open proceedings

DynASt: Efficient Maintenance of Agree-Sets Against Dynamic Datasets

Khalid Belhajjame
PSL, Université Paris-Dauphine, LAMSADE
Paris, France
khalid.belhajjame@dauphine.fr

ABSTRACT

Constraint discovery is a fundamental task in data profiling, which involves identifying the dependencies that are satisfied by a dataset. As published datasets are increasingly dynamic, a number of researchers have begun to investigate the problem of dependencies' discovery in dynamic datasets. Proposals this far in this area can be viewed as schema-based in the sense that they model and explore the solution space using a lattice built on the basis of the attributes (columns) of the dataset. It is recognized that proposals that belong to this class, like their static counterpart, tend to perform well for datasets with a large number of tuples but a small number of attributes. The second class of proposals that have been examined for static datasets (but not in dynamic settings) is data-driven and is known to perform well for datasets with a large number of attributes and a small number of tuples. The main bottleneck of this class of solutions is the generation of agree-sets, which involves pairwise comparison of the tuples in the dataset.

We present in this paper *DynASt* , a system for the efficient maintenance of agree-sets in dynamic datasets. We investigate the performance of *DynASt* and its scalability in terms of the number of tuples and the number of attributes of the target dataset. We also show that it outperforms existing (static and dynamic) state-of-the-art solutions for datasets with a large number of attributes.

1 INTRODUCTION

Data dependencies (data constraints) are key ingredients for a range of data management functionalities including schema normalization [11], query optimization [12], data cleaning [10] and data integration [7, 25]. Because of this, a large body of work was dedicated to the discovery of data dependencies of different expressiveness in the last two decades, ranging from functional dependencies (FDs), candidate keys, inclusion dependencies to more expressive dependencies such as denial constraints.

While identifying the dependencies that hold for static datasets has been thoroughly investigated, discovering the dependencies that hold over time for dynamic datasets is increasingly needed in practice with only a few proposals dedicated to the subject so far [8, 26, 30]. Indeed, checking the validity of dependencies over time provides in-depth information, e.g., in data exploration scenarios, to understand the complex semantic relationships between attributes and their evolution over time. Such an assessment has also practical benefits and applications. It can be used to identify dependencies that are resilient over time, thus assisting the user in determining which dependencies are reliable and which are not. This is particularly the case when the system supports approximate dependencies, which are accompanied by a confidence/error. Also, sudden changes in previously reliable dependencies can flag up data quality problems, i.e., erroneous updates. This is particularly relevant in the scenario where the dataset under analysis is continuously updated using information extraction pipelines fed by uncertain and changing data sources. Furthermore, in such contexts, dynamic dependency discovery can be used to analyse the impact of adding or removing a new data source in the information extraction pipeline. For example, if the addition of a new source invalidates previously valid and reliable dependencies, this may mean that the new source suffers from data quality problems. On the other hand, if the removal of a data source produces new data dependencies, which the domain expert had identified as invalid, it may mean that the removal of the data source in question has an impact on the representativeness (completeness) of the dataset produced by the information extraction pipeline.

Motivated by this, several researchers have begun to investigate incremental dependency discovery in dynamic datasets. For example, Wang *et al.* [30] proposed a solution to maintain FDs when tuples are deleted from the target relation. Caruccio *et al.* [8], on the other hand, have explored the problem of maintaining FDs when tuples are inserted in the target relation. The proposal by Schirmer *et al.* [26] is the most complete in the sense that it maintains FDs under both tuple deletion and insertion.

The above proposals are schema-driven: they model the search space of FDs using a lattice whose nodes represent the attribute sets of the relational schema, and whose edges specify the containment relationships between the attribute sets. Different strategies are then used to explore such a space. For example, Wang *et al.* [30] has shown how FDs can be maintained in the case of tuple suppression by traversing the lattice in an ascending or descending manner. Schirmer [26], on the other hand, limits the search to the middle part of the lattice, where non-FDs may become minimal FDs following the removal of tuples, and minimal FDs may become Non-FDs following the insertion of tuples. It is well recognized that the time complexity of schema-driven dependency discovery algorithms depends mainly on the size of the lattice, which is predicated by the number of attributes of the target relation. Because of this, such algorithms tend to perform well for relations that have a large number of tuples but with a small number of attributes (see Papenbrock *et al.* [23]).

The second class of dependency discovery algorithms, which complements schema-driven algorithms, are data-driven. This class of algorithms proceeds in two steps. In the first step, tuples in a dataset are compared in pairs to identify attributes on which they agree, which are called agree-sets. To illustrate this, consider the Employee relation illustrated in Table 1 providing information about employees, who are characterized by their first name FN, last name LN, position P and salary S, and focus on the first four tuples, which represent the initial state of the relation before deletion and insertion of tuples. To identify the FDs that are valid

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-088-2 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Table 1: **Example of a relation with four initial tuples, and a batch composed of one tuple to delete and two tuples to insert indicated by the signs "-" and "+", respectively.**

	FN	LN	P	S
t_1	John	Bond	Manager	2500
t_2	Marie	Miller	Manager	3000
t_3	Marie	Bond	Employer	2000
$-\ t_4$	Tom	Gray	Manager	3000
$+\ t_5$	John	Miller	Employee	2000
$+\ t_6$	Anna	Scott	Manager	3000

in such a relation, the four tuples are compared in pairs to identify the attributes on which they agree, e.g., the first two tuples give rise to the following agree-set $\{P\}$. This first step results in the following set of agree-sets: $\{\{P\}, \{LN\}, \{FN\}, \{P, S\}, \emptyset\}$. In the second step, FDs are derived based on such agree-sets. If we take the example of DepMiner [21], to identify the FDs that have the attribute FN in their rhs, i.e., FDs of the form $X \longrightarrow FN$, it starts by identifying maximal agree-sets that do not contain FN and are different from the empty set, i.e. $max(r, FN) = \{\{LN\}, \{P, S\}\}$. It goes on to construct the complementary of such a set, disregarding the attribute FN. More precisely, the complement of $\{LN\}$ is $\{P, S\}$ and the complement of $\{P, S\}$ is $\{LN\}$, thus giving rise to the complement set: $cmax(r, FN) = \{\{P, S\}, \{LN\}\}$. The FDs $X \longrightarrow FN$ are obtained by identifying the minimal traversals of the hypergraph formed by $cmax(r, FN)$. In the above case, there are two minimal traversals, namely $\{LN, P\}$ and $\{LN, S\}$, thus giving rise to the following FDs: $LN, P \longrightarrow FN$ and $LN, S \longrightarrow FN$.

Unlike schema-driven algorithms, data-driven algorithms perform well for relations with a large number of attributes but with a small number of tuples (see Papenbrock *et al.* [23]). The main bottleneck of this class of solutions is the generation of agree-sets. We present in this paper *DynASt*, a system for the efficient maintenance of agree-sets in dynamic datasets. To cater for the discovery of approximate dependencies in dynamic settings, *DynASt* utilizes prefix trees for the efficient storage and retrieval of computed agree-sets. We investigate the performance of *DynASt* and its scalability in terms of the number of tuples and the number of columns of the target dataset. We also show that it outperforms existing (static) state-of-the-art solution by up to three orders of magnitude, and that it outperforms existing dynamic state of the art solution, namely *DynFD*, for datasets with a large number of attributes. Accordingly, the contributions of this paper are as follows:

(1) *Data structures* that are carefully elaborated to cater for the efficient computation of agree-sets in a dynamic setting in Section 4.
(2) An algorithmic solution for incrementally computing agree-sets and incrementally maintaining the underlying data structures in Section 5.
(3) A mechanism for the indexing and efficient retrieval of agree-sets in Section 6.
(4) Evaluation exercises that assess the efficiency of *DynASt*, and compare its performance against a state-of-the-art techniques in Section 7.

In addition, we lay the foundations of our work in Section 2. We provide an overview of our solution in Section 3. We also analyze and compare existing proposals to ours and close the paper in Section 8.

2 FOUNDATIONS

Given a relation r instance of a relational schema R, a data-driven dependency discovery algorithms [13, 19, 20, 33] start by comparing in a pair-wise manner the tuples of r to identify the attributes on which they agree.

Definition 2.1 (Agree-set). Given a relation r instance of the relational schema R, and a pair of tuples $\{t_i, t_j\}$ of r. The agree-set of $\{t_i, t_j\}$, denoted by $as(t_i, t_j)$, is the set of attributes of R for which t_i and t_j have the same values.

$$as(t_i, t_j) = \{A \in R\ s.t.\ t_i[A] = t_j[A]\}$$

Consider, for example, the Employee relation illustrated in Table 1 . The tuples t_1 and t_2 of such a relation have the same value for the attribute P, i.e., $as(t_1, t_2) = \{P\}$. To store and manipulate agree-sets efficiently, we use the bitset data structure. The agree-set $as(t_i, t_j)$ is represented by a bitset whose size is the number of attributes in R. The bit at position k is set to 1 if the tuples t_i and t_j have the same value for the attribute at position k in the relational schema R, and is set to 0, otherwise. For example, $as(t_1, t_2)$ of the tuple t_1 and t_2 in Table 1 will be represented in DynASt using the the bitset 0010, since the two tuples agree only on the third attribute P.

Definition 2.2 (Relation Evidence Set). A relation evidence set is a set, the elements of which are agree-sets. Specifically, the evidence set of a relation r is defined as follows:

$$ES(r) = \{as(t_i, t_j)\ s.t.\ ((t_i, t_j)\ \in\ r \otimes r) \wedge (t_i \neq\ t_j)\}$$

$\otimes$ denotes unordered Cartesian product, which is a weaker form of the Cartesian product.

Consider again the example illustrated in Table 1. The initial Employee relation, denoted by Emp^{curr}, which contains the four initial tuples prior to deleting tuple t_4 and inserting tuples t_5 and t_6. The computation of the agree-sets of every pair in Emp^{curr} allows us to derive the following relation evidence set for Emp^{curr}: $ES(Emp^{curr}) = \{\{P\}, \{LN\}, \{FN\}, \{P, S\}, \emptyset\}$, which using the bitset representation can be rewritten as follows: $ES(Emp^{curr}) = \{0010, 0100, 1000, 0011, 0000\}$,

In the second phase, data-driven algorithms derive dependencies from agree-sets. We illustrated in the introduction how agree-sets are used by DepMiner [21] to infer functional dependencies. Agree-sets are also used by key discovery algorithms. For example, Gordian [27] uses agree-sets to identify non-keys, e.g. P is found to be a non-key given the agree-set $a(t_1, t_2) = \{P\}$. Non-keys are then used to identify maximal non-keys, which in turn are used to discover minimal keys.

Beyond exact data dependencies, agree-sets are key ingredients to the computation of approximate data dependencies [19]. Indeed, data-driven dependency discovery algorithms can be adapted to the discovery of approximate dependencies. This is achieved by associating each agree-set with a cardinality specifying the number of times it occurs within a relation r. More details on how our solution can be used in the context of approximate dependency discovery are presented in Section 6.

The most computationally expensive phase in data-driven dependency discovery algorithms, including DepMiner [21] and FastFD [33], is the first phase, which consists of computing the relation evidence set. In terms of computing the relation evidence set, the most sophisticated proposal among the above ones is DepMiner, in the sense that it avoids comparing all tuple pairs by constructing maximal sets. Maximal sets refer to sets of tuples that have the same value for at least one attribute. Thus, it avoids

comparing tuples that are known to share no attribute values. We will use this method as a baseline for our empirical evaluation, and focus in what follows on presenting the solution we designed for maintaining the relation's evidence set under tuple insertion and deletion.

3 OVERVIEW OF DYNAST

Consider a relation r^{curr} instance of R for which the evidence set has been computed, and let $b = (b_{del}, b_{ins})$ a batch of tuples that are to be deleted from and inserted into r^{curr}: $r^{next} = r^{curr} \cup b_{ins} \setminus b_{del}$. It is worth noting here that tuple update is translated into the deletion of the existing tuple and the insertion of a new tuple with the updated attribute values. Figure 1 illustrates the architecture of *DynASt* that we adopt to incrementally compute the dependencies of r^{next} given the changes in b. The first step consists of computing the agree-sets that need to be added and those that need to be removed given a batch of b. The second step updates the relation evidence set given the computed deltas in agree-sets. Finally, the data dependencies are updated given the new relation evidence set. This may entail removing some existing data dependencies and the discovery of new data dependencies.

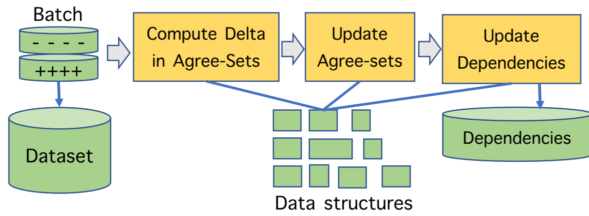

Figure 1: Processing of a single batch of tuple inserts and deletes by *DynASt*.

Let us now consider in more details the problem of computing the relation evidence set $ES(r^{next})$ given a batch b and a previously computed relation evidence set $ES(r^{curr})$. This operation involves computing deltas, that is the agree-sets that need to be added to and those that need to be removed from $ES(r^{curr})$ given the batch b. The insertion of a tuple $t \in b_{ins}$ yields agree-sets that may need to be added to $ES(r^{curr})$, if they are not in already. Specifically, the set of agree-sets that needs to be unionied with $ES(r^{curr})$ as a result of the insertion of the tuples in b_{ins} can be specified using the following set comprehension:

$$\{as(t_i, t_j)\ s.t.\ ((t_i, t_j) \in (r^{next} \otimes b_{ins})) \wedge (t_i \neq t_j)\}$$

The deletion of the tuples in b_{del} from r^{curr}, on the other hand, may mean the deletion of member agree-sets from $ES(r^{curr})$. However, tuple deletion is trickier to handle: we cannot proceed as with insertion by deleting from $ES(r^{curr})$ the agree-sets obtained using b_{del}. The reason is that a member set of a relation evidence set can be obtained by more than one tuple-pair. To illustrate this, consider the Emp^{curr} relation that contains the first four tuples illustrates in Table 1. And consider that tuple t_4 was deleted, i.e., $Emp^{next} = Emp^{curr} \setminus \{t_4\}$. We have the following agree-set $as(t_1, t_4) = \{P\}$, which is a member of the relation-evidence set of $ES(Emp^{curr})$. Notice that such a set is also member of the relation evidence set $ES(Emp^{next})$, even after the deletion of t_4. The reason is that there are two tuple pairs that yield such a set, specifically $as(t_1, t_4) = as(t_1, t_2) = \{P\}$.

To overcome the above problem, we extend the previous notion of relation evidence set so that it becomes a multiset, in which a member agree-set appears as many times as the number of tuple-pairs in the relation that gives rise to such a set.

Definition 3.1 (Relation Evidence Multiset). The relation evidence multiset of a relation r, denoted by $ES^m(r)$, is defined by the the following multiset comprehension:
$ES^m(r) = \{\{as(t_i, t_j)\ s.t.\ ((t_i, t_j) \in r \otimes r)\ \wedge\ (t_i \neq t_j)\}\}$

We use double braces $\{\{\}\}$, to distinguish multisets from sets. To know the number of times a given agree-set appears within the multiset $ES^m(r)$, we assume the existence of the function $occ(ES^m(r), es)$, which given a multiset and a member thereof, returns a positive number designating the number of times es appears in $ES^m(r)$. $occ(ES^m(r), es) = 0$ if es is not a member of $ES^m(r)$. Note that, by definition, it follows that $ES(r) = supp(ES^m(r))$, where $supp()$ is a function that given a multiset, returns the corresponding support set.

Using multisets, we can handle insertion and deletion in a similar manner. Specifically, given the set of tuples b_{del}, we compute $\Delta^-_{ES^m}(r)$, the multiset of agree-sets that needs to be subtracted (in the multiset sense) from $ES^m(r^{curr})$ using the following multiset comprehension:

$$\Delta^-_{ES^m}(r) = \{\{as(t_i, t_j)\ s.t.\ (\{t_i, t_j\} \in (r^{curr} \otimes b_{del}))\ \wedge\ (t_i \neq t_j)\}\}$$

In other words, given a tuple $t_i \in b_{del}$, all the agree-sets that are obtained by pairing t_i which each of the tuples in r^{curr} are considered. To illustrate this, consider our example where $\mathtt{b_{del}} = \{\mathtt{t_4}\}$. Given that $Emp^{curr} = \{t_1, t_2, t_3, t_4\}$, we have:
$\Delta^-_{ES^m}(Employee) = \{\{as(t_4, t_1), as(t_4, t_2), as(t_4, t_3)\}\}$. By computing the member agree-sets, we have
$\Delta^-_{ES^m}(Employee) = \{\{\{P\}, \{P, S\}, \emptyset\}\}$

Similarly, given the set of tuples b_{ins}, we compute $\Delta^+_{ES^m}(r)$, the multiset of agree-sets that needs to be unionised (in a multiset sense) with $ES^m(r^{curr})$ using the following multiset comprehension:

$$\Delta^+_{ES^m}(r) = \{\{as(t_i, t_j)\ s.t.\ (\{t_i, t_j\} \in (r^{next} \otimes b_{ins}))\ \wedge\ (t_i \neq t_j)\}\}$$

In other words, given a tuple $t_i \in b_{ins}$, all the agree-sets that are obtained by pairing t_i which each of the tuples in r^{next}, i.e. $r^{curr} \setminus b_{del} \cup b_{ins}$, are added. Note that in doing so, t_i is not paired with the tuples in b_{del}. This is because tuples' deletion are processed before tuples' insertion. To illustrate this, consider our example, where $b_{ins} = \{t_5, t_6\}$. Given that $Emp^{next} = \{t_1, t_2, t_3, t_5, t_6\}$, we have:
$\Delta^+_{ES^m}(Employee) = \{\{as(t_5, t_1), as(t_5, t_2), as(t_5, t_3),$
$as(t_5, t_6), as(t_6, t_1), as(t_6, t_2), as(t_6, t_3)\}\}$
After computing the member agree-sets, we have
$\Delta^+_{ES^m}(Employee) = \{\{\{FN\}, \{LN\}, \{P, S\}, \emptyset, \{P\}, \{P, S\}, \emptyset\}\}$
Notice that the same agree-set, e.g., {P,S}, appears as many times as the number of unordered tuple pairs that yield it. The solution we propose gives the same results whether tuple insertions are processed before deletions, or vice versa. That said, we opt to process deletions first, as this avoids computing agree-sets given tuple insertion, that will need to be removed anyway later because of tuple deletion.

We can now formulate the following property for computing $\mathtt{ES^m(r^{next})}$ in an incremental manner.

PROPERTY 1. *Consider the evidence multiset $ES^m(r^{curr})$ of a relation r^{curr}, and a batch of tuples $b = (b_{ins}, b_{del})$ that are inserted and deleted from r^{curr}: $r^{next} = r^{curr} \cup b_{ins} \setminus b_{del}$. The evidence*

Table 2: A compressed representation of the Employee relation.

	FN	LN	P	S
t_1	0	0	0	0
t_2	1	1	0	1
t_3	1	0	1	2
$-\ t_4$	2	2	0	1
$+\ t_5$	0	1	1	2
$+\ t_6$	3	3	0	1

multiset of r^{next} can be incrementally computed as follows:

$$ES^m(r^{next}) = ES^m(r^{curr}) \oplus \Delta^+_{ES^m}(r) \ominus \Delta^-_{ES^m}(r)$$

where $\oplus$ stands for multiset union, and $\ominus$ for multiset subtraction. Specifically, consider two multisets ms_1, ms_2, the union $ms_3 = ms_1 \oplus ms_2$, and the subtraction $ms_4 = ms_1 \ominus ms_2$. We have $occ(ms_3, x) = occ(ms_1, x) + occ(ms_2, x)$, and $occ(ms_4, x) = max(0, occ(ms_1, x) - occ(ms_2, x))$. The proof for this property can be found in the appendix.

Applying the above property to our example, we have: $ES^m(Emp^{next}) = ES^m(Emp^{curr}) \ominus \Delta^-_{ES^m}(Employee) \oplus \Delta^+_{ES^m}(Employee)$, where Emp^{next} is the relation obtained by deleting t_4 and inserting t_5 and t_6 to Emp^{curr}. That is:
$ES^m(Emp^{next}) = \{\{\{P\}, \{LN\}, \{P\}, \{FN\}, \{P, S\}, \emptyset\}\}$
$\oplus \{\{\{FN\}, \{LN\}, \{P, S\}, \emptyset, \{P\}, \{P, S\}, \emptyset\}\}$
$\ominus \{\{\{P\}, \{P, S\}, \emptyset\}\}$
Which yields the following relation evidence multiset:
$ES^m(Emp^{next}) = \{\{\{LN\}, \{P\}, \{FN\}, \{FN\}, \{LN\}, \{P, S\}, \emptyset, \{P\}, \{P, S\}, \emptyset\}\}$

Note that agree-sets that are empty-sets, specifying that the tuple pairs in questions disagree on all attributes, are not informative as far as dependency discovery is concerned. As such, they can be safely discarded when computing the evidence multiset and the associated deltas.

The computation of $\Delta^+_{ES^m}(r)$ and $\Delta^-_{ES^m}(r)$ can be costly. $\Delta^+_{ES^m}(r)$ involves computing the agree-set of every pair in $r^{next} \otimes b_{ins}$. Specifically, it requires $(|(r^{next}| \cdot |b_{ins}|) \cdot |R|$ attribute value-pair comparisons, where $|R|$ designates the arity of the relation. Given that *(i)* $r^{next} = r^{curr} \setminus b_{del} \cup b_{ins}$, *(ii)* $b_{del} \subseteq r^{curr}$, and that *(iii)* $r^{curr} \setminus b_{del}$ and b_{ins} are mutually disjoint, we end up with $(|r^{curr}| - |b_{del}| + |b_{ins}|) \cdot |b_{ins}| \cdot |R|$ attribute value comparison. Computing $\Delta^-_{ES^m}(r)$, on the other hand, involves $(|(r^{curr}| \cdot |b_{del}|) \cdot |R|$ attribute value-pair comparisons. We will see later how judiciously chosen indexing structures can be used to avoid such a costly operation.

4 DATA STRUCTURES

DynASt uses a compressed representation for relations, where each attribute value is mapped to a number that uniquely identifies that value in the attribute column. Table 2 shows a compressed representation of the Employee relation in Table 1. A similar representation has already been used in HyFD [24] and DynFD [26]. In addition to consuming less memory, this allows for a more efficient equality comparison of attribute values.

The most expensive operation in data-driven FD discovery algorithms is the computation of the relation evidence multiset. We present in this section new data structures that underpin *DynASt* to reduce the cost of this operation in a dynamic setting where tuples are inserted and deleted.

Table 3: Attribute-value evidence sets for the relation in Table 1

EV(FN,"John")	EV(LN,"Bond")	EV(P,"Manager")	EV(S,2500)
$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\1\\0\end{pmatrix}$	$\begin{pmatrix}1\\1\\0\\1\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$
EV(FN,"Marie")	EV(LN,"Miller")	EV(P,"Employee")	EV(S,3000)
$\begin{pmatrix}0\\1\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\\1\end{pmatrix}$
EV(FN,"Tom")	EV(LN,"Gray")		EV(S,2000)
$\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$		$\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$

4.1 Attribute-Value Evidence Vector

Given an attribute A of R, and given a value v of A, the evidence vector for the attribute value v, denoted by $EV(A, v)$ distinguishes the tuples of r whose attribute A takes the value v from those whose attribute A takes a value that is different from v. In doing so, $EV(A, v)$ is implemented as a bit vector, in which the ith bit is set to 1 if the the tuple it represents takes the value v for the attribute A, and 0, otherwise. Notice that doing so assume that the tuples of r are associated with a position from 1 to $|r|$. To do so, we assume the existence of a bijective function, $getTupleAt(r, i)$, implemented using a list, which maps each integer value i, where $1 \leq i \leq |r|$, to a tuple in r. WLOG, we consider for exposition sake that $getTupleAt(r, i) = t_i$.

Table 3 shows the evidence vectors of the attribute values of the $Employee^{curr}$ relation, i.e., considering the first four tuples in Table 1. Take for example the vector $EV(FN, \text{“}John\text{”})$, the first bit of which is set to 1 indicating that tuple t_1 has "John" as a first name, whereas the remaining bits are set to 0 indicating that the tuples t_2, t_3 and t_4 have first names that are different from "John". Another example is that of $EV(P, \text{“}Manager\text{”})$ in which only one bit is set to 0 indicating that the tuple t_3 has a position different from "Manager", whereas the remaining three tuples have all "Manager" as a position.

Formally, an attribute-value evidence vector $ES(A, v)$ is defined as follows:

$$ES(A, v)[i] \;=\; \begin{cases} 1, & if\ getTuple(i)[A] = v \\ 0, & otherwise \end{cases}$$

where $1 \leq i \leq |r|$. Attribute-value evidence sets are implemented using bit-vectors whose size corresponds to the cardinality $|r|$.

In what follows, we assume the existence of an index $AVIdx$ providing access to the attribute value evidence vector. Such an index can be used to have access to all the available attribute value evidence vectors for a given relation r using the operations $AVIdx.getVectors(R)$, as well as for respectively retrieving, adding and removing a given attribute value vector: $AVIdx.getVector(A, v, R)$, $AVIdx.addVector(A, v, R)$ and $AVIdx.removeVector(A, v, R)$ where v is a value of the attribute A of R. We omit the argument R, when it is clear from context.

Notice that given a relation r, the number of attribute-value evidence vectors that are generated is equal to the number of distinct attributes values that appear in r. For example, Emp^{curr} contains 11 different attribute values (see the initial four tuples in Table 1), each of which gives rise to an attribute-value evidence vector.

4.2 Tuple-evidence Multiset

Given a relation r^{curr} and a batch $b = (b_{ins}, b_{del})$, to compute $\Delta^{+}_{ES^m}(r)$ and $\Delta^{-}_{ES^m}(r)$, we make use, as we will see in Section 5, of the notion of tuple-evidence multiset, which is defined as follows.

Definition 4.1 (Tuple-evidence Multiset). Given a tuple t of r, the evidence multiset of t given r, denoted by $ES^m(t,r)$ is a multiset, the members of which are the agree-sets $as(t,t_i)$, where $t_i \in r \setminus \{t\}$. That is: $ES^m(t,r) = \{\{as(t,t_i)\ s.t.\ t_i \in r \setminus \{t\}\}\}$

Essentially, a tuple-evidence multiset $ES^m(t,r)$ provides information about the agree-sets that the tuple t contributes to the relation evidence multiset $ES^m(r)$.

Notice that computing a tuple-evidence multiset involves comparing the attribute values of the tuple t, with the attribute values of every tuple t_i in r except for t itself. To reduce the cost of this operation, we make use of attribute-value evidence vectors that we have introduced earlier at the beginning of this section. Specifically, to compute $ES^m(t,r)$, we start by performing a zip-like product operation of the attribute-value evidence vectors of the attribute values in t, giving rise to what we refer to as tuple-evidence vector $EV(t, Emp^{curr})$.

Definition 4.2 (Tuple evidence vector). Given $[A_1, \ldots, A_m]$, the list of attributes of R, and given a tuple t of r, the zip-bit product (denoted by the symbol $\odot$) of the attribute-value evidence vectors $EV(A_i, t[A_i])$ with i in $[1, m]$, generates the tuple-evidence vector $EV(t,r)$:

$$EV(t,r) = \bigodot_{A_i in [A_1,\ldots,A_m]} EV(A_i, t[A_i])$$

where the jth element of $EV(t,r)$ is a bitset that is obtained by concatenating the jth bits of the attribute-value evidence vectors $EV(A_1, t[A_1]), \cdots, EV(A_m, t[A_m]$.

As an example, consider the tuple t_1 of the Emp^{curr} relation. The tuple-evidence vector $EV(t_1, Emp^{curr})$ is obtained using the following zip-bit product: $EV(FN, t_1[FN]) \odot EV(LN, t_1[LN]) \odot EV(P, t_1[P]) \odot EV(S, t_1[S])$, which is depicted in Figure 2. Notice that the jth element of the tuple-evidence vector is a bitset that is obtained by concatenating the bits in the jth position of the attribute-value evidence vectors. Take for example the second element in the tuple-evidence vector $EV(t_1, Emp^{curr})$, i.e., 0010. The first bit is set to 0 because the 2nd element in the attribute-value evidence vector $EV(FN, t_1[FN])$ is set to 0, while the third bit of such bitset is set to 1 because the second bit of the attribute-value evidence vector $EV(P, t_1[P])$ is set to 1.

Once a tuple-evidence vector $EV(t,r)$ is computed, the tuple evidence multiset $ES^m(t,r)$ is obtained by constructing a multiset the elements of which correspond to the bitsets in $EV(t,r)$, with the exception of the bitset corresponding to $as(t,t)$. Going back to our example, the tuple-evidence multiset of t_1 is obtained by extracting the components bitsets of the vector $EV(t_1, Emp^{curr})$ and discarding the bitset corresponding to $as(t_1, t_1)$ to obtain the following multiset: $\{\{0010, 0100, 0010\}\}$.

Note that bitsets with all bits set to 0 are not informative with respect to data dependencies, since they simply indicate that two tuples disagree on all attribute values. As such, they can be removed. Let $getMultiSet()$ be a function that given a vector of bitsets, generates a multiset containing the elements of the vector, except the bitset representing the agreeset $ag(t,t)$ and except the bitsets having all of their bits set to 0. We can now state the following property:

Figure 2: Generating a tuple-evidence vector using attribute-value evidence vectors

$$\underset{EV(FN,t_1[FN])}{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}} \quad \underset{EV(LN,t_1[LN])}{\begin{pmatrix}1\\0\\1\\0\end{pmatrix}} \quad \underset{EV(P,t_1[P])}{\begin{pmatrix}1\\1\\0\\1\end{pmatrix}} \quad \underset{EV(S,t_1[S])}{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}} \quad \Bigg| \quad \underset{EV(t_1,Emp^{curr})}{\begin{pmatrix}1111\\0010\\0100\\0010\end{pmatrix}}$$

Property 2 (Soundness of the generation of tuple-evidence multiset). *Given a tuple t, and the list of attributes $[A_1, \ldots, A_m]$ of R, the multiset obtained by first applying the zip-bit product of the attribute-value evidence vectors $EV(A_i, t[i])$ with $i \in [1 : m]$, and then applying the function $getMultiSet()$ is the tuple-evidence multiset of t, $ES^m(t,r)$. That is:*

$$ES^m(t,r) = getMultiSet(\bigodot_{A_i in [A_1,\ldots,A_m]} EV(A_i, t[A_i]))$$

The proof for this property can be found in the appendix.

Inverted Attribute Index

As well as the above data structures, we need a means to efficiently retrieve tuples having a given attribute value. Given an attribute A of a relational schema R, the inverted attribute index of A given the value v, designated by $IAIdx(R, A, v)$, is a data structure implemented using a hash-table that maps each value v of A to a set containing the identifiers of the tuples having v as the value for A.

Inverted attribute index is similar to position list index (PLIs) (see Abedjan *et al.* [2]). These data structures are typically used in schema-driven FD discovery algorithms, such as TANE [17], to check the validity of candidate FDs. To do so, they intersect the PLIs of the attributes involved in the candidate FD. In our context, they are only intended for indexing unique attribute values. As such, the construction of the inverted attribute index takes a linear time since it can be generated using a single scan of the relation in question.

5 HANDLING INSERTS AND DELETES

Having described *DynASt*'s architecture and underlining data structures, we are now in a position to present the algorithms it uses to incrementally compute the delta evidence multisets following the insertion or deletion of tuples.

5.1 Handling Inserts

Given a relation r^{curr} and a set of tuples b_{ins} that are inserted, Algorithm 1 details how the delta evidence multiset $\Delta^{+}_{ES^m}$ that needs to be added to obtain the new relation evidence multiset, is computed. To do so, for each tuple in b_{ins}, the tuple-evidence multiset is computed and added to the delta evidence multiset (line 11). The soundness of this processing for computing the delta evidence multiset hinges on the soundness of the computation of the tuple-evidence multiset (see Property 2).

As well as computing the delta evidence multiset, the algorithm updates the indices. In particular, an entry (bit) for each inserted tuple is appended to the attribute-value evidence vectors (lines 17-19). For those attribute-value vectors having the same value as the attribute of the inserted tuple, the added bit is set to 1 (lines 23-24). If an attribute value of the inserted tuples does not exist, in which case there is no attribute-value vector that

represents such value within the AVIdx, a new vector is created (line 26). The entry bits of that vector are all set to 0 except for the last one, which is set to "1". Indeed, because none of the other tuples have that attribute value, the associated entry is 0, except for the last entry bit which represents the inserted tuple, which is associated with a 1-bit (lines 27-28). The algorithm also updates the inverted attribute index by adding the inserted tuple to the associated attribute-value clusters in IAIdx (line 21).

Algorithm 1 Compute-ES-Delta+

```
1:  Inputs: r^curr                      ▷ relation r prior to receiving the batch
2:          b_ins                                      ▷ set of inserted tuples
3:  Output: Δ+_ES(r)             ▷ Multiset of evidence sets to be add to
    ES^m(r^curr)
4:  Begin
5:  Δ+_ES(r) = ∅
6:  size = |r^curr|
7:  r^temp = r^curr
8:  for t_i ∈ b_ins do
9:      size + +
10:     update-Index-Given-Insertion(t_i, size)
11:     Δ+_ES(r) = Δ+_ES(r) ⊕ ES^m(t_i, r^temp)
12:     r^temp = r^temp ∪ {t_i}
13: end for
14: Return Δ+_ES(r)
15: End
16:
17: Procedure   update-Index-Given-Insertion(t, size)
18:     for (v ∈ AVIdx.getVectors()) do
19:         v.addBit("1")
20:     end for
21:     for (A ∈ R) do
22:         IAIdx.get(A, t[A]).add(t)
23:         if (!AVIdx.containsKey(A, t[A])) then
24:             v = AVIdx.get(A, t[A])
25:             v.setLastBit("1")
26:         else
27:             v = newBitVector(size)
28:             v.setAllBits("0")
29:             v.setLastBit("1")
30:             AVIdx.add(A, v, t)
31:         end if
32:     end for
33: End
```

5.2 Handling Deletes

Given a set of tuples b_{del} that are deleted from the relation r^{curr}, Algorithm 2 details how the delta evidence multiset $\Delta^-_{ES^m}$ that needs to be subtracted to obtain the new relation evidence multiset, is computed. To do so, for each tuple in b_{del}, the tuple relation evidence multiset is computed and added to the delta evidence multiset (line 8).

The algorithm also updates the indices. In particular, the entry (bit) representing the deleted tuple is removed from each attribute-value evidence vector (lines 16-18). The inverted attribute index is also updated by removing the ID of the deleted tuples from the corresponding clusters in IAIdx (lines 19-21). Note that if a cluster becomes empty as a result of this operation, the corresponding (attribute, value) in *IAIdx* is removed.

5.3 Optimizing the Computation of tuple-evidence Vectors

Algorithm 2 Compute-EV-Delta-

```
1:  Inputs: r^curr                  ▷ relation r prior to receiving the batch
2:          b_del                                  ▷ set of deleted tuples
3:  Output: Δ-_ES(r)         ▷ Multiset of evidence sets subtract from
    ES^m(r^curr)
4:  Begin
5:  Δ-_ES(r) = ∅
6:  r^temp = r^curr
7:  for t_i ∈ b_del do
8:      Δ-_ES(r) = Δ-_ES(r) ⊕ ES^m(t_i, r^temp)
9:      position = getTuplePositin(t_i)
10:     update-Index-Given-Deletion(t_i, position)
11:     r^temp = r^temp \ {t_i}
12: end for
13: Return Δ-_ES(r)
14: End
15:
16: Procedure   update-Index-Given-Deletion(t, position)
17:     for (v ∈ AVIdx.getVectors()) do
18:         v.removeBitAtPosition(position)
19:     end for
20:     for (A ∈ R) do
21:         IAIdx.remove(A, t[v], t)
22:     end for
23: End
```

The essence of the solution we have just described consists in computing tuple-evidence vectors of a given batch (of inserted or deleted tuples) using already precomputed attribute-value evidence vectors. Consider, for example, that a new batch consisting of four tuples t_1, t_2, t_3 and t_4 that have been inserted into a relation r with four attributes, and suppose that computing tuple-evidence vectors corresponding to such tuples involves computing the following zip-bit products, where `vij` are attribute-value evidence vectors:

- $Ev(t_1, r) = v_{11} \odot v_{12} \odot v_{13} \odot v_{14}$
- $Ev(t_2, r) = v_{21} \odot v_{12} \odot v_{13} \odot v_{14}$
- $Ev(t_3, r) = v_{31} \odot v_{32} \odot v_{13} \odot v_{14}$
- $Ev(t_4, r) = v_{41} \odot v_{42} \odot v_{13} \odot v_{14}$

So far, we have considered that tuple-evidence vectors of a given batch are computed independently of each other. However, by doing so, we miss opportunities to optimise the number of zip-bit products that need to be performed for their computation. To illustrate this, consider the above tuple-evidence vectors $Ev(t_i, r)$ with i in $[1, 4]$. Their computation involves performing 12 zip-bit products, 3 zip-bit products per tuple-evidence vector. For example, $Ev(t_1, r)$ is computed by first computing the zip product $v_{11} \odot v_{12}$, which yields a vector that we denote by $v_{11,12}$, followed by the zip product $v_{11,12} \odot v_{13}$, which gives the vector $v_{11,12,13}$, and finally $v_{11,12,13} \odot v_{14}$, which results in $v_{11,12,13,14}$ representing the tuple-evidence vector $Ev(t_1, r)$.

The number of zip-bit products that need to be performed to compute such tuple-evidence vectors can, however, be reduced. Indeed, notice that the tuple-evidence vectors $Ev(t_1, r)$ and $Ev(t_2, r)$ share the last three attribute-value evidence vectors, and that the four tuple-evidence vectors share the last two attribute-value evidence vectors. Therefore by carefully selecting the order in which the zip-bit products are performed we can reduce the total number of zip-bit products that need to be performed for computing the four tuple-evidence vectors. Specifically, if we are to start by first performing the zip-bit product

$v_{13} \odot v_{14}$ to obtain $v_{13,14}$ (which is shared by the four tuple-evidence vectors), and then the zip-bit product $v_{12} \odot v_{13,14}$ to obtain $v_{12,13,14}$ (which is shared by the first two tuple-evidence vectors), and then processing the remaining zip-bit products, then we will end up performing 8 zip-bit products instead of 12 to compute the four tuple-evidence vectors. For large batches, the number of zip-bit products saved can be significant especially if the number of unique attribute values is small.

Finding the optimal ordering of zip-bit products can be expensive, however. The problem outlined above is related to common subexpression elimination [29] and code expression factorization [6], and it is known that identifying terms (in our case the zip-bit products) that can be used as factors with the view to perform the minimum overall number of operations quickly becomes a large combinatorial problem. Indeed, just with our example, with 4 attributes, each tuple yields 5 ways of performing the zip-bit products, given the associative nature of the zip-bit products. Comparing the obtained rewriting obtained among the tuples to identify the largest chunks of zip-bit products that can be used in common to reduce the overall number of performed zip-bit products is combinatorially challenging. Therefore, we need a solution that can identify promising factors in a reasonable processing time and requiring small memory consumption.

Using existing factorization or common-subexpression elimination techniques [6, 29] is not fit for our purpose since our zip-bit product is associative but is not commutative. We therefore elaborated a new algorithm (Algorithm 3 below) that adapts existing factorization techniques to our problem and utilizes two data structures that can be constructed in a single pass of the batch to be processed: T_to_V P_to_T.

- T_to_V is a hash data structure that specifies the attribute-value evidence vectors of each tuple. In the case of the above example, this gives rise to the following data structure. $T_to_V = \{t_1 : \{v_{11}, v_{12}, v_{13}, v_{14}\},\ t_2 : \{v_{21}, v_{12}, v_{13}, v_{14}\},\ t_3 : \{v_{31}, v_{32}, v_{13}, v_{14}\},\ t_4 : \{v_{41}, v_{42}, v_{13}, v_{14}\}\}$
- P_to_T, on the other hand, specifies for each zip-bit product the tuples in which is involved. Going back to our example, this gives rise to the following data structure. $P_to_T = \{(v_{11}, v_{12}) : \{t_1\},\ (v_{12}, v_{13}) : \{t_1, t_2\},\ (v_{13}, v_{14}) : \{t_1, t_2, t_3, t_4\},\ (v_{21}, v_{12}) : \{t_2\},\ (v_{31}, v_{32}) : \{t_3\},\ (v_{32}, v_{13}) : \{t_3\},\ (v_{41}, v_{42}) : \{t_4\},\ (v_{42}, v_{13}) : \{t_4\}\}$.

Algorithm 3 operates iteratively, wherein each iteration decides on which zip-bit product to perform. The zip-bit product selected is the one that is associated with the set with the maximal cardinality in P_to_T, that is the one involved in the computation of the maximum number of evidence sets. To illustrate this, consider the above zip-bit product to tuples mapping P_to_T. The zip-bit product occurring most of times are (v_{13}, v_{14}), which is involved in the computation of four evidence sets (Algorithm 3, line 7). The zip-bit product $v_{13} \odot v_{14}$ is computed, giving rise to the vector $v_{13,14}$ (Algorithm 3, line 7). The two data structures T_to_V and P_to_T are updated accordingly by removing the pair (v_{13}, v_{14}) from P_to_T, and updating both data structures T_to_V and P_to_T , when necessary to account for the creation of the new evidence vector (Algorithm 3, lines 10-25) as follows.

- $T_to_V = \{t_1 : \{v_{11}, v_{12}, v_{13,14}\},\ t_2 : \{v_{21}, v_{12}, v_{13,14}\},\ t_3 : \{v_{31}, v_{32}, v_{13,14}\},\ t_4 : \{v_{41}, v_{42}, v_{13,14}\}\}$
- $P_to_T = \{(v_{11}, v_{12}) : \{t_1\},\ (v_{12}, v_{13,14}) : \{t_1, t_2\},\ (v_{21}, v_{12}) : \{t_2\},\ (v_{31}, v_{32}) : \{t_3\},\ (v_{32}, v_{13,14}) : \{t_3\},\ (v_{41}, v_{42}) : \{t_4\},\ (v_{42}, v_{13,14}) : \{t_4\}\}$.

In the second iteration, the algorithm chooses to perform the zip-bit product of the pair $(v_{12}, v_{13,14})$ since this is the new most frequent pair used in the processing of the tuples t_1 and t_2. The new zip-bit product gives rise to the vector $v_{12,13,14}$, and the data structures T_to_V and P_to_T are updated as illustrated before. The algorithm terminates when all the necessary zip-bit products have been performed, i.e. when P_to_T becomes an empty set. It is worth noting that Algorithm 3 performs iteratively both the ordering and the execution of zip-bit products, and it can be used for the computation of tuple-evidence vectors in the case of insertion as well as deletion.

Algorithm 3 Compute-Tuple-Evidence-Vectors

```
1:  Inputs: P_to_T            ▷ zip-bit product to tuples mapping.
2:          T_to_V            ▷ Tuple to attribute-value evidence vector
    mapping.
3:  Output: TR_EV             ▷ A dictionary of tuple-evidence vectors
4:  Begin
5:  TR_EV = {}
6:  while (P_to_T ≠ ∅) do
7:      (v_i, v_j) = EdgeWithMaxCard(P_to_T)
8:      v_ij = v_i ⊙ v_j
9:      P_to_T.remove(v_i, v_j)
10:     for r ∈ P_to_T.getRows(v_i, v_j) do
11:         row = T_to_V.getRows(r)
12:         row.remove(v_i, v_j)
13:         if ∃v_x s.t. row.contains(v_x, v_i) then
14:             row.replace((v_x, v_i), (v_x, v_ij))
15:             P_to_T.getRows(v_x, v_i).remove(r)
16:             P_to_T.getRows(v_x, v_ij).add(r)
17:         end if
18:         if ∃v_y s.t. row.contains(v_j, v_y) then
19:             row.replace((v_j, v_y), (v_ij, v_y))
20:             P_to_T.getRows(v_j, v_y).remove(r)
21:             P_to_T.getRows(v_ij, v_y).add(r)
22:         end if
23:         if row = ∅ then
24:             TR_EV.add(r, v_ij)
25:         end if
26:     end for
27: end while
28: End
```

Note that the algorithm returns a dictionary specifying for each tuple, designated by its row position r, the associated tuple-evidence vector. Once the evidence vectors are computed, the evidence multiset is computed. In doing so, some agree-sets need to be discarded in the case of tuple deletion, and others need to be added in the case of tuple insertion.

6 INDEXING AND RETRIEVING AGREE-SETS

To make use of the obtained agree sets, DynASt provides a means for exploring and retrieving agree-sets. For applications that seek to discover exact dependencies, it suffices to provide them with the currently holding agree-sets or the delta in agree-sets with respect to the last increment. For applications that seek the discovery of approximate data dependencies, they may need additional mechanisms that assist such solutions. Specifically, for the discovery of approximate FDs and approximate keys,

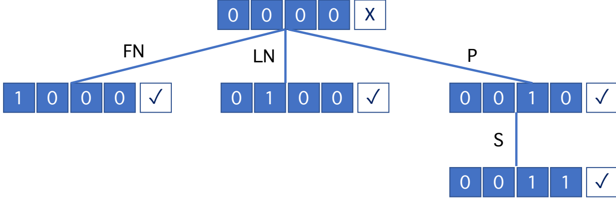

Figure 3: Example illustrating the AStree indexing the agree-sets that holds in $Employee^{next}$.

underlying solutions need a mechanism for efficiently computing errors of approximate dependencies. WLOG, we focus in the remainder of this section on approximate FDs. The same solution presented here for the retrieval of agree-sets can be utilized for estimating the errors in candidate approximate keys.

Approximate FDs are FDs that may not be completely satisfied by tuple pairs in a relation r. Specifically, an approximate FD $fd : X \longrightarrow A$ is associated with an error specifying the percentage of unordered tuple pairs $\{t_1, t_2\}$ in $r \otimes r$ that violate fd, i.e., that have the same values for the attributes X and different values for at least one attribute in A [9, 19]. Given that the number of unordered tuple pairs in $r \otimes r$ is $\frac{|r| \cdot (|r|-1)}{2}$, the error of an approximate FD can be defined as follows:

Definition 6.1 (Error of Approximate FD). The error of an approximate FD $fd : X \longrightarrow A$ of a relation r is defined as follows:

$$err(fd, r) = \frac{2 \cdot |\{\{t_1, t_2\} \in (r \otimes r)\ s.t.\ (t_1[X] = t_2[X]) \land (t_1[A] \neq t_2[A])\}|}{|r| \cdot (|r| - 1)}$$

When exploring the space of candidate approximate FDs, there is, therefore, a need for a mechanism that identifies the proportion of tuple-pairs (in other words agree-sets) that satisfy the dependency in question. Therefore, the above definition can be reformulated as follows.

Definition 6.2 (Error of Approximate FD). Given a relation r of a relational schema R, the error of the approximate FD $fd : X \longrightarrow A$ given r is defined as follows:

$$err(fd, r) = \frac{2 \cdot |\{as \in ES^m(r)\ s.t.\ \neg satisfy(as, fd))\}|}{|r| \cdot (|r| - 1)}$$

where $satisfy(as, fd)$ is a boolean function that returns true if the agree-set as satisfies the FD FD, and is false, otherwise.

Notice that the element that is computationally expensive when computing the error is determining the magnitude of the set $\{as \in ES^m(r)\ s.t.\ \neg satisfy(as, fd))\}$. To efficiently determine the magnitude of such a set, we use two data structures for indexing agree-sets. The first is a hash table that specifies for each agree-sets the number of times it occurs in the relation evidence multiset $ES^m(r)$. The second data structure is an extended prefix-tree [28], which we refer to as AStree specifying the agree-sets that occur in $ES^m(r)$. Figure 3 illustrates the AStree indexing the agree-sets of the relation Emp^{next}. We recall here that $ES^m(Emp^{next}) = \{\{0010, 0010, 0100, 0100, 1000, 1000, 0011, 0011\}\}$. Each edge is labeled by an attribute. An agree-set is represented by a path in the tree. For convenience, we label the nodes in the tree by bitsets representing the agree-set they represent. Moreover, each node is associated with a flag (a bit), which is set for agree-sets that hold. In or example AStree, all nodes, except the root, represent agree-sets that hold that is why they are all checked. In the general case, however, we may have internal nodes that are not checked because they correspond to agree-sets that do not hold.

The AStree can be used for effectively computing the error of a given candidate approximate function dependency. To illustrate how, consider a candidate FD $X \longrightarrow A$, where X is a subset of the attribute of the target relational table and X is an attribute of the relation such that $A \notin X$. To compute the error of such a dependency, we need to compute the number of agree-sets that violate it. Such agree-sets can be characterized as bitsets the size of which correspond to the arity of the relational table, and in which the bits representing the attributes in X are set to 1 whereas the bit representing the A attribute is set to 0. This is perhaps better explained using a concrete example. Consider the *Employee* relation illustrated in Table 1, and consider the FD $LN \longrightarrow P$. The bitsets representing agree-sets that violate such a dependency are as follows: 0100, 1100, 0101 and 1101. To identify the error of such a dependency, we need to identify for each of such bitsets the one that exists in $ES^m(Employee^{next})$, and add up their cardinalities. This operation can be costly if the number of bitsets that need to be examined is large. This is the case for relational tables with a large number of attributes. The AStree allows performing this operation efficiently. Indeed, to determine the number of agree-sets that violate the dependency $LN \longrightarrow P$, we simply perform a search in the AStree that is performed in a single pass of the tree, and that returns all the agree-sets that are equal to or special the agree-set 0101. An agree-set as_1 specializes and an agree-set as_2 if every bit that is set to 1 in as_1 is also set to 1 in as_2. Given the returned agree-sets, their cardinalities are retrieved using the hash table used for this purpose.

The reader may wonder why we do not store the cardinality of each valid agree-set in the AStree by using an integer instead of a boolean flag specifying valid agree-sets. The reason we do that is to reduce the number of updates that need to be performed to the AStree when processing inserted or deleted tuples. Indeed, given the insertion and/or deletion of tuples to the target relation, many agree-sets will remain valid but their cardinalities are likely to change. For such agree-sets, we do not need to update the AStree, but simply update their associated cardinalities in the hash table.

7 VALIDATION

The solution we have presented for incremental computation of agree-sets over dynamic data sets raises a number of questions. In particular, how effective is *DynASt*, what parameters affect its performance, and for which kind of datasets is it suitable? How does it compare to state-of-the-art solutions solutions? Is the insertion and deletion performance comparable? What is the cost of maintaining the ASTree, and how do search queries perform against them?

To answer the above questions, we conducted a series of experiments, which we report on in this section. We conducted the experiments on datasets, which were selected taking into account two parameters that can impact the performance of the presented solutions, namely the number of rows and the number of columns in the datasets. Table 4 summarizes the characteristics of the 6 datasets we used for the evaluation. The first two datasets (Bridges and Iris) have a small number of rows and columns. The Adult and Claims datasets have a large number of rows but a small number of columns, while the last two datasets have a small number of rows but a large number of columns. It should be noted here that to our knowledge, none of the existing related work considers datasets with a large number of columns, such as the Uniprot dataset with 222 columns, for evaluation purposes. The table also contains information on the number of agree-sets and attribute-value evidence vectors in each dataset.

Table 4: Characteristics of the datasets used in the evaluation.

Dataset	#Columns	#Rows	#Agree-sets	#Attribute-value evidence vectors
Bridges	13	108	644	337
Iris	5	150	27	126
Adult	15	32,561	9,555	22,146
Claims	10	**101,984**	512	20,657
Flight	109	1,000	19,099	4,203
Uniprot	**222**	1,000	**326,698**	**29,396**

Since our solution is intended for dynamic datasets, we partitioned each dataset into a number of batch of size n, and measured the time required to compute agree-sets given the insertion (and deletion) of batches. We considered different batch sizes.

Regarding the competitive evalution, we used DeepMiner, which we implemented in Java based on the description of the algorithm given in [21]. We also used the dynamic solution DynFD[1], which is the most recent schema-based solution that caters for the insertion and deletion of triples. We implemented *DynASt* using Java[2], and used for the empirical evaluation a server with a single processor with a speed of 1.6 GHz, and 16 GB or memory. This server runs on macOS 10.14.6 and uses JDK 1.8.

7.1 Batch Processing Performance

In our first experiment, we evaluated the performance of DynASt in processing the insertion of batches of size equal to 50 on all datasets. For the Bridges and Iris datasets, the time required was low (less than 8 milliseconds). Figure 4 illustrates the results for each of the remaining datasets, showing the time required to process a batch after each quantile. (The figure also illustrates the processing time for processing batch deletion, which we will examine later in Section 7.4.) For the Adult and Claims datasets, the figure shows that Dynast's performance is affected by the size of the dataset into which the batch is inserted, in particular the processing time increases slightly with dataset size. Notice that for some batches, especially for the batch inserted after the 9th quantile of the adult dataset, the processing time increases. This can be explained by the fact that at this batch, new attribute values appeared given the newly inserted tuples, resulting in the creation of new attribute-value evidence vectors. It should be noted, however, that the performance is good given that the maximum processing time for batch insertion is 1.4 second and is recorded in the case of the Claims dataset. The figure also illustrates the results for the Flight and Uniprot datasets, and shows that the performance of DynASt is better for these datasets (compared to the Adults and Claims datasets), despite the large number of columns characterizing the Flight and Uniprot datasets. The maximum time recorded is 40 ms for the Flight dataset and 160 ms for the Uniprot dataset. This first experiment shows that the performance of DynAST is good when inserting small batches, and that the performance is better for datasets with a large number of columns than for datasets with a large number of rows.

7.2 Scalability

To evaluate the impact of batch size, we performed the previous experiment by varying the batch size. For the Adults and Claims dataset, the batch size varies between 10 and 1000, while for the Flight and Uniprot dataset, the batch size varies between 10 and 150 tuples, since both datasets have a small number of rows. Figure 5 illustrates the results. For a given batch size n, the figure shows the average processing time recorded when successively processing all batches of size n that make up the dataset. The figure shows that DynASt performs better for smaller batch sizes, and that the larger the batch size, the higher the required processing time. This observation applies to all datasets. We also note that DynASt performs better for the Flight and Uniprot datasets despite the large number of columns characterizing these datasets, regardless of the batch size. Processing batches of 100 or less for the Claims and Adult datasets requires little time. However, this time increases for batches with 500 and 1000 tuples. This shows that our solution scales well for datasets with a large number of columns. It can also be used for datasets with a large number of rows, provided that the batch size remains small (less than or equal to 100 tuples).

To evaluate the impact of schema size (number of columns) on DynASt's performance, we considered for each dataset the case where a portion of the dataset containing 500 tuples has already been processed and examined the time required to process the insertion of a batch of 50 tuples. Figure 6 illustrates the results obtained. It shows that DynAST scales much better at the schema level compared with the number of rows. For example, the Flight dataset, which has 109 columns, ~ 11 times the number of columns in the Claims dataset and 7 times the number of columns in the Adult dataset, requires 100 ms to process a batch of 50 tuples, a time comparable to the processing times recorded in the case of the Adult and Claims datasets. This same observation applies for the Uniprot dataset, which is characterized by a large schema consisting of 222 columns, that is, for example, 22 times the number of columns in the Claims dataset, and yet requires 311 ms. This time is higher than the time required by other datasets, but is still small in proportion to the schema size for the Uniprot dataset. We also note that the time required for processing the Flight dataset is smaller than that required by the Adult dataset, which is characterized by a much smaller schema. This can be explained by the fact that the number of attribute-value evidence vectors in the case of the Flight dataset is smaller, which is due to the fact that the number of distinct attribute values is small in the Flight dataset compared with the Adult dataset. This same observation also applies to the comparison of processing time between the Flight and Uniport dataset. In other words, Dynast is sensitive to the number of attribute-value evidence vectors.

7.3 Competitive Evaluation

The objective of this experiment is to compare the performance of our incremental solution to a baseline bulk-based solution. As a baseline solution, we utilised the algorithm proposed by DepMiner [21]. We are interested in identifying the batch sizes

[1] github.com/HPI-Information-Systems/dynfd
[2] github.com/khalidb/DynASt

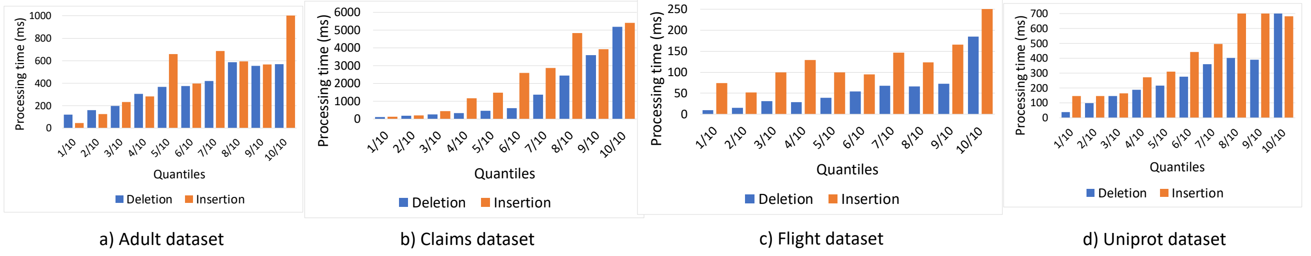

Figure 4: Processing of batches of size 50.

Table 5: Speedup of insertion processing required by our solution compared with that of the baseline solution.

Dataset \ Batch size ratio	1%	10%	100%	1000%
Adult	439.1	15.62	12.13	9.94
Claims	45.59	5.83	0.23	0.18
Flight	581.64	74.37	17.33	14.36
Uniprot	6610.56	457.46	30.82	27.27

Table 6: Speedup of deletion processing required by our solution compared with that of the baseline solution.

Dataset \ Batch size ratio	1%	10%	50%	75%
Adult	439.26	50.13	2.62	0.42
Claims	23.87	2.97	0.23	0.03
Flight	153.39	28.05	1.20	0.14
Uniprot	3984.29	536.88	10.53	0.24

for which our incremental solution outperforms the baseline solution. To carry out this experiment, we changed the batch size from small to very large (even larger than the initial dataset). To do this, we defined the batch size relative to the initial dataset. Specifically, we considered the cases where the batch size is equal to 1%, 10%, 100%, and 1000% of the size of the initial dataset, and computed the time required to generate agree-sets using our incremental solution and the baseline solution, respectively.

Table 5 shows that for all datasets the incremental methods outperforms largely the baseline solution, and that this performance decreases as the batch size ratio increases. The table also showed that even for a large batch size (1000%), our incremental solution outperforms the baseline solution, with the exception of the Claims dataset. Take, for example, the Adult dataset. With a batch size ratio of 1000%, our solution is nearly 10 times faster than the baseline solution. This implies that our solution has merit also for medium-sized datasets such as the Adult dataset containing 32,561 tuples. On the other hand, our solution is less suitable for large batch size ratios when dealing with large datasets (in terms of number of rows) such as the Claims dataset.

We also conducted an experiment comparing the performance of DynASt with DynFD [26], a system for dynamic maintenance of FDs. For this purpose, we extended DynASt to be able to generate FDs for given agree-sets. Rather than developing such an extension from scratch, we used and adapted an algorithm, viz. the FDEP algorithm [13], that was developed under the aegis of the Metanome project[3] to generate FDs given agree-sets that were computed using DynASt. We performed the same experiment as the one described above, but with DynFD. The results

[3]hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html

Table 7: Speedup of DynFD compared to DynASt in processing datasets with large number of rows.

Dataset \ Batch size ratio	1%	10%	100%	1000%
Adult	2.66	15.61	11.03	13.12
Claims	4.07	11.66	16.22	24.23

Table 8: Speedup of DynASt compared to DynFD in processing datasets with large number of columns .

Dataset \ Batch size ratio	1%	10%	100%	1000%
Flight	22.48	18.19	11.36	10.70
Uniprot	X	X	X	X

obtained can be divided into two classes, depending on whether the datasets to be processed has a large number of rows or a large number of columns. DynFD outperforms DynASt when processing datasets with a large number of rows (see table 7). The larger the number of rows to be inserted, the greater the difference in processing. For example, DynFD is up to 25 times faster when processing the largest dataset (Claims). DynFD showed similar performance in processing deletions when it came to such datasets, i.e. Adults and Claims.

On the other hand, DynASt outperforms DynFD when processing datasets with a large number of columns (see table 8). In particular, DynASt can be up to 22 times faster than DynFD. It is also worth mentioning that DynFD was unable to process Uniprot: the program crashes after a long waiting time (20 minutes and 11 seconds waiting time in the case of processing the insertion of 1% of the Uniprot dataset). It is also worth noting that DynFD was not able to process deletions for both Flight and Uniprot, regardless of the percentage of deleted tuples. DynASt, on the other hand, was able to gracefully process deletions for both of these datasets. Specifically, maintaining agree-sets when deleting 75% of the Flight and Uniprot datasets required 1.08 seconds and 6.56 seconds, respectively.

The results of the above experiment support the hypothesis made in the introduction regarding the superiority of the schema-based solutions, i.e. DynFD, for datasets with a large number of rows but a small number of attributes, and the superiority of data-driven solutions, in this case DynASt, for datasets with a large number of attributes but a small number of rows. The study thus confirms DynASt's position as a complement to existing schema-based solutions when it comes to handling dynamic datasets.

7.4 Batch Deletion Processing

We conducted experiments like those described above where the tuples in the batch are deleted from the initial datasets (instead

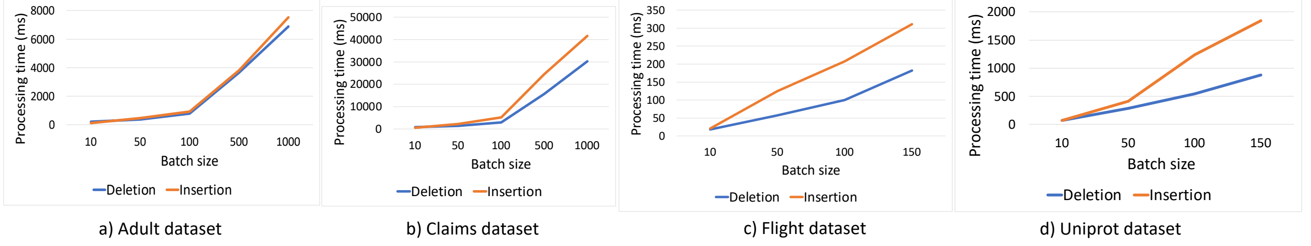

Figure 5: Batch Size Scalability.

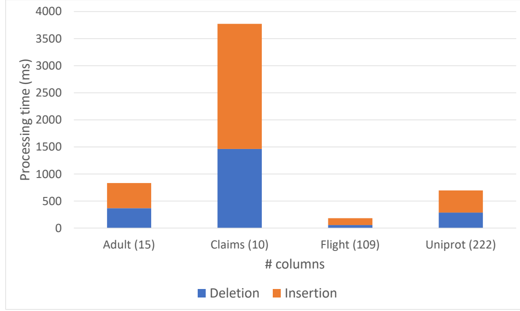

Figure 6: Schema scalability.

of them being inserted into it). The results are reported in the previous figures (Figures 4, 5 and 6), and the conclusions are the same as with tuple insertion. There are two observations that need to be reported, however, by comparison with batch insertion. First, batch deletion tends to take a slightly smaller time compared with batch insertion with datasets with a small number of rows. This is the case for the Flight and Uniprot dataset. On the other hand, we observed that for datasets for a large number of rows, e.g., the Adult and Claims datasets, the processing time tends to increase. This can be explained by the fact that in the case of the tuple insertion, attribute-value evidence vectors are updated by appending a new element, whereas in the case of tuple deletion, attribute-value evidence vectors are updated by deleting an existing element with positions in the middle of the vectors. This last operation involves a search within the attribute-element vector to identify the elements that represent the tuple deleted, an operation that is can be costly compared with a simple append, especially when the size of the attribute-value vector (which is equal to the number of rows in the dataset) is large.

7.5 ASTree Processing

We have advocated the use of ASTree as a means of storing, maintaining and querying agree-sets. We performed a series of experiments examining the processing time required to maintain the agree-sets tree considering the insertion and deletion of tuples. This experiment showed that for datasets with a small number of columns, e.g., Adults and Claims, the required processing time is low even when processing large batches. In contrast, maintaining the agree-sets tree for datasets with a large number of attributes, especially Uniprot, proved to be more expensive. It should be noted, however, that not only the number of columns affects the processing time, but also the number of agree-sets that need to be added or removed. To illustrate this, the table 9 shows the processing time required for inserting and removing batches of 50 tuples in each of the datasets we considered.

Table 9: Agree-Set update given a batch of 50 tuples.

	Adult (10)	Claims (15)	Flight (109)	Uniprot (222)
Insertion	3.2	0.4	16.95	1590
Deletion	6.11	1.2	34	2300

It shows that the processing time is quite low for Adults and Claims. It increases slightly for the Flight dataset, even though the number of columns increases from 15 to 109. For the Uniprot dataset, the processing time increased substantially, due to the large number of columns that this dataset induces.

Note, however, that the processing time for deletion can be decreased by not processing the deletion after each batch. Indeed, we have an auxiliary structure (a map structure) that informs us about the cardinalities of the agree-sets. Thus, such agree-sets can be kept in the agree-set tree even when their cardinality reaches the value 0. Indeed, using this strategy, we noticed that the retrieval of agree-sets (or more specifically the specialization of a given agree-set, which is necessary for the treatment of approximate constraints for example) is negligibly affected, it takes less than a millisecond.

8 RELATED WORK AND CONCLUDING REMARKS

Data profiling [1], in general, and data dependency discovery [18], in particular, are classic problems that date back to the end of the 90's and that have gained momentum in recent years due to the growing interest in the broad area of data-driven research and analysis and the advent of data lakes.

8.1 Static Dependency Discovery

For data dependency discovery, existing solutions can be classified into two categories: schema-driven solutions and data-driven solutions. Schema-driven solutions model the space of possible solutions using a lattice whose nodes are subsets of the attributes of the target relationship. Regarding FDs, for example, pioneered by TANE algorithm [17], various solutions have been proposed subsequently, including DFD [5], FD_MINE [34], and FUN [22]. Regarding key discovery, HCA [3], DUCC [16] and the proposal by Giannella and Wyss [15] are schema-driven. The solutions in this class propose different ways to efficiently traverse and prune the lattice in the search for valid dependencies. Solutions that fall in this category are known to perform well for datasets with a large number of rows but a small number of columns (attributes).

The second class of solutions that have been pursued are data-driven, and are known to perform well for datasets with a large number of columns but a small number of rows. Solutions that fall in this category compare the row of the datasets to identify data dependencies. For example, the well-known key-discovery Gordian algorithm [27] is data Driven. Regarding FDs, Dep-Miner [21], FDEP [21] and FastFD [33] are the pioneering algorithms in

data-driven dependency discovery algorithms. In essence, such algorithms are based on agree-sets (or difference-sets), and algorithms in this area attempt to reduce the number of agree-sets that need to be computed to identify dependencies. For example, DepMiner is based on the notion of maximal sets, which are used to derive the tuple-pairs that need to be compared, thereby discarding useless tuple-pairs that share no attribute values and as such are not informative as far as FD discovery is concerned.

As well as the above schema-based and data-based approach, a number of hybrid proposals have been made where the two approaches are used in tandem, e.g., HyFD[24], PYRO [19], HCA-Gordian [3] and the proposal by Wei and Link [31, 32]. Such proposals use both tuple comparison with lattice traversal (which is translated into PLI intersections in the case of FD discovery), and they differ in how they combine the two techniques.

8.2 Dynamic Dependency Discovery

The proposals by Gasmi *et al.* [14] is perhaps the first to tackle the problem of incremental discovery of FDs under tuple insertion. In the same lines, the proposal by Caruccio *et al.* [8] have explored the problem of maintaining FDs when tuples are inserted in the target relation. The former is data-driven while the latter is schema-driven. The two proposals are confined to tuple insertion, and as such do not address the case of tuple deletion. Moreover, the proposal by Gasmi *et al.* tackles the insertion of a single tuple at a time, making it unsuitable in situations where updates occur in batches. Wang *et al.* [30], on the other hand, tackles tuple deletion using a schema-driven algorithm, and it does so that by traversing the lattice of possible FDs to identify the FDs that can be generalized. DynFD [26] is to our knowledge the most complete in the sense that it maintains FDs under both tuple deletion and insertion and in a batch-processing manner. This proposal is also a schema-based that utilizes PLIs together with covers representing minimal FDs and maximal non-FDs. Given a set of tuples that are inserted, it inspects existing FDs to identify the ones that have been invalidated. Conversely, given tuples' deletion, it examines the FDs that may have become non-minimal as a result. Regarding key discovery, the only proposal that we are aware of is SWAN [4]. It operates similarly to the proposal by Schirmer *et al.*, in the sense that it maintains metadata information about minimal keys and maximal non-keys. Given tuple insertion or deletion, it examines the keys or non-keys that may have been affected as a result.

From the above, it transpires that existing complete solutions capable of handling batches of insertions and deletions adopt a schema-driven approach. *D*ynASt complements the above proposals by adopting a data-driven approach, focusing on the computationally intensive step, namely the computation of agree-sets and their maintenance against dynamic data sets. The data structures we have adopted as well as the associated algorithms allow us to handle datasets with a large number of attributes, as shown in the evaluation section. Moreover, empirical evaluation has shown that our incremental solution outperforms the state of the art of data-driven solutions even in static settings, and the most recent dynamic solution DynFD for datasets with large number of columns. In future work, we intend to investigate how our proposed method for dynamic maintenance of agree-sets can be used in tandem with a schema-based method to efficiently handle data dependency discovery in dynamic settings.

9 APPENDIX: PROOFS

Proof for Property 1. To prove that $ES^m(r^{next}) = ES^m(r^{curr}) \oplus \Delta^+_{ES^m}(r) \ominus \Delta^-_{ES^m}(r)$, we start by showing that:
$r^{next} \otimes r^{next} = (r^{curr} \times r^{curr}) \cup (r^{next} \times b_{ins}) \setminus (r^{curr} \times b_{del})$
We have:
$r^{next} \otimes r^{next} = (r^{curr} \cup b_{ins} \setminus b_{del}) \otimes (r^{curr} \cup b_{ins} \setminus b_{del})$
Given that unordered Cartesian product, just like ordered Cartesian product, distributes over union and set difference, we have:
$r^{next} \otimes r^{next} = (r^{curr} \otimes r^{curr}) \cup (r^{curr} \otimes b_{ins}) \cup (b_{ins} \otimes r^{curr}) \cup (b_{ins} \otimes b_{ins}) \setminus (r^{curr} \otimes b_{del}) \setminus (b_{ins} \otimes b_{del}) \setminus (b_{del} \otimes r^{curr}) \setminus (b_{del} \otimes b_{ins}) \setminus (b_{del} \otimes b_{del})$
Unordered cartesian product is, by definition, symmetric: $(A \otimes B) = (B \otimes A)$, and therefore $(A \otimes B) \cup (B \otimes A) = (A \otimes B)$. By applying this property to the above formula, we have:
$r^{next} \otimes r^{next} = (r^{curr} \otimes r^{curr}) \cup (r^{curr} \otimes b_{ins}) \cup (b_{ins} \otimes b_{ins}) \setminus (r^{curr} \otimes b_{del}) \setminus (b_{ins} \otimes b_{del}) \setminus (b_{del} \otimes b_{del})$
Using again the fact that unordered Cartesian product distributes over union and set difference, we factorize the above formula as follows: $r^{next} \otimes r^{next} = (r^{curr} \otimes r^{curr}) \cup ((r^{curr} \cup b_{ins} \setminus b_{del}) \otimes b_{ins}) \setminus ((r^{curr} \cup b_{del}) \otimes b_{del})$
Given that $r^{next} = r^{curr} \cup b_{ins} \setminus b_{del}$, we have:
$r^{next} \otimes r^{next} = (r^{curr} \otimes r^{curr}) \cup (r^{next} \otimes b_{ins}) \setminus ((r^{curr} \cup b_{del}) \otimes b_{del})$
We have that $b_{del} \subseteq r^{curr}$, which implies that $r^{curr} \cup b_{del} = r^{curr}$, thereby reducing the above formula as follows:
$r^{next} \otimes r^{next} = (r^{curr} \otimes r^{curr}) \cup (r^{next} \otimes b_{ins}) \setminus (r^{curr} \otimes b_{del})$

We use the following two multiset properties to reach the desired conclusion. Given two sets A and B such that $A \cap B = \emptyset$:

$$\{\{f(x) \ s.t. \ x \in A \cup B\}\} = \{\{f(x) \ s.t. \ x \in A\}\} \oplus \{\{f(x) \ s.t. \ x \in B\}\}$$

Given two sets B and C such that $B \subseteq C$:

$$\{\{f(x) \ s.t. \ x \in B \setminus C\}\} = \{\{f(x) \ s.t. \ x \in B\}\} \ominus \{\{f(x) \ s.t. \ x \in C\}\}$$

Using the above two properties, we have:
$ES^m(r^{next}) = \{\{as(t_i, t_j) \ st. \ \{t_i, t_j\} \in r^{next} \otimes r^{next}\}\}$

$ES^m(r^{next}) = \{\{as(t_i, t_j) \ st. \ \{t_i, t_j\} \in (r^{curr} \otimes r^{curr}) \cup (r^{next} \otimes b_{ins}) \setminus (r^{curr} \otimes b_{del})\}$

Given that $(r^{curr} \otimes b_{del}) \subseteq (r^{curr} \otimes r^{curr})$, we have:
$ES^m(r^{next}) = \{\{as(t_i, t_j) \ st. \ \{t_i, t_j\} \in (r^{curr} \otimes r^{curr}) \cup (r^{next} \otimes b_{ins})\}\} \ominus \{\{as(t_i, t_j) \ st. \ \{t_i, t_j\} \in (r^{curr} \otimes b_{del})\}\}$

And, given that $(r^{curr} \otimes r^{curr}) \cap (r^{next} \otimes b_{ins}) = \emptyset$, we have:
$ES^m(r^{next}) = \{\{as(t_i, t_j) \ st. \ \{t_i, t_j\} \in (r^{curr} \otimes r^{curr})\}\} \oplus \{\{as(t_i, t_j) \ st. \ \{t_i, t_j\} \in (r^{next} \otimes b_{ins})\}\} \ominus \{\{as(t_i, t_j) \ st. \ \{t_i, t_j\} \in (r^{curr} \otimes b_{del})\}\}$

Which yields the conclusion we are after:

$$ES^m(r^{next}) = ES^m(r^{curr}) \oplus \Delta^+_{ES^m}(r) \ominus \Delta^-_{ES^m}(r)$$

Proof for Property 2. Consider the ith element of the vector $EV(t, r)$, and let us examine the jth bit in such element. Such a bit is set to 0 if t and the tuple in the ith position, $t_i = getTupleAt(r, i)$, have the same value for the attribute A_j, and 0 otherwise. This allows us to conclude that the the ith element of the vector $EV(t, r)$ represents the agree-set $as(t, t_i)$ where $t_i = getTupleAt(r, i)$. Given that i^{th} ranges from 1 to the size of r, we can conclude that the elements in the vector $EV(t, r)$ cover all the agree-sets $as(t, t_i)$ where $t_i \in r$, including t itself. We can therefore conclude that the multiset that is created from such a vector by eliminating the element compose of bits set to 0 is the tuple evidence multiset $ES^m(t, r)$.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDB J.* 24, 4 (2015), 557–581. https://doi.org/10.1007/s00778-015-0389-y

[2] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00878ED1V01Y201810DTM052

[3] Ziawasch Abedjan and Felix Naumann. 2011. Advancing the discovery of unique column combinations. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, Craig Macdonald, Iadh Ounis, and Ian Ruthven (Eds.). ACM, 1565–1570. https://doi.org/10.1145/2063576.2063801

[4] Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2014. Detecting unique column combinations on dynamic data. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 1036–1047. https://doi.org/10.1109/ICDE.2014.6816721

[5] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. 2014. DFD: Efficient Functional Dependency Discovery. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, Jianzhong Li, Xiaoyang Sean Wang, Minos N. Garofalakis, Ian Soboroff, Torsten Suel, and Min Wang (Eds.). ACM, 949–958. https://doi.org/10.1145/2661829.2661884

[6] Alfred V. Aho and Stephen C. Johnson. 1975. Optimal Code Generation for Expression Trees. In *Proceedings of the 7th Annual ACM Symposium on Theory of Computing, May 5-7, 1975, Albuquerque, New Mexico, USA*, William C. Rounds, Nancy Martin, Jack W. Carlyle, and Michael A. Harrison (Eds.). ACM, 207–217. https://doi.org/10.1145/800116.803770

[7] Naser Ayat, Hamideh Afsarmanesh, Reza Akbarinia, and Patrick Valduriez. 2012. Pay-as-you-go data integration using functional dependencies. In *International Conference on Availability, Reliability, and Security.* Springer, 375–389.

[8] Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, and Giuseppe Polese. 2019. Incremental Discovery of Functional Dependencies with a Bit-vector Algorithm. In *Proceedings of the 27th Italian Symposium on Advanced Database Systems, Castiglione della Pescaia (Grosseto), Italy, June 16-19, 2019 (CEUR Workshop Proceedings)*, Massimo Mecella, Giuseppe Amato, and Claudio Gennaro (Eds.), Vol. 2400. CEUR-WS.org. http://ceur-ws.org/Vol-2400/paper-21.pdf

[9] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2020. Mining relaxed functional dependencies from data. *Data Min. Knowl. Discov.* 34, 2 (2020), 443–477. https://doi.org/10.1007/s10618-019-00667-7

[10] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE).* IEEE, 458–469.

[11] E. F. Codd. 1983. A Relational Model of Data for Large Shared Data Banks (Reprint). *Commun. ACM* 26, 1 (1983), 64–69. https://doi.org/10.1145/357980.358007

[12] Marius Eich, Pit Fender, and Guido Moerkotte. 2016. Faster plan generation through consideration of functional dependencies and keys. *Proceedings of the VLDB Endowment* 9, 10 (2016), 756–767.

[13] Peter A. Flach and Iztok Savnik. 1999. Database Dependency Discovery: A Machine Learning Approach. *AI Commun.* 12, 3 (1999), 139–160. http://content.iospress.com/articles/ai-communications/aic182

[14] Ghada Gasmi, Lotfi Lakhal, and Yahya Slimani. 2012. An incremental approach for maintaining functional dependencies. *Intell. Data Anal.* 16, 3 (2012), 365–381. https://doi.org/10.3233/IDA-2012-0529

[15] Chris Giannella and C Wyss. 1999. Finding minimal keys in a relation instance.

[16] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. 2013. Scalable Discovery of Unique Column Combinations. *Proc. VLDB Endow.* 7, 4 (2013), 301–312. https://doi.org/10.14778/2732240.2732248

[17] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* 42, 2 (1999), 100–111. https://doi.org/10.1093/comjnl/42.2.100

[18] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning.* ACM. https://doi.org/10.1145/3310205

[19] Sebastian Kruse and Felix Naumann. 2018. Efficient Discovery of Approximate Dependencies. *Proc. VLDB Endow.* 11, 7 (2018), 759–772. https://doi.org/10.14778/3192965.3192968

[20] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient Discovery of Functional Dependencies and Armstrong Relations. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings (Lecture Notes in Computer Science)*, Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust (Eds.), Vol. 1777. Springer, 350–364. https://doi.org/10.1007/3-540-46439-5_24

[21] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient discovery of functional dependencies and armstrong relations. In *International Conference on Extending Database Technology.* Springer, 350–364.

[22] Noël Novelli and Rosine Cicchetti. 2001. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Jan Van den Bussche and Victor Vianu (Eds.), Vol. 1973. Springer, 189–203. https://doi.org/10.1007/3-540-44503-X_13

[23] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proc. VLDB Endow.* 8, 10 (2015), 1082–1093. https://doi.org/10.14778/2794367.2794377

[24] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 821–833. https://doi.org/10.1145/2882903.2915203

[25] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. 2002. Translating Web Data. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002.* Morgan Kaufmann, 598–609. https://doi.org/10.1016/B978-155860869-6/50059-7

[26] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 253–264. https://doi.org/10.5441/002/edbt.2019.23

[27] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. 2006. GORDIAN: Efficient and Scalable Discovery of Composite Keys. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 691–702. http://dl.acm.org/citation.cfm?id=1164187

[28] Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, Byeong-Soo Jeong, and Young-Koo Lee. 2009. Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences* 179, 5 (2009), 559–583.

[29] Jeffrey D. Ullman. 1973. Fast Algorithms for the Elimination of Common Subexpressions. *Acta Informatica* 2 (1973), 191–213.

[30] Shyue-Liang Wang, Wen-Chieh Tsou, Jiann-Horng Lin, and Tzung-Pei Hong. 2003. Maintenance of Discovered Functional Dependencies: Incremental Deletion. In *Intelligent Systems Design and Applications*, Ajith Abraham, Katrin Franke, and Mario Köppen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 579–588.

[31] Ziheng Wei and Sebastian Link. 2019. Discovery and Ranking of Functional Dependencies. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019.* IEEE, 1526–1537. https://doi.org/10.1109/ICDE.2019.00137

[32] Ziheng Wei and Sebastian Link. 2021. Embedded Functional Dependencies and Data-completeness Tailored Database Design. *ACM Trans. Database Syst.* 46, 2 (2021), 7:1–7:46. https://doi.org/10.1145/3450518

[33] Catharine M. Wyss, Chris Giannella, and Edward L. Robertson. 2001. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances - Extended Abstract. In *Data Warehousing and Knowledge Discovery, Third International Conference, DaWaK 2001, Munich, Germany, September 5-7, 2001, Proceedings (Lecture Notes in Computer Science)*, Yahiko Kambayashi, Werner Winiwarter, and Masatoshi Arikawa (Eds.), Vol. 2114. Springer, 101–110. https://doi.org/10.1007/3-540-44801-2_11

[34] Hong Yao, Howard J. Hamilton, and Cory J. Butz. 2002. FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan.* IEEE Computer Society, 729–732. https://doi.org/10.1109/ICDM.2002.1184040