Incremental discovery of functional dependencies with a bit-vector algorithm

Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, and Giuseppe Polese

University of Salerno, Department of Computer Science
via Giovanni Paolo II n.132, 84084 Fisciano (SA), Italy
{lcaruccio,scirillo,deufemia,gpolese}@unisa.it

Abstract. Functional dependencies (FDs) were conceived in the early '70s, and were mainly used to verify database design and assess data quality. Nowadays they are automatically discovered from data since they can be exploited for many different purposes, such as query relaxation, data cleansing, and record matching. In the context of big data, the speed at which new data is being created demand for new efficient algorithms for FD discovery. In this paper, we propose an incremental FD discovery approach, which is able to update the set of holding FDs upon insertions of new tuples to the data instance, without having to restart the discovery process from scratch. It exploits a bit-vector representation of FDs, and an upward/downward search strategy aiming to reduce the overall search space. Experimental results show that such algorithm achieves extremely better time performances with respect to the re-execution of the algorithm from scratch.

Keywords: Functional Dependency · Discovery Algorithm · Instance Updating · Incremental Discovery.

1 Introduction

With the advent of Big Data, both industry and research communities have manifested a tremendous interest in technologies capable of extracting information and their correlations among data. One way to represent such relationships is to use functional dependencies (FDs), which represent relationships among database columns that can be used for several advanced database operations, such as query optimisation, data cleansing, and data integration [12].

While FDs were originally specified at database design time, as properties of a schema that should hold on every instance of it, there has been the need to automatically discover them from data for reducing the design effort and for supporting their evolution in the application domains. This is made possible also thanks to the availability of big data collections, and to the contributions of several research areas, such as machine learning and knowledge discovery [9].

The problem of discovering FDs is extremely complex, since the number of holding FDs for a given database instance could be exponential with respect to the number of database columns. In fact, most of the discovery algorithms described in the literature aimed to provide solutions in which the search space complexity is reduced by exploiting the theoretical properties of FDs. However, while the availability of big data collections stimulates the discovery of meaningful FDs from data, the necessity to update them according to the evolution of database instances require that also FDs are updated. In fact, one of the characteristics describing big data is the *velocity*, which includes the fact that data are continuously produced.

In this paper, we propose an incremental discovery approach for FDs, which updates the complete set of holding FDs according to the new added tuples. The approach exploits a bit-vector representation of FDs, and an upward/downward search strategy aiming to prune the search space. Experimental results achieved on twenty-eight datasets show that the proposed approach achieves better time performances with respect to the re-execution of the algorithm from scratch.

The paper is organised as follows. Section 2 reviews the FD discovery algorithms existing in the literature. Section 3 provides some background definitions about FDs and formulates the problem of FD discovery. Section 4 presents the proposed methodology for the incremental discovery of FDs from data, whose evaluation is reported in Section 5. Finally, summary and concluding remarks are included in Section 6.

2 Related Work

The FD discovery problem dates back to the '80s, when the first discovery algorithms were defined [11], and many of the latest proposals are based on theoretical foundations introduced within older solutions [7, 8]. It is a extremely complex problem, since the number of potential FDs can be exponential, and their detection requires analysing a huge number of column combinations [1].

In the literature there are two main categories of methods to automatically discover FDs from data, namely column-based and row-based methods. The former exploit an attribute lattice to generate candidate FDs, which are successively tested to verify their validity. The whole process is made efficient by exploiting valid FDs to prune the search space for new candidates. Examples of top-down approaches include the algorithms TANE [8], FD_Mine [18], and DFD [2]. Row-based methods derive candidate FDs from two attribute subsets, namely *agree-sets* and *difference-sets*, which are built by comparing the values of attributes for all possible combinations of tuples pairs. Examples of bottom-up approaches include the algorithms DepMiner [10], FastFD [17], and FDep [6].

Experimental results show that column-based algorithms usually outperform row-based ones on datasets with many rows and few columns, whereas on datasets with few rows and many columns the row-based algorithms usually perform better [13]. Recently, an hybrid algorithm has been proposed in order to obtain better performance in all cases [14]. It combines row- and column-efficient discovery techniques by managing two separated phases, one in which it calculates FDs on a randomly selected small subset of records (column-efficiency), and the other in which it validates the discovered FDs on the entire dataset.

One of the first theoretical proposal of incremental algorithm for FD discovery has been proposed in 2001 [16]. It exploits the concepts of tuple partitions and monotonicity of FDs to avoid the re-scanning of the database. Another proposal exploits the concept of functional independency in order to maintain the set of FDs updated over time [3]. Finally, in order to discover and maintain functional dependencies in dynamic datasets, the DynFD algorithm has been proposed [15]. It continuously adapts the validation structures of FDs in order to evolve them with batch of inserts, updates, and deletes of data. Some of the methodologies surveyed above (i.e. [16, 3]) are not implemented, whereas others need to store the data structure produced during the discovery process. To this end, the advantage of the proposed methodology is that it relies only on the list of discovered FDs, so it can be applied to any column-based FD discovery algorithm.

3 The Discovery Problem

Before discussing the problem of discovering FDs, let us first recall the definition of the canonical FD. Given a relational database schema $\mathcal{R}$, defined over a set of attributes $attr(\mathcal{R})$, derived as the union of attributes from relation schemas R composing $\mathcal{R}$, assuming w.l.o.g. they all have unique names. For each attribute $A \in attr(R)$, its domain is denoted by $dom(A)$. Moreover, given an instance r of R and a tuple $t \in r$, we use $t[A]$ to denote the projection of t onto A; similarly, for a set X of attributes in $attr(R)$, $t[X]$ denotes the projection of t onto X. An FD over R is a statement $X \rightarrow Y$ (X implies Y) with $X, Y \subseteq attr(R)$, such that, given an instance r over R, $X \rightarrow Y$ is satisfied in r if and only if for every pair of tuples (t_1, t_2) in r, whenever $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$. X,Y are also named Left-Hand-Side (LHS) and Right-Hand-Side (RHS) of an FD.

The goal of dependency discovery is to find meaningful dependencies holding among columns of a dataset. They represent domain knowledge and can be used to assess database design and data quality [9]. Moreover, discovering dependencies from data permits to capture the evolution of real-world domains.

Discovering FDs over a relation instance r entails finding all the possible column combinations, and for each of them find all the possible partitions forming the LHS and RHS of candidate FDs. Without loss of generality, we can consider only candidates with a single attribute on the RHS. Given this, the FD discovery problem has an extremely large search space. In fact, given a relation r with M attributes and n tuples, we need to consider all the possible combinations of 2 to M attributes, counting each of them as many times as the number of attributes in it, in order to account for the number of different candidates with a single RHS attribute. This complexity is synthesised by the following formula:

$$\sum_{k=2}^{M} \binom{M}{k} k \qquad (1)$$

Since this complexity represents only the number of candidate FDs that could be potentially checked, the discovery algorithms need to tackle several other issues, such as the validation of each candidate whose complexity is linear in the number of tuples.

The general FD discovery problem is described by the following definition.

Definition 1. *Given a relation instance r of a relation R, an* FD *discovery algorithm has to find the minimal cover set of* FD*s holding in r, having the property that tuples equal on the LHS must be equal also on the RHS.*

According to this definition, an FD discovery algorithm has to identify the *minimal cover* of FDs holding on a relation instance r. The minimal cover will contain all valid FDs even if they have low support. Thus, an holding FD does not appear in the minimal cover P only when it can be inferred by other FDs in P. For these reasons, given a set P of all minimal FDs holding on r, and computed at time τ, it is necessary to update P whenever one or more tuples have been added from time τ to time τ', since new tuples can violate one or more FDs of P. Details on how we deal with such a problem are discussed into the next section.

4 Incremental Discovery of Functional Dependencies

This section presents the structure of the proposed incremental approach for FDs discovery by introducing the data representation and the search strategies. Data structures, pruning, and validation processes of FDs will, therefore, be described.

4.1 Incremental Methodology

FD discovery algorithms operating on static datasets require to be re-execute upon the insertion of new tuples. This is a computationally expensive operation, especially for database instances with a large number of rows and columns. The main problem with the database instance evolution is that FDs found at time τ can be invalidated at time τ'. For this reason, it is necessary to implement incremental approaches that reduce the time for searching and validating FDs.

In our study, we propose an incremental discovery approach that takes in input the FDs associated to a relation instance r, represents FDs using bitsets, and performs an upward and downward search strategy. Figure 1 shows the binary representation of an FD with k attributes. In particular, each dependency $X \rightarrow A$ requires two bitsets: the first contains all the LHS attributes, while the second contains the RHS attribute of the FD.

Each location of a bitset B represents an attribute of a relation instance r. In particular, given an FD $X \rightarrow A$ on r, if $B_X[i] = 1$ then the i-th attribute in r belongs to X. The size of each bitset corresponds to the number of attributes of the considered instance r. In this way, it is possible to represent dependencies with hundreds of attributes in a compact and lightweight fashion.

In our methodology, all binary FDs holding on a instance at time τ are stored in a linked ordered hash map. This data structure was used to ensure that information are quickly saved and retrieved. The map uses as key the bit-vector representation of FDs, and as value the next dependency, according to an ordering criteria based on the LHS cardinality. Moreover, two fast arrays link the map in order to reduce the insertion times of the FDs. This data organisation allows to perform a level-wise discovery based on a lattice search strategy. Furthermore, it guarantees an immediate pruning of dependencies that are not minimal.

As said above, inserting new tuples in a relation instance could affect the validity of the FDs. In particular, the main effects that can be produced by new tuples are:

– **Invalidation of functional dependencies**. Let $X \rightarrow A$ be a minimal FD with $X, A \subseteq attr(R)$ at time τ, such that for each pair of tuples (t_1^τ, t_2^τ) belonging to the instance r, whenever $t_1^\tau[X] = t_2^\tau[X]$ then $t_1^\tau[A] = t_2^\tau[A]$. If at time $\tau + 1$ there exist a new tuple $t_3^{\tau+1}$ such that:

$$t_1^\tau[X] = t_3^{\tau+1}[X] \wedge t_1^\tau[A] \neq t_3^{\tau+1}[A]$$

or two new tuples $t_4^{\tau+1}$, $t_5^{\tau+1}$, which verify the following property:

$$t_4^{\tau+1}[X] = t_5^{\tau+1}[X] \wedge t_4^{\tau+1}[A] \neq t_5^{\tau+1}[A]$$

then the FD $X \rightarrow A$ is no longer valid.

– **Confirmation of minimality**. If a minimal functional dependency is valid at time τ and has not been invalidated by any tuple at time $\tau + 1$, then this dependency is minimal also at time $\tau + 1$.

Based on these considerations, the methodology considers the first functional dependency extracted at time τ with the smallest LHS cardinality and the attribute partitions of the relation instance. In the pre-processing phase, the FDs holding at time τ are mapped within the linked ordered hash map and the partitions are updated according to the set of new tuples. The use of tuple partitions avoids to access data during the discovery and allows to validate the FDs through the refinement property [8].

Definition 2. ***Refinement.*** *A functional dependency $X \rightarrow A$ holds on r iff $|\pi_X| = |\pi_{(X \cup A)}|$, where π_X and $\pi_{(X \cup A)}$ are the sets of equivalence classes, i.e., they are the partitions of r on X and $X \cup A$.*

If a minimal FD $X \rightarrow A$ is invalidated at time $\tau + 1$, then new candidate dependencies need to be analysed. In particular, it is necessary to validate all

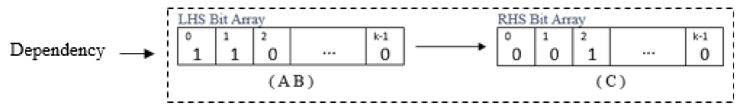


Fig. 1. Bit-vector representation of a functional dependency.

Algorithm 1 Incremental Discovery Algorithm

```
INPUT: a set $\Sigma_\tau$ of valid and minimal FDs at time $\tau$
OUTPUT: a set $\Sigma_{\tau+1}$ of new valid and minimal FDs at time $\tau+1$
 1: for all $X \rightarrow A \in \Sigma_\tau$ do
 2:     if REFINEMENT($X \rightarrow A$) is not valid then
 3:         $L_{l+1} \leftarrow$ NEXTLEVEL($X \rightarrow A$)
 4:         for all $X_{l+1} \in L_{l+1}$ do
 5:             if not INFERENCE($X_{l+1} \rightarrow A$ then)
 6:                 $\Sigma_\tau \leftarrow \Sigma_\tau \cup \{X_{l+1} \rightarrow A\}$
 7:     else
 8:         $L_{l-1} \leftarrow$ PREVLEVEL($X \rightarrow A$)
 9:         for all $X_{l-1} \in L_{l-1}$ do
10:             if not INFERENCE($X_{l-1} \rightarrow A$) then
11:                 $\Sigma_\tau \leftarrow \Sigma_\tau \setminus \{X \rightarrow A\}$
12: $\Sigma_{\tau+1} \leftarrow \Sigma_\tau$
```

non-trivial FDs $XB \rightarrow A$ for each $B \in attr(R)$, with $B \notin X \cup A$. This means that a higher level in the lattice search strategy is considered. In fact, only candidates of higher levels w.r.t. the holding FDs at time τ can hold at time $\tau+1$. Moreover, it is not necessary to verify the dependencies $X \setminus B \rightarrow A$ for each $B \in X$, since such FDs were not valid at time τ, and cannot be valid at time $\tau+1$. However, it is necessary to guarantee the minimality of $XB \rightarrow A$. In fact, such dependency can be considered as minimal at time $\tau+1$ if and only if it cannot be inferred by other holding FDs at time $\tau+1$.

Definition 3. ***Inference.*** *A functional dependency $XB \rightarrow A$ is inferred by another dependency $XB \setminus Z \rightarrow A$ with $Z \subseteq XB$ iff $XB \setminus Z \rightarrow A$ holds at time $\tau+1$, and $Z \neq \emptyset$.*

The inference checking is performed in the proposed search process by exploiting the previously introduced linked ordered hash map. In fact, it is necessary to verify if does not exist an FD, among those already validated at time $\tau+1$, which is minimal with respect to $XB \rightarrow A$. In particular, in order to speed up the inference checking process, we used a hash map that links the possible FD's RHS with all minimal FDs already validated at time $\tau+1$. This allows to prune the number of FDs to be checked.

The complete discovery process defined according to the proposed methodology is described by Algorithm 1. In particular, for each FD holding at time τ (line 1), it verifies if $X \rightarrow A$ also holds at time $\tau+1$ through the `REFINEMENT` function (line 2). Next, if $X \rightarrow A$ is not valid, the algorithm generates new candidate FDs at a higher level, by excluding those that can be inferred (lines 3-6). Instead, if $X \rightarrow A$ is valid, the latter is added to the result set only iff it cannot be inferred by other holding FDs at a lower level (lines 8-11).

In general, the proposed approach can drastically reduce the execution time for two main reasons:

1. the bit-vector representation of FDs permits to quickly process new configurations of FDs by computing the new LHSs in terms of attribute supersets or subsets. In other words, it optimises the downward/upward search strategies;
2. the ordered linked hash map permits to extremely prune the search process, since it guarantees a fast minimality test.

Example 1. Table 1 shows a database instance r in which three new tuples have been added at time $\tau + 1$. Starting from the partitions of the instance r at time τ, that is, $\pi_A = \{[0, 1], [3, 4], [2]\}$, $\pi_B = \{[3, 4], [0], [1, 2]\}$, $\pi_C = \{[0, 2], [4], [1, 3]\}$, $\pi_D = \{[0, 3], [4], [2], [1]\}$, the incremental algorithm computes the updated partitions, that is, $\pi'_A = \{[0, 1], [3, 4, 7], [2, 5, 6]\}$, $\pi'_B = \{[3, 4], [0], [6, 7], [1, 2, 5]\}$, $\pi'_C = \{[0, 2, 5, 7], [4], [1, 3, 6]\}$, $\pi'_D = \{[0, 3, 5], [7], [4], [2], [6], [1]\}$, and loads the minimal FDs holding at time τ. The latter are represented in terms of bit-vectors and inserted into the ordered linked map.

Figure 2 shows the linked ordered hash map obtained at time $\tau+1$ for the new instance of Table 1. It visualises all the considered candidate FDs. In particular, each bit-vector in Figure 2 represents: 1) a minimal FD holding at time $\tau + 1$, 2) a candidate FD that has been invalidated at time $\tau + 1$, or 3) a candidate FD that is not minimal at time $\tau + 1$.

Initially, the dependency with the lowest LHS cardinality is selected by using a fast array to directly link the first dependency with a given LHS cardinality. In Figure 2 this corresponds to $(0110) \rightarrow (0001)$. Using the refinement property, this FD is removed because $|\pi_X| \neq |\pi_{(X \cup A)}|$. Starting from this, the algorithm calculates the next candidates, which consider LHS supersets. For each of them the inference checking is applied, and only those that cannot be inferred are added to the map. Therefore, in the example, the FDs $(1110) \rightarrow (0001)$ and $(0111) \rightarrow (1000)$ are added to the ordered map, also updating the links that preserve the map ordering. The execution proceeds until all FDs are explored. In particular, the FD $(0111) \rightarrow (1000)$ is removed because it is not minimal with respect to $(0101) \rightarrow (1000)$, which has been previously validated. Finally, the algorithm returns the new minimal set of FDs holding on the instance at time $\tau + 1$, so avoiding the re-execution of the discovery algorithm from scratch.

Table 1. An example of a relation instance updated at time $\tau + 1$.

	Row Id	A	B	C	D
	0	Andorra	1	French	French
	1	Andorra	2	English	Russian
	2	San Marino	2	French	Italian
	3	Cipro	4	English	French
	4	Cipro	4	Greek	Spanish
+	5	San Marino	2	French	French
+	6	San Marino	3	English	German
+	7	Cipro	3	French	Arabic

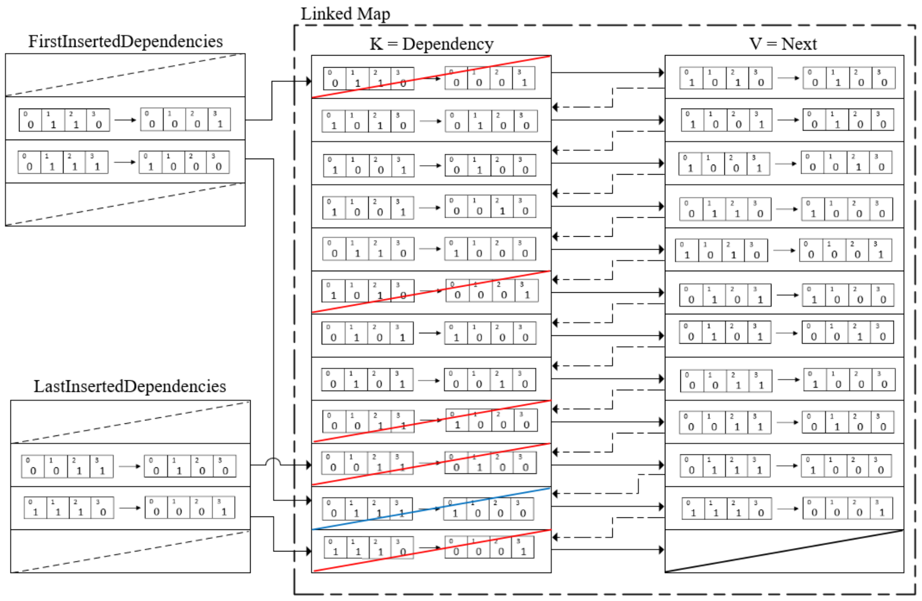


Fig. 2. The Linked Ordered Hash Map related to Example 1.

5 Evaluation

In the following, we present experimental results concerning the performance of the proposed approach in discovering FDs. In particular, the performed tests show how much the new proposed approach can improve time performances with respect to the re-execution of the FD discovery algorithm.

Implementation details. The algorithm has been developed in Java 11.0.2. In particular, in order to improve the performance of the algorithm and avoid the re-calculation of partitions, we also introduced a methodology for caching partitions, since the latter are widely used for the validation of candidate FDs.

Hardware and datasets. The tests has been performed on a Mac with an Intel Xeon processor at 3.20 GHz 8-core and 64GB of RAM. Moreover, to ensure a proper execution on the considered datasets, the Java memory heap size allocation has been set to 40GB to permit a proper execution on datasets with a high number of tuples/attributes.

We evaluated the proposed approach on several real world datasets, previously used for testing FD discovery algorithms [13]. Statistics on the characteristics of the considered datasets are shown in Table 2. Such datasets are composed of one relation, since FD discovery algorithms always consider de-normalised databases.

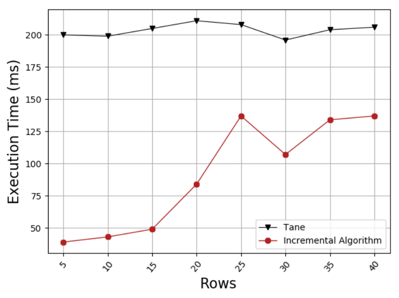
Table 2. Characteristics of the used datasets and discovery results.

Dataset	Tuples	Attributes	%Inserts	#fds	TANE time(ms)	Incremental time(ms)
Abalone	4176	9	50 %	137	308	**183**
Adult	32561	15	50 %	78	43033	**1085**
Balance-scale	624	5	50 %	1	130	**6**
Breast-cancer-wisc.	699	11	50 %	46	316	**284**
Breast-cancer	285	10	50 %	1	230	**132**
Bridges	108	13	50 %	142	218	**195**
Bupa	344	7	50 %	25	134	**19**
CalIt	10081	3	50 %	1	176	**30**
Cars	407	9	50 %	67	162	**48**
Car_data	1727	7	50 %	1	197	**7**
Chess	1999	7	50 %	6	184	**7**
Citations	1000	9	50 %	76	203	**182**
Citeseer	10K	7	50 %	10	456	**116**
Citeseer	20K	6	50 %	4	721	**816**
Cmc	1472	10	50 %	1	477	**15**
DBLP	20K	8	50 %	28	364	**172**
Echocardiogram	132	13	50 %	538	191	**363**
Ecoli	335	9	50 %	54	139	**24**
Haberman	305	4	50 %	0	110	**1**
Hayes-roth	132	6	50 %	5	119	**2**
Iris	149	5	50 %	4	116	**6**
Letter	20K	17	50 %	61	177087	**6203**
Mammography	960	6	50 %	0	150	**1**
Nursery	12960	9	50 %	1	1065	**45**
Servo	166	5	50 %	1	115	**3**
Tae	150	6	50 %	2	122	**14**
Tax	100K	15	50 %	364	29368	**28598**
Wine	178	14	50 %	1374	209	**131**

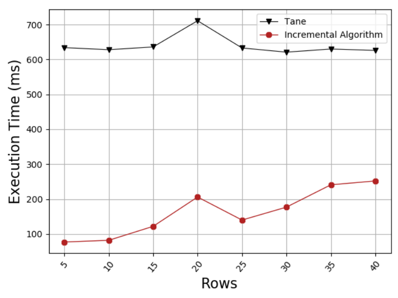
Evaluation process. We carried out different tests by using as starting point the minimal FDs extracted through the TANE algorithm [8]. All the tests were performed on datasets split into two parts. The first part represents the relation instance at time τ, and it has been given in input to TANE. The complete dataset represents the relation instance at time $\tau+1$ analysed by the proposed incremental discovery algorithm. Moreover, we re-executed the TANE algorithm on the complete dataset. This allowed us to analyse the resulting FDs, and to compare execution times of our approach w.r.t. a complete re-execution of TANE. Moreover, we executed other two experiments: 1) by varying the number of tuples inserted at time $\tau+1$; 2) by varying the number of rows/columns of the dataset.

Analysis of the results. The first test has been conducted by simulating the insertion of 50% of tuples of the complete dataset. The results are reported in Table 2, where a comparison between the times of TANE and the incremental approach are shown. From the results we can notice that the proposed approach is in general more efficient, despite the variability of the number of rows/columns. This is mainly due to the pruning strategies introduced by the incremental approach. However, there are few cases in which the execution times are equivalent to those of TANE. This happens when there is a huge number of holding FDs at time τ.

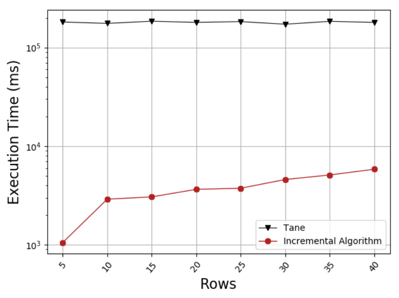
The second test aimed to evaluate the relationship between the execution times and the sizes of the datasets. In particular, we selected datasets by varying



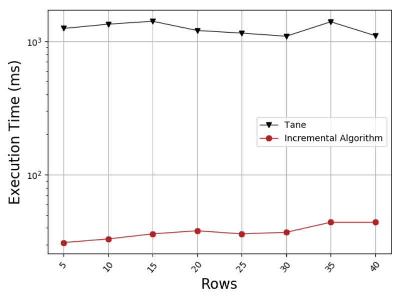
(a) Citations



(b) Citeseer



(c) Letter



(d) Nursery

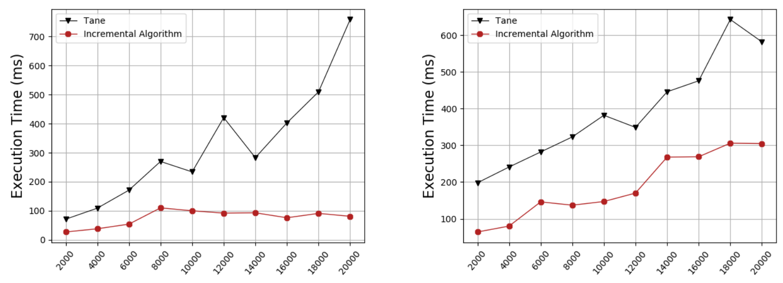
Fig. 3. A comparison between execution times of TANE with respect to our proposal by varying the percentage of inserted tuples at time $\tau + 1$.

the number of tuples inserted at time $\tau + 1$ in the range of 5%-40% w.r.t. the complete dataset. Results are shown in Figure 3. In particular, the results for *nursey* (Figure 3(d)) and *citeseer* (Figure 3(b)) show an extremely reduction of the execution times when considering the incremental approach. This is probably due to the fact that the number of minimal FDs at time τ was small. Moreover, although this number is higher for *citations* and *letter*, the time performances of our approach is still lower than the execution of TANE (Figures 3(a) and 3(c)).

The last test allowed us to evaluate the discovery time of the incremental algorithm on datasets with a fixed number of tuples but a variable number of columns (first session), and with a fixed number of attributes but a variable number of tuples within an interval range of $[2000 - 20000]$ tuples with step 2000 (second session). For these experiments *dblp* and *citeseer* datasets have been selected. Table 3 contains the characteristics of the datasets and the results of the experiments with a variable number of attributes. Instead, the results of the second session are shown in the Figure 4. By analysing Table 3, we can notice that the execution times of the proposed approach are always lower than the execution times of TANE. Moreover, we noticed that for our incremental approach the *dblp* dataset is more critical than *citeseer* when the number of attributes increases. Also in this case, this is probably due to the higher number

Table 3. A comparison between execution times of TANE with respect to our proposal by varying the number of attributes.

Dataset	Tuples	Attributes	%Inserts	TANE time(ms)	Incremental time(ms)
DBLP	20K	2	3 %	220	**32**
DBLP	20K	3	3 %	277	**78**
DBLP	20K	4	3 %	293	**126**
DBLP	20K	5	3 %	366	**162**
DBLP	20K	6	3 %	418	**232**
DBLP	20K	7	3 %	490	**245**
Citeseer	20K	2	3 %	175	**2**
Citeseer	20K	3	3 %	203	**2**
Citeseer	20K	4	3 %	360	**3**
Citeseer	20K	5	3 %	376	**45**
Citeseer	20K	6	3 %	435	**73**
Citeseer	20K	7	3 %	456	**89**



(a) Citeseer (b) DBLP

Fig. 4. A comparison between execution times of TANE with respect to our proposal by varying the number of tuples.

of FDs discovered at time $\tau + 1$. Different results have been obtained when we consider the variation in the number of tuples (Figure 4). In this case, although non monotonic the time trend grows faster for TANE.

6 Final Remarks

In this paper we have proposed an incremental approach for discovering FDs, which permits to update the set of holding FDs without the need of exploring the complete search space. A bit-vector representation of the FDs allowed us to optimise this process. Experimental results show that the proposed approach considerably reduces the execution times with respect to a re-execution from scratch.

In the future, we would like to further improve this approach in order to automatically updates FDs even for other database instance modifications, such as the deletion and the updating of tuples. Another interesting issue concerns the possibility of updating also Relaxed Functional Dependencies (RFDs) [4], for which the discovery process is even more complex [5].

References

1. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. The VLDB Journal **24**(4), 557–581 (2015)
2. Abedjan, Z., Schulze, P., Naumann, F.: DFD: Efficient functional dependency discovery. In: Proceedings of the 23rd ACM International Conference on Information and Knowledge Management. pp. 949–958. CIKM '14 (2014)
3. Bell, S.: Discovery and maintenance of functional dependencies by independencies. In: KDD. pp. 27–32 (1995)
4. Caruccio, L., Deufemia, V., Polese, G.: Relaxed functional dependencies – A survey of approaches. IEEE TKDE **28**(1), 147–165 (2016)
5. Caruccio, L., Deufemia, V., Polese, G.: On the discovery of relaxed functional dependencies. In: Proceedings of 20th International Database Engineering & Applications Symposium. pp. 53–61. IDEAS '16 (2016)
6. Flach, P.A., Savnik, I.: Database dependency discovery: A machine learning approach. AI Commun. **12**(3), 139–160 (1999)
7. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: Efficient discovery of functional and approximate dependencies using partitions. In: ICDE. pp. 392–401 (1998)
8. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal **42**(2), 100–111 (1999)
9. Liu, J., Li, J., Liu, C., Chen, Y.: Discover dependencies from data - A review. IEEE Transactions on Knowledge and Data Engineering **24**(2), 251–264 (2012)
10. Lopes, S., Petit, J.M., Lakhal, L.: Efficient discovery of functional dependencies and armstrong relations. In: Proceedings of the 7th International Conference on Extending Database Technology. pp. 350–364. EDBT '00 (2000)
11. Mannila, H., Räihä, K.J.: Dependency inference. In: Proceedings of the 13th International Conference on Very Large Data Bases. pp. 155–158. VLDB '87 (1987)
12. Naumann, F.: Data profiling revisited. ACM SIGMOD Record **42**(4), 40–49 (2014)
13. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J.P., Schönberg, M., Zwiener, J., Naumann, F.: Functional dependency discovery: An experimental evaluation of seven algorithms. Proceedings of the VLDB Endowment **8**(10), 1082–1093 (2015)
14. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: Proceedings of the 2016 International Conference on Management of Data. pp. 821–833. ACM (2016)
15. Schirmer, P., Papenbrock, T., Kruse, S., Hempfing, D., Meyer, T., Neuschäfer-Rube, D., Naumann, F.: DynFD: Functional dependency discovery in dynamic datasets. In: To appear in EDBT (2019)
16. Wang, S.L., Shen, J.W., Hong, T.P.: Incremental discovery of functional dependencies using partitions. In: Proceedings Joint 9th IFSA World Congress and 20th NAFIPS International Conference (Cat. No. 01TH8569). vol. 3, pp. 1322–1326. IEEE (2001)
17. Wyss, C., Giannella, C., Robertson, E.: FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In: Procs of Intl Conf. on Data Warehousing and Knowl. Disc. pp. 101–110. DaWaK '01 (2001)
18. Yao, H., Hamilton, H.J., Butz, C.J.: FD_Mine: Discovering functional dependencies in a database using equivalences. In: Proceedings of IEEE International Conference on Data Mining. pp. 729–732. ICDM '02 (2002)