithm 2 line 16-17), DynFD switches to a strategy that we call *violation search*. This strategy is a best effort approach to find violations for formerly valid FDs via comparing records: Given two records r_i and r_j with their dictionary-encoded signature, we can easily compute the set of attributes X, in which r_i and r_j hold same values, and the set of attributes Y, in which r_i and r_j hold different values. It follows that $X \rightarrow Y$ are all non-FDs.

Any newly inserted record can cause violations only with those partner records that have at least one value with the inserted record in common; records that do not share any value with the inserted record need not be considered. With DynFD's Plis, we can easily retrieve all those partner records by simply collecting all Pli clusters of the inserted record. Comparing the inserted record to all records in these clusters would, in fact, reveal all new violations, but the comparison costs are quadratic in the number of records, which is usually too expensive. For this reason, DynFD compares an inserted or changed record only to a small, promising subset of partner records.

Related work has shown that record pairs with possibly many overlapping values are promising candidates for finding new violations [13]. A sorting approach was demonstrated that moves pairs with high overlap closer together so that near neighborhoods become promising candidates. These neighborhoods are then progressively explored by moving ever larger windows over the sortings. If the violation search becomes inefficient, which is when less than 10% of the comparisons reveal new violations, the search ends. DynFD implements the exact same progressive search, but it compares only those record pairs that include at least one inserted (or updated) record.

For every discovered non-FD, DynFD needs to update both the positive and the negative cover. Algorithm 3 shows the necessary steps: It first updates the positive cover (lines 1-9) and then the negative cover (lines 10-13). To update the positive cover, the algorithm collects all invalidated FDs and removes them from the cover (lines 1-3). For each of these invalidated FDs, it also generates all direct specializations adding the minimal ones to the positive cover (lines 4-9). To update the negative cover, Algorithm 3 first checks if it contains a specialization (line 10); only if there is no specialization, the current non-FD is maximal and we add it to the negative cover (lines 11-12).

Algorithm 3: Dependency induction from a non-FD

Data: relational schema R, *nonFd*, positive cover *fds*, negative cover *nonFds*
Result: updated *fds*, *nonFds*

```
1  invalid ← fds.getGeneralizations(nonFd);
2  for fd ∈ invalid do
3      fds ← fds \ {fd};
4      rhs ← fd.getRhs();
5      for r ∈ R \ (nonFd.getLhs() ∪ {rhs}) do
6          newLhs ← fd.getLhs() ∪ {r};
7          spec ← (newLhs → rhs);
8          if ¬fds.containsGeneralization(spec) then
9              fds ← fds ∪ {spec};
10 if ¬nonFds.containsSpecialization(nonFd) then
11     nonFds.removeGeneralizations(nonFd);
12     nonFds ← nonFds ∪ {nonFd};
13 return fds, nonFds
```

Algorithm 4: Lattice-based non-FD validation

Data: relational schema R, positive cover *fds*, negative cover *nonFds*
Result: updated *fds*, *nonFds*

```
1  for level ∈ |R|...0 do
2      validFds ← ∅;
3      for nonFd ∈ nonFds.getLevel(level) do
4          if needsValidation(nonFd) ∧ isValid(nonFd) then
5              validFds ← validFds ∪ {nonFd};
6      for fd ∈ validFds do
7          nonFds ← nonFds \ {fd};
8          fds.removeSpecializations(fd);
9          fds ← fds ∪ {fd};
10         for r ∈ fd.getLhs() do
11             newLhs ← fd.getLhs() \ {r};
12             gen ← (newLhs → fd.getRhs());
13             if ¬nonFds.containsSpecialization(gen) then
14                 nonFds ← nonFds ∪ {gen};
15     if |validFds| / |nonFds.getLevel(level)| > 0.1 then
16         depthFirstSearch(validFds, fds, nonFds)
17 return fds, nonFds
```

5 HANDLING DELETES

To handle deletes, we propose a lattice traversal approach that validates the non-FDs in the negative cover level-wise starting with the most specific non-FDs and successively proceeding to more general ones. Because the maximal non-FDs imply all other non-FDs, the validation algorithm checks and, if necessary, generalizes only maximal non-FDs. Due to the level-wise iteration of the lattice, we can test any newly derived generalization for specializations in the negative cover to ensure that only maximal non-FDs are added back into the negative cover (maximality pruning). We describe this main validation process of the negative cover in more detail in Section 5.1 and its optimizations in Section 5.2 and Section 5.3.

5.1 Incremental non-FD validation process

Algorithm 4 shows the lattice traversal-based *non-FD* validation algorithm. This algorithm basically inverts the lattice traversal algorithm for inserts (see Algorithm 2): It operates on the negative cover instead of the positive cover (line 3), it traverses the cover from the most special non-FDs to the most general non-FDs instead of from most general to most special FDs (line 1), it transfers updates to the positive cover instead of to the negative cover (lines 8-9), and it generalizes de-facto-valid non-FDs instead specializing de-facto-invalid FDs (lines 10-14).

The validation function *isValid()* (line 4) is the same validation function that we already introduced for the validation of FDs in the insert scenario, but we now expect the outcomes to be mostly non-FDs. For non-FDs, DynFD adds an additional pruning technique to the validation that we explain in Section 5.2.

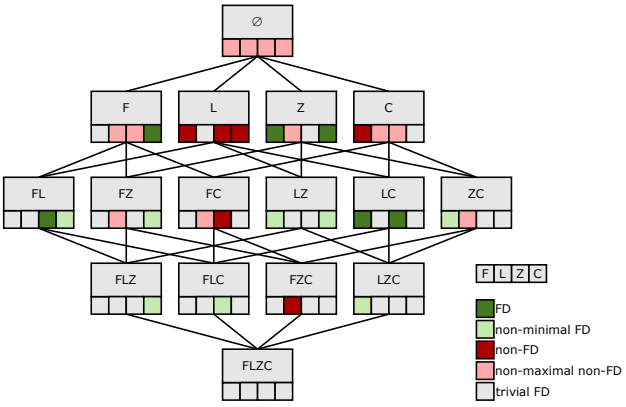
One major difference to the insert scenario, though, is that deleted records *resolve* violations and do not *introduce* them. For this reason, the progressive violation search, which compares promising record pairs in the quest for new non-FDs, makes no sense due to the lack of new non-FDs. Instead, we propose optimistic *depth-first searches* if the number of valid FDs exceeds 10% of all validated non-FDs in a current level (lines 15-16). We explain these optimistic depth-first searches in Section 5.3.

Applying the algorithm to our example, we start with the initial candidates for maximal non-FDs $fzc \rightarrow l$, $fl \rightarrow z$, $fl \rightarrow c$, $fc \rightarrow z$, $l \rightarrow f$, and $c \rightarrow f$. They are the dark red cells in Figure 3 and we traverse the lattice from bottom to top. Starting with the most specific candidate, nothing changes. In the next step, it turns out that both $fl \rightarrow z$ and $fl \rightarrow c$ become valid. Thus, they are both candidates for new minimal FDs. Furthermore, we need to add the generalizations $f \rightarrow c$, $l \rightarrow z$, and $l \rightarrow c$ to the negative cover. $f \rightarrow z$ is not maximal and therefore not a new candidate. Validating the remaining five candidates shows that also $f \rightarrow c$ is valid and thus a candidate for a new minimal FD. $fl \rightarrow c$ is not a minimal FD anymore. There are no new candidates for maximal non-FDs and we end up with six minimal FDs. The corresponding lattice is shown in Figure 4.

5.2 Validation pruning

Most FD candidates that we validate in the delete scenario are non-FDs and the purpose of validation is to confirm that there is still at least one violation to each candidate. Although the validation algorithm terminates as soon as it finds the first violation to a candidate, in many cases it still checks a lot of matching

Figure 4: Lattice after validating non-FDs



Algorithm 5: Depth first search for FDs

```
Data: fd, positive cover fds, negative cover nonFds
Result: updated fds, nonFds
1 for r ∈ fd.getLhs() do
2     newLhs ← fd.getLhs() \ {r};
3     newFd ← (newLhs → rhs);
4     if fds.containsGeneralization(newFD) ∨
        isValid(newFd) then
5         depthFirst(newFd, fds, nonFds);
6 deduceNonFds(fd, fds, nonFds);
7 return fds, nonFds
```

Algorithm 6: Dependency induction from a FD

```
Data: fd, positive cover fds, negative cover nonFds
Result: updated fds, nonFds
1  valid ← nonFds.getSpecializations(fd);
2  for nonFd ∈ valid do
3      nonFds ← nonFds \ {nonFd};
4      rhs ← nonFd.getRhs();
5      for r ∈ fd.getLhs() do
6          newLhs ← nonFd.getLhs() \ {r};
7          gen ← (newLhs → rhs);
8          if ¬nonFds.containsSpecialization(gen) then
9              nonFds ← nonFds ∪ {gen};
10 if ¬fds.containsGeneralization(fd) then
11     fds.removeSpecializations(fd);
12     fds ← fds ∪ {fd};
13 return fds, nonFds
```

value combinations, which is expensive. To avoid many of these checks, DynFD stores a violating record pair, which is a pair of two identifiers whose records contradict the FD, as a *surrogate violation* for every maximal non-FD in the negative cover. As long as these two records exist in the data, the algorithm does not need to check the corresponding non-FD.

So what DynFD does is the following: Whenever the algorithm creates a non-FD, it also attaches the record pair that made the FD invalid to the respective non-FD lattice node. When this lattice node needs to be validated (see Algorithm 4 line 4), the algorithm first calls the function *needsValidation()* to check whether one of its two attached records was deleted in the current batch. In case both records are still present, which is usually true, no validation is needed; otherwise, the algorithm has to run the validation to, depending on the result, either attach a new violating record pair or remove the non-FD.

In order to consistently attach violating record pairs to all non-FDs in the negative cover throughout the dynamic discovery process, we need to consider two procedures in DynFD that identify new non-FDs: the candidate validation (*isValid()* function) and the sampling (Section 4.3). Both procedures know a violating record pair whenever an FD candidate is invalidated so that they can simply attach this record pair to a new non-FD. Conversely, if records are deleted, they need to be consistently removed from the non-FDs. For this purpose, we index all non-FD annotations (*recordID* → *nonFD*) and, for every batch of deletes, remove from the negative cover all record identifiers (and their respective partner record identifiers) if a batch deletes them. The initial non-FD annotations in the negative cover are calculated on the fly with the first batch, because whenever a maximal non-FD lacks a violation annotation, Algorithm 4 validates it anyway.

5.3 Depth-first searches

If a prior non-FD becomes valid, we successively check all its generalizations. These checks can continue for many levels in the lattice and stretch out to an exponential number of candidates (exponential in the number of attributes); this makes the validation very expensive. The new non-FDs are, however, often covered by only a few maximal non-FDs. Hence, we propose optimistic depth-first searches that target maximal non-FDs of small Lhs-arity; these non-FDs prune many candidate non-FDs from the lattice.

If more than 10% of the non-FDs in one level of the negative cover became true FDs (Algorithm 4 line 15), we start the optimistic depth-first search. This subroutine takes the *validFds*, which are the former non-FDs that have been found valid, as input. For a sample of 10% of these *seed FDs*, DynFD aggressively searches their generalizations for new maximal non-FDs. We consider only a sample of the seed FDs, because the depth-first searches are an optimistic optimization attempt and should not change the search strategy entirely – most FDs still change only a bit making breath-first search in general more effective. The 10% efficiency threshold and the 10% seed sample are hard-coded parameters that have shown to be efficient settings for most datasets. The non-FDs discovered in the depth-first searches might be new overall maximal non-FDs and are, hence, used to update both the negative cover *nonFds* and the positive cover *fds*.

The algorithm that performs an optimistic depth-first search for one seed FD is depicted in Algorithm 5. It implements a recursive depth-first traversal of the positive cover starting with the seed FD *fd*. Given a valid FD, Algorithm 5 first generates all its direct generalizations by gradually removing each attribute from the Lhs (lines 1-3). For each generalization, the algorithm then checks if it has an own generalization in the positive cover. In that case, the generalization is true and it must not be validated; otherwise, Algorithm 5 validates the generalization (line 4). For every valid generalization, we continue the depth-first search (line 5). After handling all generalizations, the current FD is used to deduce new non-FDs in the negative cover (line 6). This step is done at last, because the deduction is expensive and, if some more general FDs have already been used for deduction in some recursion, there is less to be deduced for the current FD.

The function *deduceNonFds()* updates both negative and positive cover with a new, true FD. It is basically an exact inversion of the deduction Algorithm 3 for non-FDs as we now specialize the negative cover and generalize the positive cover accordingly. Technically, we switch negative and positive cover, and generalization and specialization, which yields Algorithm 6. This algorithm starts by retrieving all non-FD specializations of the known FD from the negative cover (line 1). All these FDs are valid now. Hence, it then removes each such FD from the negative cover (lines 2-3). To generalize the new valid FDs into possibly true non-FDs, the algorithm removes each attribute of the valid FD's Lhs once (lines 5-7). If such a generalization is maximal, it is added to the negative cover (lines 8-9). After updating the negative cover, Algorithm 6 also updates the positive cover with the known FD by adding the FD to the positive cover and removing all its specializations (lines 10-12).

6 EVALUATION

We now evaluate the performance of our FD maintenance algorithm DynFD on multiple real-world datasets. The evaluation covers a detailed analysis of our proposed pruning rules and techniques and compares DynFD to the repeated execution of HyFD, the state-of-the-art FD discovery algorithm for static setups.

To evaluate DynFD, we need some datasets with a change history and a special experimental setup. That is what we discuss first. We then evaluate DynFD's batch processing times and its throughput. Afterwards, we analyze the algorithm's performance w. r. t. different batch sizes and, then, compare the batch times to HyFD's batch times. As a final set of experiments, we evaluate our four main pruning strategies: *cluster pruning*, *violation search*, *validation pruning*, and *depth-first searches*.

6.1 Datasets and experimental setup

Datasets. For our experiments, we use six real-world datasets and their change history: The *artist* relation from the MusicBrainz database [9], the *claims* relation from the airport baggage claims dataset published by Homeland Security [15], and the Wikipedia infobox relations *cpu*, *disease*, *actor*, and *single* put together by Google [7]. The six datasets and their characteristics, i. e., number of columns, rows, changes, and FDs, are listed in Table 3. The insert, delete, and update percentage columns refer to the number of changes that we have for these datasets. For each dataset, we highlight the characteristics that make the dataset interesting: The selection contains wide (*actor*) and long (*artist*) relations, ones with insert (*single*), and update (*cpu*) heavy changes, and a dataset with particularly many changes (*claims* and *disease*). Hence, it fairly represents real-world data in general.

Changes. The six datasets are given as a series of dataset dumps in different versions. Because DynFD requires the individual change operations that transformed one version into its successor version, we extracted all inserts, deletes, and updates from the change history of each dataset. The first version in each history serves as the initial dataset and the sequence of changes broken down into fixed-sized batches constitutes the dynamic input for the FD maintenance task. In practice, the size of the batches depends on the change rate, the size of the data, and use case specific currentness requirements; in our experiments, we test different batch sizes and their impact on the FD maintenance performance.

Experiments. Given the initial dataset and the batched sequence of changes, all experiments process the entire sequence of batches as fast as possible. This gives us an upper bound for the throughput performance: If the actual change rate is higher, the maintenance would apply back-pressure and throttle the database. The research question is, though, what throughput we can achieve.

Hardware. All experiments have been executed on a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB RAM. This server runs on CentOS 6.4 and uses OpenJDK 64-Bit 1.8.0_111 as Java environment.

6.2 Batch processing performance

In this first experiment, we evaluate the batch processing performance of DynFD on all datasets by fixing the batch size to 100 and measuring the processing time of each such batch. We run up to 100 batches of each change history, which corresponds to 10,000 changes per dataset; for *cpu* and *actor*, we process only 1,463 and 5,647 changes respectively, because this is their entire change history length. The results are shown in Table 4.

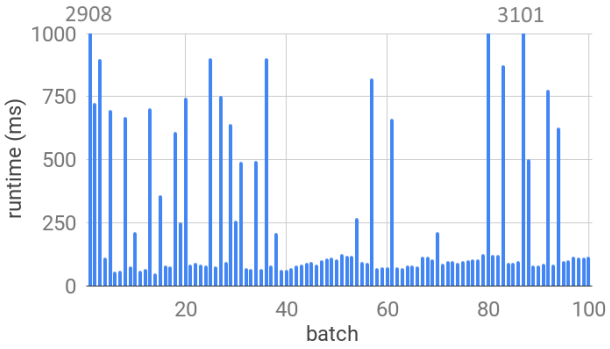


Figure 5: Runtime per batch (size 100) on *single*.

The measurements show that both the accumulated *runtime* and the *throughput* are affected by the width (#Column) and the length (#Rows) of the data: Although *single* has about three times more rows than *actor*, *actor*'s three times more columns result in almost half the throughput performance; if the number of rows is, however, significantly larger as for the *artist* dataset, the throughput also drops significantly, i. e., to only 17 changes per second. Considering the complexity of FD discovery, which is also the complexity of processing each batch, this observation is no surprise. Note that the comparably low throughput for *cpu* is due to the fact that the dataset is very short (62 rows) so that every batch (100 changes) basically rewrites the entire dataset.

The *average batch times* together with the *percentile* times show that the batch times have many outliers, i. e., runtime spikes that differ greatly from the average runtime of a batch. For most batches, the set of minimal FDs does not change much and our maintenance strategies do a good job skipping through these batches; for some batches, however, the FDs do change, sometimes even significantly. Then, DynFD needs to evolve FDs and invest some additional effort, which we tried to minimize with our pruning strategies.

Figure 5 supports this observation by plotting the individual batch times for the *single* dataset: The majority of batches is processed very quickly while some batches take orders of magnitude longer. This plot looks similar for the other datasets, namely a default batch processing time with occasional runtime spikes.

6.3 Batch size scalability

In the previous experiment, we fixed the batch size to 100 changes per batch. To see the impact of the batch size, we now scale it from 10 to 1,000 changes per batch. Figure 6 shows the average runtime of DynFD per batch w. r. t. the different batch sizes. The average in this experiment is calculated over the first 10,000 changes per dataset and both axes of the chart are in log-scale.

Most batch processing costs, such as the costs for updating the data structure and for checking whether the deletes resolved any violations, scale linearly with the size of the batch. The experimental results in Figure 6, however, show that 100 times more changes in a batch cause only about 10 times longer batch times on all datasets. This means that increasing the batch size also increases the throughput, as the processing costs per change decrease. This is because some batch activities, such as the expensive validation of the positive cover, constitute constant costs per batch regardless of its size (if no FD changes). The number of

Table 3: Characteristics of the datasets used in our evaluation.

Dataset	#Columns	#Rows (initial)	#Changes	#FDs (initial)	#FDs (final)	%Inserts	%Deletes	%Updates
cpu	15	62	1,463	209	327	4.3	0.2	**95.5**
disease	13	1,600	**361,828**	23	29	1.0	0.6	98.4
actor	**83**	3,655	5,647	347	326	64.9	0.5	34.6
single	26	12,451	12,614	193	248	**96.1**	0.0	3.9
artist	18	**1,122,887**	25,470	226	278	61.8	3.7	34.5
claims	8	1054	**202,913**	32	3	100.0	0.0	0.0

Table 4: Performance of DynFD on all datasets.

Dataset	runtime [sec]	throughput [changes/sec]	avg batch time [ms]	99th percentile [ms]	95th percentile [ms]	90th percentile [ms]
cpu	1.1	1,318.0	74.0	201.4	158.8	116.8
disease	1.5	6,844.6	14.6	62.2	26.4	19.0
actor	25.9	218.0	454.5	1,375.9	1,132.2	931.0
single	26.8	373.0	268.0	2,715.8	834.1	717.1
artist	577.1	17.3	5,771.0	15,033.1	13,979.2	12,367.9
claims	1.8	5,602.2	17.9	89.6	56.2	44.1

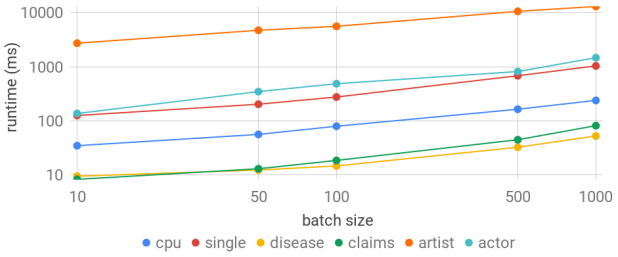


Figure 6: Average runtime for different batch sizes.

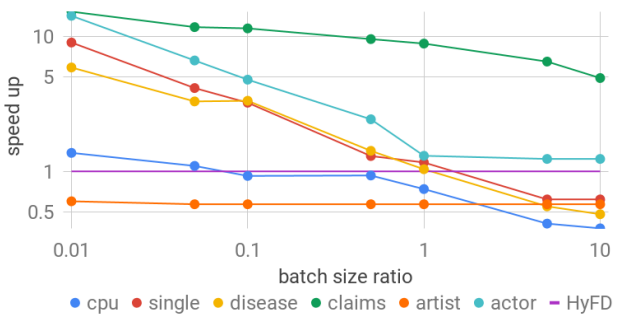


Figure 7: Speedup of DynFD relative to the runtime of HyFD.

changing FDs per batch does also not increase linearly with the batch size: Intuitively, if we look at the data less often, we probably overlook some changes, e. g., if $A \rightarrow B$ changes to $AC \rightarrow B$ and then to $ACD \rightarrow B$, we might *observe* only the change of $A \rightarrow B$ to $ACD \rightarrow B$. Hence, we observe that the average batch processing time increases sub-linearly when increasing the batch size in DynFD. It is, furthermore, remarkable that the runtime increase is similar for all datasets although they differ greatly in size and change patterns.

6.4 Competitive evaluation

Our next experiment compares the batch processing performance of DynFD to repeated executions of the static profiling algorithm HyFD. For this comparison, it is especially interesting to see for which batch sizes our dynamic algorithm exhibits superior performance compared to a static profiling approach. For this purpose, we now scale it up from small to excessively large. We also now define the batch size relative to the initial dataset size, to make the measurements comparable across differently sized datasets: We start with a batch size containing as many changes as 1% of the initial dataset length, i. e., 1% of #Rows, and increase the batch size to 1000% of initial dataset size. Then, we plot the runtime of DynFD relative to the runtime of HyFD for each dataset, e. g., a speedup of 5 indicates that DynFD was 5 times faster than HyFD with a particular batch size ratio on a particular dataset; 1 is equally fast and speedups smaller than 1 are slower than HyFD.

The results depicted in Figure 7 show that DynFD is more than an order of magnitude faster than HyFD for small and medium batch sizes. This superiority decreases as we increase the batch size – until HyFD becomes faster for most datasets. DynFD's poor performance on *artist* is due to the fact that a batch size of 1% initial dataset size already covers 11,228 changes – about half of the entire change history. The first batch in each experiment is also more expensive than later batches, because this is when DynFD collects the initial violation annotations for the negative cover. Because 10% batch size ratio already covers the entire change history for *artist*, the performance does not change after that measurement. For *cpu*, we observe the opposite effect: The dataset is so tiny that simply re-profiling it with every batch is the best option anyway.

According to the measurements for *disease*, *single*, and *actor*, the inflection point at which static profiling (with HyFD) tends to overtake dynamic profiling (with DynFD) is at about 100% batch size ratio, i. e., when a batch basically re-writes the entire dataset.

	cpu	single	disease	claims	artist	actor
-	674.4	16749.4	20015.4	26516.4	459258.0	
4.3	545.6	12790.6	20585.6	25732.2	443418.2	
5.3	460.6	15359.2	19859.0	26065.2	441666.6	10717.2
4.2	641.8	14396.0	19963.8	24431.0	408986.0	
5.2	607.0	20076.0	14298.0	28156.0	498740.0	
4.3+5.3	427.8	13445.8	19107.2	24516.0	395145.6	10960.8
4.3+5.3+4.2	448.8	11898.8	20074.4	24021.0	361354.6	8212.8
4.3+5.3+4.2+5.2	387.0	11699.0	13957.0	25018.0	364466.0	8597.0

Figure 8: Runtime with different sets of pruning strategies and a fixed batch size of 1,000.

	cpu	single	disease	claims	artist	actor
-	11107.6	15523.6	56107.0	218269.8	47712.6	
4.3	10618.4	12080.6	56588.6	218695.0	46280.0	
5.3	8779.5	14456.9	54894.2	216246.6	30812.2	17001.2
4.2	10796.0	15371.0	53761.0	179788.0	45790.0	
5.2	9752.5	20060.0	48473.0	202904.0	52094.0	
4.3+5.3	8540.7	13460.7	53510.6	180654.4	28275.2	17370.8
4.3+5.3+4.2	8369.8	10887.4	55088.4	181156.6	26608.3	13188.0
4.3+5.3+4.2+5.2	6466.0	10315.0	45130.0	158946.0	27848.0	12404.0

Figure 9: Runtime with different sets of pruning strategies and a relative batch size of 10% initial dataset size.

6.5 In-depth performance analysis

DynFD proposes an FD maintenance algorithm with four major pruning strategies. The basic algorithm performs the incremental data structure updates and the level-wise valuations of the positive and negative cover; it basically enables the dynamic evolution of FDs. The four pruning strategies, which are *cluster pruning* (see Section 4.2), *violation search* (see Section 4.3), *validation pruning* (see Section 5.2), and *depth-first searches* (see Section 5.3), aim to reduce the maintenance effort as much as possible. Let us now evaluate how effective each individual strategy is and what the performance implications are.

In this final set of experiments, we execute DynFD with different sets of pruning strategies on all datasets. Because the *violation search* is so important for the algorithm, i. e., the performance drops significantly without any form of this strategy, we let the baseline algorithm run a naive sampling that compares changed records only to their direct neighbors w. r. t. some sorting. For each combination of strategies and dataset, we measure the processing time for all batches of fixed size 1,000 (Figure 8) and relative size 10% (Figure 9), respectively.

The measurements for both batch sizes show that the composition of all pruning strategies performs best in general. It is not the optimal composition of pruning strategies for all datasets, but the performance is reliably good throughout all our datasets. With a few exceptions, every strategy tends to improve the performance a bit. One exception is the *violation search* (4.3) on the *disease* dataset, because a naive version of this strategy is already in use by the baseline; and the optimized version, which is the version that runs progressively increasing windows, does not improve upon this naive strategy. The second exception is the *validation pruning* (5.2) that performs poorly on *claims*, *single*, and *artist*: On *claims*, the batches simply do not contain any deletes so that the strategy introduces the overhead of labeling non-FDs with violations without ever needing them. On *single* and *artist*, the annotations did not effectively prevent violations, because

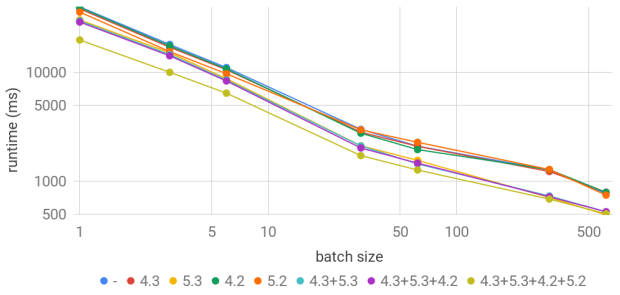


Figure 10: Runtime on *cpu* with different sets of pruning strategies and different batch sizes.

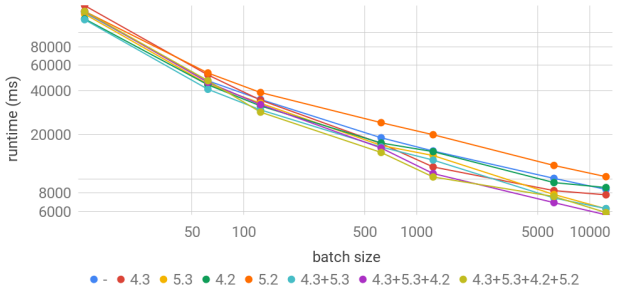


Figure 11: Runtime on *single* with different sets of pruning strategies and different batch sizes.

many of the annotated violations did vanish, i. e., the violations in these datasets are simply not stable enough to prevent enough violations that could balance the overhead of maintaining the violations.

The measurements in Figure 10 and Figure 11 present the runtimes of the different strategy compositions for different batch sizes. The lines show that the composition of *all* pruning strategies also performs reliably well, i. e., best or close to best, regardless of the batch size.

7 RELATED WORK

DynFD is the first algorithm to maintain FDs under inserts, updates, and deletes in dynamic data. Nonetheless, we identify two categories of related work, namely (i) the discovery of FDs in static data and (ii) the maintenance of metadata in dynamic data in general.

7.1 Discovering FDs in static data

Research has brought up many FD discovery algorithms for static data, which can be classified into *column-based*, *row-based*, and *hybrid* algorithms [12]. In the following, we briefly describe one popular representative of each class.

One of the earliest FD discovery algorithms is the column-based algorithm Tane that models the search space, i. e., the set of all candidate FDs, as a powerset lattice of attribute combinations [8]. It traverses this lattice in a level-wise bottom-up fashion until it arrives at the non-trivial, minimal FDs. Although, DynFD adopts the lattice-shaped search space, it traverses it both bottom-up and top-down in response to data changes. For the candidate validation, Tane proposed *stripped partitions* (also: *position list indexes*, Plis) that we also use in DynFD.

The row-based Fdep algorithm proposes a fundamentally different strategy [6]: It compares all pairs of records in the input relation to deduce the complete *negative cover*, i. e., all candidate FDs that are violated by some tuple pair. The non-trivial, minimal FDs are derived from the negative cover via cover inversion. DynFD also maintains a negative cover, but for the purpose of processing tuple deletions and not to infer the positive cover.

The hybrid algorithm HyFD combines column- and row-based techniques to avoid possibly many ineffective candidate validations and tuple comparisons [13]. By interleaving the two discovery principles and by having them exchange intermediate results, HyFD significantly outperforms all non-hybrid competitors. In DynFD, we tailor the hybrid discovery approach to work on dynamic data. Our experiments show that the proposed extensions and adjustments have a great, positive impact on the performance of the approach.

7.2 Maintaining metadata in dynamic data

Arguably, maintaining metadata in dynamic data has received much less attention than static data profiling. Therefore, this section widens its scope to maintaining *any* kind of metadata, not only FDs.

To the best of our knowledge, the only existing FD maintenance algorithm was proposed by Wang et al. in [17]. The algorithm deals only with tuple deletions and neither inserts nor updates. Similar to DynFD, the algorithm uses bottom-up and top-down approaches as well as indexes to evolve the FDs. Unlike DynFD, however, it does not maintain a negative cover of non-FDs that significantly improves and distinguishes the handling of deletes from the static case.

The Swan algorithm by Abedjan et al. is an incremental discovery algorithm for unique column combinations (UCCs), i. e., key candidates in dynamic datasets [1]. Starting from a pre-calculated set of UCCs, Swan actively applies all insertions and deletions to the current metadata set. The algorithm groups change operations into batches and uses various indexes to optimize the change calculations – two principles that are also used by DynFD. In contrast to Swan, however, DynFD operates on both a positive and a negative cover representation of the metadata, which enable additional pruning strategies.

Lastly, Shaabani and Meinel proposed an incremental algorithm to maintain inclusion dependencies (INDs) in dynamic data [16]. The algorithm uses a concept called *attribute clustering* that can be used in an incremental setup to evolve INDs w. r. t. sets of data changes. Besides the fact that this algorithm also handles both inserts and updates in batches, it has little in common with DynFD, because IND discovery is very different from FD discovery.

8 CONCLUSION

In this paper, we introduced DynFD, a novel algorithm that maintains the functional dependencies of dynamic datasets. To evolve the set of FDs with every batch of inserts, updates, and deletes, the algorithm continuously adapts its validation structures as well as a negative and a positive cover of FDs. With DynFD, we proposed a new cover inversion algorithm and four pruning strategies that stabilize the maintenance performance. Our evaluation shows that, if the batch size is small, the dynamic algorithm is more than an order of magnitude faster than repeated executions of a state-of-the-art, static profiling algorithm.

Because profiling static data is already a challenging task, profiling dynamic data is a research direction that has not been studied much thus far. With DynFD, we made a first approach to solve this task for functional dependencies. Our experiments show that this algorithm greatly improves upon using static data profiling algorithms in dynamic scenarios, but it leaves several interesting research questions open:

(1) *Devising a measure for interestingness* could serve to track only interesting and, hence, fewer dependencies; this would greatly improve the maintenance performance.

(2) *Incorporating knowledge about existing database constraints* into the maintenance process could help to prune further validations; FDs with a key constraint on their Lhs, for instance, cannot invalidate.

(3) *Exploiting the specifics of update operations*, such as the fact that most updates do not alter all attribute values but only a few, could help to devise further pruning rules.

REFERENCES

[1] Z. Abedjan, J. Quiané-Ruiz, and F. Naumann. Detecting unique column combinations on dynamic data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1036–1047, 2014.

[2] P. Bohannon, W. Fan, and F. Geerts. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 746–755, 2007.

[3] C. R. Carlson, A. K. Arora, and M. M. Carlson. The application of functional dependency theory to relational databases. *The Computer Journal*, 25(1):68–73, 1982.

[4] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[5] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[6] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

[7] Google. Distributing the Edit History of Wikipedia Infoboxes. https://research.googleblog.com/2013/05/distributing-edit-history-of-wikipedia.html, 2013. [Online; accessed 8-October-2018].

[8] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[9] Internet Archive. MusicBrainz Data Dumps. https://archive.org/details/musicbrainzdata, 2018. [Online; accessed 8-October-2018].

[10] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - a review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(2):251–264, 2012.

[11] R. J. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[12] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment (PVLDB)*, 8(10):1082–1093, 2015.

[13] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 821–833, 2016.

[14] G. N. Paulley. *Exploiting Functional Dependence in Query Optimization.* PhD thesis, University of Waterloo, 2000.

[15] H. Security. TSA Claims Data. https://www.dhs.gov/tsa-claims-data, 2018. [Online; accessed 8-October-2018].

[16] N. Shaabani and C. Meinel. Incremental discovery of inclusion dependencies. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 2:1–2:12, 2017.

[17] S.-L. Wang, W.-C. Tsou, J.-H. Lin, and T.-P. Hong. *Maintenance of Discovered Functional Dependencies: Incremental Deletion*, pages 579–588. Springer, Heidelberg, 2003.