Efficient Discovery of Conditional Dependencies with Desbordante

Ivan Kozhukov, Dmitry Fedoseev, Maksim Emelyanov, Artem Smola,
Pyotr Senichenkov, Pavel Anosov, George Chernishev
Saint-Petersburg State University
Saint-Petersburg, Russia
{ivan.s.kozhukov, dmitrii.a.fedoseev, m.emelyanov.m, artem.m.smola, p.senichenkov, pavel.i.anosov, chernishev}@gmail.com

***Abstract*—Conditional functional dependencies (CFDs) are functional dependencies with a restricted scope: they specify the context in which a dependency holds and are useful for data-quality tasks, specifying complex integrity constraints, and extracting valuable insights from data.**

We study the CFD discovery problem, which is computationally demanding. We build on the state-of-the-art CFDFinder algorithm and introduce a set of algorithmic and engineering improvements, including a parallelization strategy, to produce ParCFDFinder. Our implementation is integrated into Desbordante — a high-performance open-source data profiler written in C++ that exposes a Python interface, enabling CFD discovery to be invoked from any Python program.

Experimental results show that our enhancements speed up the algorithm by up to $318\times$ ($118\times$ on average) and reduce memory usage by up to $23\times$ ($14\times$ on average) compared with the existing Java-based implementation of Metanome. Integrating ParCFDFinder into Desbordante makes it possible, for the first time, to conveniently discover CFDs on datasets with hundreds of thousands of rows on a commodity machine within a reasonable time.

I. Introduction

Conditional functional dependencies (CFDs) are a special type of functional dependency [1] that apply not to the entire table but only to a subset of it.

For example, consider Table I, which contains product sales data. The CFD $Product \longrightarrow Price$ for $Country \in \{US, CA\}$ holds in this table, while the functional dependency $Product \longrightarrow Price$ does not.

TABLE I. Sales
(example)

SaleID	Product	Category	Price	Country
1	Smartphone X	Electronics	699.0	US
2	T-Shirt Classic	Clothing	25.0	US
3	Smartphone X	Electronics	699.0	CA
4	Office Chair	Furniture	150.0	US
5	Smartphone X	Electronics	749.0	GB
6	Coffee Maker	Home	89.0	GB
7	T-Shirt Classic	Clothing	25.0	CA
8	Office Chair	Furniture	150.0	CA
9	T-Shirt Classic	Clothing	23.0	GB
10	Smartphone X	Electronics	699.0	GB

In this context, the problem of discovering CFDs from data arises. Unlike approximate FDs, CFDs provide an explicit context that specifies where a dependency holds, making their scope transparent and enabling discovery of nontrivial, context-specific insights about the data.

In practice, CFDs enable both knowledge extraction — supporting hypothesis generation for scientific discovery — and a range of data-quality interventions (e.g., typo detection, near-duplicate removal, and schema matching). CFDs also serve to specify complex integrity constraints in data. Finally, in machine-learning workflows CFDs aid feature engineering and can guide the design of ablation studies.

Desbordante (Spanish for *boundless*) [2] is an *open-source high-performance* data profiling tool designed for discovery and validation of complex patterns in data. It provides a range of algorithms, including specialized methods for CFD discovery. Tasks in this domain are computationally intensive, and CFD discovery is particularly demanding: it requires both substantial CPU time and large amounts of memory; therefore, algorithms are frequently memory-bound. Thus, there is a clear need for fast, memory-efficient discovery algorithms. Currently, there are roughly half a dozen published approaches; the most recent and fastest among them is CFDFinder [3], which is integrated into the Metanome data profiler [4].

However, Metanome is more of a research prototype than a production-ready tool. First, it is implemented in Java — a language that favors development speed over raw execution speed. As our group's work has shown [5]–[8], simply reimplementing pattern-discovery algorithms in C++ can yield $2-3\times$ speedups in some (though not all) cases. Second, Metanome lacks a mature user interface, which makes deployment and everyday use difficult. Finally, the Metanome project is no longer actively maintained.

We therefore decided to implement the algorithm in Desbordante and create its high-performance variant. To that end, we first developed a baseline version and then proposed a set of algorithmic and engineering optimizations, including a parallelization strategy that was absent in the original CFDFinder. The discussion of the design and evaluation of these enhancements forms the core of this paper.

Experimental results on several datasets show that our implementation is faster by a factor of $17-318\times$ (on average $118\times$) and uses $2-23\times$ (on average $14\times$) less memory compared to Metanome.

Overall, we have extended the practical applicability of CFD discovery: for the first time, users can process hundreds

of thousands of records on a commodity machine within a reasonable time.

The contributions of this paper are the following:

- A set of algorithmic and engineering optimizations for the original CFDFinder algorithm, including a parallelization approach. We call the resulting algorithm ParCFDFinder.
- An open-source implementation of the ParCFDFinder algorithm is included in the Desbordante profiler. Thus, the most efficient CFD discovery algorithm is now ready for practical use and can be employed in any Python program.
- An experimental evaluation of the proposed algorithm was conducted on multiple datasets, with comparisons against Metanome. We measured both row- and column-scalability and examined the effect of multithreading.
- A case study that illustrates the interpretation of CFD discovery results using real data.

This paper is organized as follows. In Section II, we provide the core definitions necessary for understanding the content of the paper. In Section III, we discuss the existing studies concerning CFD discovery. The original CFDFinder algorithm is described in Section IV, while ParCFDFinder is discussed in Section V. The evaluation is presented in Section VI. The case study is detailed in Section VII. We conclude the paper in Section VIII.

II. Background

Let r be a relation over the schema R, defined on a set of attributes $\mathcal{A}$.

Definition 1: A **Functional Dependency (FD)** $f : X \to A$, where $X \subseteq R$ and $A \in R$, **holds in** r if for all tuples $t_i, t_j \in r$:

$$t_i[X] = t_j[X] \Longrightarrow t_i[A] = t_j[A].$$

Here, X is called the **left-hand side (LHS)** and A the **right-hand side (RHS)**.

Definition 2: An FD $f : X \to A$ is a **generalization** of another FD $g : Y \to A$ if $X \subset Y$. Conversely, f is a **specialization** of g if $Y \subset X$.

Definition 3: An FD $f : X \to A$ is **minimal** if there exists no attribute $B \in X$ such that the FD $X \setminus B \to A$ holds in r. In other words, f has no generalization that is also valid in r.

Definition 4: A **Conditional Functional Dependency (CFD)** φ is a pair $(f : X \to Y, T_p)$, where $f : X \to Y$ is a functional dependency (called the **embedded FD** of φ) and T_p is a table over attributes $X \cup Y$ (called the **pattern tableau**). For every attribute $A \in X \cup Y$ and every tuple $t_p \in T_p$, the value $t_p[A]$ is either a constant $a \in \text{dom}(A)$ or the wildcard symbol "_" (which matches any value from $\text{dom}(A)$). A tuple $t_p \in T_p$ is called a **pattern tuple** (or simply a **pattern**).

Definition 5: A tuple $t \in r$ **matches** a pattern $t_p \in T_p$ on an attribute set $S \subset R$, denoted $t \asymp t_p$, if for every attribute $B \in S$: $t_p[B] =$ "_" or $t_p[B] = t[B]$.

Definition 6: A relation r **satisfies a CFD** $\varphi = (X \to Y, T_p)$ if for every $t_i, t_j \in r$ and every pattern $t_p \in T_p$:

$$t_i[X] = t_j[X] \asymp t_p[X] \Longrightarrow t_i[Y] = t_j[Y] \asymp t_p[Y].$$

Definition 7: For a CFD $\varphi = (X \to A, T_p)$ and a relation r, the **cover** of a pattern $p \in T_p$ is defined as:

$$\text{cover}(p) = \{t \in r \mid t[X] \asymp p[X]\}.$$

Definition 8: The **local support** of a pattern p is defined as:

$$\text{local\_support}(p) = \frac{|\text{cover}(p)|}{|r|}.$$

Definition 9: The **global support** of a pattern tableau T_p is defined as:

$$\text{global\_support}(T_p) = \frac{\left|\bigcup_{p \in T_p} \text{cover}(p)\right|}{|r|}.$$

We refer to the **support of a CFD** φ as the global support of its pattern tableau.

Definition 10: The set of **keepers** of a pattern p is the set of tuples covered by p that do not cause a violation of the embedded FD or any single-tuple violation.

Definition 11: The **local confidence** of a pattern p is defined as:

$$\text{local\_confidence}(p) = \frac{|\text{keepers}(p)|}{|\text{cover}(p)|}.$$

Definition 12: The **global confidence** of a pattern tableau T_p is defined as:

$$\text{global\_confidence}(T_p) = \frac{\left|\bigcup_{p \in T_p} \text{keepers}(p)\right|}{\left|\bigcup_{p \in T_p} \text{cover}(p)\right|}.$$

We refer to the **confidence of a CFD** φ as the global confidence of its pattern tableau.

III. Related Work

A. Discovering data quality rules

In their seminal work, Chiang and Miller [9] introduced several metrics — such as *Support*, χ^2, and *Conviction* — to identify interesting Conditional Functional Dependencies (CFDs). These metrics help filter out trivial dependencies, thus preventing the common issue where discovery algorithms generate an overwhelming number of CFDs, only a few of which are truly significant. While this aspect aligns with many studies in CFD discovery, a distinctive feature of their work is an algorithm that performs two key tasks: discovering CFDs and detecting violations of them. These violations can then be flagged for data analysts to review and correct.

The experimental evaluation revealed two key complexity characteristics of the algorithm: it scales exponentially with the number of attributes and approximately linearly with the domain size of the attributes. Furthermore, the study evaluated and compared several metrics for quantifying the "interestingness" of discovered CFDs. The results demonstrated that *Conviction* outperformed the others, proving to be the superior measure for highlighting the most useful dependencies. The metrics of *Confidence*, *Support*, and *Interest* followed in decreasing order of effectiveness, though they still provided valuable results.

Importantly, the authors investigated the use of *Support* and *Conviction* metrics for identifying data exceptions (i.e., CFD violations). Detecting exceptions via *Support* highlighted infrequently occurring values, which are not necessarily errors. In contrast, using *Conviction* targeted data that violate an approximate CFD due to statistical independence. Their experimental results demonstrated an advantage of *Conviction*: it yielded a much richer set of interesting violations for analyst review.

B. Discovering Conditional Functional Dependencies

Fan et al. [10] present three CFD mining algorithms that differ in their features and applicability: CFDMiner, CTANE, and FastCFD.

1) CFDMiner: CFDMiner is an efficient algorithm for discovering constant CFDs, namely minimal, k-frequent, and left-reduced constant CFDs. A key feature of the algorithm is that the authors successfully reduce the problem of finding minimal constant CFDs to the established problem of discovering all k-frequent closed itemsets [11]. Afterwards, the authors matched them with their corresponding free itemsets.

2) CTANE: CTANE is an extension of the TANE algorithm for levelwise discovery of FDs. The output of CTANE is the set of all minimal, k-frequent CFDs of a relational instance, i.e., its complete canonical cover. The principle of CTANE's operation is based on a systematic, levelwise enumeration of attribute combinations X and patterns t_p (patterns may contain wildcards). The algorithm starts with single attributes and at each subsequent level constructs more complex candidates. Its efficiency is achieved through a specialized mechanism of auxiliary structures C^+, which precompute possible right-hand sides for each candidate and prune provably unpromising search branches.

3) FastCFD: The FastCFD algorithm, like CTANE, discovers all minimal k-frequent CFDs; however, it employs a depth-first search strategy instead of a levelwise traversal. The set of optimizations used in the algorithm, such as decomposing the problem into subproblems for each RHS and utilizing minimal covers of difference sets, is primarily aimed at handling data with high arity (more than 15 attributes). Furthermore, FastCFD leverages the results produced by CFDMiner in its operation.

4) Applicability Scope: The experimental results provided by the authors suggest potential application scenarios for these algorithms. CFDMiner can be employed for discovering exclusively constant CFDs. CTANE is applicable to datasets with low arity and high support thresholds, while FastCFD is suitable for scenarios featuring high arity and moderate data volumes.

C. Discovering (frequent) Constant Conditional Functional Dependencies

Diallo, Novelli, and Petit [12] proposed the CFUN algorithm, which, similar to CFDMiner, focuses exclusively on discovering frequent *constant* CFDs. However, unlike the approaches based on itemsets (like CFDMiner) or hypergraphs,

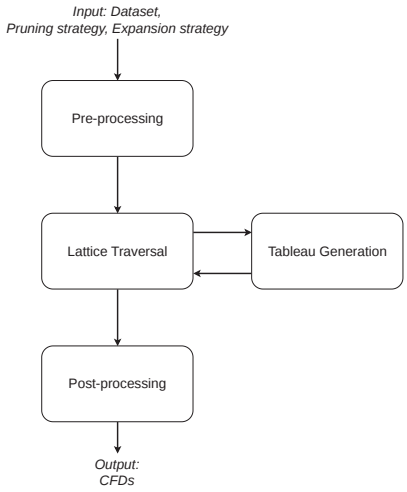

Fig. 1. Schema of CFDFinder algorithm

CFUN is an extension of the FUN algorithm (originally designed for standard functional dependencies) that has been adapted for the conditional setting. The algorithm introduces the concepts of *conditional agree sets*, *conditional closure*, and *quasi-closure*. Operating in a levelwise manner (akin to Apriori), CFUN identifies the canonical cover of satisfied CFDs by computing the difference between the closure and quasi-closure of candidate attribute sets.

Regarding applicability, the authors demonstrate that CFUN scales linearly with the number of tuples, making it feasible for datasets with a large number of rows. Its memory usage is also shown to be efficient. Unlike generalized algorithms (like CTANE), CFUN is optimized specifically for constant patterns, offering a direct method to derive a non-redundant cover without the intermediate step of mining closed itemsets.

D. Splitting the "C" from the "FD"

Rammelaere and Geerts [13] revisited the problem of approximate CFD discovery, presenting it as a combination of functional dependency (FD) discovery and pattern mining. They described three approaches that differ in how they traverse the search space:

1) Integrated Approach: This strategy traverses a combined lattice of constant and variable patterns simultaneously. This is the approach employed by the classic CTANE algorithm.

2) Itemset-First: This new approach decouples the process by first performing frequent itemset mining to discover constant patterns (the "C"). Then, for each pattern found, a data projection is formed, on which the functional dependency discovery algorithm is run.

3) FD-First: This strategy summarizes the idea behind the work [9] by initially searching for variable FDs on the entire dataset. For FDs that do not hold globally but have sufficient confidence, it subsequently runs a specialized pattern mining step to identify the specific constant conditions under which these dependencies are valid.

IV. CFDFinder

This section provides a step-by-step analysis of the CFDFinder algorithm, aimed at a deeper understanding of the approach under consideration. All stages of the algorithm are schematically shown in Figure 1. In addition, some implementation details that are key to the optimization methods we propose are discussed here.

A. Pre-processing

During the pre-processing stage, CFDFinder constructs auxiliary structures and identifies the complete set of maximal non-FDs for the relation. This set subsequently serves as the foundation for generating CFD candidates. At this stage, CFDFinder employs a modified version of the HyFD FD discovery algorithm [14].

The initial stage of HyFD involves creating **Position List Indexes (PLIs)** and **compressed records**. A PLI is constructed for a specific attribute X as a collection of index lists, where each list groups the row indices (row IDs) that share the same value for attribute X. Effectively, a PLI partitions the relation into a set of equivalence classes, referred to as **clusters**. The compressed representation of records is then generated by replacing each attribute value with the index of the cluster to which the corresponding record belongs.

Subsequently, the operation of the HyFD algorithm consists of three repeating stages: **sampling**, **induction**, and **validation**. During the **sampling stage**, pairs of records are selected that match only on some attributes and have different values for the remaining attributes. Based on these pairs, a set of non-FDs of the form $X \to A$ is constructed, where X is the set of matching attributes and A is the mismatching attribute. The task of iterating through all possible pairs of tuples has quadratic complexity, which results in poor scalability. To avoid performance issues, only a subset of record pairs is selected via specialized sampling techniques during the sampling stage. At the **induction stage**, FD candidates are formed based on the set of non-FDs obtained during the sampling stage. These candidates are then verified during the **validation stage**. During both the induction and validation stages, non-FDs are identified, including those dependencies that do not become candidates during induction and those that become candidates but were determined to be non-FDs during validation. In the original HyFD, these non-FDs are discarded, but in CFDFinder, they are added to a set of non-FDs from which any non-FD with a specialization in the tree is removed, meaning only maximal non-FDs remain.

The final output of HyFD is a set of minimal FDs. CFDFinder augments the accumulated maximal non-FDs by adding dependencies derived from these minimal FDs — those obtained by removing a single attribute from the left-hand side. Only those generalizations that are themselves maximal non-FDs (i.e., those for which no specialization exists in the current set) are included. The obtained set of maximal non-FDs is the set of all maximal non-FDs of the relation.

B. Lattice traversal

During the lattice traversal stage, an attempt is made to generate a pattern tableau for CFD candidates, and new candidates are created based on existing ones. The pseudocode for this part of the algorithm is presented in Algorithm 1.

Algorithm 1 Top-down lattice traversal

```
1: Data: N — set of maximal non-FDs, R — relation
2: for 1 ≤ i ≤ |R| do
3:     L_i ← ∅
4: end for
5: for n = (X_n, A_n) ∈ N do
6:     l = |X_n|
7:     L_l ← L_l ∪ {n}
8: end for
9: p ← |R|
10: while p > 0 do
11:     for c = (X_c, A_c) ∈ L_p do
12:         T ← buildTableau(c)
13:         if T ≠ ∅ then
14:             S ← {(X', A_c) | X' ⊂ X_c ∧ |X_c \ X'| = 1}
15:             L_{p−1} ← L_{p−1} ∪ S
16:         end if
17:     end for
18:     p ← p − 1
19: end while
```

First, all remaining levels of the lattice are prepared (lines 2–4 of the pseudocode). Then, initial candidates from the set of maximal non-FDs obtained in the previous step are added to their corresponding levels; the level number corresponds to the number of attributes in the candidate's LHS (lines 5–8).

Next, the lattice is traversed top-down, from the level of candidates with the maximum number of attributes in the relation down to the level of candidates with a single attribute (lines 9–19). For each candidate, an attempt is made to create a pattern tableau, the validity of which is determined according to the pruning strategy. If a valid tableau is obtained for a candidate, that CFD candidate is considered a valid CFD, and new candidates — generated by removing one attribute from the LHS of the resulting CFD — are added to the lattice at the level below.

C. Pattern Tableau generation

In the pattern tableau generation stage, the algorithm attempts to construct a pattern tableau for a CFD candidate. It iteratively adds new patterns to the tableau as long as the chosen strategy permits. The stage concludes when no further patterns can be added. The resulting tableau is then evaluated against predefined mining parameters (e.g., confidence and support thresholds). If it satisfies these constraints, the corresponding CFD is recorded in the final set of mining results. The pattern tableau generation stage is presented in Algorithm 2.

Before delving into the details of the algorithm, we introduce a few additional definitions.

Definition 13: A pattern p_p that contains exactly one more specified constant than p_c is called a **parent pattern** of p_c. Similarly, a pattern p_c that contains exactly one fewer specified constant than p_p is called a **child pattern** of p_p. A single

Algorithm 2 The tableau generation algorithm of CFDFinder

```
1: Data: d — CFD candidate, r — relation instance, ps — pruning strategy, es — expansion strategy
2: T ← ∅; F ← ∅; p0 ← es.generateNullPattern(d)
3: for all t ∈ r do
4:     cover(p0) ← cover(p0) ∪ {t}
5:     kp(p0) ← kp(p0) + kp(t)
6:     ct(p0) ← ct(p0) + ct(t)
7: end for
8: margSupport(p0) ← ct(p0)
9: conf(p0) ← kp(p0)/ct(p0)
10: F ← F ∪ {p0}
11: while F ≠ ∅ and ps.continue(T) do
12:     p ← argmax_{x∈F} margSupp(x)
13:     F ← F \ {p}
14:     if ps.addPattern(p) then
15:         T ← T ∪ {p}
16:         for all p′ ∈ F do
17:             cover(p′) ← cover(p′) \ cover(p)
18:             margSupport(p′) ← Σ_{t∈cover(p′)} ct(t)
19:             if ¬ps.considerPattern(p′) then
20:                 F ← F \ {p′}
21:             end if
22:         end for
23:     else
24:         for all c ∈ es.expand(p) do
25:             if ps.valid(c) then
26:                 cover(c) ← determineCover(cover(p), c)
27:                 margSupport(c) ← |cover(c)|
28:                 if ps.considerPattern(c) then
29:                     F ← F ∪ {c}
30:                 end if
31:             end if
32:         end for
33:     end if
34: end while
35: if ps.continueGeneration(T) then
36:     return T
37: end if
38: return ∅
```

pattern may have multiple child patterns and multiple parent patterns.

Definition 14: A pattern that consists exclusively of the wildcard symbol "_" for all attributes is called the **null pattern**.

The tableau generation algorithm is heuristic. It relies on the premise that if the selected patterns individually exhibit a local confidence higher than a predefined global confidence threshold, then the entire tableau will also meet or exceed that threshold.

Initially, the algorithm creates a **null pattern** p_0 and adds all unique tuples to its cover. This is possible because the null pattern matches every tuple. In the listing, $kp(p)$ denotes the number of keepers of pattern p, and $ct(p)$ represents the size of its cover (i.e., $|\text{cover}(p)|$).

The cover, support, and confidence of the null pattern are then computed. Afterwards, p_0 is added to the **frontier** F. The frontier is a set of pattern candidates sorted by **marginal support** — the additional support a pattern would contribute to the current tableau. This ordering enables a greedy strategy that always selects the candidate with the highest marginal support.

Next, the algorithm checks whether the current tableau T can be improved according to the chosen strategy via the $continue(T)$ function. For instance, this function may verify whether the tableau already meets a fixed support threshold. If the check passes, the algorithm proceeds to process the pattern with the highest marginal support from the frontier.

If a pattern p can be added to the tableau, its cover is subtracted from the cover of every other candidate p' in the frontier. This update ensures that the **marginal support** of each candidate correctly reflects only the tuples not yet covered by T, thereby minimizing overlap and guaranteeing that the global support increases by exactly the marginal support of p. Additionally, the function $considerPattern(p')$ determines whether it is still worthwhile to keep p' in the frontier. For example, candidates with zero marginal support are removed to improve performance.

If a pattern p is **not** selected for inclusion, it is **expanded**: all its child patterns are generated. The on-demand expansion [15] uses the $valid(c)$ function. According to the selected pruning strategy, this function verifies either the completion of processing for all parent nodes or the absence of the current pattern from the set of previously visited ones. If c satisfies the mining parameters, it is added to the frontier. The cover of a child pattern can be computed efficiently from the cover of its parent with Algorithm 3, reducing the computational overhead of repeatedly scanning the entire relation.

Algorithm 3 Computing the cover of a child pattern

```
1: Data: p_c — child pattern, cover(p) — parent cover
2: cover_c ← ∅
3: for each cluster ∈ cover(p) do
4:     matches ← true
5:     t ← cluster[0]
6:     for A ∈ attr(p_c) do
7:         if ¬p_c[A].Matches(t[A]) then
8:             matches ← false
9:         end if
10:    end for
11:    if matches = true then
12:        cover_c ← cover_c ∪ cluster
13:    end if
14: end for
15: return cover_c
```

The algorithm terminates when the frontier is empty or when a stopping condition defined by the strategy is met. It then either returns a valid pattern tableau for the CFD candidate or — if the strategy's pruning criteria cannot be satisfied — returns an empty set.

D. Pruning Strategies

CFDFinder implements several pruning strategies.

1) Minimum Confidence and Support Thresholds: According to this pruning strategy, tableau generation continues if the frontier contains at least one pattern and if the minimum support of the pattern tableau has not reached a certain

specified value. Only patterns with local confidence no less than the specified value are added to the tableau, while patterns with zero support are discarded. A CFD candidate is added to the resulting CFD set if the generated pattern tableau satisfies the specified thresholds.

2) Minimum Support Gain and Maximum Support Drop Thresholds (Support Independent Strategy):

Definition 15: **Minimum support gain** is the lower bound of the local support of patterns in the tableau.

Definition 16: **Maximum support drop** is the upper bound of the difference between the support of the current CFD candidate and the support of its generalizations.

Definition 17: **Maximum pattern tableau length** is the upper limit of the number of patterns in the pattern tableau.

The algorithm terminates its operation if any of the following conditions is met: 1) there are no patterns remaining in the frontier, or 2) the number of patterns in the tableau equals the maximum tableau length, or 3) there are no patterns in the frontier that meet the minimum support gain threshold.

Using this strategy, we consider only those patterns whose support gain is greater than the specified threshold. A CFD candidate is added to the resulting set if the generated pattern tableau satisfies the specified confidence threshold and the support drop does not exceed the specified upper bound.

3) Partial FD:

Definition 18: The **g1** metric is the fraction of record pairs violating a given FD.

Definition 19: Let $s \in [0, 1]$; then a **partial FD** $f : X \to A$, where $X \subseteq R$ and $A \in R$, is said to **hold in** r if $g1 \leq s$.

When this strategy is chosen, no patterns are generated except for null patterns. The tableau creation function *buildTableau* is simplified to generating only the null pattern and returning a tableau containing only the null pattern if the $g1$ metric value for this pattern does not exceed a specified threshold value; otherwise, an empty tableau is returned.

It is easy to see that the result of the algorithm's operation with the $g1$ strategy is a set of partial FDs.

E. Expansion Strategies

During the expansion stage, the algorithm processes patterns that have sufficient local support but lack the required local confidence to be included directly in the resulting pattern tableau. These patterns cannot be pruned entirely, as they may lead to viable specializations. The idea is to specialize them into new pattern candidates and add these candidates to the frontier. A key property of specialization is that the cover of any child pattern is always a subset of the cover of its parent, regardless of the specific expansion strategy employed.

An **expansion strategy** defines a method for enumerating child patterns. The CFDFinder algorithm employs three expansion strategies proposed in [15].

1) Expansion by Adding Constants: This strategy is suitable for standard CFDs, where attributes may take either constant values or the wildcard symbol "_". A pattern is specialized by replacing each wildcard symbol with constants drawn from the attribute's domain. This approach generates all possible combinations of constants across the wildcard positions, thereby producing a complete set of child patterns. The process is illustrated in Algorithm 4.

Algorithm 4 Expansion by Adding Constants

```
children ← ∅
for t ∈ cover(p) do
    for A ∈ attr(p) do
        if p[A] ="_" then
            c ← p
            c[A] ← t[A]
            children ← children ∪ {c}
        end if
    end for
end for
return children
```

2) Expansion for Negative Conditions: To support negative conditions, the assignment operation $c[A] \leftarrow t[A]$ must be extended. For each constant $a \in \text{dom}(A)$, we now generate two types of conditions: equality $A = a$ and inequality $A \neq a$. This modification doubles the search space and significantly affects the pruning process.

Specializing p by constraining an attribute A to a value a yields two child patterns: $c_{[=a]}$ (with $A = a$) and $c_{[\neq a]}$ (with $A \neq a$). Their covers partition $cover(p)$: the tuples covered by $c_{[\neq a]}$ are exactly those in $cover(p)$ not covered by $c_{[=a]}$.

3) Expansion for Range Conditions: This strategy employs a range-based representation of attribute bindings. We assume a total order on each attribute domain and assign indices to values accordingly. For a CFD $\varphi : (X \to Y, T_p)$, each pattern tuple $t_p \in T_p$ assigns a range $t_p[A] = [l, r]$ to every attribute $A \in X \cup Y$, where l and r are indices of values in $\text{dom}(A)$ and $l \leq r$. A data tuple t satisfies this condition iff the value $t[A]$ corresponds to an index i such that $l \leq i \leq r$. Range bindings generalize both wildcard symbols (when l is the first index and r is the last) and constant bindings (when $l = r$ corresponds to a single value).

The null pattern p_0 is initialized with ranges covering the entire domain of each attribute (i.e., from the first index to the last). During expansion, for each attribute A with $p[A] = [l, r]$ and $l < r$, two child patterns are generated: $[l + 1, r]$ and $[l, r - 1]$. This effectively enumerates all contiguous subranges by removing one boundary index at a time.

V. Proposed Approach

The Java version of the CFDFinder algorithm, developed within the Metanome data profiling framework[1], was taken as the baseline version. This implementation was originally described in [3].

We implemented two improved variants of the algorithm in C++ within the high-performance data profiler Desbordante. The first version is a re-implementation of the Java version of the algorithm with minimal changes necessitated by differences in the data structure implementations in Java and C++. The second version contains a series of our improvements and

[1] http://www.metanome.de

code optimizations that provide a significant increase in the performance and scalability of the algorithm.

A. C++ Re-implementation

In the Java version, the frontier is implemented using *PriorityQueue*. When porting to C++, the equivalent *std::priority_queue* from the STL was considered. However, it lacks a method to search for elements within the queue, which is required for the algorithm. Therefore, it was replaced with *boost::multi_index*, which allows for the definition of two indices simultaneously: one for priority ordering (similar to a priority queue) and one for hash-based element lookup.

Overall, the C++ port did not yield significant performance improvement on most datasets. However, for datasets where the Java version spent a substantial portion of its execution time on frontier lookups, the replacement with *boost::multi_index* resulted in a noticeable speedup. This observation motivated us to pursue further algorithmic and technical optimizations.

B. CFDFinder Optimization

1) Improved Structures: After generating a child pattern during the expansion process, the algorithm, according to the selected pruning strategy, checks whether all parent patterns have been fully considered or whether the current pattern has already been processed. The **visited** structure, representing a set of visited patterns, is used for this check. In the Java version, this is implemented using a *HashSet* (a hash table using the chaining method). In the improved version, we used *boost::unordered_flat_set* instead of *std::unordered_set* from the C++ Standard Template Library. This container provides higher performance due to its use of open addressing and the resulting good data locality.

2) Pattern Generation Optimizations: The pattern generation stage is the most computationally intensive part of the algorithm, accounting for approximately 99% of the total execution time across all tested datasets. Our optimizations target this stage by leveraging a key observation: a child pattern differs from its parent in only one attribute. This allows us to restructure the workflow: instead of immediately constructing full child patterns, we first work with lightweight *candidates* — potential patterns represented only by the attribute binding being specialized. All validation and filtering steps are performed using only this binding and the parent's metadata. Only after a candidate passes all checks do we materialize the full pattern and its cover.

We organize the optimizations into four complementary techniques, each addressing a specific inefficiency in the original implementation.

a) Eliminating Duplicate Pattern Generation: The Constant and NegativeConstant expansion strategies generate child patterns by replacing a wildcard with values taken from representative tuples of each cluster in the parent's cover. Different clusters often share the same value for the expanded attribute, leading to duplicate child patterns. In the original algorithm, each duplicate undergoes full cover construction and validation before being discarded during the duplicate check.

Our solution preprocesses the cluster representatives to extract only *unique values* for the attribute being expanded. This simple filtering eliminates redundant computations and reduces memory allocations by ensuring that each distinct value generates at most one candidate pattern.

b) Deferred Cover Construction with Bitmask Pruning: Originally, to determine whether a child pattern meets the support threshold, the algorithm first constructed its full cover (by copying clusters from the parent cover), computed support, and then discarded the cover if validation failed. This resulted in excessive memory operations.

We introduce a lightweight **cover mask** — a bitset with a length equal to the number of clusters in the parent cover. For each candidate child pattern (identified by its attribute binding), we compute the mask in a single pass over the parent clusters, setting a bit for each matching cluster. From this mask, we can calculate the pattern's support without materializing the cover. Only if the support satisfies the threshold do we proceed to construct the actual cover by selecting the clusters indicated by the mask. This approach eliminates unnecessary copying and reduces memory pressure.

c) Batch Processing of Multiple Candidates: The original implementation processed each child pattern sequentially: for each generated candidate, it performed validation, support calculation, and (if successful) cover construction. This pattern-by-pattern approach exhibits poor data locality: each iteration operates on different attribute bindings, causing frequent cache misses and preventing the compiler from applying vectorized optimizations.

We restructure the workflow into a batched pipeline that processes all candidates for a given attribute together. For the attribute being expanded, we precompute the values from each parent cluster representative and store them in a contiguous array. This data layout enables efficient cache utilization and allows the subsequent operations to be performed uniformly across all candidates:

1) Generate all candidate child patterns from the parent (using unique values to avoid duplicates).
2) Apply validation filters to the entire batch, removing invalid candidates early.
3) Compute cover masks and support for all remaining candidates in a single pass over the precomputed attribute values.
4) Filter candidates that fail support thresholds.
5) Finally, construct full covers only for the surviving patterns.

This batch-oriented approach progressively reduces the number of candidates before performing expensive operations, enables uniform processing across candidates, and significantly improves cache locality by operating on contiguously stored data.

d) Reusing Intermediate Results for Negative Constants: For the *NegativeConstantStrategy*, after computing the cover mask for a valid constant, the mask for its negation is obtained

simply as the bitwise complement. Similarly, its support is computed as the difference between the total number of rows and the support of the constant being negated. Our batched pipeline naturally reuses these intermediate results, thereby avoiding the redundant independent computations performed in the original implementation.

3) Parallelization of lattice traversal: The pattern tableau generation stage for each CFD candidate is computationally expensive, as it involves exploring a large space of candidate patterns. Fortunately, these generation tasks are independent: the attempt to construct a tableau for one candidate does not affect the others. This means that when traversing candidates of the same level, we can process them in parallel, as noted in [3].

To leverage this property, we parallelized the lattice traversal using a thread pool implemented using the Boost.Asio library. We divide the list of candidates into several batches and process them concurrently using the thread pool. The pool distributes the batches across multiple CPU cores, executing multiple tableau generation tasks in parallel. Boost.Asio provides an efficient work-stealing mechanism that balances the load dynamically, ensuring that cores remain busy even when some tasks complete earlier than others. This approach significantly reduces the overall runtime of the mining process and scales well with the number of available cores.

VI. Evaluation

A. Methodology

To verify the correctness of our C++ implementation, we compare its mined CFDs with those produced by the original CFDFinder algorithm integrated into the Metanome framework.

During testing, we observed that the original algorithm does not fully specify the order in which candidate patterns are traversed. Specifically, when multiple patterns have identical support and confidence values, the order of selection becomes ambiguous. Depending on the chosen order, a different pattern may be added to the tableau, which in turn affects the recalculation of candidate parameters, leading to divergent subsequent choices. As a result, the final pattern tableau may differ, making it impossible to determine whether discrepancies stem from implementation errors or merely from non-deterministic traversal. Furthermore, this non-determinism also affects execution time, as different traversal orders lead to processing different sets of candidates and performing different numbers of support recomputations, rendering runtime comparisons equally unreliable.

To address this issue, we introduced additional ordering criteria that enforce a deterministic and unambiguous traversal of candidates. The primary ordering remains based on support and confidence; however, when ties occur, we proceed to compare the attribute values of the patterns. Each pattern is represented as a list of elements, each encoding a condition determined by the chosen expansion strategy. For each type of condition, we define a fixed local order; the exact ordering is not critical as long as it is applied consistently and identically across both implementations. With this deterministic ordering in place, the two implementations can be compared correctly and reproducibly.

Reference outputs are obtained by running the modified version of the Metanome implementation under the same strategies and parameter settings.

B. Experimental Setup and Configuration

a) Hardware and software: Experimental studies were conducted on a testbed with the following hardware and software configurations. Hardware: Intel Xeon Gold 6338 (Icelake), 4 physical cores (8 threads, MT), 48 GiB RAM, 16 MiB L2 Cache. Software: Ubuntu 24.04.4 LTS (Noble Numbat), gcc (Ubuntu 13.3.0-6ubuntu2 24.04.1) 13.3.0, CMake 3.28.3, Boost 1.89.0, openjdk 1.8.0_482, OpenJDK Runtime Environment (build 1.8.0_482-8u482-ga us1-0ubuntu1 24.04-b08), OpenJDK 64-Bit Server VM (build 25.482-b08, mixed mode).

b) Datasets: To conduct the testing, a collection of datasets was assembled, including both real-world and synthetic data. Several of these datasets were used in the paper [3]. These include the following datasets: Wisconsin Breast Cancer, Bridges, Echocardiogram, ncvoter, abalone and Wine Reviews. Additional datasets were added to ensure a more comprehensive study. These include: Students[2], BMW Global Sales[3], Biocase multimedia object and Biocase gathering namedareas[4]. The characteristics of the datasets are provided in Table II.

TABLE II. Used datasets

Dataset	Num_rows	Num_cols	Size (KB)
Bridges	108	13	6
Echocardiogram	132	13	6
Wisconsin Breast Cancer	699	11	20
BMW Global Sales	1000	14	69.67
ncvoter	1000	19	151
abalone	4177	9	191.9
Students	5000	21	516.81
Biocase multimedia object	18,784	15	9,100
Wine Reviews	150,935	11	17,540
Biocase gathering namedareas	137,710	11	21,200

c) Configuration: To replicate the experiments from the original CFDFinder, *SupportIndependentStrategy* and *ConstantExpansionStrategy* were chosen as the main study strategies. The following values were selected as fixed pruning parameters: a minimum support gain of 5% of the number of dataset rows, a maximum support drop of 10%, a minimum confidence of 1.0, and a maximum pattern tableau length of 2000 patterns.

[2]URL:https://www.kaggle.com/datasets/amar5693/student-performance-dataset

[3]URL:https://www.kaggle.com/datasets/payaldhokane/bmw-global-sales-and-market-data?resource=download

[4]URL:https://my.hidrive.com/share/rt.v.6myak#$/

C. Research questions

We evaluated the performance and scalability of our proposed solution through a series of controlled experiments. This evaluation aimed to substantiate the advantages of implementing the algorithm in C++ with the proposed optimizations. The study addresses three primary research questions:

- **RQ1**: What performance gains in terms of execution time and memory consumption are achieved by the optimized C++ version of CFDFinder compared to its Java implementation in Metanome?
- **RQ2**: To what degree does the multi-threaded ParCFDFinder surpass both the baseline implementation in Metanome and the single-threaded version of ParCFDFinder, and what overall speedup is achieved across a variety of datasets?
- **RQ3**: How effectively does ParCFDFinder scale as the number of execution threads increases from 1 to the limit of available physical cores, and how close is the achieved speedup to the ideal linear scaling?

D. Experiments

Experiment 1. In this initial experiment, we reproduce the evaluation scenario performed by the CFDFinder authors on the presented datasets in order to compare our proposed implementation options — the multi-threaded ParCFDFinder and the single-threaded ParCFDFinder — with the original Java baseline implementation.

Experiment 2. In the second experiment, we evaluate the scalability of three algorithm versions (the Metanome, single-threaded, and multi-threaded implementations of ParCFDFinder) with respect to two key dimensions: the number of rows and the number of columns.

Experiment 3. In the last experiment, we examine the scalability of ParCFDFinder under varying degrees of parallelism. By adjusting the number of threads in the pool, we measured the resulting impact on both execution time and memory consumption.

E. Discussion

Experiment 1.

In this experiment, we evaluate the algorithm's performance on the datasets across several metrics: execution time, memory usage, and the number of mined CFDs. For the C++ implementations, we also report speedup and memory factor relative to the Java baseline. Execution time was limited to three hours; a dagger (†) next to a dataset indicates that the Java version timed out on that test. The complete results are presented in Table III.

On approximately half of the datasets, Java times out, while both C++ versions successfully complete all of them. When Java finishes within the time limit, its execution time and memory consumption are an order of magnitude higher than those of the C++ implementations. The multi-threaded version is on average 3–4× faster than the single-threaded one, though the speedup varies by dataset.

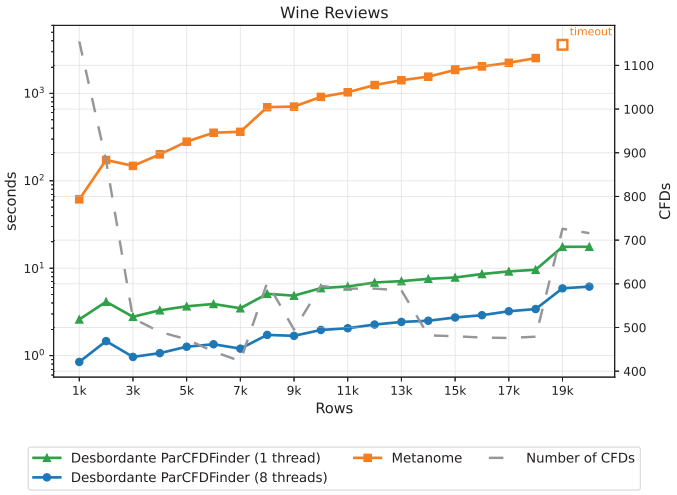

Fig. 2. Scalability with respect to the number of rows: comparison of Java and C++ implementations on datasets up to 20,000 rows.

Memory usage of the multi-threaded version is nearly identical to that of the single-threaded one — an observation discussed in detail in Experiment 3. The memory improvement over Java stems from our optimizations and the absence of JVM overhead. For datasets where Java times out, memory factor is omitted.

Regarding the number of mined CFDs: when Java does not time out, our implementation mines the same number of CFDs (due to our methodology) but more rapidly. When Java times out, it mines an order of magnitude fewer CFDs than our C++ implementations.

It should be noted that in some cases where Java reached the time limit, our implementation utilized more memory. This is explained by the fact that Java managed to process only a negligible number of levels before reaching the time limit, which is confirmed by the small number of discovered CFDs.

Experiment 2.

To evaluate row scalability, we use the *Wine Reviews* dataset containing 150,935 rows. For column scalability, we use the *ncvoter* dataset due to its large number of columns. These datasets were previously employed for scalability analysis in [3].

The results for row scalability are presented in two separate figures because the Java implementation fails to complete within the one-hour time limit once the number of rows exceeds approximately 18,000. In contrast, our C++ implementations efficiently process the full dataset within the time limit. Therefore, Figure 2 shows all three versions on smaller instances (where Java completes), while Figure 3 focuses on the C++ versions across the entire row range. For column scalability, both Java and C++ versions are presented in a single figure, as shown in Figure 4.

Figure 2 presents the results on datasets truncated to 20,000 rows. The C++ implementations are several orders of magnitude faster than the Java version, and all three exhibit approximately linear growth in execution time as the number

TABLE III. OVERALL RESULTS

Dataset	Algorithm	Time (s)	Mem. (MB)	#CFDs	Speedup	Mem. factor
Avocado Prices†	Metanome	TL	3215.3	30	–	–
	ParCFDFinder (1 thread)	727.35	**5379.5**	5285	$> 4.9\times$	–
	ParCFDFinder (8 threads)	**157.67**	5834.4	5285	$>$ **22.8**$\times$	–
BMW Global Sales	Metanome	463.14	2806.8	6468	–	–
	ParCFDFinder (1 thread)	21.96	**241.3**	6468	$21.1\times$	**11.6**$\times$
	ParCFDFinder (8 threads)	**6.59**	270.9	6468	**70.3**$\times$	$10.4\times$
Biocase gathering namedareas†	Metanome	TL	3333.8	1	–	–
	ParCFDFinder (1 thread)	46.22	**948.8**	80	$> 77.9\times$	–
	ParCFDFinder (8 threads)	**30.11**	2957.6	80	$>$ **119.6**$\times$	–
Biocase multimedia object†	Metanome	TL	6135.5	56	–	–
	ParCFDFinder (1 thread)	72.63	**769.2**	256	$> 49.6\times$	–
	ParCFDFinder (8 threads)	**19.44**	3022.6	256	$>$ **185.2**$\times$	–
Bridges	Metanome	47.37	2723.9	13874	–	–
	ParCFDFinder (1 thread)	9.83	**141.8**	13874	$4.8\times$	**19.2**$\times$
	ParCFDFinder (8 threads)	**2.78**	144.5	13874	**17.0**$\times$	$18.8\times$
Echocardiogram	Metanome	25.72	2814.5	9180	–	–
	ParCFDFinder (1 thread)	3.66	**123.5**	9180	$7.0\times$	**22.8**$\times$
	ParCFDFinder (8 threads)	**1.17**	123.9	9180	**22.1**$\times$	$22.7\times$
Students†	Metanome	TL	2859.9	4717	–	–
	ParCFDFinder (1 thread)	707.93	**13897.2**	84044	$> 5.1\times$	–
	ParCFDFinder (8 threads)	**239.35**	15153.3	84044	$>$ **15.0**$\times$	–
Wine Reviews†	Metanome	TL	3681.1	7	–	–
	ParCFDFinder (1 thread)	310.62	**4984.8**	1439	$> 11.6\times$	–
	ParCFDFinder (8 threads)	**102.20**	5465.7	1439	$>$ **35.2**$\times$	–
Wisconsin Breast Cancer	Metanome	212.81	2823.2	4711	–	–
	ParCFDFinder (1 thread)	18.25	**160.0**	4711	$11.7\times$	**17.6**$\times$
	ParCFDFinder (8 threads)	**5.00**	240.9	4711	**42.6**$\times$	$11.7\times$
abalone	Metanome	156.63	318.5	458	–	–
	ParCFDFinder (1 thread)	1.81	**130.6**	458	$86.5\times$	**2.4**$\times$
	ParCFDFinder (8 threads)	**0.66**	153.8	458	**237.3**$\times$	$2.1\times$
ncvoter†	Metanome	TL	4090.9	4294	–	–
	ParCFDFinder (1 thread)	7477.56	**8012.9**	326518	$> 2.9\times$	–
	ParCFDFinder (8 threads)	**2153.15**	8072.3	326518	$>$ **10.0**$\times$	–
nursery	Metanome	143.26	2799.2	255	–	–
	ParCFDFinder (1 thread)	1.46	**122.6**	255	$98.1\times$	**22.8**$\times$
	ParCFDFinder (8 threads)	**0.45**	148.0	255	**318.4**$\times$	$18.9\times$

of rows increases. However, the Java implementation times out when the row count slightly exceeds 18,000, whereas both C++ versions process this scale successfully.

Figure 3 shows the benchmark on the full dataset. Both C++ implementations complete without timing out, and the execution time scales linearly with the number of rows — a favorable property for large-scale datasets. The multi-threaded version achieves a speedup of approximately $3\times$ over the single-threaded C++ implementation on the largest instance.

Figure 4 demonstrates scalability with respect to the number of columns. All algorithms exhibit exponential growth in execution time, which is expected for a problem of this combinatorial nature. The Java version exhausts the three-hour time limit when the number of columns reaches approximately 15, while both C++ implementations successfully process all column configurations up to 19 columns. The multi-threaded version achieves a speedup of approximately 3–4$\times$ on the instance with the highest column count. As the number of columns increases, the advantages of parallelization become evident, and the multi-threaded version consistently outperforms the single-threaded one.

Experiment 3.

These measurements were conducted on the *Students* dataset, as it demonstrated one of the highest runtimes and the largest number of detected CFDs across our datasets.

Figure 5 shows the speedup achieved by the multi-threaded implementation as the number of threads increases, along with the corresponding memory consumption. We observe logarithmic growth in the speedup factor, which is a good result in practice. Additionally, the measured memory footprint remains very close to the baseline (single-threaded) level within the measurement variance.

Typically, increasing the thread count leads to higher memory consumption due to per-thread overhead (e.g., thread

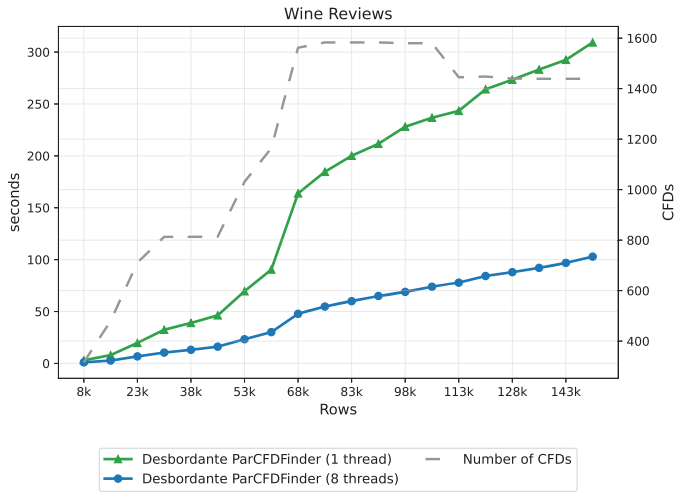

Fig. 3. Scalability with respect to the number of rows: C++ implementations on the full dataset (150,935 rows).

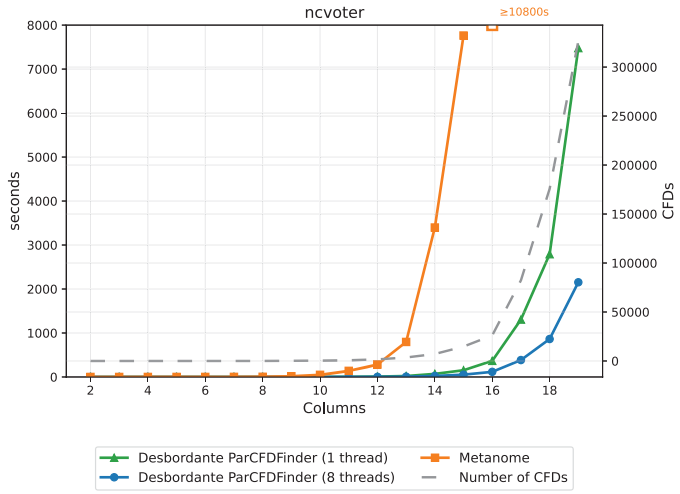

Fig. 4. Scalability with respect to the number of columns.

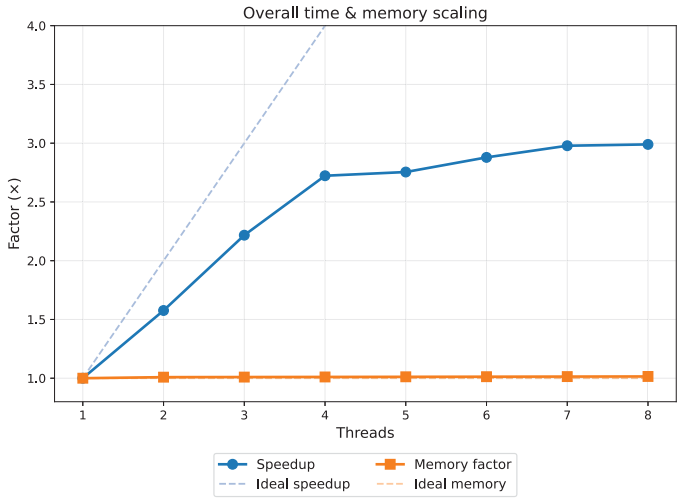

Fig. 5. Scalability with respect to the number of threads.

stacks and synchronization primitives). In our implementation, however, no such increase is observed. Two factors explain this behavior:

First, peak memory is dominated by the *support_map*, which stores CFD candidates with their support values for checking support decay during specialization. This structure accounts for 90–95% of the total memory, making any per-thread overhead negligible.

Second, the algorithm traverses the pattern lattice from the top to the bottom level. The *support_map* accumulates candidates monotonically: all previous candidates remain stored, and new specialized ones are added at each level. Memory peaks at the deepest level, where the number of cumulative candidates is at its largest. Per-thread overhead at intermediate levels is effectively overshadowed by this monotonic growth.

Thus, any thread-related memory overhead is too small to be distinguished from the dominant *support_map* footprint, given our measurement precision.

VII. Case Study

To evaluate the practical results of the algorithm, the implementation was tested on a dataset containing data regarding the effects and side effects of medications across various populations, titled "1000 drugs and side effects"[5]. The dataset consists of 1,000 records and includes nine columns: *Patient_ID*, *Age*, *Gender*, *Condition*, *Drug_Name*, *Dosage_mg*, *Treatment_Duration_days*, *Side_Effects*, and *Improvement_Score*.

The algorithm was executed with the same configuration used in the experiments, yielding a total of 269 CFDs. Below, we examine several of the discovered CFDs.

First, consider the following CFD:

```
[Age,Condition,Drug_Name,Dosage_mg,
Treatment_Duration_days,Side_Effects]
                -> Improvement_Score
PatternTableau {
        (_|Infection|_|_|_|_)
        (_|Diabetes|_|_|_|_)
        (_|Hypertension|_|_|_|_)
        (_|Depression|_|_|_|_)
}
Support: 0.792
Confidence: 1.0
```

This CFD indicates that for conditions such as Infection, Diabetes, Hypertension, and Depression, if the patient's age, medication, dosage, side effects, and treatment duration are known, the treatment outcome (*Improvement_Score*) is uniquely determined. In other words, for these specific conditions, a particular therapy within a given age group yields a stable and reproducible result. The dependency has a support value of 0.792, demonstrating that it models a widespread pattern within the dataset.

As another example, consider the following CFD:

```
[Age,Gender,Condition,
Treatment_Duration_days] -> Side_Effects
PatternTableau {
```

[5]URL:https://www.kaggle.com/datasets/palakjain9/1000-drugs-and-side-effects/data

```
        (_|_|Pain Relief|_)
        (_|Female|Diabetes|_)
}
Support: 0.31
Confidence: 1.0
```

In this sample, we specifically observe that for female patients with diabetes, the side effects of the treatment are entirely determined by their age and treatment duration (as shown in the second pattern of the tableau). This CFD has a support of 0.31 and a confidence of 1.0, representing a precise rule that holds true for specific patient subgroups.

VIII. Conclusion

We have presented ParCFDFinder, an improved implementation of CFDFinder obtained by combining algorithmic and engineering techniques available in C++, together with a parallelization strategy that yields additional speedup. The resulting algorithm is up to $318\times$ faster than the state-of-the-art baseline and reduces memory usage by up to $23\times$. ParCFDFinder is integrated into Desbordante, an open-source data profiler with a Python interface, enabling users to import and invoke the algorithm from Python using only a few lines of code.

Consequently, we have extended the practical applicability of CFD discovery: for the first time, users can process hundreds of thousands of records on a commodity machine within a reasonable time. This enables analysis of meaningfully sized datasets, extraction of actionable insights, and development of data-quality applications. Our case study illustrated, through a concrete example, how ParCFDFinder serves these purposes.

Future work includes expanding the pool of CFD discovery algorithms available in Desbordante, as different algorithms yield complementary sets of CFDs and expose different tunable parameters.

Acknowledgments

We would like to express our gratitude to Thorsten Papenbrock and Felix Naumann for providing access to the text of the master's thesis [3].

References

[1] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: an experimental evaluation of seven algorithms," *Proc. VLDB Endow.*, vol. 8, no. 10, p. 1082–1093, jun 2015. [Online]. Available: https://doi.org/10.14778/2794367.2794377

[2] G. Chernishev, M. Polyntsov, A. Chizhov, K. Stupakov, I. Shchuckin, A. Smirnov, M. Strutovsky *et al.*, "Desbordante: from benchmarking suite to high-performance science-intensive data profiler," in *Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD)*, ser. CODS-COMAD '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 234–243. [Online]. Available: https://doi.org/10.1145/3703323.3703725

[3] M. Grundke, "Discovering interesting conditional functional dependencies," Master's thesis, Information Systems Chair Hasso-Plattner-Institute, Potsdam, 2018.

[4] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann, "Data profiling with Metanome," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1860–1863, Aug. 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824086

[5] A. Shlyonskikh, M. Sinelnikov, D. Nikolaev, Y. Litvinov, and G. Chernishev, "Lightning fast matching dependency discovery with desbordante," in *2024 36th Conference of Open Innovations Association (FRUCT)*, 2024, pp. 729–740.

[6] A. Smirnov, A. Chizhov, I. Shchuckin, N. Bobrov, and G. Chernishev, "Fast discovery of inclusion dependencies with desbordante," in *2023 33rd Conference of Open Innovations Association (FRUCT)*, 2023, pp. 264–275.

[7] Y. Kuzin, D. Shcheka, M. Polyntsov, K. Stupakov, M. Firsov, and G. Chernishev, "Order in desbordante: Techniques for efficient implementation of order dependency discovery algorithms," in *2024 35th Conference of Open Innovations Association (FRUCT)*, 2024, pp. 413–424.

[8] M. Strutovskiy, N. Bobrov, K. Smirnov, and G. Chernishev, "Desbordante: a framework for exploring limits of dependency discovery algorithms," in *2021 29th Conference of Open Innovations Association (FRUCT)*, 2021, pp. 344–354.

[9] F. Chiang and R. Miller, "Discovering data quality rules," *PVLDB*, vol. 1, pp. 1166–1177, 08 2008.

[10] W. Fan, F. Geerts, L. V. S. Lakshmanan, and M. Xiong, "Discovering conditional functional dependencies," pp. 1231–1234, 2009.

[11] J. Li, G. Liu, and L. Wong, "Mining statistically important equivalence classes and delta-discriminative emerging patterns," *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 430–439, 08 2007.

[12] T. Diallo, N. Novelli, and J.-M. Petit, "Discovering (frequent) constant conditional functional dependencies," *Int. J. Data Mining, Modelling and Management Int. J. Data Mining, Modelling and Management*, vol. 120, 01 2010.

[13] J. Rammelaere and F. Geerts, *Revisiting Conditional Functional Dependency Discovery: Splitting the "C" from the "FD": European Conference, ECML PKDD 2018, Dublin, Ireland, September 10–14, 2018, Proceedings, Part II*, 01 2019, pp. 552–568.

[14] T. Papenbrock and F. Naumann, "A hybrid approach to functional dependency discovery," pp. 821–833, 06 2016.

[15] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu, "On generating near-optimal tableaux for conditional functional dependencies," *Proc. VLDB Endow.*, vol. 1, no. 1, p. 376–390, Aug. 2008. [Online]. Available: https://doi.org/10.14778/1453856.1453900