

# Efficient Validation of Functional Dependencies during Incremental Discovery

Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia and Giuseppe Polese

*Department of Computer Science, University of Salerno, via Giovanni Paolo II n.132, 84084 Fisciano (SA), Italy*

## Abstract

The discovery of functional dependencies (FDs) from data is facing novel challenges also due to the necessity of monitoring datasets that evolves over time. In these scenarios, incremental FD discovery algorithms have to efficiently verify which of the previously discovered FDs still hold on the updated dataset, and also infer new valid FDs. This requires the definition of search strategies and validation methods able to analyze only the portion of the dataset affected by new changes. In this paper we propose a new validation method, which can be used in combination with different search strategies, that exploits regular expressions and compressed data structures to efficiently verify whether a candidate FD holds on an updated version of the input dataset. Experimental results demonstrate the effectiveness of the proposed method on real-world datasets adapted for incremental scenarios, also compared with a baseline incremental FD discovery algorithm.

## Keywords

Data Profiling, Functional Dependency, Incremental Discovery

## 1. Introduction

The usefulness of functional dependencies (FDs) has been widely demonstrated due to their application in several contexts, such as for cleaning data [1], evaluating the feasibility of classification models [2], and so forth. Although in the last decades many algorithms have been proposed for the automatic discovery of FDs from data, e.g., [3, 4, 5], they lack the capability of managing dynamic data. In this context, the set of discovered FDs should be kept consistent with the possible data changes over the time, e.g., by means of insert, update, and delete operations. As a consequence, incremental FD discovery algorithms have to adopt discovery strategies that limit the search of candidate FDs to specific parts of the search space according to previously discovered FDs.

Most of the incremental FD discovery algorithms existing in literature extend traditional FD search strategies with mechanisms that exploit the knowledge generated by the previous executions [6, 7]. However, none of these algorithms introduce optimizations in their validation methods aiming to consider data changes only.

---


*SEBD 2021: The 29th Italian Symposium on Advanced Database Systems, September 5-9, 2021, Pizzo Calabro (VV), Italy*

✉ lcaruccio@unisa.it (L. Caruccio); scirillo@unisa.it (S. Cirillo); deufemia@unisa.it (V. Deufemia); gpolese@unisa.it (G. Polese)

🆔 0000-0002-6711-3590 (L. Caruccio); 0000-0003-0201-2753 (S. Cirillo); 0000-0002-6711-3590 (V. Deufemia); 0000-0002-8496-2658 (G. Polese)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

Traditional validation methods, such as the partition-based ones, exploit the possibility to organize data into partitions according to attribute candidate sets, which will consistently be redefined by following a level-by-level search strategy. In this way, the validation can be performed by testing properties of partitions involved in each candidate FD [5]. Instead, row-based discovery algorithms directly derive candidate FDs by comparing attribute values for all possible combinations of tuples pairs, so implicitly inferring the validation of the FDs [4, 8].

In this paper, we propose an FD incremental discovery algorithm, named REXY (RegeX-based incremental discoverY), which includes a new validation method exploiting regular expressions (RegExs) to extract the subset of data affecting discovery results. It employs a compressed data representation, which limits the memory load and optimizes the data analysis. The incremental search strategy of REXY is based on an upward/downward candidate generation method based on the set of previously holding FDs without the need of exploring the complete search space. We experimentally evaluated REXY on 22 real-world datasets. In particular, we evaluated the effectiveness of the algorithm in its incremental identification of FDs using as performance benchmark the incremental algorithm described in [9]. The results demonstrate that REXY improves the time performances of a baseline algorithm of several orders of magnitude.

The remainder of the paper is organized as follows. Section 2 reviews the main discovery strategies and algorithms existing in the literature. Section 3 introduces the incremental discovery problem. Section 4 describes the proposed validation method and its integration in an incremental search strategy, whereas the whole discovery algorithm is reported in Section 5. Experimental results are discussed in Section 6, and conclusions are included in Section 7.

## 2. Related Work

In the literature several algorithms for the discovery of FDs from data have been defined. They are mainly devoted to the analysis of data from scratch, yielding specific methodologies to explore and prune the search space. In particular, column-based and row-based represent the two main strategies they follow.

Column-based strategies model the search space as an attribute lattice, which permits to consider candidate dependencies at each level in terms of lattice's edges, enabling the representation of the Left-Hand-Side (LHS) and the Right-Hand-Side (RHS) of an FD. After the generation of FD candidates at each level, it is necessary to validate each of them, entailing the evaluation of combination values or making the FD representations efficient, such as data partitions [3, 5, 10]. By following a level-by-level search strategy, column-based strategies exploit the possibility to progressively organize data, and to test the validation of a candidate FD by considering how data are collected into partitions. On the contrary, row-based strategies directly derive candidate FDs by analyzing all possible combinations of tuples pairs, aiming to derive two attribute subsets in terms of *agree-sets* or *difference-sets*, and entailing the automatic derivation of all holding FDs [4, 8]. In this case, the validation of an FD is implicitly inferred by the overall analysis over the set of data. Finally, in order to improve the efficiency of discovery algorithms, hybrid strategies have also been proposed [11, 12]. The latter calculate FDs on a randomly selected small subset of records (row-based), and validates the discovered FDs on the entire dataset, by focusing on a small subset of the search space determined by FD candidates

and validation results (column-based).

In literature, there exist several FD discovery algorithms for incremental scenarios. In particular, one of the first algorithms exploits the concepts of tuple partitions and monotonicity of FDs to avoid the re-scanning of the database [7]. Another proposal is based on the concept of functional *independency*, through which the algorithm maintains the set of FDs updated over time [13]. Finally, the DynFD algorithm is able to discover and maintain FDs in dynamic datasets [6]. In particular, it extends an aforementioned hybrid strategy, by continuously adapting the FD validation structures according to a batch of insert, update, and delete data operations. With respect to these incremental approaches, REXY uses a new validation method that focuses the verification of candidate FDs only on the subset of data affected by the data changes. This can improve the efficiency of FD discovery in incremental scenarios.

### 3. Incremental discovery of Functional Dependencies

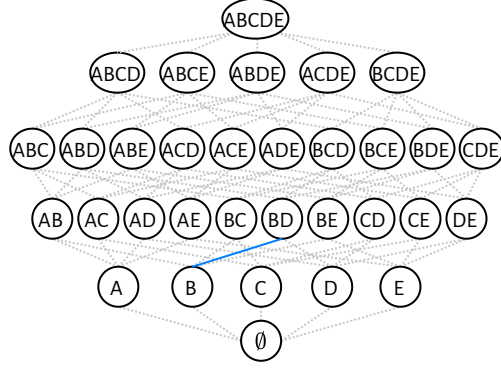
Functional Dependencies (FDs) express relationships between attributes of a database relation. An FD  $X \rightarrow Y$  states that the values of an attribute set  $Y$  are uniquely determined by the values of  $X$ . More Formally, given a relation schema  $R$ , an FD over  $R$  is a statement  $X \rightarrow Y$  ( $X$  determines  $Y$ ), with  $X, Y \subseteq \text{attr}(R)$ , 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  $t_1[Y] = t_2[Y]$ .  $X$  and  $Y$  are also named Left-Hand-Side (LHS) and Right-Hand-Side (RHS) of an FD, respectively. The latter is said to be *non-trivial* if and only if  $X \supsetneq Y$ , and *minimal* if and only if there is no attribute  $B \in X$  such that  $X \setminus B \rightarrow Y$  is also an FD holding on  $r$ . For sake of simplicity and w.l.o.g, in the rest of the paper we assume that  $Y$  consists of a single attribute.

In general, the FD discovery problem aims at finding a set of all *minimal* FDs holding on a relation instance  $r$ . It entails searching for a partition of tuples sharing the same values on RHS attributes whenever they share the same value on LHS attributes [14]. To do this, it is possible to first generate FD candidates, and then verifying their validity and minimality, yielding a column-based strategy.

Column-based strategies model the search space as a graph representation of a lattice, which contains a collection of attribute sets, where *Level 0* maps the empty set, *Level 1* singleton sets, one for each attribute, *Level 2* the pair sets, one for each possible combination of two attributes, and so forth. Finally, the last level, namely *Level M*, will contain a single set of all the attributes from  $R$ . It permits to consider candidate FDs at each level in terms of edges. For instance, the edge highlighted in blue in Figure 1 defines the candidate FD  $B \rightarrow D$ .

The validation process of column-based strategies performs simple computations on the cardinalities of the partitions calculated over attribute combinations, which correspond to the lattice nodes. This process is particularly efficient when it is possible to follow a level-by-level search strategy from *Level 1* to *Level M*, and to gradually construct partitions over ever-increasing attribute set sizes. This could not be guaranteed when it is necessary to explore the search space with different starting points, such as a set of FDs.

Incremental scenarios require to update a set of minimal FDs  $\Sigma_\tau$  discovered over data collected until time  $\tau$  according to the set of tuple modifications  $D_{\tau+1}$  at time  $\tau + 1$ . This yields the necessity to (re)-consider all FDs in  $\Sigma_\tau$ , and possibly generates new FD candidates according to validation



**Figure 1:** The lattice search space representation for the attribute set  $\{A, B, C, D, E\}$ .

results. In fact, the addition of a new tuple can entail the *invalidation* of a previously holding FD, whereas the deletion of a tuple can entail to a previously holding FD be no longer *minimal*. More formally, let  $X \rightarrow A$  be an FD holding at time  $\tau$ , and  $t$  a tuple yielding a dataset modification, then we need to consider the following effects:

- **Invalidation:** when a tuple  $t$  is added, it could produce the invalidation of  $X \rightarrow A$  at time  $\tau + 1$ . Consequently, one or more FD candidates on the next lattice level having the same RHS and a superset of its LHS could be validated. Thus, it is necessary to consider all possible FD candidates  $XB \rightarrow A$  such that  $B \notin X$  and  $B \neq A$ .
- **Minimality:** when a tuple  $t$  is removed,  $X \rightarrow A$  could be no longer minimal at time  $\tau + 1$ . Consequently, one or more FD candidates on the previous lattice level having the same RHS and a subset of its LHS could be validated, yielding a minimal FD. Thus, it is necessary to consider all possible FD candidates  $X \setminus B \rightarrow A$  such that  $B \in X$ .

Notice that, a tuple update can always be managed as the deletion of the stored tuple and the subsequent addition of the tuple containing the modified values.

Invalidation and minimality checks determine how FD candidate should be generated and managed throughout the search space according to previously holding FDs and validation results. This entails moving the focus on different parts of the search space, by focusing the attention on a subset of data characterized by specific candidate FDs under analysis. Thus, we propose a new validation method based on RegExs, which checks how modified data entail some violations, and/or verifies the minimality of a candidate FD.

## 4. The RegEx-based validation method

In this section, we first describe the compressed data structures used to optimize the time and space usage of the discovery algorithm. Then, we introduce the new RegEx-based validation method, and how it can be adopted within an incremental FD discovery process.

	Identif	River	Location	Erected	Purpose	Length	Lanes	Clear-G	T-Or-D	Material	Span	Rel-L	Type
	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)	(K)	(L)	(M)
$t_1$	E22	A	24	1876	Highway	1200	4	G	Through	Wood	Short	S	Wood
$t_2$	E26	M	12	1883	RR	1150	2	G	Through	Steel	Medium	S	Simple-T
$t_3$	E28	M	3	1884	Highway	1000	2	G	Through	Steel	Medium	S	Arch
$t_4$	E31	M	8	1887	RR	1161	2	G	Through	Steel	Medium	S	Simple-T
$t_5$	E34	O	41	1888	RR	4558	2	G	Through	Steel	Long	F	Simple-T
$t_6$	E35	A	27	1890	Highway	1000	2	G	Through	Steel	Medium	F	Simple-T

(a) Before the mapping step.

	Identif	River	Location	Erected	Purpose	Length	Lanes	Clear-G	T-Or-D	Material	Span	Rel-L	Type
	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)	(K)	(L)	(M)
$t_1$	1	1	1	1	1	1	1	1	1	1	1	1	1
$t_2$	2	2	2	2	2	2	2	1	1	2	2	1	2
$t_3$	3	2	3	3	1	3	2	1	1	2	2	1	3
$t_4$	4	2	4	4	2	4	2	1	1	2	2	1	2
$t_5$	5	3	5	5	2	5	2	1	1	2	3	2	2
$t_6$	6	1	6	6	1	3	2	1	1	2	2	2	2

(b) After the mapping step.

Table 1: Snippet of *Bridges* to illustrate validation and discovery strategies.

#### 4.1. Compressed data structures

Before introducing the new validation method, it is necessary to provide details about its data structures. In particular, in order to minimize the memory load, we represent a dataset by using lightweight references to its tuples without losing significance. To do this, we map each attribute data to a unique numeric value, which allows to efficiently identify data changes. In particular, whenever new tuples are added to the dataset, their values are mapped into their corresponding numerical values. This representation leads to a fast tuple comparison during the FD discovery process, and permits to create RegExs in a simple way, avoiding character encoding issues.

**Example 1.** Let us consider the snippet of the *Bridges* dataset in Table 1(a) containing 13 attributes. As shown in Table 1(b), after the mapping phase, each value has been mapped to a unique ID for each attribute. Let us now consider the insertion of the following two tuples in the *Bridges* dataset:

$t_7$	E37	M	18	1891	RR	1350	2	G	Through	Steel	Medium	S	Simple-T
$t_8$	E34	O	41	1888	RR	4558	2	G	Through	Steel	Long	F	Simple-T

then each of these values is parsed according to the previous values of the dataset, and mapped to the following two tuples:

$t_7$	7	2	7	7	2	6	2	1	1	2	2	1	2
$t_8$	5	3	5	5	2	5	2	1	1	2	3	2	2

Although this data representation allows us to quickly retrieve information, it is necessary to define a new structure to support the validation step of candidate FDs. To this end, we exploit a hashmap, namely RegExHashMap, whose keys are unique ID combinations, whereas the associated values correspond to the number of their occurrences in the dataset. In this way, it is possible to avoid duplicate entries and to quickly perform insertion and/or removal operations.

**Example 2.** Let us consider Example 1, where the mapped tuples  $t_7$  and  $t_8$  should be inserted in Table 1(b). Thus, for  $t_7$  the ID “7, 2, 7, 7, 2, 6, 2, 1, 1, 2, 2, 1, 2” is inserted into the RegExHashMap as key, whose associated value is set to 1, since no other tuples in Table 1(b) contains the same values of  $t_7$ . Instead, for  $t_8$  the key “5, 3, 5, 5, 2, 5, 2, 1, 1, 2, 3, 2, 2” is already included into the RegExHashMap, then the associated value is set to 2, since tuple  $t_8$  is equal to  $t_5$ , yielding to two occurrences of the same key.

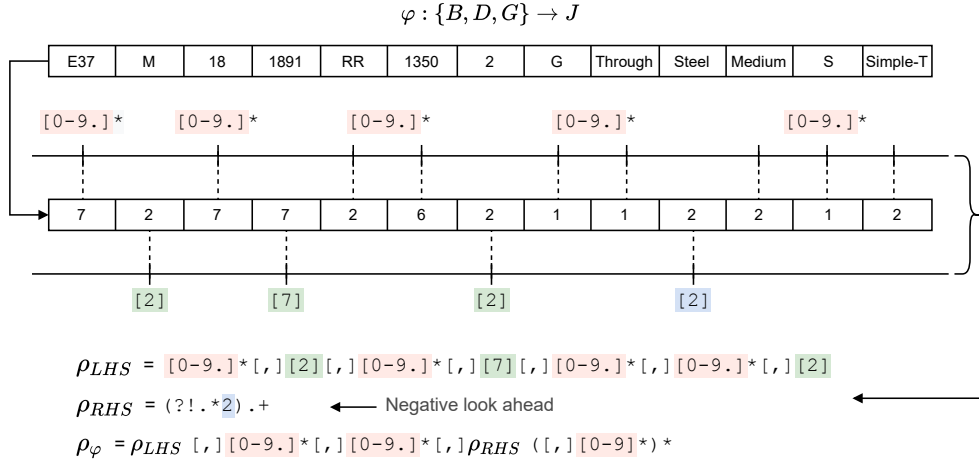
## 4.2. Validation approach

An efficient discovery methodology should permit to quickly validate candidate FDs on the set of data under analysis, and ensure high adaptability to possible data changes. The proposed validation method relies on RegExs and exploits the RegExHashMap for fast retrieval operations. In particular, let  $\varphi : X \rightarrow A$  a candidate FD over a dataset  $D$ . The proposed approach aims to create a RegEx  $\rho$  to validate  $\varphi$  over  $D$ . For sake of clarity, we describe the creation of  $\rho$  by considering a new tuple  $t$ . However, this strategy could be easily adapted to consider a large number of new tuples by chaining different RegExs.

The validation algorithm considers the projection of attributes in  $X$  and  $A$  onto the tuple  $t$  to select value combinations that must be considered into the validation process. More specifically, to validate  $\varphi$ , it is necessary to create a RegEx for the attribute set  $X = X_1, X_2, \dots, X_k$  by considering  $t[X_1], t[X_2], \dots, t[X_k]$ , in the following way:

$$\begin{aligned} \rho_{LHS} &= t[B_1][,]([0-9]^*[,])^*t[B_2][,]([0-9]^*[,])^*\dots([0-9]^*[,])^*t[B_k] \\ \rho_{RHS} &= (?!.^*t[A]).+ \\ \rho_\varphi &= ([0-9]^*[,])^*\rho_{LHS}([,][0-9]^*)^*\rho_{RHS}([,][0-9]^*)^* \end{aligned}$$

We can notice that  $\rho_{LHS}$  contains comma character as separator, whereas each value could be followed from zero or more numeric characters, i.e.,  $[0-9]^*$ , representing all the possible IDs for the attributes that are not in  $\varphi$ . In fact, one of the strengths of this approach is to avoid considering specific values for attributes that must not be considered during the validation step. Similarly to the LHS, it is necessary to define a RegEx for the RHS, i.e.,  $\rho_{RHS}$ . To this end, we can define  $\rho_{RHS}$  as the negative look ahead of  $t[A]$  (i.e., all values not equal to  $t[A]$ ), so enabling to consider only the tuples in which  $t[A]$  does not appear. The combination of  $\rho_{LHS}$  and  $\rho_{RHS}$  allows us to define a new RegEx  $\rho_\varphi$  to validate the candidate  $\varphi$  after the insertion of  $t$ . In this way, it is possible to verify if there exists at least one key in the RegExHashMap satisfying  $\rho_\varphi$ .



**Figure 2:** An example of RegEx creation for *Bridges* dataset.

**Example 3.** Let us consider the snippet dataset in Table 1(b), a candidate FD  $\varphi : \{B, D, G\} \rightarrow J$ , and the new tuple  $t_7$  defined in Example 1. The validation algorithm should verify if this new tuple leads to the invalidation of  $\varphi$ , according to the strategy defined above. To this end, it is necessary to define the RegEx  $\rho_\varphi$  to validate  $\varphi$  shown in Figure 2. We can observe that the validation approach permits to consider the new value  $t_7[B]$ ,  $t_7[D]$ , and  $t_7[G]$  for the attributes  $B$ ,  $D$ , and  $G$ , while it considers a generic numeric value (i.e.,  $[0-9]^*$ ) for the remaining attributes. As a consequence, the regex  $\rho_{LHS}$  has been defined as shown in Figure 2. We can notice that  $\rho_{LHS}$  allows the validation strategy to check if already exists at least one tuple in the dataset with the same value for the attributes  $B$ ,  $D$ , and  $G$ . However, it is necessary to define the regex  $\rho_{RHS}$  for the attribute  $J$  by exploiting the negative look ahead of  $t_7[J]$ , in order to check if exist at least one key in the RegExHashMap with the same value for the attributes  $B$ ,  $D$ , and  $G$ , but a different value for  $J$ .

### 4.3. An incremental search strategy

The proposed validation method can be used in any discovering strategy that, by working incrementally, can properly define FD candidates to be validated. In particular, the algorithm REXY uses a version of the algorithm described in [15], adapted for incrementally managing any kind of data changes. In particular, given a dataset  $D$  at time  $\tau$ , REXY considers the set of FDs holding at time  $\tau$ , denoted as  $\Sigma_\tau$ , as starting points, and collect them in a hash map ordered by LHS cardinality [15]. Then, it follows a bottom-up search strategy by considering  $\Sigma_\tau$  as the set of candidate FDs, and by validating each of them according to the validation method described above. Consequently, given a candidate FD  $\varphi$ , REXY performs a *downward* discovery step when a tuple is removed or  $\varphi$  is a novel candidate (i.e.,  $\varphi \notin \Sigma_\tau$ ). It consists in the generalization of the candidate  $\varphi$  performed by iteratively removing one attribute on its LHS, until REXY meets a valid FD in the lower levels. Conversely, when a new tuple is added and  $\varphi$  is not valid, an *upward* discovery step is accomplished in order to find the possible new holding FDs. It consists

in the specialization of  $\varphi$  performed by adding a new attribute on its LHS.

The usage of an ordered hash map into a bottom-up search strategy permits to reuse the analysis performed at the previous levels, and forward the validation of newly generated candidate FDs to the next level, yielding an optimization in the pruning of the search space.

## 5. The REXY Algorithm

The main procedure of REXY is shown in Algorithm 1. Given a set of minimal FDs  $\Sigma_\tau$  valid at time  $\tau$ , the set of tuples  $D_{\tau+1}$  updating  $D$  at time  $\tau + 1$ , and an instance of the RegExHashMap  $\Lambda_\tau$  at time  $\tau$ , the main procedure of REXY starts the discovery process by considering  $\Sigma_\tau$  as the set of candidate FDs. In particular, REXY performs a discovery step in ascending order by the LHS cardinalities of  $\Sigma_\tau$  (line 3) and extracts all the candidate FDs from  $\Sigma_\tau$  with a specific LHS cardinality (line 4). Then, for each of them, it checks if the candidate  $\varphi : X \rightarrow A$  is still valid after the updating of the dataset according to the validation approach defined in Section 4.2 (line 6). If the analyzed FD is not valid at time  $\tau + 1$ , the procedure first removes it from the set of the analyzed candidates (line 7), and then generates new candidate FDs at a higher lattice level (line 8), by discarding those that can be inferred from other FDs already validated at time  $\tau + 1$  (lines 9-10). On the other hand, if the analyzed FD is still valid at time  $\tau + 1$ , REXY tries to find if there exist other FDs at a lower lattice level (line 12) that have been validated after update operations (lines 13-18). At the end of each iteration, the procedure removes all the FDs that are not minimal at time  $\tau + 1$  w.r.t. the FDs already validated in the previous iterations (line 19). Finally, RegExHashMap is updated with the new tuples (line 20), and the procedure returns a new set of minimal and valid FDs at time  $\tau + 1$  (line 21).

The validation procedure of REXY is shown at the bottom of Algorithm 1. Given a candidate FD  $\varphi : X \rightarrow A$  at time  $\tau + 1$ , an instance of the RegExHashMap  $\Sigma_\tau$  at time  $\tau$ , and the set of the new tuples  $D_{\tau+1}$  at time  $\tau + 1$ , the procedure creates a new RegEx for each new tuple in  $D_{\tau+1}$ , in order to check the validation of the candidates on the updated dataset. In particular, the procedure starts by analyzing each new tuple in  $D_{\tau+1}$  (line 24). Then, for each of them, REXY defines a new RegEx according to the approach defined in Section 4.2. More specifically, for each attribute  $B$  in  $attr(R)$ , if  $B$  does not belong to the attributes of  $\varphi$ , the procedure adds to  $\rho_\varphi$  a generic value (lines 28-30). On the other hand, if  $B$  belongs to the attributes on the LHS, then the procedures add to the final RegEx  $\rho_\varphi$  the corresponding value for the analyzed tuple  $t[B]$  (line 32). Otherwise, if none of the previous cases is verified, the attribute  $B$  belongs to the RHS, so the procedure adds to  $\rho_\varphi$  the negative look ahead of this value (line 33). After the creation of the RegEx  $\rho_\varphi$ , REXY checks if exists at least a tuple in the RegExHashMap that matches this RegEx. If true, it means that  $\varphi$  is not valid on the dataset at time  $\tau + 1$ , since there exists at least one pair of tuples that invalidate  $\varphi$  (line 35). Otherwise, if the previous case is never verified for any other tuple, it means that the candidate is valid at time  $\tau + 1$  (line 36).

It is worth notice that, while the implementation of REXY considers several code optimizations and takes advantage of the defined data structures, for sake of clarity, the pseudo-codes of Algorithm 1 do not show such optimizations. They mainly guarantee that each possible candidate FD is validated at most once.



---

**Algorithm 1: Main Procedures of REXY**

---

**Input:** A set  $\Sigma_\tau$  of minimal FDs at time  $\tau$ ; Updated tuples  $D_{\tau+1}$  for the dataset  $D$  at time  $\tau + 1$ ; An instance of the RegExHashMap  $\Gamma_\tau$  at time  $\tau$

**Output:** The set of new minimal FDs at time  $\tau + 1$ , i.e.,  $\Sigma_{\tau+1}$

```
1 Function MAIN_PROCEDURE:
2    $\Sigma_{\tau+1} \leftarrow \emptyset$ 
3   for  $c$  in  $[1, \dots, |attr(R)|]$  do
4      $\Sigma_c \leftarrow \Sigma_\tau.CARDINALITY\_FILTER(c)$ 
5     for each  $X \rightarrow A \in \Sigma_c$  do
6       if  $VALIDATION\_REGEX(X \rightarrow A, \Gamma_\tau, D_{\tau+1})$  is False then
7          $\Sigma_c \leftarrow \Sigma_c \setminus X \rightarrow A$ 
8          $\Sigma_s \leftarrow SPECIALIZE\_CANDIDATE(X \rightarrow A)$ 
9         for each  $\{Z \rightarrow A\} \in \Sigma_s$  do
10          if  $IS\_MINIMAL(\{Z \rightarrow A\}, \Sigma_\tau)$  is True then
11             $\Sigma_\tau \leftarrow \Sigma_\tau \cup \{Z \rightarrow A\}$ 
12        else
13           $\Sigma_g \leftarrow GENERALIZE\_CANDIDATE(X \rightarrow A)$ 
14          for each  $\{W \rightarrow A\} \in \Sigma_g$  do
15            if  $VALIDATION\_REGEX(\{W \rightarrow A\}, \Gamma_\tau, D_{\tau+1})$  is False then
16               $\Sigma_g \leftarrow \Sigma_g \setminus \{W \rightarrow A\}$ 
17               $\Sigma_g \leftarrow \Sigma_g \cup GENERALIZE\_CANDIDATE(\{W \rightarrow A\})$ 
18             $\Sigma_c \leftarrow \Sigma_c \cup \Sigma_g$ 
19        if  $|\Sigma_c| > 0$  then  $\Sigma_{\tau+1} \leftarrow \Sigma_{\tau+1} \cup MINIMALITY\_CHECK(\Sigma_{\tau+1}, \Sigma_c)$ 
20     $\Gamma_{\tau+1} \leftarrow \Gamma_{\tau+1} \cup D_{\tau+1}$  // Updating of the RegExHashMap
21    return  $\Sigma_{\tau+1}$ 
22 Function  $VALIDATION\_REGEX(X \rightarrow A, \Gamma_\tau, D_{\tau+1})$ :
23    $\rho_\varphi \leftarrow \emptyset$ 
24   for each  $t \in D_{\tau+1}$  do
25      $ATTRS \leftarrow attr(R)$ 
26     for  $c$  in  $[1, \dots, |attr(R)| - 1]$  do
27        $B \leftarrow ATTRS[c]$ 
28       if  $B \notin X \wedge B \neq A$  then
29         if  $i == 0$  then  $\rho_\varphi \leftarrow \rho_\varphi \cup ([0 - 9]^*[ , ])^*$ 
30         else  $\rho_\varphi \leftarrow \rho_\varphi \cup ([ , ] [0 - 9]^*)^*$ 
31       else
32         if  $B \in X$  then  $\rho_\varphi \leftarrow \rho_\varphi \cup t[B]$ 
33         else if  $B \in A$  then  $\rho_\varphi \leftarrow \rho_\varphi \cup (?!.^*t[B]).+$ 
34         if  $i < |ATTRS| - 1$  then  $\rho_\varphi \leftarrow \rho_\varphi \cup [ , ]$ 
35       if  $\Gamma_\tau.EXIST\_MATCHING(\rho_\varphi)$  is True then return False
36   return True
```

---

## 6. Experimental results

REXY has been developed in Java 15. The execution tests have been performed on an iMac with an Intel Xeon processor at 3.40 GHz, 36-core, and 128GB of RAM. Furthermore, in order

#	Dataset	Cols [#]	Rows [#]	FDs [#]	Exec (Avg) [ms]	Validations [#]	Validations (Min) [ms]	Validations (Max) [ms]	Validations (Avg) [ms]	Memory [MB]
1	Iris	5	150	4	1	766	11	12	12	95
2	Balance-scale	5	625	1	1	1330	8	38	13	100
3	Bupa	6	345	16	18	6979	11	36	14	120
4	Appendicitis	7	107	72	39	7890	9	37	23	116
5	Chess	7	2000	6	2	17142	13	13	13	39
6	Abalone	9	4148	137	335	705498	12	33	18	114
7	Nursey	9	12960	1	82	59137	5	18	16	121
8	Tic-tac-toe	9	958	9	28	14613	5	6	5	122
9	Glass	10	214	123	140	28138	14	21	18	117
10	Cmc	10	1473	1	13	17832	20	31	24	138
11	Yeast	10	1484	50	12	95531	11	11	11	46
12	Breast-cancer	11	699	46	175	56269	8	15	11	66
13	Fraud-detection	11	28000	48	1482	1723242	5	230	207	414
14	Poker-hand	11	264000	1	19	2401240	9	12	10	156
15	Echocardiogram	13	132	538	160	65932	8	13	8	59
16	Bridges	13	108	142	153	34017	11	31	11	94
17	Australian	14	690	535	374	501588	10	26	12	124
18	Adult	15	32560	60	281	3747386	8	23	12	803
19	Tax	15	6848	310	7038	2378734	10	532	287	202
20	Lymphography	19	147	2730	71844	5156312	14	34	20	4921
21	Hepatitis	20	155	8250	194553	4528681	11	170	126	48878
22	Parkinson	24	195	1724	524195	874789	16	16470	10492	127

Table 2: Details of the considered real-world datasets and REXY performances.

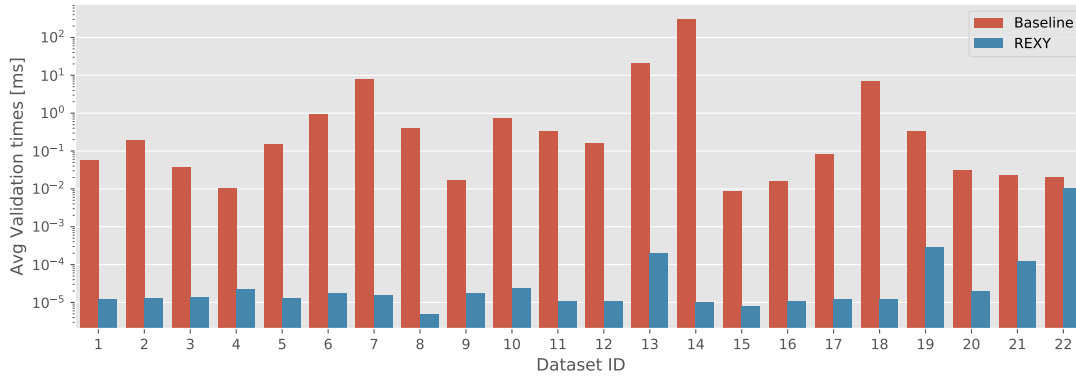
to ensure proper executions of REXY on the considered datasets, the Java memory heap size allocation has been set to 100GB.

Table 2 summarizes the time performances of REXY on 22 real-world datasets varying in the number of rows and columns<sup>1</sup>. In our experimental session, we simulate a scenario of continuous insertions of tuples by considering one tuple at a time. This allowed us to deeply analyze the performances of the validation process on each candidate FD. More specifically, Table 2 reports the minimum, the maximum, and the average times of the validation process for each dataset. We also report the number of resulting FDs and the number of validations performed at the end of the last execution of REXY on each dataset.

The results highlight that the average execution time is almost always less than 10 seconds, except for the *Lymphography*, *Hepatitis*, and *Parkinsons* datasets. This is probably due to the large number of FD invalidated during the discovery process, possibly leading to a larger number of new candidates to be validated. Concerning the validation process, the average time of this process is almost always less than 25 milliseconds per FD, and the maximum time almost never exceeds 40 milliseconds, except for the *Fraud-detection*, *Tax*, *Hepatitis*, and *Parkinsons* datasets. Despite these time peaks, the average times demonstrate that such values can be considered as outliers. In general, although execution times increase when the number of attributes increases, on average they remain low for validating each candidate FD.

We performed a comparative evaluation of REXY with respect to the incremental FD discovery algorithm presented in [9] (baseline). In particular, both the algorithms follow the same search strategy, but differ on how they organize data, also yielding to completely different validation methods. Thus, the comparison allowed us to highlight the improvements in terms of execution times of the proposed validation strategy with respect to the traditional partition-based used in

<sup>1</sup>The datasets are available on: <https://github.com/DastLab/TestDataset>



**Figure 3:** Validation times of REXY and the baseline algorithm proposed in [9].

[9]. Figure 3 shows time performances on average of both algorithms on the 22 datasets. We can notice that, REXY outperforms the baseline algorithm with several orders of magnitude, ranging from 2 to 7, except for the *Parkinson* dataset for which the baseline takes advantage from a caching strategy that allows to reuse the computation performed in the previous iterations.

In general, partition-based validation strategies, such as the one included in [9], are particularly efficient when a complete discovery process has to be performed, since partitions can be incrementally computed following a level-by-level bottom-up search strategy. However, the incremental scenario requires the discovery process to browse the search space starting from the previously holding FDs (i.e., not all FD candidates have to be considered in each level). Thus, the proposed validation method is able to perform the verification without having the necessity to compute sparse partitions, yielding faster execution times as can be noted in the discussed experimental results.

## 7. Conclusion

In this paper we presented REXY, an incremental algorithm for FD discovery. It exploits a new Regex-based validation method to efficiently select tuples affected by dataset updates. REXY follows an incremental strategy to explore the search space based on the previously holding FDs. Experimental results over real-world datasets show that REXY can efficiently update the set of holding FDs, without having the necessity to execute algorithms from scratch. In the future, we would like to include REXY in a visual monitoring tool [16], and extend it for updating Relaxed Functional Dependencies [17] in incremental scenarios.

## References

- [1] X. Chu, I. F. Ilyas, P. Papotti, Holistic data cleaning: Putting violations into context, in: Proc. of IEEE International Conference on Data Engineering, 2013, pp. 458–469.
- [2] M. Le Guilly, J.-M. Petit, V.-M. Scuturici, Evaluating classification feasibility using func-

- tional dependencies, in: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLIV*, Springer, 2020, pp. 132–159.
- [3] Z. Abedjan, P. Schulze, F. Naumann, DFD: Efficient functional dependency discovery, in: *Proc. of the 23rd ACM International Conference on Information and Knowledge Management*, 2014, pp. 949–958.
  - [4] P. A. Flach, I. Savnik, Database dependency discovery: A machine learning approach, *AI communications* 12 (1999) 139–160.
  - [5] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen, TANE: An efficient algorithm for discovering functional and approximate dependencies, *The Computer Journal* 42 (1999) 100–111.
  - [6] P. Schirmer, T. Papenbrock, S. Kruse, D. Hempfing, T. Meyer, D. Neuschäfer-Rube, F. Naumann, DynFD: Functional dependency discovery in dynamic datasets., in: *Proc. of 22nd International Conference on Extending Database Technology*, 2019, pp. 253–264.
  - [7] S.-L. Wang, J.-W. Shen, T.-P. Hong, Incremental discovery of functional dependencies using partitions, in: *Proc. of Joint 9th IFSA World Congress and 20th NAFIPS International Conference*, volume 3, 2001, pp. 1322–1326.
  - [8] C. Wyss, C. Giannella, E. Robertson, FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances, in: *Proc. of International Conference on Data Warehousing and Knowledge Discovery*, 2001, pp. 101–110.
  - [9] L. Caruccio, S. Cirillo, V. Deufemia, G. Polese, Incremental discovery of functional dependencies with a bit-vector algorithm, in: M. Mecella, G. Amato, C. Gennaro (Eds.), *Proceedings of the 27th Italian Symposium on Advanced Database Systems*, volume 2400 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019.
  - [10] H. Yao, H. J. Hamilton, C. J. Butz, FD\_Mine: Discovering functional dependencies in a database using equivalences, in: *Proc. of IEEE International Conference on Data Mining*, 2002, pp. 729–732.
  - [11] T. Papenbrock, F. Naumann, A hybrid approach to functional dependency discovery, in: *Proc. of the 2016 International Conference on Management of Data*, 2016, pp. 821–833.
  - [12] Z. Wei, S. Link, Discovery and ranking of functional dependencies, in: *Proc. of IEEE 35th International Conference on Data Engineering*, IEEE, 2019, pp. 1526–1537.
  - [13] S. Bell, Discovery and maintenance of functional dependencies by independencies, in: *Proc. of the 1st International Conference on Knowledge Discovery and Data Mining*, 1995, pp. 27–32.
  - [14] L. Caruccio, V. Deufemia, G. Polese, Mining relaxed functional dependencies from data, *Data Mining and Knowledge Discovery* 34 (2020) 443–477.
  - [15] L. Caruccio, S. Cirillo, Incremental discovery of imprecise functional dependencies, *J. Data and Information Quality* 12 (2020).
  - [16] B. Breve, L. Caruccio, S. Cirillo, V. Deufemia, G. Polese, Visualizing dependencies during incremental discovery processes, in: A. Poulouvasilis et al. (Ed.), *Proceedings of the Workshops of the EDBT/ICDT 2020 Joint Conference*, volume 2578 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020.
  - [17] L. Caruccio, V. Deufemia, F. Naumann, G. Polese, Discovering relaxed functional dependencies based on multi-attribute dominance, *IEEE Transactions on Knowledge and Data Engineering* (2021) To appear.