Information Sciences 690 (2025) 121531

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Non-blocking functional dependency discovery from data streams

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

Department of Computer Science, University of Salerno, Fisciano, Salerno, Italy

ARTICLE INFO

Keywords:
Data stream management
Data stream profiling
Functional dependency
Continuous discovery
Data cleaning
Online learning

ABSTRACT

With the proliferation of sensors and IoT technologies, there is an increasing need to analyze information from data streams that they produce dynamically. However, the volume and velocity of this data require algorithms that mine knowledge as data are read from streams. The capability of dynamically extracting functional dependencies (FDs) from data streams would not only permit to assess and improve the quality of data, but also provide knowledge on the evolution of data correlations within streams, allowing to understand the relevance that each feature has in predicting unknown features. In this paper, we propose a new discovery algorithm, namely COD3, which allows to continuous discovery FDs holding on a data stream, as the data are read from it. COD3 represents the first proposal to use a non-blocking architectural model for discovering FDs from data streams. Furthermore, we present novel data structures and a validation method to handle dynamic discovery and reduce data load inbound streams. Experimental evaluations demonstrate its effectiveness on both adapted real-world datasets and real data streams, such as those from air quality sensors. Moreover, by integrating COD3 with Bleach, a well-known FD-based data stream cleansing framework, we demonstrate its effectiveness in a real-world use case.

1. Introduction

Nowadays, there are many sources of data streams, whose data are generated by traffic sensors, health sensors, transaction logs, and so on. Typically, such sources keep sending data in extremely short time intervals, creating a continuous flow of data that often needs to be processed quickly, without the possibility of fully archiving them and sometimes without any control over the order in which the data arrives from different sensors [1,2]. The need to involve these data in advanced decision-making and analytical processes has raised considerable interest in defining approaches for filtering and cleaning data streams and performing continuous learning tasks [3,4].

In this scenario, Data Profiling plays a critical role, since profiling metadata is a useful tool to evaluate the quality of data and identify anomalies in Big Data sources. Among the profiling metadata, functional dependencies (FDs) capture important semantic relationships among data, which can be exploited for several purposes, such as to identify and repair erroneous values during data preparation steps [5,6], to accomplish feature engineering tasks [7], or to clean data streams [8]. However, designing methodologies for extracting such a type of metadata is per se a complex problem, since they can be exponential in the number of attributes, and their extraction requires analyzing a huge number of attribute combinations [9]. This becomes more complex when considering continuous data streams due to the dynamic nature of the data which makes most existing discovery techniques unsuitable for this purpose. In fact, the continuous discovery of functional dependencies from data streams entails the continuous update of the set of discovered

* Corresponding author.
E-mail addresses: lcaruccio@unisa.it (L. Caruccio), scirillo@unisa.it (S. Cirillo), deufemia@unisa.it (V. Deufemia), gpolese@unisa.it (G. Polese).

https://doi.org/10.1016/j.ins.2024.121531
Received 30 July 2024; Received in revised form 19 September 2024; Accepted 6 October 2024

Available online 10 October 2024

FDs as data are read from the stream. This requires the design of extremely fast and incremental discovery processes, which should also be able to manage considerably long, possibly infinite, executions. Moreover, while in a static dataset, it is important to manage insertion, update, and deletion operations, even if they occur less frequently than queries, in a data stream only the insertion of new tuples matters [1]. Nevertheless, while the focus is primarily on fast and continuous insertion operations, it would be useful to provide the ability to forget information about less recently processed tuples.

In this paper, we present COD3, an efficient and incremental algorithm for discovering all functional dependencies holding on a data stream. It relies on a novel data structure that enables *i*) a fast exploration of the search space according to the discovered functional dependencies, *ii*) a compressed representation of both the data read from the stream and the FDs holding on it, and *iii*) a new fast validation process for functional dependencies. The search strategy underlying COD3 has been defined by extending the one provided in [10]. We have proposed different new data structures, such as the validation graph and the path matrix, which enables COD3 to rely on a completely new efficient validation process. Moreover, differently from [10], COD3 uses a sliding window mechanism to continuously run the discovery process on new data read from the stream, while also keeping track of aging tuples. COD3 exploits a stream processing computation framework, namely Apache Storm, which enabled to combine the flexibility of a non-transactional Storm topology, with innovative discovery techniques and structures to guarantee non-blocking and continuous processing of data. The use of Apache Storm also enabled us to perform two different types of experiments, one on real-world stream adapted datasets, and another on a real data stream derived from air quality sensors. Experimental results demonstrate the effectiveness of COD3 in dynamically extracting all functional dependencies holding on a data stream at a given time instant. Moreover, the usefulness of COD3 has been proved by evaluating its discovery results in a stream data cleaning scenario. We integrated COD3 with Bleach [8], which is one of the most-known frameworks that exploit functional dependencies to detect and repair violations on a dirty data stream.

Paper Organization. Section 2 introduces the existing algorithms for discovering functional dependencies in both classical and incremental scenarios. Section 3 presents background concepts on the discovery problem, by describing both the problem complexity and the data stream management. Section 4 introduces the problem statement in terms of data management in incremental scenarios, whereas Section 5 presents the discovery strategy and the validation process underlying COD3, whose algorithm is detailed in Section 6. Then, experimental results are discussed in Section 7, whereas the application of COD3 in a case study is provided in Section 8. Finally, summary and concluding remarks are included in Section 9.

2. Related work

Among the automatically extracted metadata, functional dependencies can play an important role in data quality [11], query optimization [12], entity matching [13], machine learning [14], and several other contexts. For instance, in the context of machine learning, many proposals use database principles to support machine learning tasks. In [15] the authors use functional dependencies to build decision trees, aiming to derive a more compact structure in order to improve accuracy. In [16] authors present a framework for training and evaluating a class of statistical learning models inside a relational database. In particular, they exploit this profiling metadata to reduce the dimensionality of the underlying optimization problem. In fact, the experimental evaluation demonstrates that the usage of functional dependencies permits to obtain best results by optimizing some parameters that functionally determine others. Moreover, a recent study uses functional dependencies to tackle the problem of evaluating the feasibility of classification in machine learning models [17]. Experimental results show evidence that functional dependencies provide a tight upper bound for the accuracy of classification tasks on real-world datasets.

All these studies highlight the relevance of these metadata in different domains, from advanced data analysis to data quality assessment, up to methodologies supporting decision-making models. Nevertheless, the use of functional dependencies in data stream mining and incremental learning contexts is still an open challenge. Functional dependencies that identify essential relationships among data within evolving datasets, can be used to identify and prevent the degradation of models' performance over time. Furthermore, functional dependencies can serve as a guiding structure, allowing the learning process to prioritize and refine relevant features, thereby reducing the impact of noise and irrelevant information.

In the literature, several discovery algorithms for functional dependencies have been proposed, which are the backbone of several data profiling platforms [18,19]. However, while there are several approaches to extract metadata from data streams [3], including methods for clustering them [20,21], to the best of our knowledge there is no solution for the automatic extraction of functional dependencies from them. Most of the methods for discovering functional dependencies defined in the literature focus on static datasets, and rely on either column- and row-based strategies or a hybrid combination.

Column-based strategies model the search space as an attribute lattice, which permits to consider candidate FDs at each lattice level in terms of edges. Such candidates need to be validated, which entails evaluating value combinations or efficient representations of them. Approaches falling in this category usually exploit several pruning rules during the discovery process in order to reduce the search space. Examples of column-based approaches include the algorithms TANE [22], FD_Mine [23], and DFD [24].

Row-based strategies derive candidate FDs by analyzing the cross product between all possible combinations of tuples pairs, aiming to derive two attribute subsets, namely *agree-set* and *difference-set*. In particular, they search for attribute sets that agree on the values of certain tuple pairs, since they can functionally determine other attributes agreeing on the same tuple pairs. Once all the *agree-sets* are calculated, it is possible to derive all valid functional dependencies from them. Examples of row-based approaches include the algorithms DepMiner [25] and FastFD [26]. It has been experimentally demonstrated that column-based algorithms perform well on datasets with many rows and few columns, while row-based algorithms outperform the first ones on datasets with few rows and many columns [27].

Table 1
Overview of FD discovery algorithms.

Algorithm	Strategy	Type	Exploitation of previous FDs	Temporary storage of data	Type of data source
Tane [22]	Column-based	Static	No	No	Static Data Source
FD_Mine [23]	Column-based	Static	No	No	Static Data Source
DFD [24]	Column-based	Static	No	No	Static Data Source
DepMiner [25]	Row-based	Static	No	No	Static Data Source
FastFD [26]	Row-based	Static	No	No	Static Data Source
HyFD [28]	Hybrid	Static	No	No	Static Data Source
Wang et al. [38]	Column-based	Incremental	No	No	Static Data Source
Bell [39]	Column-based	Incremental	No	No	Static Data Source
DynFD [40]	Hybrid	Incremental	Yes	No	Dynamic Data Source
COD3	Column-based	Incremental	Yes	Yes	Dynamic and Data Streams

The hybrid algorithm HyFD combines both column- and row-based strategies [28]. It accomplishes the discovery process in two separate phases, one in which it calculates functional dependencies on a randomly selected small subset of records (column-efficiency), and the other in which it validates the discovered FDs on the entire dataset. HyFD has been successively extended with a new FD-tree data structure, and a new induction method for traversing the search space [29]. Finally, several distributed versions of well-known FD discovery approaches have been defined and evaluated in [30,31].

In the last few years, the literature has focused on the discovery of functional dependencies and other types of dependencies [32,33], such as inclusion dependencies [34], temporal functional dependencies [35], and approximate functional dependencies, also known as relaxed functional dependencies (RFD) [36]. In particular, since the discovery of certain types of RFDs has proven to be an NP-hard problem, in [37] authors have proposed an approximate column-based algorithm to discover RFDs from given thresholds.

All the discovery algorithms defined above need to be re-executed from scratch whenever the dataset is updated. This can be a big burden, since the size of the dataset is usually big. Thus, it would be desirable to have incremental algorithms capable of dynamically updating the result sets as the dataset instance evolves. The incremental algorithm proposed in [38] exploits the concepts of tuple partition and monotonicity of functional dependencies to avoid the re-scanning of the entire dataset. Another proposal is based on the concept of functional *independency*, through which it is possible to maintain the set of FDs updated over time [39]. Finally, in order to discover and maintain FDs in dynamic datasets, the DynFD algorithm continuously adapts the validation structures of functional dependencies to evolve them with a batch of insert, update, and delete operations [40].

Similarly to the above-mentioned incremental algorithms, COD3 exploits previously discovered functional dependencies to limit the exploration of the search space. Nevertheless, it differs from them since it is also able to continuously monitor data streams for a long time, without the necessity to completely store the whole set of data received from the stream over time. This is made possible by means of a new validation method exploiting novel data structures. In other words, other incremental proposals are mainly devoted to dynamically updating FDs according to transactional operations, by considering instances that dynamically change over time. Instead, COD3 is not related to a specific instance, but it processes all tuples read from the stream, also giving the possibility to forget their related information as time elapses. However, it is worth noting that both algorithms' general search and validation strategies could be adapted to consider both transactional and stream scenarios, i.e., by wrapping tuple arrival/removal in pre-processing steps, accordingly. (See Table 1.)

3. Preliminaries

In what follows we provide preliminaries and basic notations concerning the definition of functional dependencies, the problem of discovering them from data, and issues related to data stream management. Table 2 summarizes the notations used throughout the paper.

3.1. Discovering FDs from data

Given a relation schema R and an instance r of it, an FD holding on r is a statement $X \rightarrow Y$ (X implies Y), with X and Y attribute sets of R, such that 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), respectively, of the functional dependency. In particular, given $Z = \{A_1, \ldots, A_k\}$, $t[Z] \in dom(A_1 \times \cdots \times A_k)$ denotes the projection of t_i onto Z, i.e., the combination of values defined by t over the attributes $\{A_1, \ldots, A_k\}$, also denoted with $\Pi_{A_1,\ldots,A_k}(t)$ or $\Pi_Z(t)$.

An FD is said to be *non-trivial* if and only if $X \cap Y = \emptyset$. Moreover, an FD is said to be *minimal* if and only if there is no attribute $B \in X$ such that $X \setminus B \rightarrow Y$ holds on r. On the contrary, in some cases, it could be useful to consider the concept of *maximal* NON-FD, which defines a candidate FD that is invalid on the instance r with maximal LHS cardinality. More formally, a candidate FD is said to be *maximal* NON-FD if and only if there exists at least one attribute $B \notin X \cup Y$ such that $X \cup B \rightarrow Y$ holds on r.

The problem of discovering FDs from data is an extremely complex one, due to the exponential number of column combinations to be analyzed. More formally, given a relation schema R, with k attributes, and a relation instance r of it, with n tuples, and considering without loss of generality candidate FDs with a single attribute on the RHS, we need to validate each possible candidate FD. Thus,

Table 2
A summary of symbols used throughout the paper.

Symbol	Description
R	Relation schema
r	Relation instance
r_τ	Relation instance at time τ
φ	Functional Dependency
X, Y, Z	Attribute sets
A, B, C	Attributes
a, b, c	Attribute values
t, t^-	New tuple, Expired tuple
$t[B]$	Projection of t on the attribute B
$t[X]$	Projection of t on the attribute set X
$\Pi_Z(t)$	Combination of values defined by t over the attributes in Z
S	Data Stream
w	Number of data items in a sliding window
$S(\tau)_w$	Stream of new tuples at time τ
$S^-(\tau)_w$	Stream of expired tuples at time $\tau + 1$
T	Set of time instants
τ	Time instant
λ	Timestamp related to a tuple t
P_τ	Set of minimal FDs at time τ
Q_τ	Set of maximal NON-FDs at time τ
v_X, v_Y	Binary Vector on the attribute sets X and Y, respectively
G	Validation Graph
Γ	Set of distinct paths connecting a node of G
M	Path Matrix

the problem has a worst case complexity of $O(n^2 \cdot \frac{k}{2} \cdot \frac{k}{2} \cdot 2^k)$, since the number of candidate FDs is $\frac{k}{2} \cdot 2^k$, according to the space of candidates represented as edges of an attribute lattice [41], where the number of possible attribute combinations is 2^k, and the average number of attributes in each combination is $\frac{k}{2}$. Moreover, $n^2 \cdot \frac{k}{2}$ is the complexity of a brute-force approach for validating a candidate FD, where all pairs of tuples are compared on an average of $\frac{k}{2}$ attributes, which corresponds to the average number of attributes involved in an FD [27]. The exponential complexity of the FD discovery problem becomes particularly challenging in the context of data streams due to its high-speed requirements. Nevertheless, new tuples read from the stream can make existing minimal FD no longer valid, possibly yielding new candidate FDs. Thus, in the worst case, even one tuple can require a complete exploration of the search space, entailing the same asymptotic complexity of the general problem. Nevertheless, we analyzed specific issues related to the management of data streams, and the effects of such a dynamic context on the FD discovery problem, which will be detailed in the following sections.

3.2. Data stream management

In the past, data were generally moved through batch processes, with long periods of latency. However, these processes ran at intervals of several hours and there was the risk that processed data would become obsolete. To solve this problem, different data stream management models have been introduced [1]. In particular, such models need to handle real-time analysis processes without creating information queues, and to guarantee information integrity in order to avoid data loss on the stream. Furthermore, each model must view a data stream as a sequence of single tuples in order to simplify information management.

A data stream algorithm takes in input a sequence of data items, also known as tuples, $\langle t_1, \ldots, t_N, \ldots \rangle$, and processes them aiming to minimize the memory space and the average processing time for each stream element, being allowed to scan the sequence only once [42]. The typical approach is to maintain a light summary of the processed information by building a data structure capable of guaranteeing memory usage lower than the stream size.

More formally, given an ordered, infinite set T of discrete time instants, a *data stream* S is a mapping $S : T \to 2^{|r|}$ that at each instant $\tau \in T$ returns a finite subset from the set r of data items with common schema R [43]. Moreover, each data item t is related to a specific timestamp λ, which continuously increases values from T.

Given a data stream S, we need to consider the stream contents, which can vary according to the specific stream settings. In particular, by default the *current stream contents* $S(\tau)$ of a data stream S at time τ is the set $S(\tau) = \{t \in S : t.\lambda \leq \tau\}$ [43]. Moreover, in order to bound the number of data items to the most recent ones, in the context of data stream management, sliding windows are usually applied. Thus, when the collection of items is limited by a *sliding window* with size w, then the *current stream contents* $S(\tau)_w$ of S at time τ is the set $S(\tau)_w = \{t \in S : (\tau - w) < t.\lambda \leq \tau\}$. Consequently, a data item t^- is said *expired* if and only if $t \notin S(\tau)_w$. Furthermore, we can define the *expired stream contents* after the last sliding of the window over a stream S from time τ to time $\tau + 1$ as the set $S^-(\tau + 1)_w = \{t \in S : t \in S(\tau)_w \wedge t \notin S(\tau + 1)_w\}$.

For the FD discovery problem, sliding windows can be used to forget extremely old data items that possibly caused violations for some FDs. To this end, before evaluating novel data items in the new time instant $\tau + 1$, when a sliding window with size w is considered, it is necessary to update FDs according to all data items included in $S^-(\tau + 1)_w$.

4. Problem statement: continuous discovery of FDs

In this paper we analyzed how to combine data stream management models with a novel FD discovery strategy, aiming to continuously find all minimal FDs holding on a growing dataset, i.e., a data stream. In particular, we need to reformulate the traditional FD discovery problem by enabling the updating of minimal FDs from data that are continuously inserted. This represents the problem statement addressed by our proposal. More formally, for each new tuple read at time $\tau + 1$ from the stream, an FD discovery algorithm needs to update the set of minimal FDs P_τ holding at time τ with the set $P_{\tau+1}$ of minimal FDs holding at time $\tau + 1$. Notice that, we assume the set P_τ has already been updated according to the possibly expired tuples. Thus, the discovery problem after the insertion of a new tuple can be managed through an incremental strategy that avoids the complete exploration of the entire search space, aiming to quickly update the set of holding FDs. To this end, we might need to tackle the following issues:

- **Invalidation.** Let $X \to A$ be a minimal FD holding at time τ that is no longer valid at time $\tau + 1$, then it is necessary to consider all possible FD candidates $XB \to A$ such that $B \notin X$ and $B \neq A$.
- **Minimality check.** Let $X \to A$ be a candidate FD at time $\tau + 1$, then it is necessary to check whether there is no FD $X \setminus \{B\} \to A$, with $B \in X$, which has already been validated at time $\tau + 1$.

Notice that, in the case of tuple insertion, the minimality check is required only for candidate FDs generated at time $\tau + 1$. In fact, given a minimal FD $X \to A$ holding at time τ, it is easy to show that if it still holds at time $\tau + 1$, then it will continue to be minimal. In fact, if $X \to A$ is included in P_τ, then there exists at least one violation of any candidate FD $X \setminus \{B\} \to A$ with $B \in X$ already at time τ. Consequently, the insertion of new tuples can never remove the considered violations. This yields pruning strategies that can be applied to the search space to reduce its size.

As previously described, when a sliding window is enabled, it is necessary to consider the fact that some tuples can *expire* over time. This requires keeping the data structures and the set of minimal FDs up to date. COD3 and its underlying discovery strategy manage this particular scenario by enabling an update process of the set of minimal FDs included in P_τ before the processing of new tuples at time $\tau + 1$. In other words, no new tuple is considered at time $\tau + 1$ before processing all expired tuples t^- included in $S^-(\tau)_w$. More specifically, we manage expired tuples through the *Negative Tuple Approach (NTA)* [44], which yields the tuple expiration notification on the pipeline. Thus, expired tuples will be read from the stream as particular tuples, i.e., t^-. This strategy allows us to follow the criterion that incremental algorithms should process the data read from the stream only once or a few times [2]. In fact, a tuple t is processed by the proposed strategy at most twice, i.e., when it is read from the stream and in case it expires t^- according to the sliding window expiration times. Nevertheless, from a theoretical point of view, no novel issue might be considered when considering expired tuples. However, in the context of our strategy, the above-described issues must be taken into account in an alternative way, i.e., by also considering the set of maximal NON-FDs, Q_τ, which can be always determined by using the set of minimal FDs P_τ at time τ. More specifically, when a tuple t^- expires, we might need to tackle the following issues:

- **Validation.** Let $X \to A$ be a maximal NON-FD holding at time τ that is valid after the expiration of a tuple t^-, then it is necessary to add $X \to A$ as minimal FD in P_τ, and remove all minimal FD $XB \to A$ such that $B \notin X$ and $B \neq A$.
- **Minimality check.** Let $X \to A$ be a maximal NON-FD at time τ that is valid after the expiration of a tuple t^-, then it is necessary to check whether in Q_τ there is no FD $X \setminus \{B\} \to A$, with $B \in X$, which are also valid.

5. The discovery strategy underlying COD3

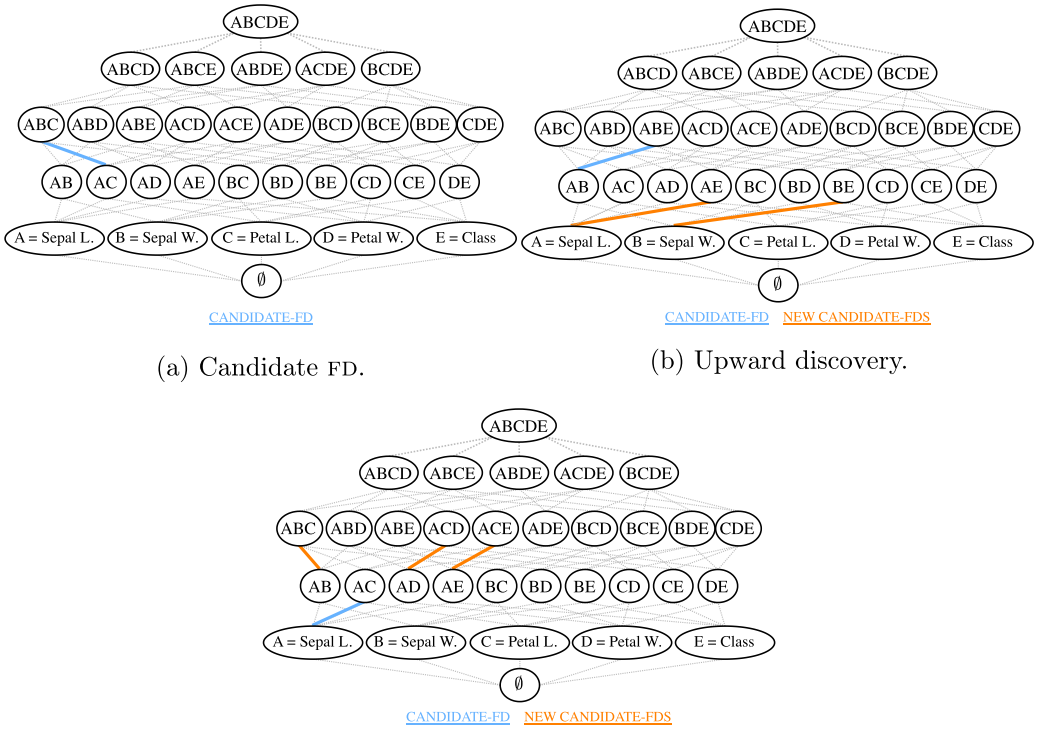
In this section, we provide a brief overview of the discovery strategy, the process topology, and the new validation methodology underlying COD3.

5.1. An overview of COD3 discovery strategy

The proposed approach follows the column-based strategy, which considers the search space as an attribute lattice, where each node contains a unique set of attributes directly connected to supersets or subsets of them. More formally, the lattice is a graph with the following properties: let $R = \{A_1, \ldots, A_k\}$ be the set of attributes of the stream tuples, then the corresponding lattice will contain the empty set at *Level 0*, singleton sets at *Level 1* (i.e., one for each attribute), pair sets at *Level 2* (i.e., one for each possible combination of two attributes), and so forth. Finally, the last level, namely *Level* k, will contain a single set with all the attributes of R. The lattice permits to consider candidate FDs at each level in terms of edges. For instance, the edge highlighted in blue in Fig. 1(a) defines the candidate FD $AC \to B$ (i.e., the common attributes AC determine the non-common attribute B).

In order to represent several FDs in a compact way, we use a binary representation. It consists of two binary vectors v_X and v_Y, representing the LHS and RHS, respectively, of an FD. Such binary vectors contain as many elements as the number of attributes in R, so that if an attribute appears in the LHS (resp. RHS) of an FD the element of v_X (resp. v_Y) associated to it will contain a 1. In this way, all the candidate FDs $X \to A$ sharing the same LHS are compressed in a single pair of vectors (v_X, v_Y), where v_Y contains a 1 for all the attributes determined by X.

Fig. 2 provides an overview of the discovery strategy underlying COD3. It starts by considering a tuple read from the stream and the set P_τ of all minimal FDs holding at time τ. Notice that, P_τ becomes the set of candidate FDs, and will be processed through a linked map, which permits to perform a discovery process in ascending or descending order of the LHS cardinality.



(a) Candidate FD.

(b) Upward discovery.

(c) Downward discovery.

Fig. 1. An example of lattice for the *Iris* dataset.

As said before, according to the NTA approach, the stream can read a novel tuple or an expired one. In the case of a novel tuple t, the strategy underlying COD3 also considers a *path matrix* mapping the impact of the new tuple on the already processed ones, as discussed in the next section. COD3 considers the set P_τ of minimal FDs at time τ as candidates at time $\tau + 1$, and performs the discovery step by processing them in ascending order of the LHS cardinality. Only for the first tuple t read from the stream at time 0, the set of candidates P_τ will consider all the edges connecting lattice level 1 nodes to level 2 ones as FD candidates, that is, those connecting nodes with one attribute to those with two attributes. In particular, for each candidate FD φ at time $\tau + 1$, the process tries to validate it, and if it does not hold (Fig. 2(a)), the process generates new candidate FDs by considering the direct supersets of its LHS. This step is named *upward discovery* (Fig. 1(b)). Vice versa, if φ is valid Fig. 2(b), and it is not contained in P_τ (i.e., it is a minimal FD at time τ), then it undergoes the minimality check, in which COD3 evaluates all the direct subsets of its LHS on the previous lattice level. Such a step is named *downward discovery* (Fig. 1(c)).

At each subsequent iteration, the process verifies the *invalidation* and *minimality check* issues defined in Section 4. In particular, let φ: $X \to A$ be the analyzed candidate FD, based on the result of validation, COD3 generates the set of candidate FDs by performing the following logical operation: the OR operation between the LHS of φ and each attribute of the dataset not contained in it (upward discovery step), and the AND operation between the LHS of φ and each attribute contained in it (downward discovery step). However, both steps exploit a minimality check strategy in order not to consider new candidates that are not minimal. The output of this step is a new set $P_{\tau+1}$ of minimal and valid FDs at time $\tau + 1$.

Example 1. Let us consider the small snippet of the *Iris* dataset[1] shown in Table 3. If the FD $A \to C$ is not valid at time $\tau + 1$, then one or more candidate FDs on the next lattice level could be valid (e.g., $AB \to C$, $AD \to C$, and $AE \to C$) as shown in Fig. 1(b). Vice versa, if the FD $AB \to E$ is valid at time $\tau + 1$, then it is necessary to check if one or more valid FDs on the previous lattice level have not already been validated (e.g., $A \to E$ and $B \to E$). If so, the FD $AB \to E$ is valid but not minimal (Fig. 1(c)).

On the other hand, when COD3 receives an expired tuple t^-, it firstly calculates the set Q_τ of maximal NON-FDs[2] at time τ, which is processed through a linked map. Thus, COD3 performs a discovery step by processing the NON-FDs in descending order of the LHS cardinality. In fact, it is important to notice that, when a tuple expires, no FDs will be invalidated, but some NON-FDs can become valid. To this end, the strategy underlying COD3 firstly checks if there exists at least one NON-FD that is valid. More formally, for

[1] https://archive.ics.uci.edu/ml/datasets/iris.

[2] A procedure to compute Q_τ is shown in [40].

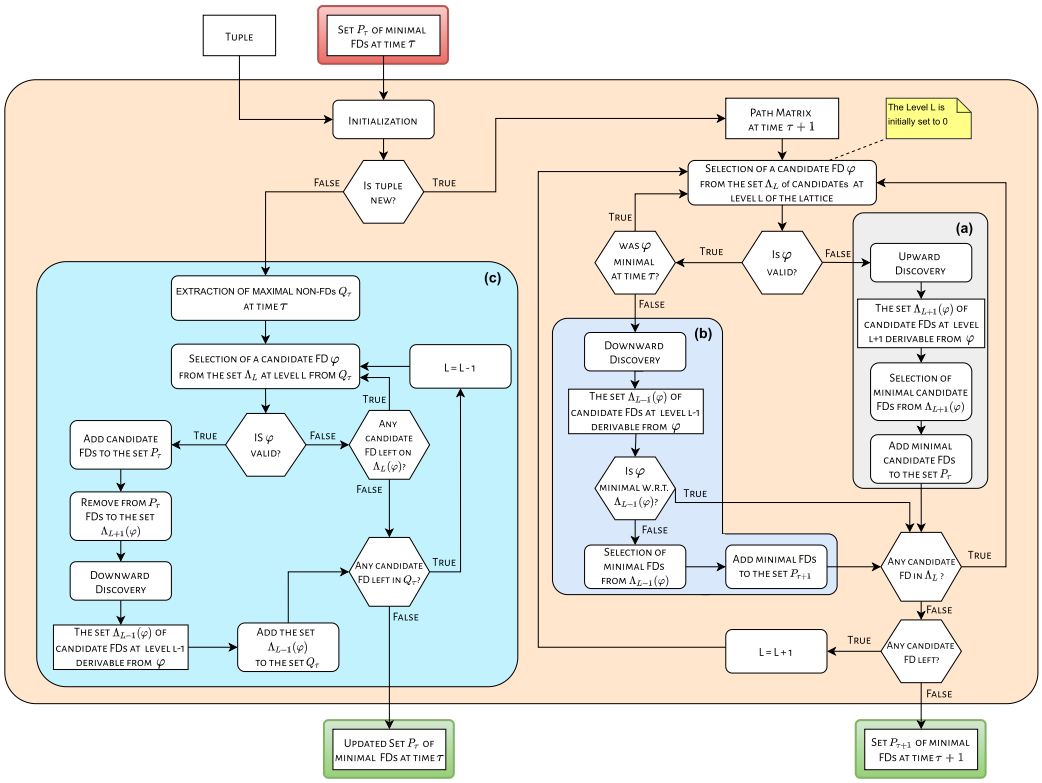


Fig. 2. Overview of the incremental discovery strategy.

each candidate $\varphi : X \rightarrow A$, if φ is valid, then COD3 verifies its minimality with respect to the already validated FDs in the set P_τ, and removes those that become not minimal, i.e., those that can be inferred by the new validated one (Fig. 2(c)). Then, COD3 performs a downward discovery to verify if there exist other NON-FDs at the lowest lattice levels that are minimal with respect to the already considered ones. The output of this step is the updated set P_τ of minimal and valid FDs at time τ.

5.2. The pipeline of COD3

In order to derive an FD discovery process suitable for data streams, we conceived a novel algorithm relying on the concept of a non-blocking strategy, in which it is not necessary to wait for the complete execution of the process on a tuple to start processing the next tuple.

Fig. 3 shows the components of the COD3 pipeline. It consists of two components representing data stream sources enabling the reading of data from static or dynamic sources: *Static Reader Component* and *Dynamic Reader Component*, respectively. The former reads data from any database instance (e.g., a real-world dataset), whereas the latter reads data from external data providers (e.g., Sensors, Social Networks, Streaming APIs, and so forth). Both these components contain a mapping mechanism transforming information into stream tuples by splitting each of them into a list of values, where each value is assigned a name.

In general, COD3 uses a single stream source type at a time whenever it considers heterogeneous sources. Each component reads the data and sends them to the next component devoted to the execution of the pre-processing steps in order to start the discovery process. In particular, in the case of a dynamic source, upon reading a new tuple from the stream COD3 exploits a negative tuple approach (NTA) to check for expired tuples, also integrating a sliding window mechanism within an *NTA Window Component*, which is responsible for reading new tuples from the source. As described above, this mechanism allows COD3 to define a time interval within

Table 3
Snippet of *iris* dataset to illustrate the discovery strategy.

Sepal Length (A)	Sepal Width (B)	Petal Length (C)	Petal Width (D)	Class Label (E)
4.8	3.0	1.4	0.1	Iris-setosa
4.8	3.0	1.4	0.3	Iris-setosa
5.0	2.0	3.5	1.0	Iris-versicolor
5.0	2.3	3.3	1.0	Iris-versicolor

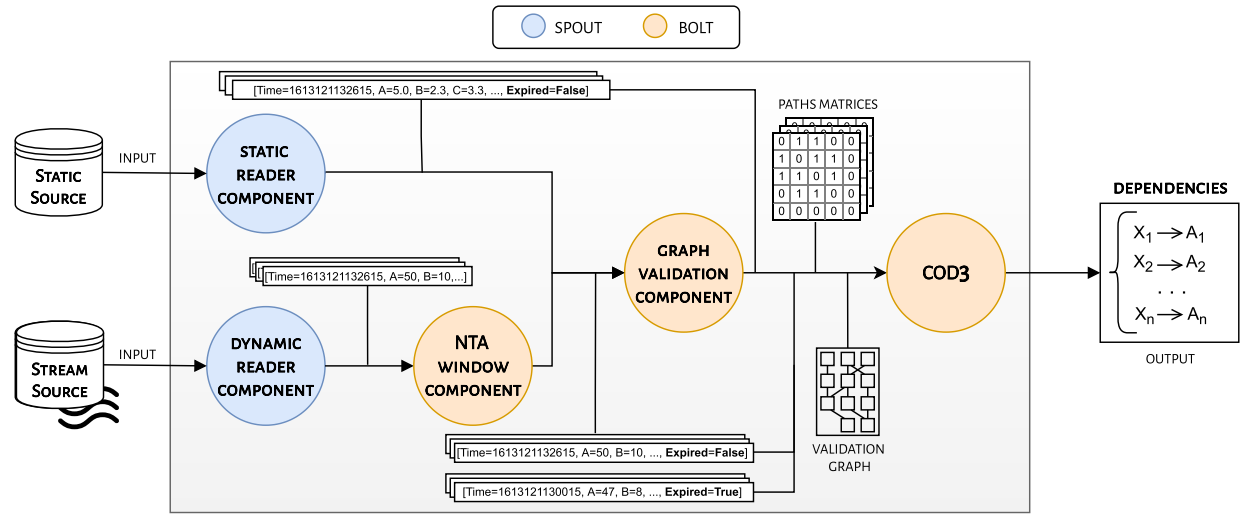


Fig. 3. Overview of the pipeline of COD3.

which the validity and minimality of the discovered FDs are guaranteed. Thus, the window scrolls according to the time interval, possibly causing the expiration of some tuples (denoted by t^-), which will be sent on the stream before processing the new ones. Notice that, the sliding window mechanism is activated according to the COD3 execution settings. By default, COD3 will perform the discovery process by only considering tuple insertions. The *NTA Window Component* sends both new and expired tuples to the *Graph Validation Component*, which in turn updates the data structures (e.g., the validation graph) based on their values. The *Graph Validation Component* can also receive tuples from static sources (e.g., real-world datasets), by means of a *Static Reader Component*, but in this case, no expired tuple is considered.

At the end of the process, the *Graph Validation Component* outputs the updated data structures and sends the read tuples to the next component that is responsible for executing the discovery process. The latter executes the discovery process according to the strategy described in Fig. 2, and extracts all the FDs holding at time $\tau + 1$. Notice that, while the discovery algorithm searches or updates the set of minimal FDs, all the previous components continue to perform their tasks by processing the newly received tuples from the stream.

The pipeline behind COD3 relies on the Apache Storm framework,[3] which is one of the most widely used technologies for managing data streams, mainly due to its adaptability. Apache Storm manages sequences of raw tuples continuously received from data providers as a collection of key-value items, and uses several control mechanisms to minimize the loss of tuples by automatically reintroducing them in the stream. The architecture of an application based on Storm is modeled as a directed acyclic graph (DAG), named *topology*, which represents a graph of independent execution modules, where nodes are some individual components, and edges represent the data passing through nodes. The components can be of type *spout* or *bolt*. A spout normally reads data from an external data source (e.g., messages, database updates, and any other static or dynamic data source), and inserts tuples into the topology. Instead, a bolt receives a set of tuples from its input stream, performs some computations on them, and then optionally inserts a new set of tuples into its output stream. A bolt processes tuples in order to send them to other bolts for further processing steps.

5.3. Graph-based FD validation

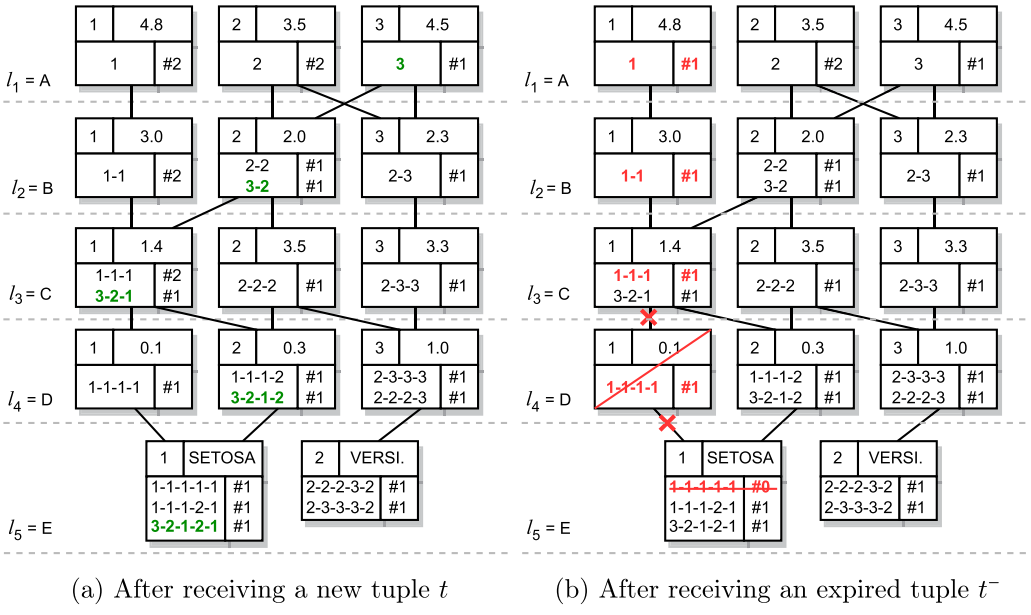
In this section, we describe the validation process underlying COD3. It relies on a new graph structure, called *validation graph*, which stores lightweight references to the tuples that continuously arrive from the stream. More formally, let k be the number of attributes of a relation schema R, a validation graph G is a structure with k levels, where each node at level l_i is associated to a value instantiated in the stream for attribute A_i, whereas each edge connects nodes at adjacent levels only if they represent attribute values appearing at least once in the same tuple read from the stream. Thus, the nodes at level l_i in G contain the value distribution of A_i in a stream of tuples instantiating R.

A node v of G can be defined as a quintuple $v = (A_i, a_i, id, l_i, \Gamma)$, where A_i is an attribute of R, a_i one of its values instantiated in the stream, id is an identifier to distinguish a_i from all other values instantiated for attribute A_i, l_i the level of A_i, and Γ is the set of all distinct paths connecting v to a node at level 1. In particular, an element of the set Γ is a pair containing a path and a counter, which states that the path has occurred that number of times. Clearly, for a node at the l_i-th level Γ will contain paths of length i.

In what follows, we introduce how the validation graph is modified according to the type of tuple read from the stream, i.e., a new tuple or an expired one, t and t^-, respectively.

When a new tuple t is read from the stream, the existing validation graph can be updated as follows:

[3] https://storm.apache.org/.



(a) After receiving a new tuple t (b) After receiving an expired tuple t^-

Fig. 4. An example validation graph for tuples read from the stream.

1) for each level l_i, $1 \leq i \leq k$, a new vertex might be added to l_i if $\Pi_{A_i}(t)$ has never occurred before in the stream;
2) for each level l_i, $1 < i \leq k$, a new edge might be added between node id_{i_j} at level l_i and node id_{i-1_p} at level l_{i-1} only if the projection $\Pi_{A_{i-1},A_i}(t)$ has never occurred in the stream;
3) for each level l_i, $1 < i \leq k$, and for each of its nodes id_{i_j}, a new path of length i might be added to its Γ set only if the projection $\Pi_{A_1,\ldots,A_i}(t)$ has never occurred in the stream.

Example 2. Let us consider the validation graph derived after reading the tuples shown in Table 3 from a stream, and let us suppose that the following tuple is successively read:

$$t = [\text{A} = \text{“4.5”}, \text{B} = \text{“2.0”}, \text{C} = \text{“1.4”}, \text{D} = \text{“0.3”}, \text{E} = \text{“Iris-setosa”}, \text{Expired} = \text{“False”}]$$

The resulting graph is shown in Fig. 4(a). Since only the value 4.5 for attribute A has never occurred before in the stream, only a new node $(A, 4.5, 3, 1, \{3 \mid \#1\})$ is added at level 1; since the values of $\Pi_{A,B}(t)$ have never occurred before, such new node is connected to node with $id = 2$ at level 2, which is in turn connected to node with $id = 1$ at level 3, because also the values of $\Pi_{B,C}(t)$ have never occurred. Finally, the new paths added to the Γ sets of some nodes are highlighted in green.

Otherwise, when an expired tuple t^- is read from the stream, the existing validation graph can be updated as follows:

1) for each level l_i, $1 \leq i \leq k$, a vertex might be removed from l_i if $\Pi_{A_i}(t^-)$ has at most one occurrence, i.e., it is no longer valid in the stream;
2) for each level l_i, $1 < i \leq k$, an edge might be removed between node id_{i_j} at level l_i and node id_{i-1_p} at level l_{i-1} only if the projection $\Pi_{A_{i-1},A_i}(t^-)$ has at least one occurrence, i.e., it is no longer valid in the stream;
3) for each level l_i, $1 < i \leq k$, and for each of its nodes id_{i_j}, a path of length i might be removed from its Γ set only if the projection $\Pi_{A_1,\ldots,A_i}(t^-)$ has at least one occurrence, i.e., it is no longer valid in the stream.

Example 3. Let us consider the validation graph derived after reading the tuples of Table 3 and the one of *Example 2* (Fig. 4(a)), and suppose that before processing new tuples the window scrolls and the following tuple is no longer valid:

$$t^- = [\text{A} = \text{“4.8”}, \text{B} = \text{“3.0”}, \text{C} = \text{“1.4”}, \text{D} = \text{“0.1”}, \text{E} = \text{“Iris-setosa”}, \text{Expired} = \text{“True”}]$$

The resulting graph is shown in Fig. 4(b), whose updates are highlighted in red. Starting from the bottom of the validation graph, the path $\{1-1-1-1-1 \mid \#0\}$ is removed from the Γ of the node with $id = 1$ at level 5, since the values of $\Pi_{A,B,C,D,E}(t^-)$ had one occurrence in the stream. Moreover, given that only the node with $id = 1$ and value 0.1 at level 4 had one path in Γ with one occurrence, the node $(D, 0.1, 1, 4, \{1-1-1-1 \mid \#0\})$ is removed from level 4. Instead, since the values of t^- have occurred more than once on the previous levels by means of other tuples read from the stream, it is necessary to only decrease the number of occurrences

Table 4
The path matrix after the insertion of the tuple in *Example 2*.

		A	B	C	D	E
	0	1	0	0	0	0
A	1	0	1	1	1	1
B	0	1	0	1	1	1
C	0	1	1	0	0	0
D	0	1	1	0	0	0
E	0	1	1	0	0	0

on the correspondent nodes in those levels. In particular, the expiration of t^- entails a decreasing of the counter for: *i*) $\Pi_{A,B,C}(t^-)$, i.e., $(C, 1.4, 1, 3, \{1-1-1 \mid \#1\})$; *ii*) $\Pi_{A,B}(t^-)$, i.e., $(B, 3.0, 1, 2, \{1-1 \mid \#1\})$; *iii*) $\Pi_{A}(t^-)$, i.e., $(A, 4.8, 1, 1, \{1 \mid \#1\})$.

As mentioned in the previous sections, in the case of reading a new tuple from the stream, COD3 exploits an additional structure, named *path matrix*, to further optimize the validation process. The path matrix is a lightweight data structure containing information related to the new nodes and edges that are added to the validation graph after the insertion of a new tuple. It is used by COD3 to isolate cases in which the validation of an FD can be performed instantly. More specifically, when a new tuple t is read from a stream, for each of its attribute values COD3 creates a binary matrix of paths before updating the validation graph G.

Formally, let us consider a stream whose tuples contain k attributes. Upon reading a new tuple t from the stream, a path matrix $M(k+1, k+1)$ with the following properties will be built for t:

1) $M[0][0] = 0$;
2) $M[0][i] = M[i][0]$ where $M[0][i] = 1$ if and only if $t[A_i]$ has never occurred in the stream for attribute A_i, 0 otherwise;
3) $M[i][j] = 1$ if and only if $\Pi_{A_i,A_j}(t)$ has never occurred in the stream for attributes A_i and A_j, 0 otherwise.

The path matrix allows the validation process to prune the search space by catching borderline cases. In what follows, we provide more details on the use of the path matrix for the validation process.

5.3.1. Validation process

The validation process of COD3 exploits the validation graph to check the validity of candidate FDs both in case of insertion of new tuples and in case of expired tuples. More specifically, let $\varphi : X \rightarrow A$ be a candidate FD after the insertion of a new tuple t at time $\tau + 1$, the validation process considers the following four different cases:

- **Case 1.** φ is valid at time $\tau + 1$ if at least one node or an edge linking a pair of attributes in X has been created on G. As an example, let us consider the FD $AC \rightarrow E$ holding on the sample dataset shown in Table 3. Since the insertion of the tuple of *Example 2* yields the creation of a new node for the attribute value $A = 4.5$, $M[1][A]$ ($M[A][1]$, respectively) will be set to 1 as shown in Table 4, and the candidate FD $AC \rightarrow E$ remains valid. The validation process for this FD is also shown in Table 5(a), which represents a snippet of the dataset *iris* for the attributes A, C, and E, where the bottom tuple is the newly inserted tuple. Thus, since the value for the attribute A, e.g., 4.5, is new, the added tuple does not invalidate the FD $AC \rightarrow E$.
- **Case 2.** φ is not valid at time $\tau + 1$ if no new path has been added across attributes in X, and a new node for attribute A has been added to G. As an example, let $C \rightarrow A$ be an FD holding on the sample dataset shown in Table 3. Since the insertion of the tuple of *Example 2* yields the creation of a new node for the attribute value $A = 4.5$, and a path from A to C, then the values of $M[1][A]$ ($M[A][1]$, respectively), and $M[A][C]$ ($M[C][A]$, respectively) are set to 1 (as shown in Table 4), invalidating the FD $C \rightarrow A$ (Table 5(b)).
- **Case 3.** φ is not valid at time $\tau + 1$ if no new node or edge has been added for attributes in X, and no new node has been added for A, but at least one edge linking a node for an attribute in X to a node on A has been added to G. As an example, let us consider the FD $B \rightarrow E$ holding on the sample dataset shown in Table 3. Since the insertion of the tuple of *Example 2* yields the creation of a new edge between the existing attribute values $B = 2.0$ and $E =$ *Iris-setosa*, the value $M[B][E]$ ($M[E][B]$, respectively) will be set to 1, as shown in Table 4, invalidating the FD, as also highlighted in Table 5(c).

Table 5
Example of candidate FDs over a snippet of the *iris* dataset.

Sepal L. (A)	Petal L. (C)	Class (E)
4.8	1.4	Iris-setosa
4.8	1.4	Iris-setosa
5.0	3.5	Iris-versicolor
5.0	3.3	Iris-versicolor
4.5	1.4	Iris-setosa

(a) $AC \rightarrow E$

Sepal L. (A)	Petal L. (C)
4.8	1.4
4.8	1.4
5.0	3.5
5.0	3.3
4.5	1.4

(b) $C \rightarrow A$

Sepal W. (B)	Class (E)
3.0	Iris-setosa
3.0	Iris-setosa
2.0	Iris-versicolor
2.3	Iris-versicolor
2.0	Iris-setosa

(c) $B \rightarrow E$

Petal L. (C)	Petal W. (D)	Class (E)
1.4	0.1	Iris-setosa
1.4	0.3	Iris-setosa
3.5	1.0	Iris-versicolor
3.3	1.0	Iris-versicolor
1.4	0.3	Iris-setosa

(d) $CD \rightarrow E$

- **Case 4.** If none of the above cases occurs, then it is necessary to check how the value paths of all the attributes in a candidate FD are linked to each other. The goal of this case is to verify if a path connecting attributes represents values of at least one previously analyzed tuple. This is due to the fact that the way in which the tuple values are linked does not comply with the transitive property, which could not otherwise be verified if the validation graph does not store value paths on the attribute nodes. For these reasons, when all other cases do not occur, the candidate FD φ is valid at time $\tau+1$ if and only if the projection of t on $X \cup A$ (i.e., $\Pi_{X,A}(t)$) forms a path contained into the Γ set associated to the deepest node among those related to the attributes in $X \cup A$. As an example, let $CD \rightarrow E$ be an FD holding on the sample dataset of Table 3, then after the insertion of the tuple of *Example 2* it is not necessary to generate any new edge connecting the nodes associated to the attributes of the FD (see sub-matrix composed of C, D, E in Table 4). Thus, let v_C, v_D, and v_E be the nodes associated to the attributes C, D, and E, respectively, then it is necessary to check if there exists at least one path in Γ_E, e.g., the set of paths at time τ of the deepest node among those associated to the attributes in $CD \rightarrow E$. In particular, since Γ contains paths of id values, it will be necessary to satisfy the pattern *{?-?-1-2-1}*, where the ? represents any possible value, and $1-2-1$ are the ids obtained from the projection of t onto $\Pi_{C,D,E}(t)$. Therefore, since there exists an item in $\Gamma_E = [\{1\text{-}1\text{-}1\text{-}1\text{-}1\}, \{1\text{-}1\text{-}1\text{-}2\text{-}1\}]$ satisfying the pattern *{?-?-1-2-1}*, then $CD \rightarrow E$ is valid, as also highlighted in Table 5(d).

Notice that, cases 1, 2, and 3 permit to perform the validation process by simply checking the path matrix. Instead, for the case 4 it is necessary to analyze the paths of id values contained in the nodes. However, the validation process is restricted to the analysis of the set Γ of a single node. This strategy permits to quick validate each candidate FD.

The above-described validation process represents the case in which a new tuple is read from the stream. Nevertheless, when a sliding window is set according to a time interval, COD3 also manages a proper validation process when the tuple read from the stream is an expired one, i.e., t^-. In particular, let $\varphi : X \rightarrow A$ be a candidate FD, similarly to Case 4, the validation process requires to check how the value paths of all the attributes in a candidate FD are linked to each other. However, in this case, it is necessary to explore all the nodes at level l_i of G, where i represents the index of the deepest level among those involved in $X \cup A$. Since possible violations of φ can reside in multiple nodes, it is necessary to check if there exist at least two distinct paths (t_1, t_2) within all paths Γ of all nodes in l_i, such that $\Pi_X(t_1) = \Pi_X(t_2)$ and $\Pi_A(t_1) \neq \Pi_A(t_2)$. As an example, let $B \rightarrow A$ be a candidate FD for the sample dataset of Table 3, then after the expiration of the tuple of *Example 3* it is necessary to check the validation of φ. Thus, let $l_2 = B$ be the deepest level involved in φ, it is possible to verify that there exist the two paths $[\{2-2\}, \{3-2\}]$, among the paths Γ_B of all nodes in l_2, which invalidate φ.

6. COD3: the proposed discovery algorithm

COD3 permits not only to update FDs when a new tuple is read from the stream, but also when a tuple expires according to the scrolling of a sliding window. Thus, for the sake of simplicity, we present the general procedure of COD3 in two parts, on the basis of the type of tuple read from the stream: a novel tuple or an expired one. Both the general procedures follow the strategy presented in Section 5.

When a new tuple is read from the stream, the procedure of COD3 is shown in Algorithm 1. Given P_τ the set of all FDs holding at time τ, a tuple t, a path matrix M related to the tuple t, and a validation graph G at time τ, COD3 starts by analyzing the candidate FDs with lowest LHS cardinality (line 1). Then, for each FD holding at time τ, COD3 uses the `INSERTION_VALIDATION` process (line 3) to verify whether $X \rightarrow A$ still holds at time $\tau+1$. This process is performed according to the validation process described in Section 5.3.1. Thus, if the analyzed FD is valid at time $\tau+1$, the algorithm checks if there exist other FDs in the previous lattice levels that have been already validated at time $\tau+1$ (line 4). In particular, if none of them infer the analyzed FD (`INFERENCE`), then it is also minimal at time $\tau+1$ (lines 4-5). Vice versa, if the analyzed FD is not valid at time $\tau+1$, COD3 generates new candidate FDs at a higher lattice level (`NEXTCANDIDATES`), by discarding those that can be inferred from other FDs (`INFERENCE`) already validated at time $\tau+1$ (lines 6-10). Notice that, the ordered discovery of the FDs allows COD3 to avoid the multiple validations of some candidate FDs.

Algorithm 1 COD3 Algorithm.

INPUT: A set P_τ of minimal FDs at time τ, a new tuple t, a path matrix M related to the tuple t, a validation graph G holding at time τ
OUTPUT: A set $P_{\tau+1}$ of minimal FDs at time $\tau+1$

```
1: Σ ← P_τ
2: for each X → A ∈ Σ do
3:   if INSERTION_VALIDATION(X → A, t, M, G) then
4:     if X → A ∉ P_τ and INFERENCE(X → A) then
5:       Σ ← Σ \ {X → A}
6:   else
7:     L_{L+1} ← NEXTCANDIDATES(X → A)
8:     for each W → A ∈ L_{L+1} do
9:       if not INFERENCE(W → A) then
10:        Σ ← Σ ∪ {W → A}
11: P_{τ+1} ← Σ
12: return P_{τ+1}
```

Algorithm 2 COD3 Algorithm for an expired tuple.

INPUT: A set Q_τ of NON-FD at time τ, a tuple t, a validation graph G holding at time τ
OUTPUT: Updated set P_τ of minimal FDs at time τ

1: $\Sigma \leftarrow Q_\tau$
2: **for each** $W \rightarrow A \in \Sigma$ **do**
3: **if** EXPIRATION_VALIDATION$(W \rightarrow A,\ G)$ **then**
4: $Q_\tau \leftarrow Q_\tau \setminus \{W \rightarrow A\}$
5: $L_{L+1} \leftarrow$ NEXTCANDIDATES$(W \rightarrow A)$
6: **for each** $Z \rightarrow A \in L_{L+1}$ **do**
7: **if** $Z \rightarrow A \in P_\tau$ **then**
8: $P_\tau \leftarrow P_\tau \setminus \{Z \rightarrow A\}$
9: $P_\tau \leftarrow P_\tau \cup \{W \rightarrow A\}$
10: $\Sigma \leftarrow \Sigma \cup$ PREVCANDIDATES$(W \rightarrow A)$
11: **else**
12: $\Sigma \leftarrow \Sigma \setminus \{W \rightarrow A\}$
13: **return** P_τ

Algorithm 3 INSERTION_VALIDATION.

INPUT: An FD $X \rightarrow A$ holding at time $\tau + 1$, a new tuple t, a path matrix M related to the tuple t, a validation graph G holding at time τ
OUTPUT: *true* if the FD is valid, *false* otherwise

1: **if** M.containsNewEdges() **then**
2: **for each** $Z \in X$ **do** ▷ Case 1
3: **if** $M[0][Z] = 1$ **then**
4: **return true**
5: **for each** $W \in X$ **do**
6: **if** $Z \neq W \wedge M[Z][W] = 1$ **then**
7: **return true**
8: **if** $M[0][A] = 1$ **then** ▷ Case 2
9: **return false**
10: **for each** $Z \in X$ **do** ▷ Case 3
11: **if** $M[Z][A] = 1$ **then**
12: **return false**
13: $W \leftarrow \emptyset$, $d \leftarrow$ -1
14: **for each** $Z \in (X \cup A)$ **do**
15: **if** $Z.depth() < d$ **then**
16: $W \leftarrow Z$
17: $d \leftarrow Z.depth()$
18: $v \leftarrow G$.getNode$(W, t(W))$ ▷ Case 4
19: **for each** $p \in \Gamma_v$ **do**
20: **if** p.contains$(t[X \cup A])$ **then**
21: **return true**
22: **return false**

The procedure of COD3 for updating the set of candidate FDs P_τ whenever at least one expired tuple is read from the stream is shown in Algorithm 2. Given Q_τ the set of all NON-FDs at time τ, a tuple t, and a validation graph G at time τ, COD3 starts by analyzing the new candidate FDs from Q_τ in descending order (lines 1-2). Then, for each candidate FD, the algorithm checks if it results valid after the expiration of t, by means of the EXPIRATION_VALIDATION process (line 3). Thus, if the candidate FD is valid, COD3 first removes it from the set Q_τ, and then removes all the FDs on the next level from P_τ (NEXTCANDIDATES), i.e., those that can be inferred from the new validated one (lines 4-8), in order to ensure the minimality of the FDs in P_τ. The new FD is then added to P_τ (line 9). Successively, COD3 calculates the candidate FDs at the lower level (PREVCANDIDATES), which will be added to the set of candidate FDs to be processed (lines 10). Vice versa, if the candidate FD is not valid, COD3 simply removes it from the set of candidate FDs (lines 11-12).

Similarly to the general procedure of COD3, and according to the validation strategies described in Section 5.3, also the validation procedure is divided into two strategies with respect to either the insertion or the expiration of a tuple, described in Algorithm 3 and 4, respectively.

In particular, in case of tuple insertion, given an FD $X \rightarrow A$, a new tuple t, a path matrix M related to the tuple t, and a validation graph G, Algorithm 3 implements the FD validation method by exploiting the path matrix and the validation graph. First of all, it is necessary to check if the path matrix related to the tuple t contains at least one new node/edge (line 1). If this is the case, it is possible to apply one of the cases 1, 2, or 3 defined above. More specifically, if the new tuple t has generated at least one node/edge according to Case 1, then $X \rightarrow A$ is valid at time $\tau + 1$ (line 2-7). However, if Case 1 does not occur and attribute A generates a new node, then $X \rightarrow A$ is not valid according to Case 2 (lines 8-9). Otherwise, the algorithm checks whether the values in X are already linked to the values in the RHS, according to Case 3 (lines 10-12), yielding the invalidation of the candidate FD. If none of the above cases occurs, the algorithm checks whether $\Pi_{X,A}(t)$ already exists on the validation graph, and only in this case the candidate FD

Algorithm 4 EXPIRATION_VALIDATION.

INPUT: An FD $X \rightarrow A$ holding at time τ, a validation graph G holding at time τ
OUTPUT: *true* if the FD is valid, *false* otherwise

```
1: i ← -1
2: for each Z ∈ (X ∪ A) do
3:   if Z.depth() < i then
4:     i ← Z.depth()
5: V ← G.getNodesByDepth(i)
6: for each v ∈ V do
7:   if |Γ_v| > 1 then
8:     for each p ∈ Γ_v do
9:       for each h ∈ Γ_v do
10:        if p ≠ h then
11:          if Π_X(p) == Π_X(h) then
12:            if Π_A(p) ≠ Π_A(h) then
13:              return false
14: return true
```

remains valid (lines 14-22). More specifically, it identifies the deepest node v between the attributes in $X \cup A$ (lines 14-17), which allows the algorithm to validate the candidate FD by searching for a path in Γ_v containing the values in $X \cup A$, according to Case 4 (lines 18-22).

On the other hand, in case of tuple expiration, given an FD $X \rightarrow A$ and a validation graph G, Algorithm 4 implements the FD validation method by exploiting the validation graph. First of all, it is necessary to find the level i of the deepest node v between the attributes in $X \cup A$ (lines 1-4). Then, COD3 extracts all the nodes at level i from the validation graph G (line 5), and for each of them it checks if the node contains at least two distinct paths (line 7). If true, COD3 compares each pair of distinct paths, and checks whether there exists at least a pair of paths p and h, respectively, such that $\Pi_X(p)$ equals $\Pi_X(h)$, while $\Pi_A(p)$ differs from $\Pi_A(h)$ (lines 8-13), yielding $X \rightarrow A$ not to be valid at time τ (lines 11-13). If none of the nodes at level i invalidate $X \rightarrow A$, then the FD continues to be valid at time τ (line 14).

7. Experimental evaluation

In this section, we describe the evaluation of COD3 on several real-world datasets and real-time streams, by also providing a qualitative analysis of the metadata extracted from a sensor-based data stream, with the aim of analyzing how metadata evolves over time.

Implementation details. COD3 has been developed in Java 12, and it has been integrated into Apache Storm 2.1.0. Furthermore, COD3 exploits the pipeline programming model of Apache Storm in order to guarantee suitable performances, continuous processing, and a trade-off between consistency, speed, and durability.

Hardware and Datasets. The experiments have been executed on an iMac Pro with an Intel Xeon CPU at 3.20 GHz and 18-cores, running macOS Mojave 6.4 and OpenJDK 12.0.2 as Java environment. The experiments were performed on several real-world datasets,[4] previously used for evaluating FD discovery algorithms. Table 6 shows the details of the evaluation datasets.

Evaluation process. In our experimental session, we performed two different types of tests to evaluate COD3 on static and dynamic sources. In the first experiment, we simulated a scenario of continuous tuple insertions by transforming datasets into dynamic sources through the COD3 pipeline components. Although in this kind of experiment COD3 is not used for what it has been conceived, we considered it in order to perform a sort of comparative evaluation with respect to well-known FD algorithms, since to the best of our knowledge there is no similar algorithm capable of directly extracting FDs from data streams. In particular, we compared COD3 with the FD discovery algorithms HyFD [28] and DynFD [40], which focus on the discovery of FDs from static and dynamic datasets, respectively. Instead, in the second experiment, we evaluated the effectiveness of COD3 on a data stream of sensors data provided by the *AQICN data portal*.[5]

7.1. Performances on real-world datasets

Our first experiment measured the execution times of COD3 on different real-world datasets (see Table 6), which are mapped into a continuous stream of data by following the strategy described in Section 5.2. The considered datasets have a different number of rows and columns in order to highlight how the COD3 execution times vary according to such parameters. In our test, we evaluated the execution times of COD3 by considering the first tuple and the initial runs of the algorithm, up to the last run on the last tuple.

Analysis of Results. Fig. 5 summarizes the time performances of COD3 for each dataset, in terms of boxplots built on the execution time distribution per tuple. The figure also reports the memory peaks reached by COD3 on each execution. As we can see, the median values of the execution times are almost always less than 10 milliseconds per tuple, except for some of the biggest datasets.

[4] https://github.com/DastLab/TestDataset.
[5] https://aqicn.org/.

Table 6
Details of the considered real-world datasets.

Dataset	Cols [#]	Rows [#]	FDs [#]
Iris	5	150	4
Chess	7	28,056	1
Abalone	9	4,177	137
Electricity	9	45,312	61
Bitcoin Heist	10	2,916,698	18
Poker-hand	11	264,027	1
Echocardiogram	13	132	538
Tsa-claims	13	25,023	129
Adult	14	32,562	60
Fd-reduced	15	250,000	4,908
Ncvoter	19	1,001	3,179
Lymphography	19	148	2,730
Hepatitis	20	155	8,250
Parkinsons	24	195	1,724
MoCap Postures	38	78,095	4,094
Sonar	60	208	97,750
Movement-libras	91	360	2,473,105
Gas-Sensors	128	4,000	302,705

In particular, for *Hepatitis*, *Sonar*, and *Gas-Sensors* the median values fall in the range $[1 - 10]$ seconds, whereas on *Movement-libras* dataset exceeds other execution times, mainly due to the thousands of functional dependencies to be validated in many executions. In general, we notice that the execution times of COD3 present small distributions (upper and lower quartiles), even though some outliers occurred, especially with datasets containing many tuples.

With respect to memory peaks, the results show that no relationship can be derived between the memory load and the dataset characteristics in terms of the number of rows and columns. However, we can observe that memory peaks slightly depending on the quantity of discovered FDs. For instance, the worst memory loads are registered for datasets exceeding two thousand holding FDs. More specifically, the memory bound is related to the number of invalidations caused by the arrival of new tuples, which more likely occur as the number of holding FDs increases.

In general, COD3 achieves good performances when the insertion of new tuples yields few invalidations. In fact, since the discovery process is performed level-by-level when one or more FDs are invalidated, COD3 considers new FD candidates from the next level of the invalidated ones.

Fig. 6 reports the average execution times of the validation process for each FD upon the insertion of a tuple (green line), also compared to the average number of validations performed for each of the four cases (colored bars). The results highlight that the average time is always less than 1 millisecond per FD, but in most cases, it does not exceed 0.2 milliseconds. The only exceptions are for *Poker-hand*, *Tsa-claims*, and *Sonar* datasets.

Concerning the number of times a specific case is executed during all validations, we can notice that for each dataset the majority of validations only exploit path matrices (Cases 1, 2, and 3), which makes the process faster. Particularly interesting are the results achieved on the biggest datasets, i.e., *MoCap Postures*, *Hepatitis*, *Sonar* and *Gas-Sensors*, where the validation process instantiates Case 4 only a few times, yielding extremely low average execution times, despite a huge number of holding functional dependencies.

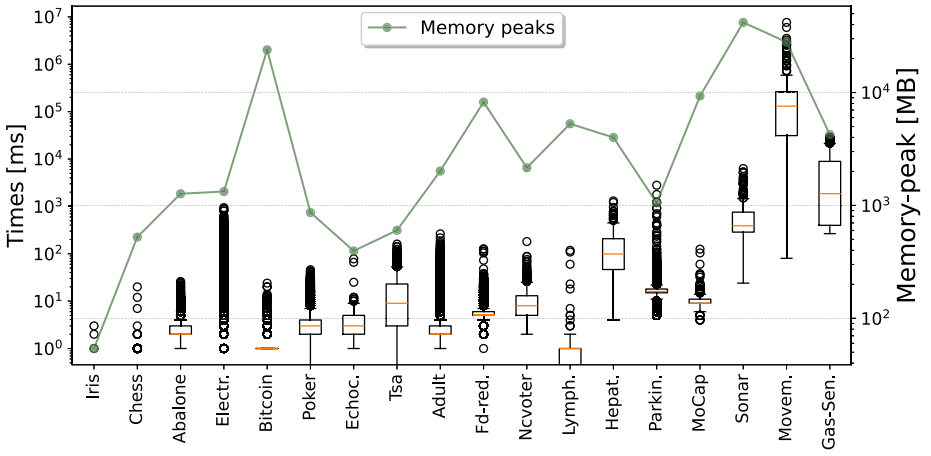


Fig. 5. Performances of COD3 over real-world dataset.

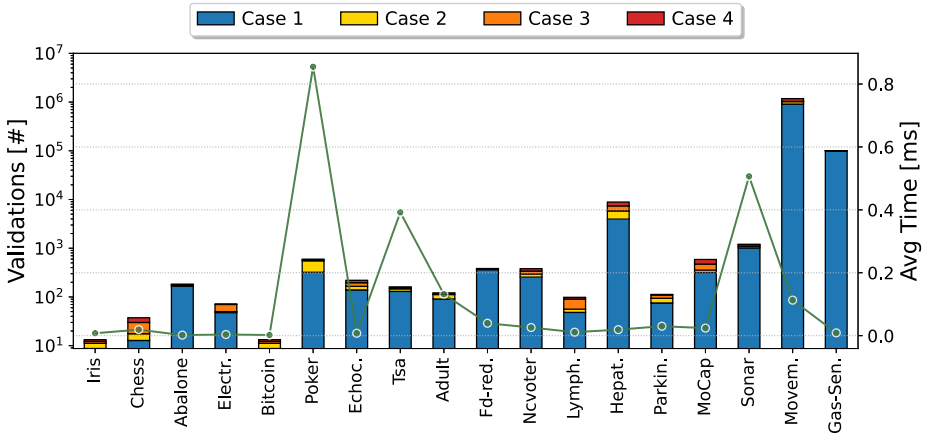


Fig. 6. Number of validations for each case defined in Algorithm 3.

7.2. Comparative evaluation

In this section, we compare the execution performances of COD3 to those of one of the best-performing non-incremental discovery algorithm (HyFD [28]), and an analogous incremental one (DynFD [40]), by considering different real-world datasets (see Table 6). In particular, HyFD is a static discovery algorithm that combines approximation techniques with several validation strategies in order to discover the set of all minimal FDs holding on a static dataset. We will show all the conditions in which COD3 under- or out-performs such a static discovery algorithm. To this end, we gradually scale up the size of the dataset, starting with a dataset containing one tuple and adding one tuple at a time, each time executing HyFD on the augmented dataset.

Instead, DynFD is an incremental discovery algorithm, which extends HyFD with the possibility of updating the set of holding functional dependencies in accordance with insert, delete, and update operations collected in batch mode. In order to compare COD3 and DynFD, we set up an insertion operation in the batch file for each tuple inserted in the dataset. More specifically, we start with a dataset containing only one tuple and simulate the insertion of one tuple at a time. To execute COD3 over static datasets, we transform the considered datasets into a continuous data flow, according to the pipeline component described in Section 5.2.

Fig. 7 shows the results of the comparative evaluation in terms of the variability of average execution times (plot at the top) and the memory load (plot at the bottom). In particular, the results show that COD3 is almost always faster than HyFD as the number of processed tuples grows, especially on the *Poker-hand*, *Fd-reduced*, and *MoCap Postures* datasets. Furthermore, we can see that COD3 has poor performances during the first runs on *Iris*, *Abalone*, *Echocardiogram*, *Ncvoter*, *Parkinsons*, *Lymphography*, and *Sonar* datasets. This is probably due to the fact that the number of validations and invalidations is quite high when the datasets contain few tuples. Moreover, COD3's poor performances on *Movement-libras* and *Gas-Sensors* datasets are due to the fact that these datasets have a large number of columns and many FDs are discovered already at the initial runs. An exception is the *Sonar* dataset, in which the execution times of both algorithms appear similar as the number of tuples increases, even though *Sonar* represents one of the biggest datasets.

Concerning the comparison between COD3 and DynFD, the results show that COD3 always outperforms DynFD in terms of execution times, except for the first tuples of the *Abalone* dataset and for the *Gas-Sensors* dataset. Similarly to HyFD, DynFD often achieves lower execution times during the first runs. This is mainly due to the fact that DynFD integrates the HyFD algorithm during initial steps to define all the underlying configurations, in order to successively work in a dynamic scenario.

It is worth noting that for *Movement-libras* and *Sonar* datasets, no results can be discussed for DynFD, since its executions exceeded the memory limit, which has been set to 50 GB. To this end, as expected, the incremental discovery algorithms (i.e., COD3 and DynFD) always require a greater amount of memory with respect to HyFD, due to the information of the previous executions that they manage. Nevertheless, COD3 almost always outperforms DynFD in terms of memory load, except for *Parkinsons* and *MoCap Postures* datasets. In general, time performance results show that COD3 can process a huge quantity of rows with suitable performances, without raising any memory issues. For instance, by considering the dataset with the highest number of rows, i.e., *Bitcoin Heist*, which contains over 2.9 million of tuples, COD3 achieves execution times extremely lower than both compared algorithms without exceeding the memory limit. This makes COD3 particularly useful for the data stream context, where the number of rows can be extremely large. Moreover, the number of attributes considered in the data stream context is generally not large, unlike the number of rows.

7.3. Discovery results over data streams

In this experiment, we measured the effectiveness of COD3 on a real-world data stream. More specifically, this experiment exploits the data of over 200 real sensors spread throughout Italy, made available by the *AQICN* portal, which monitors and shares the air quality information during the day. In particular, we selected the following 13 attributes from the data stream:

- `Particles` $PM_{2.5}$ and PM_{10} represent atmospheric aerosol particles, also known as "floating dust" or particulate matter (PM) with a diameter of 2.5 ($PM_{2.5}$), or 10 (PM_{10}) micrometers;

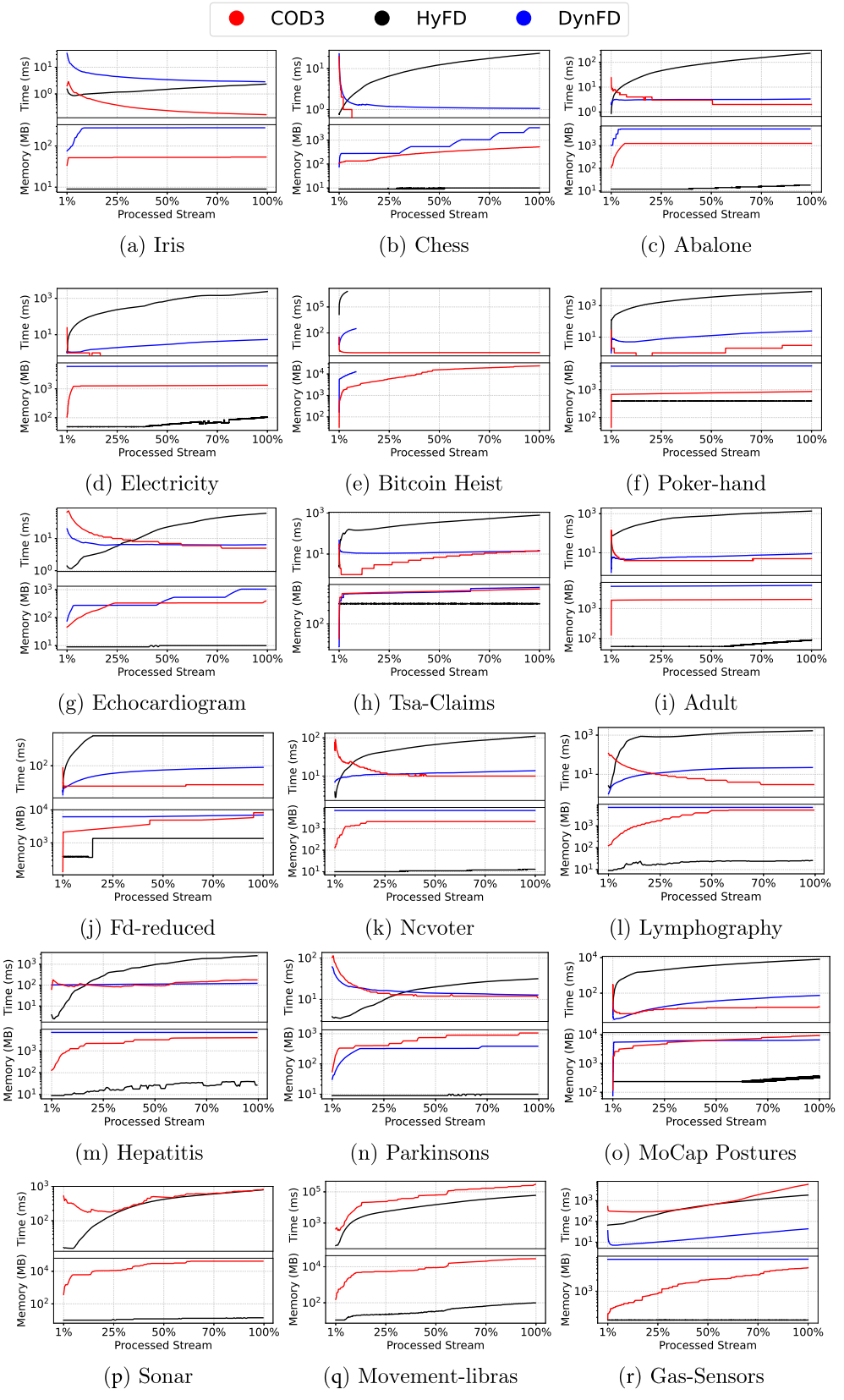


Fig. 7. Time performances and memory load by considering the variation of FDs at any time.

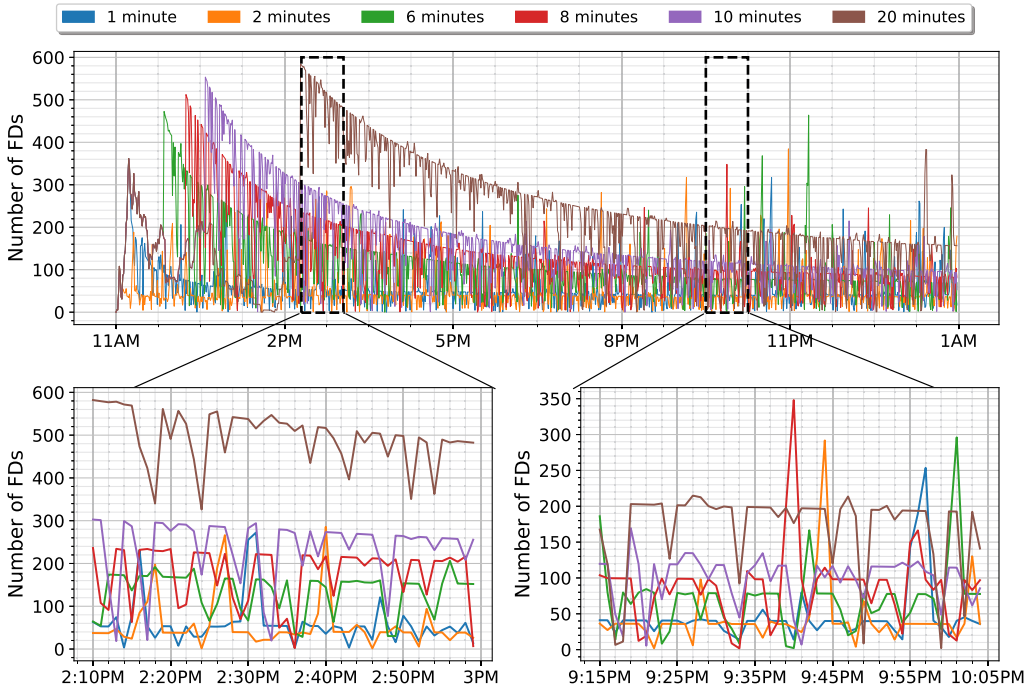


Fig. 8. Number of discovered FDs considering five expiration time intervals.

- `Nitrogen dioxide (NO2)` represents the nitrogen dioxide concentration;
- `Sulfur dioxide (SO2)` represents the sulfur dioxide concentration;
- `Dew point (dew)` represents the temperature to which air must be cooled to become saturated with water vapor;
- `Ozone (O3)` represents the ozone concentration;
- `Temperature (t)` represents temperature in centigrade degrees;
- `Humidity (h)` represents the rate of humidity;
- `Wind direction (w)` and `Wind speed (wg)` represent the information on wind direction and speed, respectively;
- `Pressure (p)` represents the value of the atmospheric pressure;
- `Rain (r)` represents the amount of rain that fell at the time of measurement;
- `Carbon monoxide (CO)` represents a colorless, odorless, and tasteless flammable gas that is slightly less dense than air.

We have considered execution sessions of COD3 on the air quality data streams, lasting about 13 hours. In particular, we set up six parallel executions by considering five different sliding windows, i.e., 1, 2, 6, 8, 10, and 20 minutes, which determine the expiration times of the tuples. The considered sliding window settings allowed us to properly emphasize how the discovery results vary according to the different times. Notice that, in a real-world scenario the sliding window times should be set up according to the nature of the application domain.

The curves shown at the top of Fig. 8 highlight the variability of holding functional dependencies discovered with the different sliding windows, whereas the zooms at the bottom provide more details in two different time windows. In particular, the line of each sliding window shows the standard deviation in the number of FDs with respect to the average number of holding FDs computed up to that time instant. As expected, the results show that the trend undertakes a bigger variability at the beginning of the discovery process, and tends to converge throughout the execution. This is particularly true for larger sliding windows. In fact, with sliding windows of 1 and 2 minutes, the trends remain almost stable right after processing the initial tuples. This can be due to the fact that the invalidation of functional dependencies does not significantly impact the number of holding dependencies, since each analyzed tuple expires in a short time. Larger sliding windows obtain their peaks afterwards, due to the fact that the sliding windows start to move later, entailing a cold start of the expiration discovery process.

In order to extract different information from air quality sensor data, and to describe the existing relations among the analyzed parameters, we compared the set of FDs mined during each execution. The results are summarized in Table 7, where for each sliding window, we grouped the results into time periods representing the two different halves of the execution period, i.e., 13 hours, and the whole period. For each of them, we report the most common attribute on the left- and right-hand-side, and the average LHS cardinality, among all FDs holding in a specific time period. Top-5 FDs in such periods are also shown, representing the most validated FDs across the different time instants.

From the results of Table 7 we can notice that the feature `Particles` (PM_{10}) appears as the most common attribute on the LHS when the sliding window is set to 1 or 2 minutes. Instead, with the larger sliding windows the attribute `Particles (PM10)` disappears,

Table 7
Summarized results obtained by COD3 across different execution sessions on real streams.

Sliding Window	Time Period	Most Common Attribute $\overline{LHS}$	Most Common Attribute RHS	$\lvert\overline{LHS}\rvert$	Top 5 FDs
1 (min)	11 AM - 5 PM	Particles (PM_{10})	Rain (r)	3	(p, dew, h) → (r) (PM_{10}, p, h) → (r) (p, t, h) → (r) (p, h, w) → (r) (p, O_3, h) → (r)
	5 PM - 12 AM	Pressure (p)	Rain (r)	4	(p, t, h) → (r) (PM_{10}, t, h) → (r) ($PM_{2.5}$, p, h) → (r) (p, t, dew, SO_2) → (r) (PM_{10}, p, h) → (r)
	13 h	Particles (PM_{10})	Rain (r)	5	(p, t, h) → (r) (p, dew, h) → (r) (PM_{10}, p, h) → (r) (NO_2, p, t, w) → (r) (p, t, CO, w) → (r)
2 (min)	11 AM - 5 PM	Particles (PM_{10})	Rain (r)	3	(p, CO, h) → (r) ($PM_{2.5}$, p, h) → (r) (p, t, h) → (r) (p, SO_2, w) → (r) (p, O_3, h) → (r)
	5 PM - 12 AM	Pressure (p)	Rain (r)	6	(PM_{10}, p, dew, h, w) → (r) (p, O_3, dew, h, w) → (r) (p, dew, SO_2, h, w) → (r) (PM_{10}, $PM_{2.5}$, NO_2, p, O_3, h, w) → (SO_2) ($PM_{2.5}$, p, dew, h, w) → (r)
	13 h	Particles (PM_{10})	Rain (r)	5	(p, dew, h) → (r) ($PM_{2.5}$, p, h) → (r) (NO_2, p, t, dew) → (r) (p, t, h) → (r) (p, CO, w) → (r)
6 (min)	11 AM - 5 PM	Wind (w)	Rain (r)	5	($PM_{2.5}$, p, h, w) → (r) (PM_{10}, t, dew, SO_2, w) → (r) (p, dew, SO_2, h, w) → (r) (p, O_3, h, w) → (r) (PM_{10}, $PM_{2.5}$, p, O_3) → (SO_2)
	5 PM - 12 AM	Particles ($PM_{2.5}$)	Rain (r)	5	(PM_{10}, $PM_{2.5}$, NO_2, t, SO_2) → (CO) (PM_{10}, t, h) → (r) (p, h, t) → (r) ($PM_{2.5}$, dew, w) → (r) (dew, p, t) → (r)
	13 h	Pressure (p)	Rain (r)	6	(p, dew, h, w) → (r) (p, dew, SO_2, w) → (r) (p, O_3, dew, h) → (r) (NO_2, p, O_3, h, w, wg) → (dew) ($PM_{2.5}$, t, h) → (r)
8 (min)	11 AM - 5 PM	Dew point (dew)	Rain (r)	5	(p, SO_2, h, t) → (r) ($PM_{2.5}$, dew, p, t) → (r) ($PM_{2.5}$, dew, w) → (r) (PM_{10}, t, h) → (r) ($PM_{2.5}$, NO_2, w) → (CO)
	5 PM - 12 AM	Particles ($PM_{2.5}$)	Rain (r)	5	(p, SO_2, h, t) → (r) ($PM_{2.5}$, dew, p, t) → (r) ($PM_{2.5}$, dew, w) → (r) (PM_{10}, t, h) → (r) ($PM_{2.5}$, NO_2, w) → (CO)
	13 h	Pressure (p)	Rain (r)	6	(PM_{10}, $PM_{2.5}$, NO_2, t, SO_2) → (CO) (PM_{10}, t, h) → (r) (p, h, t) → (r) ($PM_{2.5}$, dew, w) → (r) (dew, p, t) → (r)
10 (min)	11 AM - 5 PM	Dew point (dew)	Rain (r)	5	(PM_{10}, p, t, dew) → (r) (SO_2, dew, PM_{10}, w) → (r) (p, SO_2, h) → (r) ($PM_{2.5}$, t, CO) → (h)
	5 PM - 12 AM	Particles ($PM_{2.5}$)	Rain (r)	4	($PM_{2.5}$, NO_2, w) → (CO) (PM_{10}, t, h) → (r) (p, SO_2, h, t) → (r) ($PM_{2.5}$, dew, w) → (r) ($PM_{2.5}$, dew, p, t) → (r)
	13 h	Pressure (p)	Rain (r)	6	(p, $PM_{2.5}$, t, w) → (r) (p, dew, w) → (r) (p, CO, dew, t) → (r) (NO_2, p, O_3, h, t) → (dew) ($PM_{2.5}$, t, h, dew) → (r)
20 (min)	11 AM - 5 PM	Pressure (p)	Rain (r)	3	(p, dew, h, NO_2) → (r) (p, dew, w, O_3) → (r) (PM_{10}, p, h) → (r) (p, SO_2, h, t) → (r) (p, NO_2, O_3, h, t) → (r)
	5 PM - 12 AM	Pressure (p)	Rain (r)	4	(p, dew, SO_2, h, NO_2) → (r) (p, NO_2 h) → (r) (PM_{10}, t, h, dew) → (r) ($PM_{2.5}$, p, h, NO_2) → (r) ($PM_{2.5}$, t, h, SO_2) → (r)
	13 h	Pressure (p)	Rain (r)	5	(p, CO, t, h) → (r) (p, dew, h, SO_2) → (r) (PM_{10}, p, NO_2, t) → (r) (NO_2, h, dew) → (r) ($PM_{2.5}$, t, h, SO_2, w) → (r)

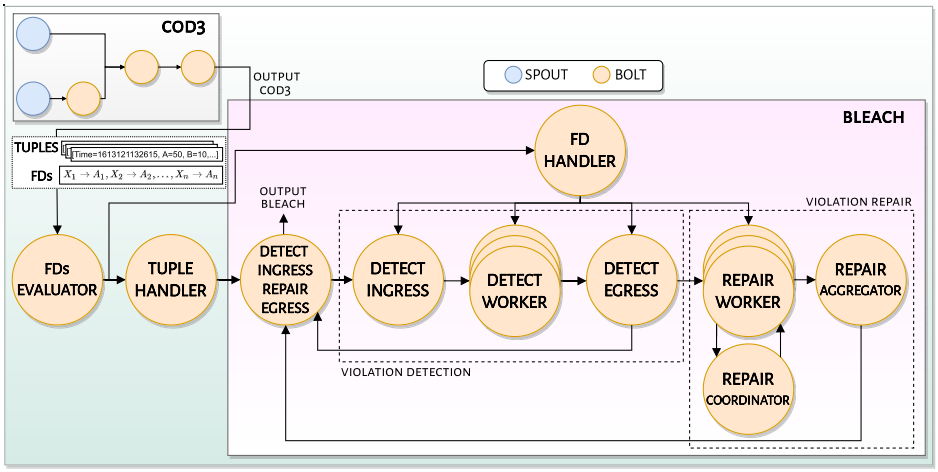


Fig. 9. The Bleach* pipeline.

and the attribute `Pressure (p)` is the most common LHS. On the contrary, the attribute `Rain (r)` represents the most common RHS attribute (e.g., the most implied) with all sliding windows. As we expected, the cardinality of LHSs increases in average with larger sliding windows. This is mainly due to the fact that more tuples possibly induce more violations, yielding to include more attributes on the LHS of holding FDs.

In what follows, we list some of the most frequent holding functional dependencies discovered with a sliding window of 1 minute, which are shared among the different time periods:

`Pressure(p), Humidity(h), Dew Point(dew)` → `Rain(r)`,
`Pressure(p), Humidity(h), Temperature(t)` → `Rain(r)`,
`Pressure(p), Humidity(h), Particles(`PM_{10}`)` → `Rain(r)`.

These FDs highlight a strong relationship between humidity, pressure, and rainfall. In particular, the humidity and the pressure with the same other data can imply the rainfall. In general, `Rain(r)` always appears as RHS in the most frequent FDs across all considered time periods. Instead, with the other sliding window not only different periods seem to include different FDs among the most frequent ones, but the latter typically have a greater LHS in size and sometimes imply a RHS attribute different from `Rain(r)` (e.g., `Sulfur dioxide(`SO_2`)`, `Dew Point(dew)`, and `Carbon monoxide(CO)`).

8. Case study: automating a stream data cleaning process with COD3

In this section, we evaluated the usefulness of COD3 in the context of stream data cleaning by integrating it with Bleach [8], a well-known tool that exploits functional dependencies to detect and repair violations in a dirty data stream. Traditionally, Bleach requires the manual specification of functional dependencies, a process that can be both time-consuming and error-prone, particularly in dynamic environments where the underlying data characteristics may evolve rapidly. By automating the extraction of functional dependencies with COD3, we enhance Bleach's capability to adapt to these changes over time, ensuring that the cleaning process remains effective as new data arrives.

8.1. Integration of Bleach and COD3 algorithm

Bleach relies on efficient and distributed data structures that allow it to store light summaries of data, which are used to clean and repair the data received from the stream. The mechanism underlying Bleach relies on a violation graph, a dynamic structure where each node represents a data cell, and edges denote detected violations involving these cells [45]. The repairing process is managed by repair components, which receive all violation messages broadcasted by an ingress router. Each component uses an incremental equivalence class algorithm to clean the data [46]. In Bleach, subgraphs within the violation graph are treated as equivalence classes, where all cells are expected to hold the same value. The value that appears most frequently within a subgraph is considered the correct one, and this is used to repair the data. Each time a violation is detected, the repair components dynamically update the subgraphs by adding new cells and merging subgraphs when necessary. This dynamic adjustment is coordinated by a coordinator component, which handles the distribution of the violation graph across the repair components. This approach allows Bleach to efficiently handle large streams of data and continuously repair them as new violations are detected.

*Pipelines of Bleach and Bleach**. Bleach is composed of multiple independent components connected to each other through a pipeline architecture, which allows to identify and repair violations on the stream. Fig. 9 shows the pipeline integrating COD3 with Bleach,

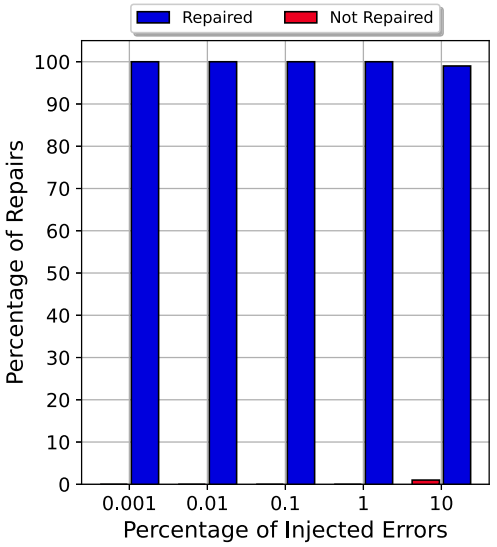


Fig. 10. Percentage of repairs performed by Bleach*.

which we refer to as *Bleach**. Two main types of Bleach components are: *violation detection* and *violation repair*. The first aims at finding input tuples that violate FDs by comparing the data history with the most recent stream data [46]. It is composed of an *ingress* router, an *egress* router, and *multiple detect* workers, which are responsible for detecting violations of specific FDs. The violation repair component receives the stream of messages generated by the detection components, and takes repairing decisions for dirty data tuples. The repairing step is performed by different repair workers that process the tuples, and emit clean data to an aggregator component. This process is managed by a coordinator component, which is responsible for updating the shared data structures to workers. The *Detect Ingress Repair Egress* component produces the output of the whole process, which consists of the processed tuples with associated repair actions.

Although Bleach is an efficient data cleaning tool for data streams, all the FDs involved in the repairing process must be defined in the configuration step, or manually injected in the pipeline. To completely automate the cleaning process, we integrated COD3 with Bleach by designing and implementing two new components: *FDs evaluator* and *Tuple Handler*. The first is responsible for evaluating the resulting FDs discovered by COD3, and determines the FDs suitable for Bleach. More specifically, since Bleach works with FDs with a single attribute on the LHS and RHS, the FDs evaluator determines the more relevant LHS attributes for each RHS of an FD. The relevance of an LHS attribute A with respect to an RHS attribute B is calculated by considering the amount of FDs having B as RHS and A in the LHS, normalized by the number of LHS attributes. The two most relevant FDs for each RHS attribute are sent to the FD handler component of Bleach. The *Tuple Handler* is responsible for the dynamic configuration of the Bleach pipeline, and for the standardization of the data read from the stream.

8.2. Evaluation process and results

Bleach* gave us the possibility to evaluate whether the continuous update of FDs can help the Bleach data cleaning system in repairing tuple values, and whether this can be performed in real-time.

Experimental settings. We considered an extended version of the *Gas-Sensors* dataset[6] shown in Table 6 to evaluate Bleach*. The dataset stores data collected by means of several types of sensors, such as gas, temperature, and humidity sensors, which have been exposed to different lab stimuli. Before processing the dataset, we performed a pre-processing step on the dataset, by removing all unique observations in the dataset that were not useful for the cleaning strategy of Bleach, i.e., those containing unique values and those that were duplicated. Thus, after a pre-processing step, the *Gas-Sensors* dataset considered in the evaluation consisted of 7 attributes and 100,000 tuples. Then, we artificially introduced several errors in the tuples, aiming to evaluate how many of them would be correctly recognized and repaired by Bleach. In particular, we considered 5 percentages of errors with respect to the total number of values of the dataset, i.e., $0.001\%, 0.01\%, 0.1\%, 1\%, 10\%$, leading to 7, 70, 700, 7,000, and 70,000 errors, respectively. The errors were introduced by replacing a value of an attribute with one randomly selected from its value distribution.

Results. Fig. 10 depicts the percentages of repairs achieved by Bleach* on the considered dataset with different percentages of injected errors. These results confirmed the usefulness of continuously updating FDs in data stream cleaning. In fact, the injected

[6] Gas Sensors for Home Activity Dataset.

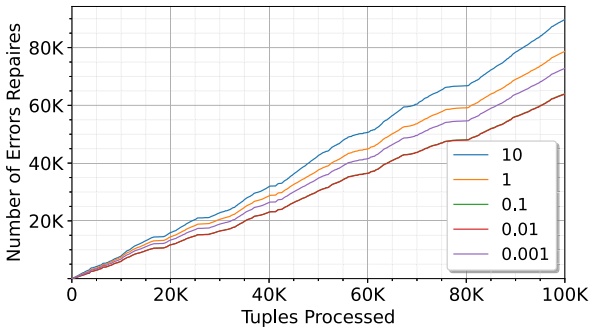


Fig. 11. Total number of repaired errors.

errors have always been repaired (i.e., reaching the 100% of repaired errors) for all considered datasets, except for the one including the highest number of errors, where the number of repaired errors was still very high, i.e., 99%.

Fig. 11 shows the total number of repaired errors across the stream. As we expected, it always increases with a (sub-)linear trend with respect to the size of the dataset. We can also note that the increasing trend is not dependent on the amount of injected errors. We do not provide a qualitative analysis of the repairs performed since they are strictly related to the Bleach cleaning strategy, which is out of the scope of our proposal, and the possibility that the considered dataset contained other errors.

Fig. 12 shows the average times for executing the whole pipeline triggered by the arrival of a single tuple. In particular, the execution times include the update of FDs (according to the COD3 strategy), the FD ranking step, and the repairing process (according to the Bleach strategy). We can notice that the average execution times vary in the range $[44, 54]$ milliseconds (ms) for each tuple, highlighting a higher variability for the processing of the first tuples. This is due to the fact that the set of discovered FDs change at the beginning of the stream and successively tends to become more stable.

The achieved results highlight that the introduction of a dynamic FD discovery process into a stream data cleaning system not only allows to continuously update metadata on which a possible cleaning process could be based, but also removes the time-consuming and error-prone step of manually configuring and updating metadata and their update, but it also adds a reduced overhead to execution times, even when processing streams with high or possibly unknown dimensionality.

9. Conclusion

In this paper, we presented COD3, an incremental algorithm for discovering FDs from data streams. It permits to continuously update the holding FDs by efficiently exploring the search space according to the previously discovered ones. We modeled the search process through a non-blocking strategy, relying on an Apache Storm topology, and exploiting a novel data structure (validation graph), which is capable of representing data in a compressed way, yielding a new validation methodology. The latter can also manage cases where candidate functional dependencies can be validated through simple checks. Experimental results demonstrate that COD3 can continuously discover FDs from both stream-simulated real-world datasets and real-world data streams, achieving good performances in terms of execution time compared to best-performing non-incremental and incremental FD discovery algorithms, like HyFD and DynFD, respectively. In particular, execution times of COD3 are mostly below 10 ms per tuple, except for large datasets where they almost always range from 1 to 10 seconds. Memory usage peaks slightly with the number of functional dependencies discovered, since in these scenarios more invalidations typically occur as new tuples arrive. COD3 is generally faster than the compared

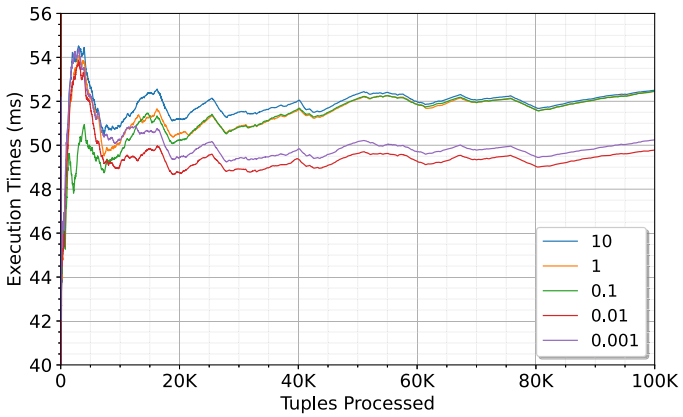


Fig. 12. Average processing time for completing the repairing process.

algorithms, especially as the number of tuples increases, while the poor performance obtained by COD3 on some datasets in the first runs is due to high validation and invalidation rates with few tuples.

The usefulness of COD3 has also been demonstrated on the real-world data stream for monitoring air quality information. In fact, through a continuous discovery of functional dependencies over the stream, it has been possible to highlight a strong relationship between humidity, pressure and rainfall. Finally, we successfully employed COD3 in a stream data cleaning scenario, by using the dynamically updated FDs to repair errors over data streams. More specifically, by automating the extraction of functional dependencies with COD3, Bleach's data cleaning tool remains effective with new data. It repaired 100% of injected errors in most datasets, and 99% even in the most error-prone case.

In the future, we would like to further improve COD3 by trying to devise an even more compressed representation of the data collected from the stream within the validation graph. Moreover, new COD3 versions exploiting parallel and distributed paradigms could be defined. Finally, another interesting issue concerns the possibility of incrementally discovering relaxed versions of functional dependencies, which would allow us to tackle the problem of approximations typically occurring in the machine learning context, and small errors in the data transmitted from sensors.

CRediT authorship contribution statement

Loredana Caruccio: Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Formal analysis, Conceptualization. **Stefano Cirillo:** Writing – review & editing, Writing – original draft, Visualization, Software, Formal analysis, Data curation, Conceptualization. **Vincenzo Deufemia:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Formal analysis, Conceptualization. **Giuseppe Polese:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link to my data.

References

[1] L. Golab, M.T. Özsu, Issues in data stream management, ACM SIGMOD Rec. 32 (2) (2003) 5–14.
[2] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, A. Bouchachia, A survey on concept drift adaptation, ACM Comput. Surv. (CSUR) 46 (4) (2014) 1–37.
[3] M.M. Gaber, A. Zaslavsky, S. Krishnaswamy, Mining data streams: a review, ACM SIGMOD Rec. 34 (2) (2005) 18–26.
[4] D. Che, M. Safran, Z. Peng, From big data to big data mining: challenges, issues, and opportunities, in: Proceedings of International Conference on Database Systems for Advanced Applications, DASFAA '13, Springer, 2013, pp. 1–15.
[5] X. Chu, I.F. Ilyas, P. Papotti, Holistic data cleaning: putting violations into context, in: Proceedings of IEEE 29th International Conference on Data Engineering, ICDE '13, 2013, pp. 458–469.
[6] T. Rekatsinas, X. Chu, I.F. Ilyas, C. Ré, HoloClean: holistic data repairs with probabilistic inference, Proc. VLDB Endow. 10 (11) (2017) 1190–1201.
[7] L.M. Ghiringhelli, J. Vybiral, S.V. Levchenko, C. Draxl, M. Scheffler, Big data of materials science: critical role of the descriptor, Phys. Rev. Lett. 114 (2015) 4105503.
[8] Y. Tian, P. Michiardi, M. Vukolić, Bleach: a distributed stream data cleaning system, in: 2017 IEEE International Congress on Big Data (BigData Congress), IEEE, 2017, pp. 113–120.
[9] Z. Abedjan, L. Golab, F. Naumann, Profiling relational data: a survey, VLDB J. 24 (4) (2015) 557–581.
[10] L. Caruccio, S. Cirillo, Incremental discovery of imprecise functional dependencies, J. Data Inf. Qual. (JDIQ) 12 (4) (2020) 1–25.
[11] F. Azzalini, C. Criscuolo, L. Tanca, E-fair-db: functional dependencies to discover data bias and enhance data equity, ACM J. Data Inf. Qual. 14 (4) (2022) 1–26.
[12] J. Kossmann, T. Papenbrock, F. Naumann, Data dependencies for query optimization: a survey, VLDB J. (2021) 1–22.
[13] l. Koumarelas, T. Papenbrock, F. Naumann, MDedup: duplicate detection with matching dependencies, Proc. VLDB Endow. 13 (5) (2020) 712–725.
[14] P. Faure-Giovagnoli, M. Le Guilly, J.-M. Petit, V.-M. Scuturici, Adesit: visualize the limits of your data in a machine learning process, in: International Conference on Very Large Data Bases, 2021, pp. 1–4.
[15] K.-W. Lam, V.C. Lee, Building decision trees using functional dependencies, in: Proceedings of International Conference on Information Technology: Coding and Computing, ITCC '04, vol. 2, IEEE, 2004, pp. 470–473.
[16] M. Abo Khamis, H.Q. Ngo, X. Nguyen, D. Olteanu, M. Schleich, In-database learning with sparse tensors, in: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '18, 2018, pp. 325–340.
[17] M.L. Guilly, J. Petit, V. Scuturici, Evaluating classification feasibility using functional dependencies, in: Transactions on Large-Scale Data- and Knowledge-Centered Systems XLIV, vol. 44, 2020, pp. 132–159.
[18] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, F. Naumann, Data profiling with metanome, Proc. VLDB Endow. 8 (12) (2015) 1860–1863.
[19] B. Breve, L. Caruccio, S. Cirillo, V. Deufemia, G. Polese, Dependency visualization in data stream profiling, Big Data Res. (2021) 100240.
[20] J. Gama, P.P. Rodrigues, An Overview on Mining Data Streams, Foundations of Computational Intelligence, vol. 6, Springer, 2009, pp. 29–45.
[21] J.A. Silva, E.R. Faria, R.C. Barros, E.R. Hruschka, A.C.d. Carvalho, J. Gama, Data stream clustering: a survey, ACM Comput. Surv. (CSUR) 46 (1) (2013) 1–31.
[22] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen, TANE: an efficient algorithm for discovering functional and approximate dependencies, Comput. J. 42 (2) (1999) 100–111.
[23] H. Yao, H.J. Hamilton, C.J. Butz, FD_Mine: discovering functional dependencies in a database using equivalences, in: Proceedings of IEEE International Conference on Data Mining, ICDM '02, 2002, pp. 729–732.
[24] Z. Abedjan, P. Schulze, F. Naumann, DFD: efficient functional dependency discovery, in: Proceedings of the 23rd ACM International Conference on Information and Knowledge Management, CIKM '14, 2014, pp. 949–958.

[25] S. Lopes, J.-M. Petit, L. Lakhal, Efficient discovery of functional dependencies and Armstrong relations, in: Proceedings of the 7th International Conference on Extending Database Technology, EDBT '00, 2000, pp. 350–364.
[26] C. Wyss, C. Giannella, E. Robertson, FastFDs: a heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances, in: Proceedings of International Conference on Data Warehousing and Knowledge Discovery, DaWaK '01, 2001, pp. 101–110.
[27] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, F. Naumann, Functional dependency discovery: an experimental evaluation of seven algorithms, Proc. VLDB Endow. 8 (10) (2015) 1082–1093.
[28] T. Papenbrock, F. Naumann, A hybrid approach to functional dependency discovery, in: Proceedings of the ACM International Conference on Management of Data, SIGMOD '16, ACM, 2016, pp. 821–833.
[29] Z. Wei, S. Link, Discovery and ranking of functional dependencies, in: Proceedings of IEEE 35th International Conference on Data Engineering, ICDE '19, IEEE, 2019, pp. 1526–1537.
[30] H. Saxena, L. Golab, I.F. Ilyas, Distributed discovery of functional dependencies, in: Proceedings of IEEE 35th International Conference on Data Engineering, ICDE '19, IEEE, 2019, pp. 1590–1593.
[31] H. Saxena, L. Golab, I.F. Ilyas, Distributed implementations of dependency discovery algorithms, PVLDB 12 (11) (2019) 1624–1636.
[32] P. Mandros, M. Boley, J. Vreeken, Discovering reliable dependencies from data: hardness and improved algorithms, in: Proceedings of IEEE International Conference on Data Mining, ICDM '18, IEEE, 2018, pp. 317–326.
[33] S. Song, F. Gao, R. Huang, C. Wang, Data dependencies over big data: a family tree, IEEE Trans. Knowl. Data Eng. (2020).
[34] L. Bornemann, T. Bleifuß, D.V. Kalashnikov, F. Nargesian, F. Naumann, D. Srivastava, Efficient discovery of temporal inclusion dependencies in Wikipedia tables, in: EDBT, 2024, pp. 399–411.
[35] C. Combi, M. Mantovani, A. Sabaini, P. Sala, F. Amaddeo, U. Moretti, G. Pozzi, Mining approximate temporal functional dependencies with pure temporal grouping in clinical databases, Comput. Biol. Med. 62 (2015) 306–324.
[36] L. Caruccio, V. Deufemia, G. Polese, Relaxed functional dependencies – a survey of approaches, IEEE Trans. Knowl. Data Eng. 28 (1) (2016) 147–165.
[37] L. Caruccio, V. Deufemia, G. Polese, Mining relaxed functional dependencies from data, Data Min. Knowl. Discov. 34 (2) (2020) 443–477.
[38] S.-L. Wang, J.-W. Shen, T.-P. Hong, Incremental discovery of functional dependencies using partitions, in: Proceedings of Joint 9th IFSA World Congress and 20th NAFIPS International Conference, vol. 3, IEEE, 2001, pp. 1322–1326.
[39] S. Bell, Discovery and maintenance of functional dependencies by independencies, in: Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining, KDD '95, 1995, pp. 27–32.
[40] P. Schirmer, T. Papenbrock, S. Kruse, D. Hempfing, T. Meyer, D. Neuschäfer-Rube, F. Naumann, DynFD: functional dependency discovery in dynamic datasets, in: Proceedings of 22nd International Conference on Extending Database Technology, EDBT '19, 2019, pp. 253–264.
[41] J. Liu, J. Li, C. Liu, Y. Chen, Discover dependencies from data—a review, IEEE Trans. Knowl. Data Eng. 24 (2) (2010) 251–264.
[42] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the 21st ACM Symposium on Principles of Database Systems, PODS '02, ACM, 2002, pp. 1–16.
[43] K. Patroumpas, T. Sellis, Window specification over data streams, in: Proceedings of International Conference on Extending Database Technology, EDBT '06, Springer, 2006, pp. 445–464.
[44] T.M. Ghanem, M.A. Hammad, M.F. Mokbel, W.G. Aref, A.K. Elmagarmid, Incremental evaluation of sliding-window queries over data streams, IEEE Trans. Knowl. Data Eng. 19 (1) (2006) 57–72.
[45] Z. Khayyat, I.F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, S. Yin, Bigdansing: a system for big data cleansing, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 1215–1230.
[46] P. Bohannon, W. Fan, M. Flaster, R. Rastogi, A cost-based model and effective heuristic for repairing constraints by value modification, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, 2005, pp. 143–154.