Incremental Discovery of Imprecise Functional Dependencies

LOREDANA CARUCCIO, University of Salerno
STEFANO CIRILLO, University of Salerno

Functional dependencies (FDs) are one of the metadata used to assess data quality and to perform data cleaning operations. However, in order to pursue robustness with respect to data errors, it has been necessary to devise imprecise versions of functional dependencies, yielding relaxed functional dependencies (RFDs). Among them, there exists the class of RFDs relaxing on the extent, i.e., those admitting the possibility that an FD holds on a subset of data. In the literature several algorithms to automatically discover RFDs from big data collections have been defined. They achieve good performances with respect to the inherent problem complexity. However, most of them are capable of discovering RFDs only by batch processing the entire dataset. This is not suitable in the era of big data, where the size of a database instance can grow with high-velocity, and the insertion of new data can invalidate previously holding RFDs. Thus, it is necessary to devise incremental discovery algorithms capable of updating the set of holding RFDs upon data insertions, without processing the entire dataset. To this end, in this paper we propose an incremental discovery algorithm for RFDs relaxing on the extent. It manages the validation of candidate RFDs and the generation of possibly new RFD candidates upon the insertion of the new tuples, while limiting the size of the overall search space. Experimental results show that the proposed algorithm achieves extremely good performances on real-world datasets.

CCS Concepts: • **Information systems** → *Inconsistent data*; *Information extraction*; Data cleaning.

Additional Key Words and Phrases: Functional Dependency, Discovery Algorithm, Tuple Insertions, Incremental Discovery, Parallelism.

ACM Reference Format:
Loredana Caruccio and Stefano Cirillo. 2019. Incremental Discovery of Imprecise Functional Dependencies. *ACM J. Data Inform. Quality* 1, 1, Article 1 (January 2019), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1

1 INTRODUCTION

Data profiling is the process of examining datasets in order to extract metadata from them [26]. Among such metadata, Functional Dependencies (FDs) play a fundamental role, since they are usually employed to assess and improve the quality of data and/or to facilitate advanced database operations, such as query optimization [15], query rewriting [8], schema design [14], and so forth. However, in several application contexts the FDs definition is too rigid to be used in practical settings. For this reason, the canonical FD definition has been extended in order to introduce approximations,

[1]This is a post-peer-review, pre-copyedit version published inJournal of Data and Information Quality (JDIQ), ACM. 12(4): 1-25 (2020). The final authenticated version is available online at: https://doi.org/10.1145/3397462

Authors' addresses: Loredana Caruccio, lcaruccio@unisa.it, University of Salerno, via Giovanni Paolo II, 132, Fisciano (SA), Italy, 84084; Stefano Cirillo, scirillo@unisa.it, University of Salerno, via Giovanni Paolo II, 132, Fisciano (SA), Italy, 84084.

1936-1955/2019/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Published in: Journal of Data and Information Quality (JDIQ).
Copyright © held by the Association for Computing Machinery, Inc. (ACM).
Authors version
The publisher version is available at https://dl.acm.org/doi/10.1145/3397462

yielding the definition of relaxed functional dependency (RFD) [6]. The latter introduces two main relaxation criteria into the canonical FD definition, such as the possibility to have approximate methods (similarity operators, order relations, and so forth) to compare attribute values (relaxation on the *attribute comparison*), or the possibility to have FDs holding on a subset of tuples (relaxation on the *extent*).

In the rest of the paper we will refer to RFDs relaxing on the extent as RFD_es. Examples of RFD_es include Approximate Functional Dependencies (AFDs) [17] Purity dependencies [35]. They mainly differ in the measure used to express the portion of the dataset on which the dependency holds, also known as coverage measure.

RFDs result particularly useful in contexts in which the presence of outliers into the data should be tolerated. As an example, in a database describing medicine characteristics there is the possibility to have an FD ActiveIngredient → Prescribed, representing the fact that whenever two medicines share the same active ingredient, then either both or none of them should require the medical prescription. However, in the real-world there can be few cases in which such an FD does not hold, whereas an RFD_e could still capture such correlation, by admitting some exceptions.

The number of FDs or RFD_es holding on a given database instance could be exponential with respect to the number of its columns. This makes the discovery problem an extremely complex one. To this end, most of the discovery algorithms described in the literature aim to provide solutions in which the search space complexity is reduced by exploiting some theoretical properties of FDs (RFD_es). Moreover, with respect to the validation of RFD_es, the computation of a coverage measure is necessary. Although several discovery algorithm described in the literature achieved good performances, most of them are not suitable in dynamic scenario, in which new data are frequently added to the dataset, like for instance in modern big data contexts, in which the datasets can grow extremely rapidly. This entails the necessity to define new incremental RFD_e discovery algorithms, capable of updating the set of RFD_e holding on a dataset upon the insertion of new data, without having to re-compute them on the entire dataset.

In this paper we propose a new discovery algorithm, named BIRD (Bit-vector based Incremental RFD_e Discoverer), which incrementally updates the set of holding RFD_es whenever new tuples are inserted into the database instance. Its start point is a set of RFD_e candidates holding on the dataset prior the insertion operations. Therefore, BIRD analyzes how new inserted tuples impact on such RFD_e candidates, checking whether they invalidate some of them, and possibly generating new candidates. Moreover, BIRD exploits effective data structures and an efficient execution strategy, which permit to split the discovery process into a level-wise parallel execution.

The main contributions of the paper are: 1) a complete analysis of the incremental RFD_e discovery problem; 2) BIRD: a new algorithm for incremental RFD_e discovery; and 3) an extensive experimental evaluation demonstrating the efficiency of our RFD_e discovery technique.

The rest of the paper is organized as follows. Section 2 reviews several discovery algorithms existing in the literature. Section 3 provides some background definitions concerning FDs and RFDs, whereas Section 4 formulates the incremental RFD_e discovery problem. The complete discovery methodology and the BIRD algorithm is presented in Section 5, whereas the experimental evaluation is reported in Section 6. Finally, summary and concluding remarks are included in Section 7.

2 RELATED WORK

In this section, we review methods proposed in the literature for both FD and RFD_e discovery. Moreover, we also describe several approaches focusing on incremental discovery paradigms.

FD *discovery.* The problem of discovering FDs dates back to the '80s, when some pioneer discovery algorithms were defined [24]. It is an extremely complex problem since it requires to consider a huge

number of column combinations in order to detect a possibly exponential number of holding FDs [1]. In the literature, there exist two main categories of FD discovery approaches, the column-based or the row-based ones. The former exploits an attribute lattice to generate candidate FDs, which are successively tested to verify their validity. The whole process is made efficient by exploiting several pruning rules, which permit to prune the search space based on previously discovered valid FDs. Examples of column-based approaches include the algorithms TANE [17], FD_Mine [39], FUN [27], and DFD [2]. Instead, row-based methods derive candidate FDs from two attribute subsets, namely *agree-set* and *difference-set*, which are built by comparing the values of attributes between all possible combinations of tuples pairs. Examples of bottom-up approaches include the algorithms DepMiner [23], FastFD [38], and FDep [12]. Experimental results show that the selection of a proper strategy can depend on the intrinsic characteristics of the dataset. In fact, column-based algorithms usually outperform row-based ones on datasets with many rows and few columns, whereas the row-based ones perform better on datasets with few rows and many columns [28]. Recently, a hybrid algorithm (HyFD) has been proposed in order to achieve a suitable performance trade-off in all cases [30]. 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 (row-based), and the other in which it validates the discovered FDs on the entire dataset. Moreover, an extension of HyFD has been recently defined [37], which extends the data structure used in HyFD, namely the FD-tree, introducing a new induction method in order to remove redundant node traversals on the lattice. Finally, several distributed versions of well-known FD discovery approaches have been defined and evaluated in [32, 33].

RFD$_e$ discovery. RFD$_e$s are FDs that must hold on 'almost' all the tuples on a relation instance r [20]. In other words, RFD$_e$s admit the possibility for a limited number of tuples in r to violate a given FD. Thus, they are validated through a coverage measure, which represents either the portion of the dataset on which an FD holds, or the one on which it is not valid. RFD$_e$s are also known in the literature with the name *Approximate Functional Dependencies* (AFDs) or *Partial Dependencies*. Among the proposed coverage measures [13], the $g3$-error is the most frequently used [20]. Most of the approaches for discovering RFD$_e$s rely on sampling [18, 20], which consists in verifying whether a candidate RFD$_e$ holds on an instance r by validating it on a sample of tuples $s \subset r$, meaning that it will hold on r with a given probability. A novel and efficient algorithm for RFD$_e$ discovery is Pyro [21]. It aims to find minimal RFD$_e$s by exploiting both samples of agree sets, and a lattice traversal strategy, which uses efficient data structures to estimate and calculate the error of RFD$_e$ candidates. Pyro also uses parallelism in order to speed-up the runtime. Instead, the method proposed in [19] exploits the error measure of super keys to determine the extent of RFD$_e$s. Finally, to cope with the complexity of the RFD$_e$ discovery problem, there are some approaches limiting the discovery only to meaningful RFD$_e$s, yielding the loss of some minimal RFD$_e$s holding on the database instance [7, 31]. Finally, the algorithm proposed in [7] permits to detect more general RFDs, i.e. those relaxing on both the attribute comparison and the extent.

Incremental discovery. The discovery algorithms presented above need to process the whole dataset. Thus, whenever the datasets are updated, they need to be re-executed from scratch, whereas it would be desirable to have some incremental discovery strategies. In what follows, we review some discovery algorithms defined for dynamic datasets. One of the first theoretical proposal of incremental algorithm for FD discovery has been provided in [36]. It exploits the concepts of tuple partitions and monotonicity of FDs to avoid the re-scanning of the entire database. Another proposal is based on the concept of functional *independency* through which it maintains the set of FDs updated over time [4]. Finally, in order to discover and maintain FDs in dynamic datasets, the DynFD algorithm has been proposed [34]. It continuously adapts the validation structures of

FDs in order to evolve them with batch of inserts, updates, and deletes of data. Moreover, another approach for discovering *Order Dependencies* (ODs) after the insertion of new tuples is defined in [40]. It is based on a strategy enabling an intelligent traversal process and reduced access to the whole dataset. Finally, the only incremental algorithm for the discovery of RFD$_e$s proposed in the literature is AD-Miner [11]. It permits to incrementally update dependency information exploiting several logical operations. The problem of incrementality has also been addressed in the context of association rule mining[25], which is somehow related to RFD$_e$s discovery problem. In particular, an association rule holding on a given dataset is defined in terms of *support* and *confidence* measures. In the literature, there exist two different types of incremental algorithms for association rule mining: single-objective[9] and multi-objective[10]; they both use the confidence as the main measure to be optimized. However, multi-objective algorithms also attempt to optimize other measures, such as comprehensibility and interestingness. Similarly to those for incremental RFD$_e$s discovery, incremental algorithms for association rule mining start using information gathered from previous executions, trying to reduce the search space. In general, the main strategy used to tackle the incrementality problem prescribes to verify how rules change after updating the data, and when it is instead necessary to re-scan the entire dataset in order to find new valid rules. More specifically, if a rule is valid on the old dataset, and it is still valid on the new increment, then it is not necessary to re-scan the entire dataset. Instead, if a rule is no longer valid, then it is necessary to execute a scan process on the entire dataset.

3 PRELIMINARES

In order to introduce the concept of FDs and RFD$_e$s let us review some basic concepts and notations.

A relational database schema $\mathcal{R}$ is a collection of relation schemas (R_1,. . ., R_n), where each R_i is defined over a fixed set of attributes $attr(R_i)$. Each attribute A_k has associated a domain $dom(A_k)$, which can be finite or infinite. A relation instance (or simply a relation) r_i of R_i is a set of tuples t such that, for each attribute $\forall\, t \in r_i$, $A_k \in attr(R_i)$, $t[A_k] \in dom(A_k)$, where denotes the projection $t[A_k]$ of a tuple t onto the attribute A_k. A database instance r of $\mathcal{R}$ is a collection of relations (r_1,. . .,r_n), where r_i is a relation instance of R_i, for $i \in [1, n]$.

Definition 3.1. Given a relational database schema $\mathcal{R}$ defined over a set of attributes $attr(\mathcal{R})$, an FD φ over $attr(\mathcal{R})$ is a statement $X \rightarrow Y$ (X implies Y) with $X, Y \subseteq attr(\mathcal{R})$, such that, given an instance r of $\mathcal{R}$, $X \rightarrow Y$ holds 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 $t_1[Y] = t_2[Y]$. In this context, the two sets X and Y are also called Left-Hand-Side (LHS) and Right-Hand-Side (RHS), resp., of φ.

RFDs extend FDs by relaxing some constraints of their definition. One class of RFDs are those relaxing on the *extent*, i.e. a dependencies holding for “almost" all tuples or for a “subset" of them [6]. In order to quantify “almost", a *coverage measure* should be specified to quantify either the portion of the dataset on which the dependency holds or does not hold. Instead, in order to identify the “subset" of tuples on which the dependency holds, a condition on the attribute domains should be specified. In this paper, we focus on RFDs holding for “almost" all tuples, which we call RFD$_e$s.

Definition 3.2. Given a database instance r of $\mathcal{R}$, and $X, Y \subseteq attr(\mathcal{R})$, LHS and RHS, respectively, of φ, a coverage measure Ψ on φ can be defined as a function $\Psi : dom(X) \times dom(Y) \rightarrow \mathbb{R}^+$ measuring the amount of tuples in r violating or satisfying φ. As an example, the *g3*-error evaluates the minimum number of tuples to be removed in order to make φ valid on the remaining dataset [17]. Although several coverage measures can be used, most of them usually return a value normalized on the total number of tuples, producing a value $v \in [0, 1]$.

By considering the possibility to include the above mentioned relaxation criterion in the FD definition, it is possible to consider a general definition for RFD$_e$s. The most general RFD definition that also considers other possible relaxation criteria can be found in [6].

Definition 3.3. Let us consider a relational database schema $\mathcal{R}$, an RFD$_e$ φ on $\mathcal{R}$ is denoted as

$$X \xrightarrow{\Psi \leq \varepsilon} Y \tag{1}$$

where

- $X, Y \subseteq attr(\mathcal{R})$, with $X \cap Y = \emptyset$;
- Ψ is a coverage measure defined on $dom(X) \times dom(Y)$.
- ε is a threshold indicating the upper bound (or lower bound in case the comparison operator is $\geq$) for the result of the coverage measure.

Given a relation instance r of $\mathcal{R}$. the RFD$_e$ φ holds on r, denoted by $r \models \varphi$, if and only if: $\forall (t_1, t_2) \in r$, if $t_1[X] = t_2[X]$, then *almost always* $t_1[Y] = t_2[Y]$. Here, *almost always* means that $\Psi \leq \varepsilon$. In fact, when Ψ is an error measure, then ε must be small to model almost all. Instead, when Ψ is a satisfiability measure, then the notation $\Psi \geq \varepsilon$ is used and ε should be big.

Notice that, for sake of simplicity and without loss of generality, we can consider RFD$_e$s with a single attribute on the RHS: $X \xrightarrow{\Psi \leq \varepsilon} A$. According to this formalization, an RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ is *non-trivial* if $A \notin X$, and it is *minimal* if there exists no attribute $B \in attr(\mathcal{R})$ such that $X \backslash B \xrightarrow{\Psi \leq \varepsilon} A$ is valid by considering the same error threshold bound ε.

In the proposed algorithm we consider one among the most frequently used coverage measures, i.e. the $g3$-error, representing the minimum fraction of tuples to be removed from a relation instance in order to make the dependency hold on the remaining tuples, yielding an RFD$_e$. More formally, let r be an instance of a relation schema R, and $X \rightarrow A$ be a candidate RFD$_e$, then the $g3$-error coverage measure can be defined as follows

$$\Psi = \frac{\min\{|r_1| \text{ s.t. } r_1 \subseteq r \text{ and } X \rightarrow A \text{ holds on } r \backslash r_1\}}{|r|} \tag{2}$$

One of the most efficient algorithms to compute the $g3$-error is defined in [16]. Given a candidate $X \rightarrow A$, and an instance r of R, it considers an efficient representation for value combinations of attributes involved into the candidate dependency, named *partition*. Given an attribute set X, a partition π_X is the collection of equivalence classes of the projection of tuples in r onto the attributes in X. In particular, given the candidate $X \rightarrow A$, the algorithm computes π_X and $\pi_{X \cup A}$, and for each equivalence class in π_X computes the maximum number of its tuples that are included in the same equivalence class of $\pi_{X \cup A}$. Summing these numbers over all the equivalence classes of π_X yields the cardinality of the biggest subset of r on which the dependency holds. Thus, the $g3$-error is computed according to the following formula [16]:

$$\Psi = 1 - \frac{\Sigma_{c \in \pi_X} \max\{|c'| \text{ s.t. } c' \in \pi_{X \cup A} \text{ and } c' \subseteq c\}}{|r|} \tag{3}$$

Example 3.4. As an example, it is likely to have the same prescription requirement for medicines having the same active ingredient. Thus, the following FD might hold:

$$\text{ActiveIngredient} \rightarrow \text{Prescribed}$$

On the other hand, in the real world there could be some exceptions, such as the medicines Oki and Ketodol, since even if they share the same active ingredient, i.e.Ketoprofen, only Ketodol

requires the prescription. For this reason, the previous FD should also admit exceptions. This can be modelled by introducing a coverage measure so that the following RFD_e holds:

$$\text{ActiveIngredient} \xrightarrow{\Psi \leq 0.10} \text{Prescribed}$$

which admit that the dependency might not hold for 10% of tuples. This means that we should remove 10% of tuples from the instance so that the dependency is valid.

4 THE DISCOVERY PROBLEM

The goal of dependency discovery is to find meaningful FDs holding between the columns of a dataset. They represent domain knowledge and can be used to assess database design and data quality [22]. Moreover, discovering dependencies from data permits to capture the evolution of big datasets within data analytics scenarios, in which new data are often added over the time.

The general RFD_e discovery problem is described in the following definition.

Definition 4.1. Given a relation instance r of a relation schema R, an RFD_e discovery algorithm aims to find all the minimal non-trivial RFD_e holding on r, that is, whenever two tuples are equal on the LHS *almost always* they must be equal on the RHS, where *almost always* is measured by $\Psi \leq \varepsilon$, with ε defined in input.

Given a set Σ of all minimal RFD_es holding on r at time τ, it is necessary the discovery algorithm must update Σ whenever one or more tuples are added between time τ and time $\tau + 1$, since new tuples can invalidate one or more RFD_es of Σ, or generate new ones. Details on how we deal with such a problem are discussed into next sections.

Discovering RFD_es over a relation instance r has the same theoretical complexity of the FD discovery problem. In fact, it entails finding all the possible column combinations, and for each of them find all the possible partitions forming the LHS and RHS of candidate RFD_es. More formally, given a relation instance 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, aiming to account for the number of different candidates with a single RHS attribute. Moreover, we need to consider the complexity of the validation of each candidate RFD_e. Thus, the problem complexity's upper bound is $O(n^2(\frac{m}{2})^2 2^m)$, where $\frac{m}{2} \cdot 2^m$ represents the number of candidates, and $\frac{m}{2} \cdot n^2$ is the complexity of a brute-force approach for the validation, where all pairs of tuples are compared on an average $\frac{m}{2}$ columns, i.e. the average number of attributes involved into an RFD_e [22, 28].

4.1 Column-based strategy

In this section, we describe the fundamentals of the column-based strategy for the discovery of FDs, which represents the baseline on which we defined the proposed RFD_e discovery algorithm. It follows the well-known APRIORI strategy for the generation of candidate dependencies [16].

More specifically, the column-based strategy models the search space as a graph representation of a lattice, which is partitioned into levels where level L_i contains all attribute combinations of size i. Each node in the lattice represents a unique set of attributes, and it is linked to nodes that contain a direct superset or subset of attributes. In other words, each edge refers to the inclusion relation between two attribute sets. Thus, a lattice permits to consider candidate RFDs at each level in terms of lattice's edges, allowing to represent the LHS and the RHS of an RFD [29]. More formally, let $R = A_1, \ldots, A_m$ be a relation schema with m attributes. The corresponding graph representation of lattice will contain a collection of attribute sets, where *Level 0* contains 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. A lattice edge links two attribute sets on two adjacent lattice levels. For instance, let AB and ABC be attribute sets on *Level 2* and *Level 3*, respectively, then edge $e(AB, ABC)$ represents the candidate $AB \rightarrow C$.

After the generation of candidates at each level, it is necessary to validate the RFD_e candidates associated to them, based on a coverage measure and the associated threshold ε provided in the input (see equation (1)).

Finally, column-based strategy employs several pruning rules during the discovery process in order to reduce the search space. In general, they permit to discard candidate RFD_es that are not minimal with respect to the already discovered ones. The possibility to apply pruning rules is of vital importance since as defined above the general RFD_e discovery problem is an extremely complex one.

4.2 Incremental discovery issues

In the previous section, we illustrated the complexity of the RFD_e discovery problem and the fundamentals of the column-based search strategy. However, if we consider an incremental scenario, in which new tuples could be inserted in the time interval between instants τ and $\tau + 1$, then the RFD_es detected at time τ might be invalidated. Thus, it is necessary to re-execute discovery algorithms on the new instance resulting at time $\tau + 1$. To this end, incremental RFD_e discovery algorithms aim to update the set of RFD_es without requiring the complete re-execution of the discovery algorithm.

The validation process must be re-executed since the $g3$-error e of an RFD_e φ holding at the time τ might increase or decrease upon the insertion of the new tuples. In particular, when the error increases, then it can lead to the invalidation of φ. On the contrary, when the error decreases, then it might happen that the $g3$-error of a candidate RFD_e φ', minimal w.r.t. φ, is below the threshold at time $\tau + 1$, even if φ' did not hold at time τ. Consequently, at time $\tau + 1$, φ' could be validated and φ be no longer minimal. More specifically, the main effects that can be produced upon the insertion of new tuples are:

- **Invalidation of RFD_e**: Let $\varphi : X \xrightarrow{\Psi \leq \varepsilon} A$ be an RFD_e holding at time τ, then when the error e increases it could produce the invalidation of φ at time $\tau + 1$. Thus, if an RFD_e holding at time τ is no longer valid a time $\tau + 1$, then one or more RFD_e candidates on the next lattice level having the same RHS and a superset of its LHS could be validated. More formally, let $X \xrightarrow{\Psi \leq \varepsilon} A$ be an RFD_e holding at time τ that is invalidated at time $\tau + 1$, then it is necessary to consider all possible RFD_e candidates $XB \xrightarrow{\Psi \leq \varepsilon} A$ such that $B \notin X$ and $B \neq A$.
- **Verification of minimality**: Let $\varphi : X \xrightarrow{\Psi \leq \varepsilon} A$ be an RFD_e holding at time τ that is valid also at time $\tau + 1$, then we need to check how the error has evolved for each RFD_e candidate on the previous level. More formally, let $X \xrightarrow{\Psi \leq \varepsilon} A$ be an RFD_e holding at time τ and $\tau + 1$, then it is necessary to consider all possible RFD_e candidates $X \setminus B \xrightarrow{\Psi \leq \varepsilon} A$ such that $B \in X$. Notice that, if at least one of these new candidates becomes valid at time $\tau + 1$, then $X \xrightarrow{\Psi \leq \varepsilon} A$ is no longer minimal, and such verification of minimality needs to be also iterated on lower lattice levels starting from the new validated RFD_es.

The necessity to perform validation and the consequent effects described above, primarily refers to all minimal RFD_es extracted at time τ. In general, an incremental discovery strategy aims to reduce the search space w.r.t. a possible navigation on the all lattice's edges. More specifically, new candidates could be generated 1) by using previously holding RFD_es, and 2) by exploring the parts of the search space that are not reachable from them.

In the first case, starting from RFD$_e$s holding at time τ, it is possible to limit the search space to their neighbors, which represent the new RFD$_e$ candidates at time $\tau + 1$. Here, given an RFD$_e$ $\varphi : X \xrightarrow{\Psi \leq \varepsilon} A$ holding at time τ, then one of its neighbors can have one of the following forms $\varphi' : XB \xrightarrow{\Psi \leq \varepsilon} A$, or $\varphi'' : X \backslash B \xrightarrow{\Psi \leq \varepsilon} A$, with $B \in attr(R) \backslash A$. This candidate generation strategy come from the FD induction strategy defined in [12], but it has been adapted to the incremental discovery scenario. Instead, since the second case would catch lattice's edges that are not neighbors of any RFD$_e$ holding at time τ, it is necessary to consider possible additional RFD$_e$ candidates, according to the three following properties:

- **Property 1.** Let Σ_τ be the set of all RFD$_e$ holding at time τ, and let

$$Z_\tau = \left\{ \bigcup_{A \in \text{attr(R)}} A \mid \exists X \xrightarrow{\Psi \leq \varepsilon} A \in \Sigma_\tau \right\}$$

  if $|Z_\tau| \neq |attr(R)|$, then for each attribute $A \notin Z_\tau$ it is necessary to add all possible RFD$_e$ candidates $B \xrightarrow{\Psi \leq \varepsilon} A$, with $B \neq A$.
- **Property 2.** Let Σ_τ be the set of all RFD$_e$ holding at time τ, and for each $A \in attr(R)$ let

$$S_\tau = \left\{ \bigcup_{B \in \text{attr(R)}} B \mid \exists X \xrightarrow{\Psi \leq \varepsilon} A \in \Sigma_\tau \wedge B \in X \right\}$$

  if $|S_\tau| \neq |attr(R)|$, then for each attribute $B \notin S_\tau$ it is necessary to add an RFD$_e$ candidate $B \xrightarrow{\Psi \leq \varepsilon} A$.
- **Property 3.** Let Σ_τ be the set of all RFD$_e$ holding at time τ, for each $A, B \in attr(R)$, $B \neq A$, let $l = min\{|X| \mid X \xrightarrow{\Psi \leq \varepsilon} A \in \Sigma_\tau \wedge B \in X\}$, and let

$$N_B^l = \left\{ \bigcup_{X \text{with} |X| = l} X \mid X \xrightarrow{\Psi \leq \varepsilon} A \notin \Sigma_\tau \wedge B \in X \right\}$$

  then for each candidate LHS $X \in N_B^l$ it is necessary to add a candidate RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ if and only if there is no $X' \subset attr(R)$ such that $X' \xrightarrow{\Psi \leq \varepsilon} A \in \Sigma_\tau$ and $X \subseteq X'$.

survival (A)	still alive (B)	age at heart attack (C)	pericardial effusion (D)	fractional shortening (E)	epss (F)	lvdd (G)
36	0	55	1	0.210	4.2	4.160
1	1	65	0	0.150	?	5.050
1	1	52	1	0.170	17.200	5.320
3	1	?	0	?	12	?
27	0	47	0	0.400	5.120	3.100
35	0	63	0	?	10	?
26	0	61	0	0.610	13.100	4.070
16	0	63	1	?	?	5.310
1	1	65	0	0.060	23.600	?
19	0	68	0	0.510	?	3.880

Table 1. Snippet of the echocardiogram dataset.

In other words, the exploration of the search space starts from all RFD$_e$s holding at time τ, and to those generated through properties 1, 2, and 3. However, starting from the latter it is possible to generate new RFD$_e$ candidates as discussed above.

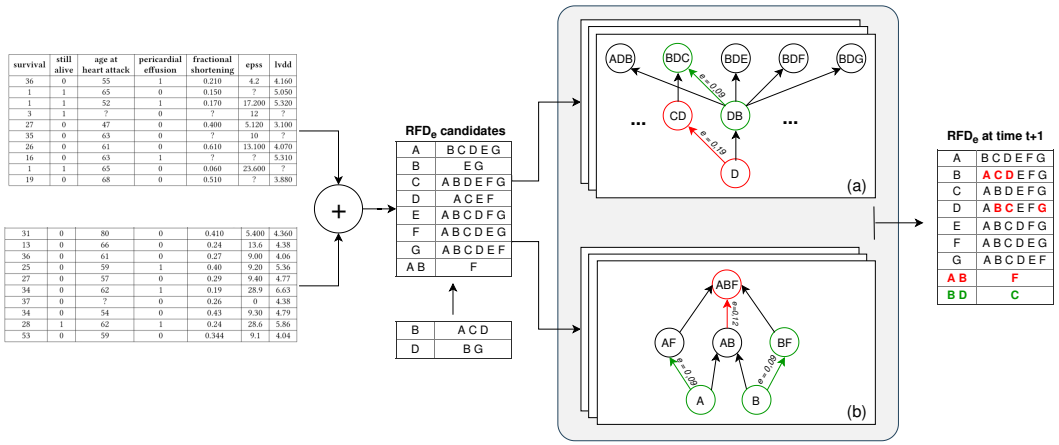

Fig. 1. An overview of the proposed discovery process by considering $\varepsilon = 0.1$.

Example 4.2. Let us consider the small snippet of the *echocardiogram* dataset shown in Table 1, then Figure 1 shows a simulation on how new RFD$_e$ candidates could be generated according to properties defined above, to the relation of Table 1, and to an error threshold $\varepsilon = 0.1$. In particular, in the middle of Figure 1 a compressed representation of the following minimal RFD$_e$s holding at time τ is shown: $A \xrightarrow{\Psi \leq \varepsilon} BCDEG, B \xrightarrow{\Psi \leq \varepsilon} EG, C \xrightarrow{\Psi \leq \varepsilon} ABDEFG, D \xrightarrow{\Psi \leq \varepsilon} ACEF, E \xrightarrow{\Psi \leq \varepsilon} ABCDFG, F \xrightarrow{\Psi \leq \varepsilon} ABCDEG, G \xrightarrow{\Psi \leq \varepsilon} ABCDEF, AB \xrightarrow{\Psi \leq \varepsilon} F$. As said above, all these RFD$_e$s become RFD$_e$ candidates at time $\tau + 1$, and according to Property 2, it is necessary to add $B \xrightarrow{\Psi \leq \varepsilon} ACD$ and $D \xrightarrow{\Psi \leq \varepsilon} BG$ as new RFD$_e$ candidates. Moreover, Figure 1(a) shows what happens when an RFD$_e$ is invalidated at time $\tau + 1$. In particular, it shows that $D \rightarrow C$ is invalidated at time $\tau + 1$, and the new RFD$_e$ candidates $AD \xrightarrow{\Psi \leq \varepsilon} C, BD \xrightarrow{\Psi \leq \varepsilon} C, DE \xrightarrow{\Psi \leq \varepsilon} C, DF \xrightarrow{\Psi \leq \varepsilon} C, DG \xrightarrow{\Psi \leq \varepsilon} C$ have been generated. Then, after the RFD$_e$ discovery algorithm performs the validation of such new RFD$_e$ candidates, only the RFD$_e$ candidate $BD \xrightarrow{\Psi \leq \varepsilon} C$ holds according to the input threshold [17]. Figure 1(b) shows what happens when an RFD$_e$ is valid at time $\tau + 1$, but it is no longer minimal. In particular, although $AB \xrightarrow{\Psi \leq \varepsilon} F$ is still valid after the insertion of the new tuples, it is no longer minimal because $A \xrightarrow{\Psi \leq \varepsilon} F$ and $B \xrightarrow{\Psi \leq \varepsilon} F$ have been validated at time $\tau + 1$. Consequently, after the insertion of the new tuples (Figure 1) the RFD$_e$s valid at time $\tau + 1$ are the following ones: $A \xrightarrow{\Psi \leq \varepsilon} BCDEG$, $B \xrightarrow{\Psi \leq \varepsilon} ACDEG$, $C \xrightarrow{\Psi \leq \varepsilon} ABDEFG$, $D \xrightarrow{\Psi \leq \varepsilon} ABCEFG$, $E \xrightarrow{\Psi \leq \varepsilon} ABCDFG$, $F \xrightarrow{\Psi \leq \varepsilon} ABCDEG$, $G \xrightarrow{\Psi \leq \varepsilon} ABCDEF$.

5 EFFICIENT LEVEL-WISE INCREMENTAL DISCOVERY

In this section, we focus our attention on the proposed incremental discovery algorithm, named BIRD (Bit-vector based Incremental RFD$_e$ Discoverer). In particular, we present its main steps by introducing the data structures, the search strategies, pruning techniques, and validation process.

BIRD is a column-based algorithm. Usually, the execution of column-based algorithms, hence its discovery process is divided into three phases: candidate generation, validation and pruning. This

means that BIRD traverses the lattice bottom-up, validating each LHS-RHS pair by using partitions. Each time candidate RFD$_e$s on the next lattice level have to be generated, then new partitions for them must be generated.

An overview of the BIRD's discovery process is provided in Figure 2. Given an instance r of a relation schema R at time τ, BIRD first performs a pre-processing step (Figure 2(a)), in which it pre-computes the partitions, updating all the partitions generated at time τ according to the new inserted tuples. In order to avoid a long pre-processing phase, only partitions associated with the first lattice level are updated, i.e. those concerning individual attributes.

Example 5.1. Starting from the example introduced in Figure 1, the partition at time τ for the attribute periodical effusion of the echocardiogram dataset is $\pi_D(\tau) = \{[0, 4, 5, 6, 7, 9], [1, 2, 3, 8]\}$. However, the new partition at time $\tau + 1$ is $\pi_D(\tau + 1) = \{[0,4,5,6,7,9,10,11,12,13,14,15,16,17,19], [1, 2, 3, 8, 18]\}$.

This operation is extremely complex, especially when considering a large number of columns and rows. The pre-processing phase also defines the starting point RFD$_e$s, storing them in an ordered linked map. The latter allows to perform an ordered discovery phase, on the basis of the LHS cardinalities of RFD$_e$s. Moreover, it also stores all the candidate RFD$_e$s generated during the discovery process.

When the process starts, the first candidate RFD$_e$ extracted from the linked map is validated by using the validation method implementing the *g3-error* described in equation 3. If the RFD$_e$ is valid (Figure 2(b)), then BIRD performs *downward* discovery, since new added tuples could reduce the error of candidate RFD$_e$s that are minimal w.r.t. the considered one, but were not valid before. However, if an RFD$_e$ is not valid, an *upward* discovery is accomplished (see Figure 2(c)) in order to find the possible new holding RFD$_e$s. In both downward and upward search strategies, BIRD incrementally performs a minimality check on each RFD$_e$. This ensures that all the RFD$_e$s in the linked map that have already been analyzed are minimal. Such minimality check strategy is much more efficient than checking the minimality after executing the discovery algorithm. At the end of each iteration, BIRD checks if there are other RFD$_e$s to analyze, and if not, it returns the minimal RFD$_e$s at time $\tau + 1$. In the following, we provide details of each component and execution step.

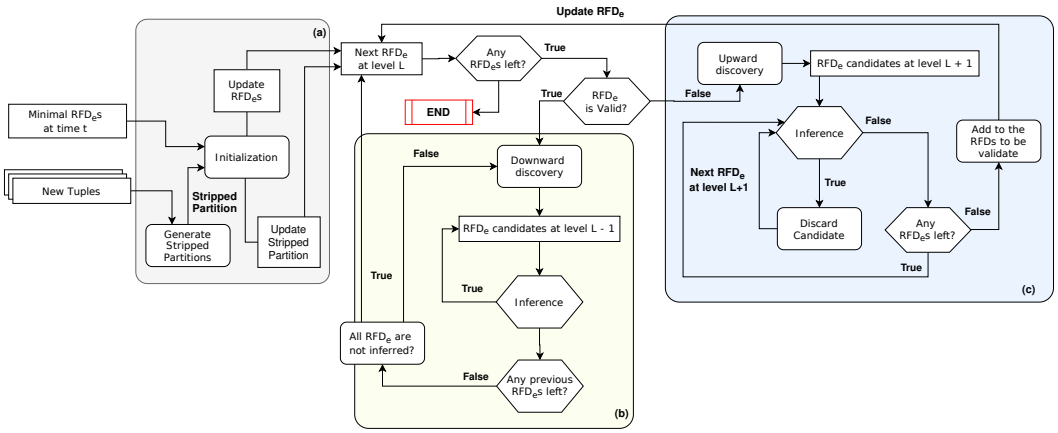

Fig. 2. An overview of BIRD.

5.1 Binary RFD$_e$ and Compressed linked ordered map

BIRD uses fast and lightweight data structures, avoiding high memory usage. In particular, the RFD$_e$s have been represented by using a compressed structure based on bit-vectors, i.e. $(0|1)^+$. Specifically, each RFD$_e$ has been represented as a pair of bit-vectors, each representing its LHS or RHS. In addition, each RFD$_e$ has associated the value e of its $g3$-error coverage measure. For instance, Figure 3 shows the representation of the candidate RFD$_e$s from the example shown in Figure 1. In particular, the vector on the left-side represents the LHS, whereas the one on the right-side represents the RHS. The bits having value 1 represent the attributes instantiated on the specific side of the RFD$_e$.

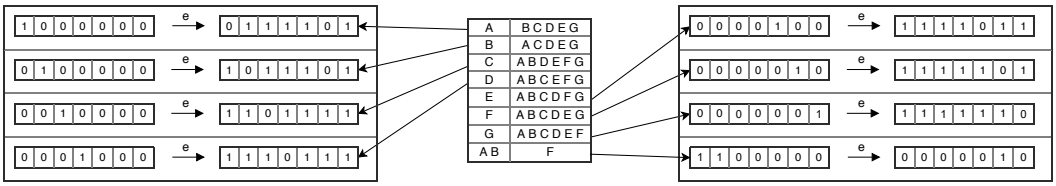

Fig. 3. Binary representation of candidate RFD$_e$s.

More specifically, let b_X be the vector that representing the LHS, and b_A be the vector representing the RHS, a binary RFD$_e$, $\varphi : (b_X, b_A, e)$, has the following properties:

- If $b_X[i] = 1$ (or $b_A[j] = 1$) then the i-th attribute is included in the LHS (or the j-th one is included in the RHS);
- $b_X \wedge b_A = (0)^+$, e.g. only non-trivial RFD$_e$s are considered;
- Each binary RFD$_e$'s side must contain at least one bit equals to 1, i.e. $b_X = (0|1)^*1(0|1)^*$ and $b_A = (0|1)^*1(0|1)^*$;
- Each binary RFD$_e$ has associated the value e representing its associated $g3$-error;
- LHS and RHS vectors have a dynamic size adapted to the number of dataset's attributes.

Moreover, let Σ be the set of RFD$_e$s holding on a relation instance r, and assuming the RFD$_e$ representation in terms of bit-vectors described above, then the compressed linked ordered map has the following properties:

- Each RFD$_e$ is inserted in the linked map according to an ordering criterion based on the number of attributes on X, i.e. considering the number of 1s in the binary representation b_X of X.
- For each RFD$_e$ there is a link to the next RFD$_e$, except for the last one;
- In order to guarantee the quick insertion of each RFD$_e$ in the linked map, a support vector contains the references of the last inserted RFD$_e$ for each b_X sharing the same number of 1s;
- For each pair of RFD$_e$s, $\varphi : (b_X, b_A, e)$ and $\varphi' : (b'_X, b'_A, e') \in \Sigma$, then $b_X \neq b'_X$.

The compressed linked ordered map ensures that information are quickly saved and retrieved. In fact, this represents a hybrid structure composed by a hash map and an external linked list, so permitting a fast data retrieval (hash map), and a fast insertion strategy (linked list). More specifically, the map uses $\varphi : (b_X, b_A, e)$ as key, and a pointer to the next RFD$_e$ as value, according to an ordering criterion based on the number of 1s into b_X.

Figure 4 shows the compressed linked ordered map for the RFD$_e$s described in the example shown in Figure 1. As we can see, all RFD$_e$s have been mapped using bit-vectors with seven bits, which represent the number of attributes in R. The proper size of bit-vectors is one of the most important properties that allows to reduce memory usage. The left-side of Figure 4 contains the support vector

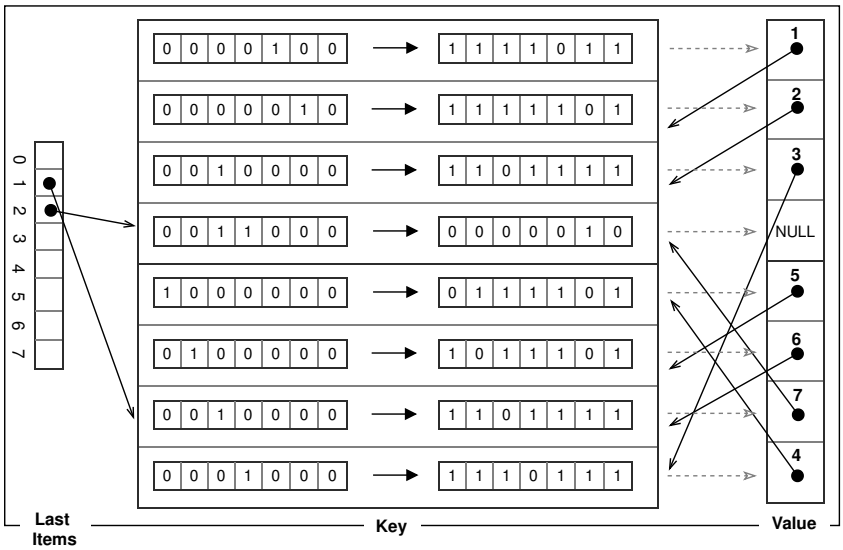

Fig. 4. The compressed linked ordered map related to the Example in Figure 1.

linking to the last inserted RFD_e among the RFD_es having the same number of attributes on the LHS. The right-side of the figure shows the value of each item in the map, representing a pointer to the next RFD_e. The discovery process starts from the first RFD_e, i.e. that with pointer *1*, and proceeds until it reaches the *null* pointer.

5.2 Generation of RFD_e candidates

The RFD_e candidate generation process is one of the main steps of the RFD_e discovery process. Given a candidate RFD_e $X \xrightarrow{\Psi \leq \varepsilon} A$ the candidate generation follows the properties defined in Section 4.2. Therefore, we consider two main cases in which the generation of new candidates is necessary, namely the *invalidation of* RFD_es and the *verification of minimality*. According to the first one, Algorithm 1 permits to generate candidate RFD_es by evaluating not valid RFD_es. In particular, it considers a candidate RFD_e not valid at time $\tau + 1$, together with the binary representation of its LHS, and it uses bit-wise operations to generate candidate RFD_es for the next lattice level. Algorithm 1 first checks whether it is possible to search candidate RFD_es at the next higher level (line 2), then it processes an attribute at the time among those not already in the LHS or RHS. This avoids the inclusion of trivial RFD_e in the linked map of candidates (lines 1-6). If these conditions are verified, then the algorithm generates a new candidate RFD_e composed of the new LHS X_{next} and the same RHS A. In particular, this is accomplished through an bit-wise OR operation between b_X and $b_{X_{\text{next}}}$ (lines 8-9). Finally, it returns a new set of generated candidates.

Example 5.2. Figure 1(a) shows an example of this operation. The RFD_e $D \xrightarrow{\Psi \leq \varepsilon} C$ is not valid at time $\tau + 1$. Using this strategy the set of candidates will contain the following RFD_es: $AD \xrightarrow{\Psi \leq \varepsilon} C, BD \xrightarrow{\Psi \leq \varepsilon} C, ED \xrightarrow{\Psi \leq \varepsilon} C, FD \xrightarrow{\Psi \leq \varepsilon} C, GD \xrightarrow{\Psi \leq \varepsilon} C$, which are all non trivial ones.

Concerning the verification of the minimality property, as said above Algorithm 2 computes possible candidate RFD_es on the previous lattice level. Similarly to the Algorithm 1, given an RFD_e $X \xrightarrow{\Psi \leq \varepsilon} A$, together with the binary representation of its LHS b_X, it first checks whether it is possible

Algorithm 1 NEXTCANDIDATES

INPUT: An RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ not valid at time $\tau + 1$, with b_X binary representation of the LHS
OUTPUT: The set Σ_{next} of new candidate RFD$_e$s at time $\tau + 1$

1: $\Sigma_{\text{next}} \leftarrow \emptyset$
2: **if** $|\text{X}| < (|attr(R)| - 1)$ **then**
3: **while** $B_i \in \text{attr}(R)$ **with** $i \leq |\text{attr}(R)|$ **do**
4: $b_{\text{X}_{\text{next}}} \leftarrow b_{\text{X}}$
5: **if** $B_i \neq A$ **then**
6: **if** $b_{\text{X}_{\text{next}}}[i] \neq 1$ **then**
7: $b_{\text{X}_{\text{next}}}[i] \leftarrow 1$
8: $\Sigma_{\text{next}} \leftarrow \Sigma_{\text{next}} \cup \{\text{X}_{\text{next}} \xrightarrow{\Psi \leq \varepsilon} A\}$
9: **return** Σ_{next}

Algorithm 2 PREVIOUSCANDIDATES

INPUT: An RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ not valid at time $\tau + 1$, with b_X binary representation of the LHS
OUTPUT: The set Σ_{prev} of new candidate RFD$_e$s at time $\tau + 1$

1: $\Sigma_{\text{prev}} \leftarrow \emptyset$
2: **if** $|\text{X}| > 1$ **then**
3: **while** $B_i \in \text{attr}(R)$ **with** $i \leq |attr(R)|$ **do**
4: $b_{\text{X}_{\text{prev}}} \leftarrow b_{\text{X}}$
5: **if** $b_{\text{X}_{\text{prev}}}[i] = 1$ **then**
6: $b_{\text{X}_{\text{prev}}}[i] \leftarrow 0$
7: $\Sigma_{\text{prev}} \leftarrow \Sigma_{\text{prev}} \cup \{\text{X}_{\text{prev}} \xrightarrow{\Psi \leq \varepsilon} A\}$
8: **return** Σ_{prev}

to search candidate RFD$_e$s at the previous lower level (line 2), and if so, it considers each $B_i \in X$ at a time, removing it from the LHS (lines 5-6). Next, the algorithm adds the new candidate RFD$_e$s to the set of candidates (line 7).

Example 5.3. Figure 1(b) shows an example of this operation. The RFD$_e$ $AB \xrightarrow{\Psi \leq \varepsilon} F$ is still valid a time $\tau + 1$. However, following the proposed discovery strategy, we check if there are RFD$_e$s in the previous lattice level with associated $g3$-error $e \leq \varepsilon$. Thus, the following new candidate RFD$_e$s need to be considered: $A \xrightarrow{\Psi \leq \varepsilon} F, B \xrightarrow{\Psi \leq \varepsilon} F$.

5.3 Inference

During the discovery process, as candidate RFD$_e$s are validated, it is necessary to check whether they are minimal. Thus, in the proposed search process, the minimality check (that we call *Inference*) is accomplished by exploiting a new compressed linked ordered map. More formally, given a relation schema R and $X, Z \subseteq attr(R)$, such that $Z \subset X$, then for each RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ validated at time $\tau + 1$ it is necessary to verify that there exists no RFD$_e$ $Z \xrightarrow{\Psi \leq \varepsilon} A$, among those already validated at time $\tau + 1$.

Algorithm 3 IS_INFERRED

INPUT: An RFD$_e$ $\varphi : X \xrightarrow{\Psi \leq \varepsilon} A$ holding at time $\tau + 1$, A set Σ_τ of candidate RFD$_e$s
OUTPUT: *true* if the φ is not minimal, *false* otherwise

```
1: if Σ_τ is empty then
2:     return false
3: else
4:     for each Z ->(Ψ≤ε) A ∈ Σ_τ do
5:         if |Z| ≤ |X| then
6:             b_(X∧Z) ← b_X ∧ b_Z
7:             if b_(X∧Z) = b_Z then
8:                 return false
9: return true
```

The inference process requires complex analysis steps to be performed on validated RFD$_e$s. For this reason, we use a linked map to group the RFD$_e$s already validated at time $\tau + 1$ sharing the same RHS. This allows to reduce the number of RFD$_e$s to be checked.

Algorithm 3 shows the proposed inference process. It considers an RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ holding at time $\tau + 1$, and a linked map containing all minimal RFD$_e$s already validated and grouped by each possible RHS, and checks if it is minimal. Algorithm 3 first checks if there exists at least one RFD$_e$ holding at time $\tau + 1$ (lines 1-3), and if so, then for each candidate RFD$_e$ $Z \xrightarrow{\Psi \leq \varepsilon} A$ checks if the cardinality of Z is lower than the one of X (lines 4-5), and it performs a bit-wise AND operation between the bit-vector representations of X and Z (lines 6-7). If the result of this operation is equal to the bit-vector of Z, then this means that $Z \subset X$, i.e $X \xrightarrow{\Psi \leq \varepsilon} A$ is not minimal (lines 8-9).

5.4 The BIRD algorithm

Algorithm 4 shown in Figure 1 implements the proposed BIRD discovery algorithm. Given Σ_τ the set of all RFD$_e$s holding at time τ, and of all RFD$_e$s generated as starting points according to the properties 1, 2 and 3 (see Section 4.2), BIRD starts by analyzing the candidate RFD$_e$s with lower cardinality (line 1). In particular, we use the compressed linked ordered map to facilitate a level-wise discovery, so as to avoid the sorting of all RFD$_e$s. Thus, for each RFD$_e$ holding at time τ, BIRD uses the VALIDATION process (line 4) to verify whether $X \xrightarrow{\Psi \leq \varepsilon} A$ still holds at time $\tau + 1$. This process is performed according to the $g3$-error computation described in equation (3). Thus, if the analyzed RFD$_e$ is not valid, BIRD generates new candidate RFD$_e$s at a higher lattice level (Algorithm 1), by discarding those that can be inferred, according to Algorithm 3 (lines 3-6). Notice that the ordering criterion of the RFD$_e$s permits to avoid the re-validation of some candidate RFD$_e$s.

Example 5.4. As mentioned in the example shown in Figure 1, if we consider the RFD$_e$ $D \xrightarrow{\Psi \leq \varepsilon} C$, it is not valid at time $\tau + 1$. Thus, the RFD$_e$s $AD \xrightarrow{\Psi \leq \varepsilon} C, BD \xrightarrow{\Psi \leq \varepsilon} C, DE \xrightarrow{\Psi \leq \varepsilon} C, FD \xrightarrow{\Psi \leq \varepsilon} C, GD \xrightarrow{\Psi \leq \varepsilon} C$ become new candidate RFD$_e$s at time $\tau + 1$. Moreover, let suppose that there exists another RFD$_e$ $E \xrightarrow{\Psi \leq \varepsilon} C$ that is not valid a time $\tau + 1$, then we should also consider all the RFD$_e$s $AE \xrightarrow{\Psi \leq \varepsilon} C, BE \xrightarrow{\Psi \leq \varepsilon} C, DE \xrightarrow{\Psi \leq \varepsilon} C, EF \xrightarrow{\Psi \leq \varepsilon} C, EG \xrightarrow{\Psi \leq \varepsilon} C$ as new candidates. However, the RFD$_e$ $DE \xrightarrow{\Psi \leq \varepsilon} C$ appears twice, but it will be validated only once thanks to the defined compressed linked ordered map.

Algorithm 4 BIRD Algorithm

INPUT: A set Σ_τ of candidate RFD$_e$s as starting points from time τ
OUTPUT: A set $\Sigma_{\tau+1}$ of new valid and minimal RFD$_e$s at time $\tau + 1$

1: **for each** $X \xrightarrow{\Psi \leq \varepsilon} A \in \Sigma_\tau$ **in ascending ordering by LHS cardinalities do**
2: **if** VALIDATION($X \xrightarrow{\Psi \leq \varepsilon} A$) *is not valid* **then**
3: $L_{l+1} \leftarrow$ NEXTCANDIDATES($X \xrightarrow{\Psi \leq \varepsilon} A$)
4: **for each** $W \xrightarrow{\Psi \leq \varepsilon} A \in L_{l+1}$ **do**
5: **if** IS_MINIMAL($W \xrightarrow{\Psi \leq \varepsilon} A$) **then**
6: $\Sigma_\tau \leftarrow \Sigma_\tau \cup \{W \xrightarrow{\Psi \leq \varepsilon} A\}$
7: **else**
8: $L_{l-1} \leftarrow$ PREVIOUSCANDIDATES($X \xrightarrow{\Psi \leq \varepsilon} A$)
9: **for each** $Z \xrightarrow{\Psi \leq \varepsilon} A \in L_{l-1}$ **do**
10: **if** VALIDATION($Z \xrightarrow{\Psi \leq \varepsilon} A$) *is valid* **then**
11: $L_{l-1} \leftarrow L_{l-1} \cup$ PREVIOUSCANDIDATES($Z \xrightarrow{\Psi \leq \varepsilon} A$)
12: **else**
13: $L_{l-1} \leftarrow L_{l-1} \setminus \{Z \xrightarrow{\Psi \leq \varepsilon} A\}$
14: **if** $|L_{l-1}| > 0$ **then**
15: $\Sigma_\tau \leftarrow \Sigma_\tau \setminus \{X \xrightarrow{\Psi \leq \varepsilon} A\}$
16: **for each** $Z \xrightarrow{\Psi \leq \varepsilon} A \in L_{l-1}$ **do**
17: **if** IS_MINIMAL($Z \xrightarrow{\Psi \leq \varepsilon} A$) **then**
18: $\Sigma_\tau \leftarrow \Sigma_\tau \cup Z \xrightarrow{\Psi \leq \varepsilon} A$
19: $\Sigma_{\tau+1} \leftarrow \Sigma_\tau$

However, if $X \xrightarrow{\Psi \leq \varepsilon} A$ is a valid RFD$_e$, the algorithm checks if other RFD$_e$s in the previous lattice level have been already validated at time $\tau + 1$ (lines 8-18). For each RFD$_e$ φ holding at time $\tau + 1$ the process tries to validate its neighbors on the previous lattice level, and φ is not removed from $\Sigma_{\tau+1}$ iff none of its neighbors are valid. Moreover, in this last case the iterations on previous candidate RFD$_e$s are stopped due to the analyzed RFD$_e$. Notice that, the minimality of candidate RFD$_e$s on previous lattice levels is always locally checked (lines 16-18).

It is worth to notice that, while the implementation of BIRD considers several code optimizations and takes advantage of the defined data structures, for sake of clarity, the pseudo-codes described above do not show such optimizations. They mainly guarantee that each possible candidate RFD$_e$ is validated at most once. Thus, in the worst case, the exploration of the search space is equivalent to the one of non-incremental RFD$_e$ discovery algorithms.

5.5 Parallelism in BIRD

As mentioned above, the RFD$_e$ discovery problem is a complex one, especially with datasets having high cardinality and dimensionality. To tackle this problem, we defined a parallel version of BIRD (named BIRD^p), which differs from the sequential version in that the generation and the validation of candidate RFD$_e$s are accomplished level-wise, i.e. one level at a time, by also using thread-safe collections. Thus, the compressed linked ordered map has been defined as a thread-safe data structure. Moreover, BIRD^p splits the space of candidate RFD$_e$s, assigning them several workers.

Algorithm 5 BIRD Parallel Version

INPUT: A set Σ_τ of candidate RFD$_e$s as starting points from time τ
OUTPUT: A set $\Sigma_{\tau+1}$ of new valid and minimal RFD$_e$s at time $\tau + 1$

```
1:  n_CPU ← number of available CPUs
2:  Σ_{τ+1} ← ∅
3:  l ← 0
4:  while l ≤ |attr(R)| do
5:      Γ ← CREATEPOOL(n_CPU)
6:      L_l ← GETRFDS(l)
7:      for all γ ∈ Γ do
8:          φ ← L_l.NEXTDEPENDENCY()
9:          γ.execute(WORKERDISCOVERY(φ, Σ_τ , Σ_{τ+1}))
10:     WAITWORKERS()
11:     l ← l + 1
12: return Σ_{τ+1}
```

More specifically, a pool of thread-workers is allocated for each level in order to continuously monitor its status, enabling the execution of each work only when necessary. In particular, let l be the selected lattice level, then L_l is the set of all candidate RFD$_e$s at time $\tau + 1$ having LHS cardinality equal to l. More specifically, the workload, i.e. the number of candidate RFD$_e$s to be validated, is distributed among all workers, assigning a validation task to each worker whenever it becomes available. The results of each worker are merged into a thread-safe set, which is updated by each thread after performing the assigned task.

Algorithms 5 and 6 implement BIRD^p. In particular, Algorithm 5 shows the initialization phase of the thread pool. Let l be the LHS cardinality of the RFD$_e$s to be analyzed at level l, then the algorithm selects only the RFD$_e$s to be checked by using a custom filter (line 6). The latter is a stream filter based on lambda expressions that selects the RFD$_e$s from the map having a specific LHS cardinality defined by l (line 3). Next, for each candidate RFD$_e$, it assigns an asynchronous task to a worker so that the discovery phase can start through the procedure `WORKERDISCOVERY` (lines 7-9). Moreover, for each iteration, it waits for the termination of each thread before moving to the next level. Finally, each worker stores the RFD$_e$s discovered at time $\tau + 1$ in a shared set $\Sigma_{\tau+1}$ that is returned after the complete execution of BIRD^p.

Instead, Algorithm 6 receives a candidate RFD$_e$ as input and tries to validate it by also checking the minimality property. Notice that, this procedure follows in part the same discovery and validation process of the sequential version (Algorithm 4) described above.

5.6 Theoretical Evaluation

From a theoretical point of view, it is necessary to guarantee that BIRD is able to find all and only minimal RFD$_e$s holding on a given dataset D. Thus, it is necessary to prove the correctness of discovered RFD$_e$s. To this end, the correctness of RFD$_e$s discovered with BIRD can be assessed through well-known methods and properties proposed in the literature. In particular, BIRD implements the one proposed in [17]. Notice that, all the proofs provided below refer the basic version of BIRD (Algorithm 4) even though they can be easily generalized to the parallel one.

Minimality. One of the evaluation dimensions of a dependency discovery algorithm is minimality, which guarantees that the discovered RFD$_e$s no longer hold upon removing an LHS attribute.

Algorithm 6 BIRD Worker

1: **function** WORKERDISCOVERY($X \xrightarrow{\Psi \leq \varepsilon} A, \Sigma_\tau, \Sigma_{\tau+1}$)
2: **if** VALIDATION($X \xrightarrow{\Psi \leq \varepsilon} A$) *is not valid* **then**
3: $L_{l+1} \leftarrow$ NEXTCANDIDATES($X \xrightarrow{\Psi \leq \varepsilon} A$)
4: **for each** $W \xrightarrow{\Psi \leq \varepsilon} A \in L_{l+1}$ **do**
5: **if** IS_MINIMAL($W \xrightarrow{\Psi \leq \varepsilon} A$ **then**)
6: $\Sigma_\tau \leftarrow \Sigma_\tau \cup \{W \xrightarrow{\Psi \leq \varepsilon} A\}$
7: **else**
8: $L_{l-1} \leftarrow$ PREVIOUSCANDIDATES($X \xrightarrow{\Psi \leq \varepsilon} A$)
9: **for each** $Z \xrightarrow{\Psi \leq \varepsilon} A \in L_{l-1}$ **do**
10: **if** VALIDATION($Z \xrightarrow{\Psi \leq \varepsilon} A$) *is valid* **then**
11: $L_{l-1} \leftarrow L_{l-1} \cup$ PREVIOUSCANDIDATES($Z \xrightarrow{\Psi \leq \varepsilon} A$)
12: **else**
13: $L_{l-1} \leftarrow L_{l-1} \setminus \{Z \xrightarrow{\Psi \leq \varepsilon} A\}$
14: **if** $|L_{l-1}| > 0$ **then**
15: $\Sigma_\tau \leftarrow \Sigma_\tau \setminus \{X \xrightarrow{\Psi \leq \varepsilon} A\}$
16: **for each** $Z \xrightarrow{\Psi \leq \varepsilon} A \in L_{l-1}$ **do**
17: **if** IS_MINIMAL($Z \xrightarrow{\Psi \leq \varepsilon} A$) **then**
18: $\Sigma_\tau \leftarrow \Sigma_\tau \cup Z \xrightarrow{\Psi \leq \varepsilon} A$
19: $\Sigma_{\tau+1} \leftarrow \Sigma_{\tau+1} \cup \Sigma_\tau$

Theorem 5.5. *Each RFD$_e$ discovered by BIRD is minimal according to the minimality property defined in Section 3.*

Proof. BIRD starts with the set of all candidate RFD$_e$s Σ_τ, and updates it according to validation results in ascending order by LHS cardinality (Line 1). Thus, if the first candidate RFD$_e$ $\rho_0 \in \Sigma_\tau$ is valid, then it is also minimal, since no PREVIOUSCANDIDATES exist. Now, assuming that all minimal RFD$_e$s with LHS cardinality k have been added to Σ_τ, then we prove by induction that if a generic RFD$_e$ $\rho_{k+1} \in \Sigma_\tau$ with cardinality $k+1$ is valid, then it is also minimal.

In Algorithm 4, when ρ_{k+1} is validated, then BIRD executes Lines 8–18. Thus, PREVIOUSCANDIDATES are collected and explored (Lines 8-9), and if there exists at least one valid RFD$_e$ in the previous lattice level, then it is necessary to explore the previous levels in order to assess the minimality property more in depth. However, if there are no valid RFD$_e$s in a previous level, then BIRD stops the exploration process, since according to the *refinement property* [17], no candidate from the previous levels can be valid. During the exploration of previous levels, if there exists at least one RFD$_e$ that is minimal w.r.t. ρ_{k+1}, then the latter is removed (Line 15). As a consequence, all valid RFD$_e$s discovered by PREVIOUSCANDIDATES in previous levels will be added to Σ after assessing their minimality (Line 17-18). Thus, if a valid RFD$_e$ ρ_{k+1} with cardinality $k+1$ is maintained into Σ_τ after the exploration by PREVIOUSCANDIDATES on those with cardinality k, then it is also minimal. □

Completeness. Another important evaluation dimension of a dependency discovery algorithm is completeness, which guarantees that the algorithm discovers *all* minimal dependencies.

Theorem 5.6. *BIRD discovers all minimal RFD$_e$s.*

Proof. At each time instant, BIRD proceeds incrementally by considering an initial set Σ_τ of candidate RFD$_e$s. Let us firstly consider the candidate RFD$_e$s appearing in Σ_τ at the beginning of the discovery process at a given time $\tau + 1$, and then in order:

- all the RFD$_e$s holding at the previous time instant (τ);
- all the candidate RFD$_e$s $B \xrightarrow{\Psi \leq \varepsilon} A$ such that there is no minimal RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ holding at time τ, for any attribute $B \neq A$ (property 1);
- all the candidate RFD$_e$s $B \xrightarrow{\Psi \leq \varepsilon} A$ such that there is no minimal RFD$_e$ $X \xrightarrow{\Psi \leq \varepsilon} A$ holding at time τ with $B \in X$ (property 2); and
- all the candidate RFD$_e$s $X \xrightarrow{\Psi \leq \varepsilon} A$ such that for each $B \in X$, B belongs to the LHS of some RFD$_e$s holding at time τ, but at time τ there exists neither a minimal RFD$_e$ $X' \xrightarrow{\Psi \leq \varepsilon} A$, with $X' \subseteq X$ and $B \in X'$, nor a minimal RFD$_e$ $X'' \xrightarrow{\Psi \leq \varepsilon} A$, such that $X \subseteq X''$ and $B \in X''$ (property 3).

Notice that property 1 guarantees the fact that at the first time instant ($\tau = 0$), for each attribute A in $attr(R)$, any candidate $B \xrightarrow{\Psi \leq \varepsilon} A$, with $B \neq A$, is added into Σ_τ.

Let us now prove by contradiction the completeness of the proposed search strategy. Let us assume that BIRD misses a minimal RFD$_e$ $\rho : X \xrightarrow{\Psi \leq \varepsilon} A$ holding on an input instance r. Then, 1) X cannot be superset of any candidate RFD$_e$s $X' \xrightarrow{\Psi \leq \varepsilon} A$, with $X' \subset X$, since if the latter has already been validated (Algorithm 4 processes candidates in increasing order of their LHS cardinality), ρ would not be minimal; instead, if $X' \xrightarrow{\Psi \leq \varepsilon} A$ does not hold on r, then ρ is analyzed by BIRD (Line 3), unless some $X'' \xrightarrow{\Psi \leq \varepsilon} A$, with $X' \subset X'' \subset X$, has already been validated; 2) X cannot be subset of any RFD$_e$ $X' \xrightarrow{\Psi \leq \varepsilon} A$ validated during the discovery process (Line 8-18), according to the aforesaid minimality proof.

Consequently, since all RFD$_e$s holding at time τ represent candidate RFD$_e$s holding at time $\tau + 1$, then points 1) and 2) prove that for each $X' \xrightarrow{\Psi \leq \varepsilon} A$ holding at time τ, $X \neq X \cup X' \neq X'$, that is, neither $X \subseteq X'$ nor $X' \subseteq X$. However, according to property 3, for each $B \in X$, if there exists at least an RFD$_e$ $X' \xrightarrow{\Psi \leq \varepsilon} A$, such that $B \in X'$, then all possible candidate RFD$_e$s at the lowest possible level are added to the set of candidate RFD$_e$s. This means that for all $B \in X$ no RFD$_e$ valid at time τ can contain B into its LHS. Thus, according to property 2, B is not included in S_τ and $B \xrightarrow{\Psi \leq \varepsilon} A$ would be added as candidate RFD$_e$ at time $\tau + 1$. We can deduce that no minimal RFD$_e$ $X' \xrightarrow{\Psi \leq \varepsilon} A$ holds at time τ for any possible attribute set $X' \subset attr(R)$, i.e. attribute A is never determined at time τ. Moreover, according to property 1, A would never be included into Z_τ, and then for each $B \neq A$ an RFD$_e$ $B \xrightarrow{\Psi \leq \varepsilon} A$ is included as candidate at time $\tau + 1$. Consequently, $X \xrightarrow{\Psi \leq \varepsilon} A$ is always considered as candidate during the discovery process, unless some $X' \xrightarrow{\Psi \leq \varepsilon} A$, with $X' \subseteq X$, has already been validated. In this case, $X \xrightarrow{\Psi \leq \varepsilon} A$ cannot be considered as minimal, which contradicts the original assumption. □

6 EVALUATION

In this section, we present the experimental evaluation of BIRD on several public datasets, comparing results with those of the TANE algorithm v.1.0.0[2]. In particular, for BIRD we split a given dataset into two portions, where the first one is used to obtain the RFD$_e$s holding at time τ, and the second

[2]We used the implementation available at: https://github.com/HPI-Information-Systems/metanome-algorithms

portion to run BIRD. This permits to simulate the insertion of new tuples in an incremental scenario. Instead, for TANE, we ran its discovery process on the entire dataset. The comparative analysis aims to show the advantages of an incremental discovery algorithm w.r.t. a complete re-execution from scratch.

Implementation details. BIRD has been developed in Java 11. Moreover, to avoid the re-computation of partitions, that are widely used for the validation of candidate RFD$_e$s, we introduced a strategy to store them in cache memory. Finally, as said above, we implemented two versions of BIRD, the sequential and parallel versions (named BIRD^p). The latter exploits functional programming techniques to properly manage parallel executions.

Hardware and Datasets. All experiments have been executed on a Mac with an Intel Xeon W 3.2GHz 32-core CPU with 128GB of RAM, running MacOS Mojave and OpenJDK 64-Bit 12.0.2 as Java environment. We evaluated BIRD on twelve public datasets previously used for testing FD and RFD$_e$ discovery algorithms [3]. Details on the considered datasets are shown in Table 2.

Evaluation process. For evaluating BIRD we simulated a tuple insertion scenario by splitting each original dataset r in two portions, based on a given percentage value. The first portion, indicated as r_τ, represents the relation instance at time τ, whereas the second one, indicated as $r_{\tau+1}$, refers to the relation instance resulting from the insertion of tuples from time τ to time $\tau + 1$. In particular, BIRD uses the minimal RFD$_e$s extracted by TANE on r_τ as starting points.

We analyzed the time performances achieved by BIRD on $r_{\tau+1}$ by varying (*i*) the percentage of tuples inserted from time τ to time $\tau + 1$, denoted as $P_{\tau+1}$, (*ii*) the $g3$-error threshold, and (*iii*) the number of execution threads. Among the different RFD$_e$ discovery algorithm proposed in literature, we chose TANE for the comparative evaluation because BIRD relies on its validation strategy.

6.1 Comparative evaluation

We performed a general comparative evaluation of BIRD versus TANE algorithm. In particular, BIRD and BIRD^p have been executed with $P_{\tau+1} = 10\%$, and all three algorithms used a $g3$-error threshold $\varepsilon = 0.3$. More specifically, times for BIRD and BIRD^p are analyzed in average, since for each dataset we performed 10 executions by considering different cut points.

Figure 5 reports the execution times achieved by the different algorithms. They highlight that BIRD and BIRD^p outperform TANE on all datasets with one or more order of magnitude, despite the variability of the number of rows/columns into datasets. In particular, execution times of BIRD are always almost an order of magnitude lower than times of TANE. The lowest performance gap is obtained with the datasets *chess*, *tsa-claims*, *ncvoter*, and *hepatitis*, for which BIRD still gets better

	Statistics				
Dataset	**[#] Columns**	**[#] Rows**	**[KB] Size**	**[#] FDs**	**[#] RFD$_e$s**
iris	5	150	5	4	13
balance-scale	5	625	7	1	5
chess	7	1999	519	1	7026
abalone	9	4176	187	137	88
nursery	9	12960	1024	1	4457
breast-cancer-wisconsin	11	699	20	46	95
bridges	13	108	6	142	340
echocardiogram	13	132	6	538	172
tsa-claims	13	145143	25608	28	355
adult	14	32561	3528	78	2289
ncvoter	19	1001	151	758	3191
hepatitis	20	155	8	8250	14973

Table 2. Statistics on the considered public datasets [3].

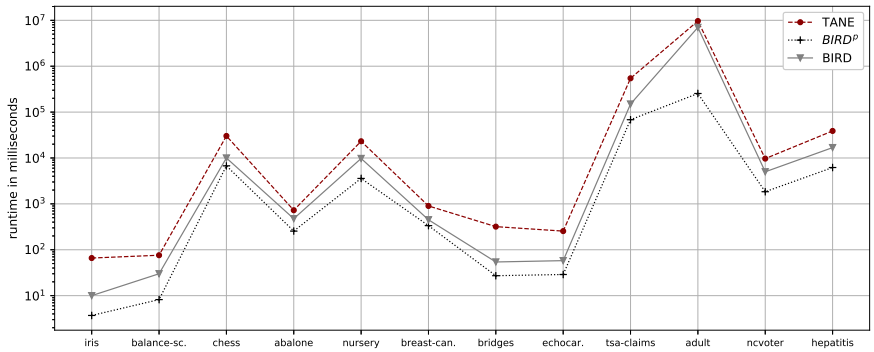

Fig. 5. Time performances of TANE, BIRD, and BIRD^p.

performances. They represent the most complex datasets due to the high number of holding RFD$_e$s, which require BIRD to manage a high number of starting points. Moreover, as expected, BIRD^p improves the execution times of the sequential version.

Another experimental session aimed to analyze the time efficiency of BIRD^p by using a configuration with $P_{\tau+1} = 60\%$, and by varying the $g3$-error thresholds in the range $[0.1, 1]$ with step 0.1. BIRD^p execution times are shown on average from 10 executions, in which we used random cut-points to split the data into two portions according to $P_{\tau+1}$. Moreover, results are always compared with a complete execution from scratch with TANE (red bars). Figure 6 highlights the execution times achieved by BIRD^p (blue bars). In general, we can notice that the execution times of BIRD^p have a decreasing trend when the $g3$-error threshold increases. This is due to the fact that the number of holding RFD$_e$s decreases as the $g3$-error threshold increases, and they typically have small LHS cardinalities, which enable the application of pruning strategies to discard lattice paths yielding non-minimal RFD$_e$s. Non-monotonic and quiet constant trends are followed by the execution times on *abalone*, *tsa-claims*, and the *iris* datasets. From the analysis of the discovered RFD$_e$s we observed that on these datasets BIRD^p validates almost always the set of RFD$_e$s received as starting points. Results of *breast-cancer-wisconsin* represent the only particular case, in which we can notice a high variability on execution times when considering smaller thresholds, whereas the trend became decreasing, starting from $\varepsilon \geq 0.4$.

Figure 6 also shows the error bars computed on the different executions performed by considering random cut-points on data. However, it is possible to notice that in most cases, the variation in times is tiny. In general, the error bars show a slightly bigger variability when the execution times present a significant gap on average w.r.t. those obtained with the previous smaller threshold. Only few exceptions are highlighted into results of *abalone* and *bridges*.

With respect to the comparison of results, it is clear that BIRD^p outperforms TANE. The latter, differently from BIRD^p, often presents quite constant and very high execution times. In general, the gap among the performances of the two algorithms is big, mainly with higher thresholds. However, the only cases in which similar time performances are obtained by the two algorithms in *abalone* with $\varepsilon = 1$, and *hepatitis* with $\varepsilon \leq 0.2$.

We also analyzed BIRD^p performances with a fixed $g3$-error threshold, and by varying the $P_{\tau+1}$ values. In particular, we varied $P_{\tau+1}$ in the range $[10\%, 90\%]$ with step 10%, and set the $g3$-error threshold to $\varepsilon = 0.2$. Also in this case, BIRD^p execution times are shown in average from 10

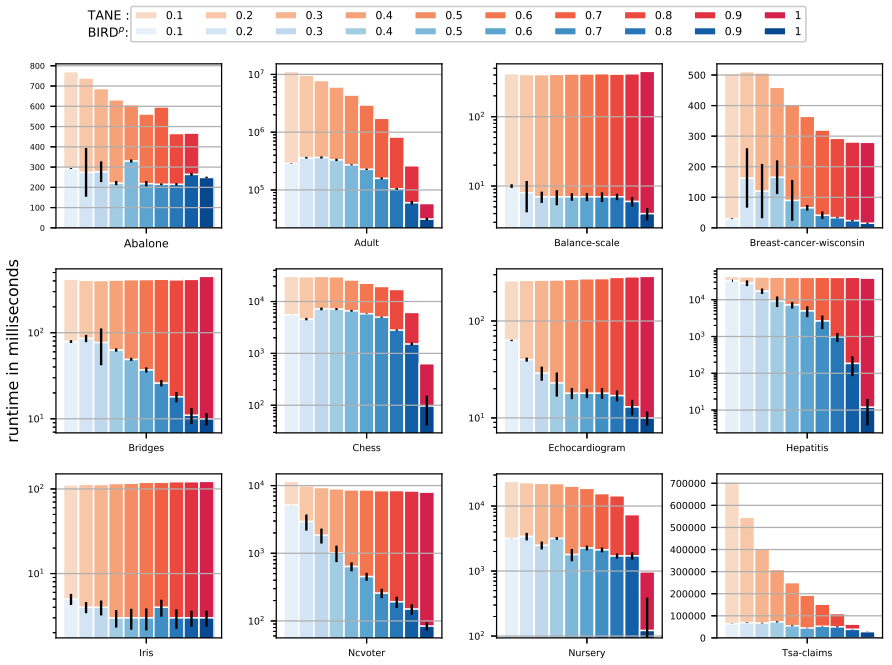

Fig. 6. Time performances with $P_{\tau+1} = 60\%$ and $g3$-error threshold in the range $[0.1, 1]$.

executions, in which we used random cut-points to split the data in two portions according to $P_{\tau+1}$. Moreover, results are always compared with a complete execution from scratch with TANE.

Figure 7 summarizes the obtained results. As opposed to the results achieved by varying the $g3$-error threshold, in this case, it is not possible to infer a general trend. In fact, we can notice a non-monotonic trend on the *abalone*, *balance-scale*, *breast-cancer-wisconsin*, *chess*, and *iris* datasets, a quite constant trend on *echocardiogram*, *nursery*, and *tsa-claim*, and an increasing trend on *adult*, *bridges*, *hepatitis*, and *ncvoter*. The latter trend is the one we expected, since when $P_{\tau+1}$ increases the number of new tuples added from time τ to time $\tau + 1$ is bigger. Thus, BIRD^p could invalidate a high number of RFD$_e$s. Usually, this occurs on datasets with high cardinality, since possible introduced violations could lead to a big variation on holding RFD$_e$s. In general, the achieved results highlight that when $P_{\tau+1}$ increases, the execution times depend on the specific dataset.

According to the error bars, we can notice that in general there is a small variability in execution times. Exceptions can be found for *balance-scale*, *breast-cancer-wisconsin*, and *iris*. The latter are the datasets with a smaller number of holding RFD$_e$s, and also present a non-monotonic trend. This could be due to the fact that the different cut-points can induce the invalidation of all or none RFD$_e$s holding at time τ, since RFD$_e$s are few in general. Finally, with respect to the times obtained from the execution of TANE (red bars) on the complete datasets, we notice that in most cases BIRD^p outperforms TANE with a different order of magnitude. The only exception is highlighted for the *hepatitis* dataset. With this, BIRD^p obtains worse performances w.r.t. TANE, when the number of tuple insertions is greater than 60%. In this case, the TANE algorithm better exploited the pruning strategies. In particular, the *hepatitis* dataset is the one containing the biggest number of RFD$_e$s. Thus, we expected that with many new tuples inserted, BIRD^p has to consider a large variation of holding RFD$_e$s.

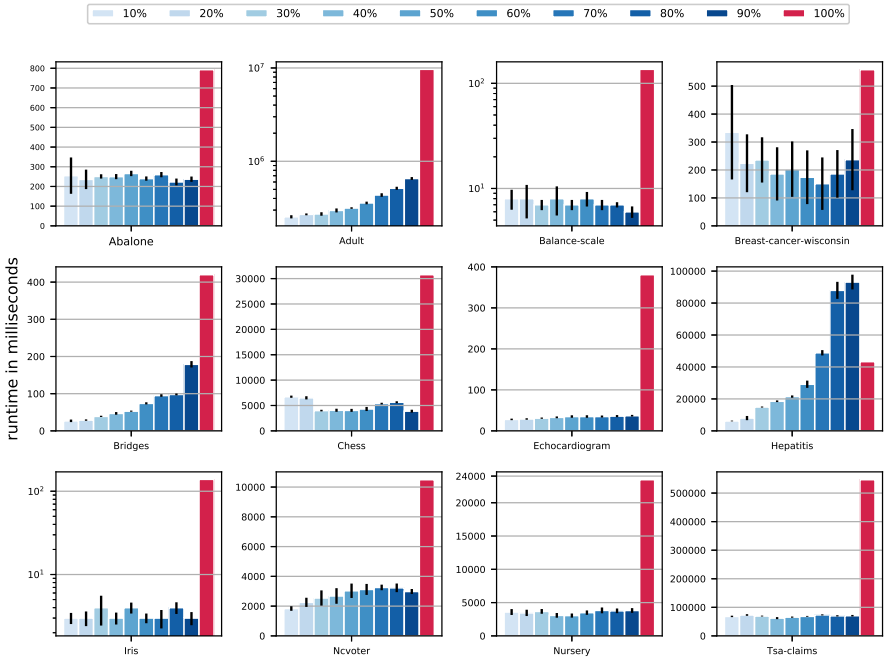

Fig. 7. Time performances with
$g3$-error threshold $\varepsilon = 0.2$ and $P_{\tau+1}$ in the range $[10\%, 90\%]$.

6.2 Scale-up evaluation

We performed a further experimental session to evaluate the communication overhead introduced by the parallelism used by BIRD^p. To this end, we considered a configuration of the algorithm with 2^i available CPUs, varying i in the range $[0, 5]$, $P_{\tau+1} = 50\%$, and $\varepsilon = 0.4$. We analyzed the time performances on *echocardiogram*, *adult*, and *ncvoter* datasets to understand how datasets with different number of rows and columns impacts on the parallelism performances.

Figure 8 shows the execution times obtained by BIRD^p for this experiment. We can notice a decreasing trend when the number of worker threads increases from 1 to 8 threads, especially for *adult* and *ncvoter*. This is due to the fact that worker threads are lock-free and they need to be synchronized at the end of their computation life-cycle, i.e., when the results are included into the final set of RFD$_e$s. This synchronization process represents the biggest BIRD^p overhead, which can worse the time performances when the number of threads exceed a given value, as depicted in Figure 8 for the *echocardiogram* dataset. For the other two datasets we can notice that the execution times reduced the sequential performances by two-thirds for *ncvoter* and one-tenth for *adult*.

7 FINAL REMARKS

We have proposed BIRD, an incremental algorithm for discovering RFDs relaxing on the extent, named RFD$_e$s, i.e. those that can be valid on a subset of the entire dataset. As new data are added to the dataset, BIRD permits to update the set of holding RFD$_e$s without the need to repeat the discovery process on the entire dataset. It relies on a compressed linked ordered map collecting the candidate RFD$_e$s, which allowed us to optimize the search process. Moreover, the generation of candidate RFD$_e$s also follows an incremental approach, since the candidate RFD$_e$s to be generated

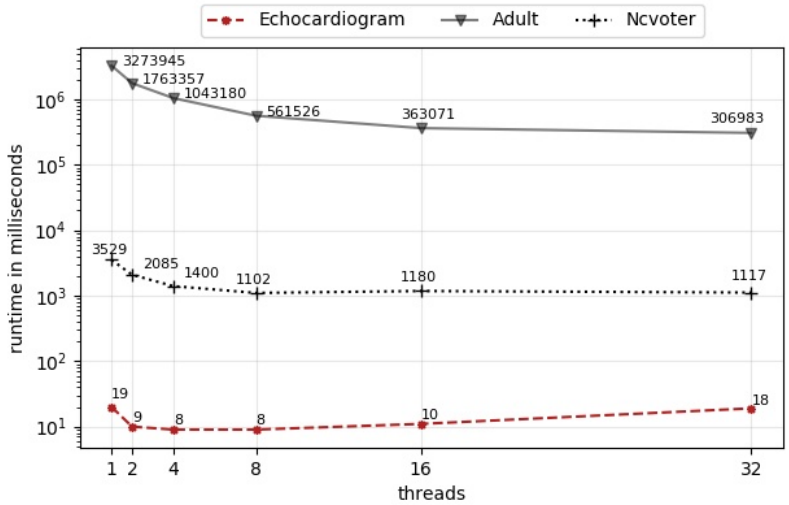

Fig. 8. Scale-up performances of BIRD^p.

change as new validation responses are collected. The level-wise analysis allowed us to define a parallel version of BIRD, in which different thread workers analyze candidate RFD$_e$s on the same level. Experimental results show that the proposed approach considerably reduces the execution times with respect to a re-execution of the discovery process from scratch, and this is further improved with the proposed parallel version.

In the future, we would like to further improve BIRD in order to automatically update RFD$_e$s even upon other database instance modification operations, such as deletion and update of tuples. Moreover, we would like to consider more complex scenarios in which data are continuously generated, like in the context of data streams [26]. Another interesting issue concerns the possibility of updating Relaxed Functional Dependencies relaxing on the attribute comparison (named RFD$_c$s) [6], for which the discovery process is even more complex [5].

REFERENCES

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

[2] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. 2014. DFD: Efficient Functional Dependency Discovery. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM '14)*. 949–958.

[3] K Bache and M Lichman. 2017. UCI Machine Learning Repository. University of California, School of Information and Computer Science, Irvine, CA (2013).

[4] Siegfried Bell. 1995. Discovery and Maintenance of Functional Dependencies by Independencies. In *Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining (KDD '95)*. 27–32.

[5] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2016. On the discovery of Relaxed Functional Dependencies. In *Proceedings of 20th International Database Engineering & Applications Symposium (IDEAS '16)*. 53–61.

[6] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2016. Relaxed Functional Dependencies – A Survey of Approaches. *IEEE Transactions on Knowledge and Data Engineering* 28, 1 (2016), 147–165.

[7] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2017. Evolutionary mining of relaxed dependencies from big data collections. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics (WIMS '17)*. ACM, 5.

[8] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2017. Learning effective query management strategies from big data. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA '17)*. IEEE, 643–648.

[9] David W Cheung, Jiawei Han, Vincent T Ng, and CY Wong. 1996. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of the twelfth international conference on data engineering*. IEEE, 106–114.
[10] Pedro G DeLima and Gary G Yen. 2005. Multiple objective evolutionary algorithm for temporal linguistic rule extraction. *ISA transactions* 44, 2 (2005), 315–327.
[11] Seyed Mostafa Fakhrahmad, MH Sadreddini, and M Zolghadri Jahromi. 2008. AD-Miner: A new incremental method for discovery of minimal approximate dependencies using logical operations. *Intelligent Data Analysis* 12, 6 (2008), 607–619.
[12] Peter A. Flach and Iztok Savnik. 1999. Database Dependency Discovery: A Machine Learning Approach. *AI Communications* 12, 3 (1999), 139–160.
[13] Chris Giannella and Edward Robertson. 2004. On approximation measures for functional dependencies. *Information Systems* 29, 6 (2004), 483–507.
[14] John Grant and Jack Minker. 1985. Normalization and axiomatization for numerical dependencies. *Information and Control* 65, 1 (1985), 1 – 17.
[15] Peter J. Haas, Ihab F. Ilyas, Guy M. Lohman, and Volker Markl. 2009. Discovering and Exploiting Statistical Properties for Query Optimization in Relational Databases: A Survey. *Statistical Analysis and Data Mining* 1, 4 (2009), 223–250.
[16] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1998. Efficient Discovery of Functional and Approximate Dependencies Using Partitions. In *Proceedings of the 14th International Conference on Data Engineering (ICDE '98)*. 392–401.
[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.
[18] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. 647–658.
[19] Ronald S. King and James J. Legendre. 2003. Discovery of Functional and Approximate Functional Dependencies in Relational Databases. *Journal of Applied Mathematics and Decision Sciences* 7, 1 (2003), 49–59.
[20] Jyrki Kivinen and Heikki Mannila. 1995. Approximate inference of functional dependencies from relations. *Theoretical Computer Science* 149, 1 (1995), 129–149.
[21] Sebastian Kruse and Felix Naumann. 2018. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment* 11, 7 (2018), 759–772.
[22] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. 2012. Discover Dependencies from Data - A Review. *IEEE Transactions on Knowledge and Data Engineering* 24, 2 (2012), 251–264.
[23] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient Discovery of Functional Dependencies and Armstrong Relations. In *Proceedings of the 7th International Conference on Extending Database Technology (EDBT '00)*. 350–364.
[24] Heikki Mannila and Kari-Jouko Räihä. 1987. Dependency inference. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*. 155–158.
[25] B Nath, DK Bhattacharyya, and A Ghosh. 2013. Incremental association rule mining: a survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 3, 3 (2013), 157–169.
[26] Felix Naumann. 2014. Data profiling revisited. *ACM SIGMOD Record* 42, 4 (2014), 40–49.
[27] Noël Novelli and Rosine Cicchetti. 2001. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies. In *Proceedings of 8th International Conference Database Theory (ICDT '01)*. 189–203.
[28] 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. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1082–1093.
[29] 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. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1082–1093.
[30] Thorsten Papenbrock and Felix Naumann. 2016. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD '16)*. ACM, 821–833.
[31] Daniel Sánchez, José María Serrano, Ignacio Blanco, Maria Jose Martín-Bautista, and María-Amparo Vila. 2008. Using association rules to mine for strong approximate dependencies. *Data Mining and Knowledge Discovery* 16, 3 (2008), 313–348.
[32] Hemant Saxena, Lukasz Golab, and Ihab F Ilyas. 2019. Distributed Discovery of Functional Dependencies. In *Proceedings of the 35th International Conference on Data Engineering (ICDE '19)*. IEEE, 1590–1593.
[33] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. 2019. Distributed Implementations of Dependency Discovery Algorithms. *Proceedings of the VLDB Endowment VLDB Endowment* 12, 11 (2019), 1624–1636.

[34] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Dennis Hempfing, Torben Meyer, Daniel Neuschäfer-Rube, and Felix Naumann. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT '19)*. 253–264.

[35] Dan A Simovici, Dana Cristofor, and Laurentiu Cristofor. 2002. Impurity measures in databases. *Acta Informatica* 38, 5 (2002), 307–324.

[36] Shyue-Liang Wang, Ju-Wen Shen, and Tzung-Pei Hong. 2001. Incremental discovery of functional dependencies using partitions. In *Proceedings Joint 9th IFSA World Congress and 20th NAFIPS International Conference (Cat. No. 01TH8569)*, Vol. 3. IEEE, 1322–1326.

[37] Ziheng Wei and Sebastian Link. 2019. Discovery and ranking of functional dependencies. In *Proceedings of the 35th International Conference on Data Engineering (ICDE '19)*. IEEE, 1526–1537.

[38] Catharine Wyss, Chris Giannella, and Edward Robertson. 2001. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances. In *Proceedings of 3rd International Conference on Data Warehousing and Knowledge Discovery (DaWaK '01)*. 101–110.

[39] Hong Yao, Howard J. Hamilton, and Cory J. Butz. 2002. FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences. In *Proceedings of the 2nd IEEE International Conference on Data Mining (ICDM '02)*. 729–732.

[40] Lin Zhu, Xu Sun, Zijing Tan, Kejia Yang, Weidong Yang, Xiangdong Zhou, and Yingjie Tian. 2019. Incremental Discovery of Order Dependencies on Tuple Insertions. In *Proceedings of the 24th International Conference on Database Systems for Advanced Applications (DASFAA '19)*. Springer, 157–174.