Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms

Maxim Strutovskiy, Nikita Bobrov, Kirill Smirnov, George Chernishev
Saint Petersburg State University
Saint Petersburg, Russia
strutovsky.m.a, nikita.v.bobrov, kirill.k.smirnov, chernishev@gmail.com

***Abstract*—Automatic discovery of various types of database dependencies (functional, inclusion, matching, and others) is a topic that has received a great deal of attention in the recent years. The problem is formulated as following: having an unexplored dataset, find all dependencies that hold on this data. Such problem formulation arises in business and scientific applications and is aimed at the discovery of patterns in data.**

Metanome is a pioneering platform which was used to benchmark existing and develop new dependency discovery algorithms. It is notable since it was the first attempt to unify all existing discovery algorithms inside a single suite. However, it should be considered a research prototype rather than a system ready for industrial use. The core reason for this is the choice of the implementation platform (Java) and the absence of optimizations.

In this paper we address the problem of high-performance dependency discovery. We present Desbordante — a platform that is intended to make the most of the available computational resources and thus to be more suitable for industrial use.

Finally, we evaluate our system experimentally and pose a number of research questions related to the obtained performance and justify its necessity. More precisely we examine 1) whether the Java implementation is indeed worse than the C++ one, 2) is it possible to use simple tricks to improve Metanome's performance, 3) what are the exact reasons behind the performance gap, and 4) what are the user-facing benefits of switching the implementations.

I. INTRODUCTION

Database dependencies [1] represent patterns contained within the data. Their discovery has several major applications [2] such as data exploration, schema engineering, data cleaning, query optimization, and data integration.

Automatic discovery of functional and other kinds of dependencies Has received a great deal of attention in the recent years. The problem is the following: having an unexplored dataset (table), find all dependencies that hold on this data. Dependency discovery is computationally-intensive: the state-of-the-art algorithms take hours or days to finish, even for megabyte-sized tables on server-class hardware [3].

Early dependency discovery algorithms were implemented inside different prototypes, using different programming languages, and compared using different datasets and baselines. To the best of our knowledge, there is only a single tool that unites the results of all existing papers. Essentially, it contains all algorithms implemented inside a single suite, alongside with many benchmarks. Its authors used their prototype to compare various existing algorithms [3]–[5] and to develop and evaluate new ones [6]–[9]. This suite is called Metanome [10] and it is implemented in Java.

Java has many well-known advantages, such as its relative ease of learning, suitability for rapid application development, a platform-agnostic computing environment, availability of visualization libraries and UI frameworks and many more.

At the same time, Java possesses a number of no less prominent drawbacks:

1) Given an equitable effort put into code, the resulting performance of Java applications is worse than that of C++, on average. This happens mostly due to JVM overhead.
2) Java application performance can be unpredictable. Since explicit memory management is not possible in Java, programs rely on an automatic garbage collector, which may be invoked at any time. Therefore, run times may significantly differ even for consecutive invocations of single-threaded programs. For this reason, in order to obtain reliable benchmarking results, sophisticated approaches like Java Benchmark Harness [11] are used.
3) Java programs usually leave a higher memory footprint than C++.

Finally, Java does not allow low-level optimizations, such as vectorization via SIMD instructions. Currently, only auto vectorization is available to Java programmers and it is not reliable, e.g. has trouble vectorizing a simple loop [12]. Vector API [13] is another option to employ SIMD in Java. However, it is still in the incubation phase and scheduled to be included [14] in JDK 16 in March 2021. Aside from that, the prospective performance of this feature compared to the C++ one is still unclear.

Another possibility to access low-level optimization for Java programmers is the JNI [15]. However, it is well-known [16], [17] that it invokes a high overhead when switching between Java and native code. Dependency discovery algorithms are data-intensive and thus, they will require a lot of such switching in performance-critical parts implemented in native languages. Therefore, this is also not an appropriate option.

Overall, the described inability to use low-level optimizations is a critical drawback for solving a high-performance computing task.

Finally, dependency discovery algorithms are very complex and frequently multi-threaded. Therefore, understanding their asymptotic computational complexity is very hard, and of limited use. Instead, in order to assess them, prospective users usually run an experimental study.

The Java implementation (Metanome) is sufficient to understand the relative performance of different algorithms. However, for industrial applications it is necessary to know their true limits to understand what datasets can be efficiently mined. To address this, we have created Desbordante (meaning "limitless" in Spanish) — a platform that is aimed to address this drawback of Metanome. It is fully written in C++, extendable and completely open-source [18].

In this paper we describe the proposed tool, its architecture, and present an experimental study that justifies its usefulness. To the best of our knowledge, Desbordante is the first tool intended specifically for high-performance dependency discovery.

Our experimental study consists of two parts: implementation-related and algorithm-related. The first part is focused on high-level implementation details of both systems. We study their impact on run-time behavior and resulting performance, and what are reasons behind it. The second part addresses the user-facing benefits in terms of improvement obtained by a specific algorithm. For this purpose, we employ Pyro [8] — an algorithm for approximate dependency discovery. Our choice is justified by the fact that Pyro is a state-of-the-art AFD discovery algorithm. The detailed list of research questions (RQs) that are addressed in this paper is presented in Section III.

This paper is organized as follows. In Section II we provide definitions of both regular and approximate functional dependencies, present the taxonomy of FD mining algorithms and briefly discuss them. In Section II-C we give a short overview of the Metanome system. Next, in Sections III and IV we discuss our experiment design and its outcomes. We conclude this study with Section V.

II. Background and Related work

This section provides the reader with all of the necessary background information: we introduce basic definitions of database dependency theory, describe the taxonomy of FD mining algorithms, and give a high-level overview of the Metanome project.

A. Functional dependencies

All the following definitions are expressed using the modern notation derived from studies [6], [8], which is slightly different from the classic [19], [20]. Nevertheless, the main idea behind *exact* functional dependencies stays the same:

Definition 1: Given a relational schema R and an instance r over R with attribute sets $X, Y \subset R$, we say that a functional dependency $X \to Y$ holds iff for any $t_1,\ t_2 \in r$, the following is true: if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$. We call the determinant set of attributes X the left-hand side (LHS) of the FD, and the dependent set the right-hand side (RHS).

However, real data contains typos, missing values, and noise. This results in scenarios when a FD can not be inferred, but common sense or domain specific knowledge implies that such a FD exists and has to be found. To address the problem of dirty data, a special class of algorithms was developed: Approximate FD mining algorithms (AFD). "Approximate" here means that a fraction of rows is allowed to disagree on RHS values if they agree on LHS. Thus, in these algorithms the discovery process is controlled by the error parameter e_{max}, which is used as a threshold to determine whether an AFD $X \to Y$ meets the condition on the maximum allowed fraction of "broken" rows (i.e. $e(X \to Y) \leq e_{max}$) or not.

Definition 2: Given an instance r and an AFD candidate $X \to Y$, its error is calculated as [8]:

$$e(X \to Y, r) = \frac{|\{(t_1, t_2) \in r^2 | t_1[X] = t_2[X] \land t_1[Y] \neq t_2[Y]\}|}{|r|^2 - |r|}$$

AFD $X \to Y$ holds on r if $e(X \to Y, r) \leq e_{max}$.

Note that setting $e_{max} = 0$ reduces the AFD mining process to FD.

B. FD mining algorithms taxonomy

Several dozen mining algorithms for different types of database dependencies have been developed during the last forty years. Each of them solves the discovery problem in their own unique way. Nevertheless, they can be grouped by one of the following fundamental approaches:

1) Algorithms based on a *lattice-traversal* approach represent dependency search space as a partially ordered set (*poset*) of a N-element set (where N is the number of attributes). In the literature, the Hasse diagram is used to visualize such a search space [19], [21], [22]. The algorithm iteratively traverses it, starting from the zeroth level. First, it considers dependencies $\varnothing \to A_1 \dots A_N$. Then it constructs the next level, containing dependency candidates which are currently not minimal and need to be verified. Candidate verification is performed via intersection of *position list indices* (denoted by $\pi(X)$): it is based on a fundamental lemma first described in [19]. A position list index is a data structure consisting of a list of *clusters*. A single cluster is a list of rows containing the same value. Next, in a successive manner, the algorithm constructs poset levels until no more candidates can be generated. The number of levels to visit and overall algorithm complexity is dependent on N, which results in a poor scalability in terms of the number of attributes [3].
2) Another approach to dependency candidate generation is based on searching for attribute subsets that agree on a certain number of tuples [23], [24]. *Agree-sets* are constructed by performing pair-wise tuple comparisons, which makes search space size dependent on the number of table rows. As a part of a pruning strategy many algorithms perform construction of *difference-sets* (i.e. agree-sets complement) before running the actual candidate validation. These sets are used by an algorithm to determine subsets of attributes that would likely be parts of an LHS and RHS of a specific candidate dependency and which are not.

3) A *hybrid* approach is modern [6]. It has been proposed to address the problem of poor column and row scalability of existing approaches. Its authors suggest to divide the mining process into two separate phases — one for calculating candidate dependencies on a small subset of data, and the other one for candidate validation on the entire dataset. This approach had proved its effectiveness not just in terms of performance, but also in the number of algorithms which inherit its core idea. The approach had already been extended to approximate dependencies inference [8] and to dependency mining in dynamically changing environments [7].
   The first reference is the paper where Pyro — an algorithm that we evaluate in the current study — was originally presented. The second reference describes the process that avoids full recalculation of the FD minimal cover and allows to maintain it in an actual state in environments where UPDATE, INSERT and DELETE operations often occur. This new domain of algorithms for incremental discovery looks especially promising if FD mining is performed in the context of big data [25], [26].
4) The last of the most prominent approaches is dependency mining via *approximation schemes* [27] or *statistical learning* [28]. This approach is different from the previous ones since it does not guarantee finding a minimal cover of exact or approximate FDs. However, the approach is worth mentioning since it contains methods which have not been applied to the FD mining domain until the most recent time.
   The basic idea is to reduce the FD discovery to the problem that was already solved for more general structures. For example, in [28] the authors develop a framework which maps FD discovery to sparse regression problem solving. Another study considers [29] FDs as undirected structures that can be learned from data with the Graphical Lasso algorithm.

C. Metanome

To the best of our knowledge, Metanome is currently the only data profiling tool that provides its user with:

1) a framework for developing and testing dependency mining algorithms,
2) support of various data types and DBMS connections,
3) a frontend web-service.

From a developer's perspective, Metanome is convenient to use. It specifies all necessary interfaces per each of the most popular types of dependency mining algorithms a user might want to implement — FD, AFD, ID, order dependency, unique column combinations, etc. Metanome core classes implement all data structures necessary for dependency discovery — position list index (on top of Apache Lucene), agree-set, and different types of nodes which can be used for search space construction. Finally, a relation builder component allows to avoid writing boilerplate code for tabular data reading functionality.

For a data analyst, Metanome is a platform that can be used for obtaining knowledge and insights about data. The platform can process plain text files or can be asked to retrieve data via a DBMS connector. Metanome's algorithm library is populated with discovery algorithms by simple drag-and-drop of a jar file containing the algorithm implementation. After data is processed, the tool maintains high-level statistics on it, preserves the history of mining algorithms invocations, and visualizes found dependencies as a sunburst chart.

D. Pyro

Pyro [8] is a state-of-the-art algorithm for mining approximate functional dependencies, which follows the *hybrid* approach. It is worth mentioning that it finds not only AFDs, but also AUCCs (approximate unique column combinations). However, we will not consider AUCCs in our Pyro overview due to two reasons:

1) this type of constraints is out of scope of our study, and,
2) any of the aforementioned processes can be applied to AUCC mining in the same way as AFD.

As it was mentioned, the hybrid approach employs both agree set and position list index as its core data structures. Besides, Pyro maintains two auxiliary data structures: the *agree set sample cache* and the *PLI cache*. It will be shown that these data structures are very useful when Pyro estimates errors of dependency candidates and performs their validation.

For now, we want to emphasize the importance of the PLI cache that can be useful not just for the discovery process performed by Pyro, but for any hybrid or lattice-traversal algorithm which is based on a rather computationally expensive PLI intersection operation. It is common that during the verification phase intersection is performed many times on the same sets of attributes. For example, in order to verify $X, Y \to W$ and $X, Y \to Z$ dependencies, the calculation of the $\pi(XY)$ intersection would likely be performed twice. To make the process faster, authors of Pyro suggest to maintain a prefix tree which contains results of partition intersection. That is, before intersecting PLIs on a set of attributes, the algorithm first checks trie whether this PLI had already been calculated or not. Decision on caching a PLI follows a *coin flip* strategy — each PLI has a 0.5 probability to be cached. More sophisticated caching approaches were developed in [36], but in the current study we use the default coin flip.

The Pyro discovery process itself is divided into two phases:

1) *error assessment* for AFD candidate hypothesizing;
2) sampling-based best-first *search space traversal*.

Dependency candidate error assessment is one of the most expensive tasks for AFD mining algorithms. Pyro solves the problem in the following way: instead of performing full error *calculation* based on PLIs intersection, the algorithm *estimates* the error first, and then extrapolates it to the whole relation. Only if the estimation satisfies the error hyperparameter value e_{max}, the algorithm performs targeted error calculation.

TABLE I. OVERVIEW OF THE EVALUATION DATASETS

Dataset	Source	Rows	Cols	Size	#FDs	#NULLs
Adult	uci [30]	32K	15	3.5 MB	78	0, 0%
BreastCancer	uci [30]	500	30	118 KB	11835	0, 0%
CIPublicHighway	data.sa.gov.au [31]	427K	18	27 MB	143	3.5M, 46.3%
EpicMeds	epa [32]	1.3M	10	55 MB	17	280K, 2.2%
EpicVitals	epa [32]	1.2M	7	33 MB	2	0, 0%
Iowa1KK	mydata.iowa.gov [33]	1M	24	210 MB	1585	1M, 4.2%
LegacyPayors	epa [32]	1.4M	4	21 MB	6	0, 0%
Neighbors100K	SDSS [34]	100K	7	6.4 MB	15	0, 0%
SG_Bioentry	BioSQL [35]	184K	8	24 MB	19	184K, 11.1%

Estimation itself is based on comparisons of tuples subsets, which makes the error assessment phase resemble the process of candidate generation described in Section II-B, as it also employs agree sets to derive sets of attributes on which subsets of tuples agree. To make estimates more precise and unbiased from small agree set samples, Pyro performs *focused sampling*. The process is called *focused* because it is defined by two conditions:

1) agree sets that are sampled must be supersets of some given attribute set X, and
2) tuples that are sampled must co-occur in the same clusters of $\pi(X)$.

Agree set samples are stored in the corresponding cache, which is used for retrieving a more appropriate agree set sample when performing error estimates for specific attribute sets.

Search space traversal is performed by Pyro in its own unique way — using a separate-and-conquer strategy. For each attribute X Pyro creates a search space which can be considered as a fragment of a partially ordered set (poset we mentioned in Section II-A) where all minimal AFDs of $Y \to X$ format are located. Each iteration over the search space is started from a *launchpad* — a single attribute at the bottom level of poset with the smallest error estimate. Starting to check candidates on a higher levels (dependencies with long LHS), algorithm *ascends* until it finds minimal dependency on its path. When the dependency is verified as minimal, e.g., $Y_1, Y_2, Y_3, \ldots, Y_N \to X$, Pyro *trickles down* to lower poset levels to estimate the errors of the generalizations of that minimal dependency, which are: $Y_2, Y_3, \ldots, Y_N \to X$, $Y_3, \ldots, Y_N \to X$ and so on. Each of them is a minimal candidate now, so Pyro recursively trickles down until the encountered candidate error estimate is larger than the error hyperparameter e_{max}. After the new candidates are checked (verified), Pyro ends the iteration by checking the complement of the generalizations. Pyro inspects unexplored search space parts at the very end of traversal. The authors emphasize that the strategy is sufficiently general and can be used for both AFD and AUCC discovery.

III. EXPERIMENTAL DESIGN

A. Experimental setup

For our experiments we have selected a number of datasets listed in Table I. In this table we list their important properties: number of rows and columns, their size, the number of exact functional dependencies, and the number of `NULL` values contained in the table (and their share).

Currently, Desbordante supports two algorithms: Tane [19] and Pyro [8]. In our experiments we consider only Pyro due to the anticipated demand.

Experiments were run on a PC with the following hardware specs: Intel(R) Core(TM) i5-7600K CPU (4 cores) @ 3.80GHz 16GB DDR4 2133MHz RAM, 2TB HDD WD20EZAZ, as well as the following software: Pop!_OS 20.10, 64-bit, C++: gcc 10.2.0, Java: OpenJDK 64-bit Server VM 11.0.9.1, GraalVM CE 21.0.0.

B. Evaluation metrics

Existing papers that evaluate dependency discovery algorithms [3], [8] consider roughly the same set of metrics concerning run times and memory usage. Every new algorithm tries to minimize them, thus extending the boundaries of what datasets can be processed.

In this study we are also interested in other metrics which characterize the efficiency of the system and the algorithm:

- the LHS size of dependency;
- the number of outliers.

For FD discovery algorithms we calculate metrics as follows:

- for each language and dataset we perform 10 launches in order to build 95% confidence intervals for run times and memory metrics;
- the RNG seed was fixed for all experiments, except RQ1, where a number of different (but same for both implementations) seeds was used. The impact of the RNG seed on algorithm performance is discussed in Section IV-D;
- for large datasets we incrementally increase LHS size to obtain the max possible value while run times memory limits are satisfied.

C. Research questions

We consider two groups of research questions (RQs). The first one aims to justify the need of creating a new system and tries to explore the exact reasons why the performance of Java-based implementation suffers.

- *RQ1: Does the C++ implementation outperform Java in terms of run-times and memory consumption metrics?*

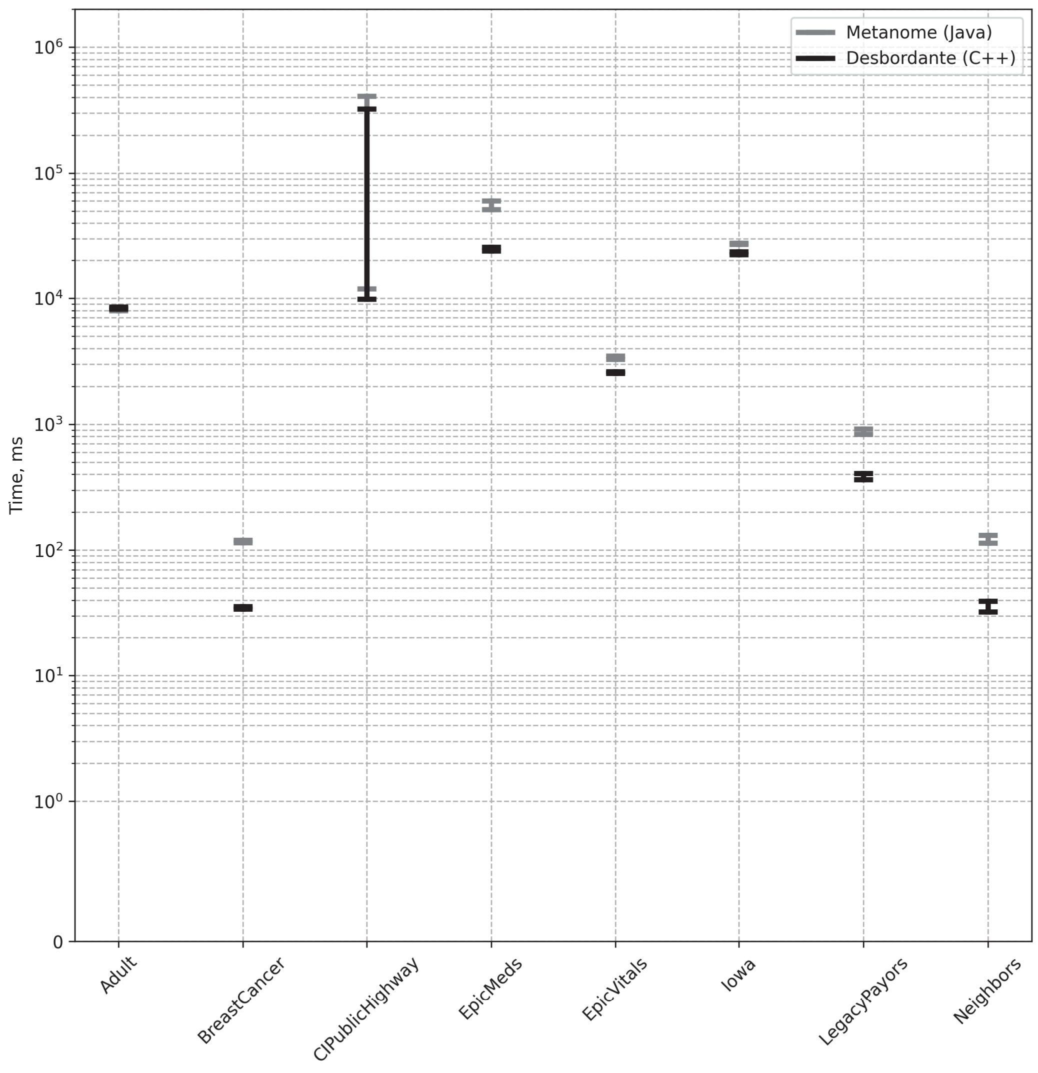

Fig. 1. RQ1: performance comparison of Desbordante and Metanome

- *RQ2: Are there any "straightforward" techniques that will allow to bring the performance of the Java implementation closer to the C++ one?* Here, straightforward means that the user will tune the run time environment and vary its parameters without touching the actual code.
- *RQ3: What are the reasons behind the differences in these metrics?*

The second group is dedicated to understanding and quantifying the benefits that we can get from moving to the C++ implementation.

- *RQ4: What are the reasons that stand behind Java runtime outliers? Is it possible to get rid of them? Can we guarantee stable run times?*
- *RQ5: How the performance is impacted, if maxLHS parameter is set?*

IV. Experimental Results and Discussion

A. RQ1: Does the C++ implementation outperform Java in terms of run-times and memory consumption metrics?

For this RQ, we experimentally evaluate out-of-the-box performance of Metanome and compare it with Desbordante's. For Metanome, we have selected the best possible set of configuration options (see RQ2). At the same time, we have not tuned Desbordante's compile options, using only the "-O3" parameter. The performance was studied using exact FD

TABLE II. RQ1: COMPARING THE DESBORDANTE AND METANOME IMPLEMENTATIONS (RUN TIMES)

Implementation	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
Desbordante	8381 ± 154	34 ± 0	166342 ± 156446	24599 ± 873	2580 ± 33	22854 ± 566	385 ± 21	35 ± 3
Metanome	8177 ± 176	117 ± 2	210615 ± 198689	55680 ± 4346	3383 ± 103	27228 ± 318	875 ± 44	122 ± 8

TABLE III. RQ1: COMPARING THE DESBORDANTE AND METANOME IMPLEMENTATIONS (MEMORY CONSUMPTION)

Method	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
Desbordante, AVG	174.03 MB	567.57 KB	858.37 MB	248.44 MB	133.63 MB	495.58 MB	88.80 MB	9.65 MB
Desbordante, MAX	311.93 MB	1.28 MB	1.40 GB	306.35 MB	181.89 MB	673.27 MB	177.00 MB	22.35 MB
Metanome, AVG	281.62 MB	16.18 MB	999.29 MB	638.68 MB	193.33 MB	725.60 MB	268.34 MB	55.68 MB
Metanome, MAX	575.57 MB	24.49 MB	1.89 GB	1.18 GB	426.83 MB	1.12 GB	493.61 MB	109.99 MB

discovery with the Pyro algorithm (i.e., the error was set to zero). The same algorithm was used in all subsequent RQs.

It is important to note that we have not exhausted the tuning potential of the C++ implementation. This concerns not only compile options, but data structures and libraries as well. Currently, Desbordante uses default C++ and Boost data structures, and we have not tuned their parameters. Desbordante does not rely on custom memory management libraries (allocators), but instead uses the C++ default. It is well-known [37] that using a special allocator is a simple way to improve performance of C++ programs.

The results of this experiment are presented in Figure 1. Since datasets have very different run times, we have plotted them using a logarithmic axis.

It can be seen that the C++ implementation (Desbordante) is clearly superior to Java (Metanome) on all but two datasets. The performance on “Adult” and “CIPublicHighway” datasets is approximately the same, and their confidence intervals heavily intersect. The obtained improvement ranged from 1.19 to 3.43 times, 2.12 on average. The exact numbers are presented in Table II.

Turning to memory consumption, we have measured both maximum and average memory consumption over the course of application. The results are presented in Table III. In terms of memory consumption, Desbordante is also superior to Metanome on all datasets: the obtained improvement ranged from 1.46 to 2.57 times, 1.88 on average (average consumption over the course). Note that the results on the “breast_cancer” dataset were not taken into account due to the clear outlier nature of the latter. The peak consumption ratios are usually even higher.

B. RQ2: Are there any “straightforward” techniques that will allow to bring the performance of the Java implementation closer to the C++ one?

This RQ is dedicated to checking whether the performance of Metanome can be improved using simple techniques that do not require code modification. Java programs heavily depend on run time specifics: many options may affect their performance. Let us list the ones that we have identified and tested in our experiments:

1) JDK versions: 11 and 15. We have considered these versions since 11 is the latest LTS (Long-Term Support), and 15 is the latest STS (Short-Term Support).
2) Compilation approach: JIT (Just-In-Time) compilation or AOT (Ahead-Of-Time). For JIT compilation, the Hotspot JVM was used and for AOT — GraalVM;
3) Compilation type: client, server or tiered compilation;
4) If tiered compilation is used, then parameters T3 and T4 can be varied;
5) Garbage collector: G1, MarkSweep, Parallel, Serial;
6) Maximum (and minimum) heap size;

The results of varying the JDK version are presented in Table IV. We can see that:

- JDK choice barely affects performance in the majority of scenarios.
- The only affected datasets are CIPublicHighway and EpicMeds: the difference between JDK versions reaches 50% and 150% respectively. Having investigated (using event counting with perf) the reasons of this behavior, we found out that the culprit is the number of native instructions, which is more than 3 times higher for GraalVM. Interestingly, data-related events (cache misses, TLB misses) are approximately the same for both implementations. Such difference in the number of instructions can be observed only on these two datasets. Therefore, we can conclude that this is a code generation anomaly of GraalVM.
- Another observation is the following: even on regular datasets, the newer HotSpot JDK shows slightly worse results, which can be explained by their purposes — one being LTS and another STS.
- Finally, smaller datasets are less impacted.

Next, we have studied the effect of JDK compilation options. The results are presented in Table V. It is evident that there are no large differences between them: at most, 10% of improvement is obtained. Additionally, in a lot of cases confidence intervals overlap, therefore we can not make reliable conclusions.

We have also tried to tinker with Client, Server, and Tiered compilation options. The first two are essentially different compilers (C1 and C2, respectively) [38] and

TABLE IV. RQ2: Varying JDK version (JIT only)

Method	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
HotSpot JDK 11	7642 ± 64	115 ± 2	41178 ± 220	51930 ± 4029	3329 ± 144	27318 ± 228	809 ± 32	152 ± 6
HotSpot JDK 15	7886 ± 62	114 ± 4	57174 ± 527	67583 ± 3318	3413 ± 55	27545 ± 472	879 ± 36	151 ± 5
GraalVM JDK 11	8888 ± 36	125 ± 4	46828 ± 620	162917 ± 177	5836 ± 60	25968 ± 1156	1082 ± 21	153 ± 9

TABLE V. RQ2: Varying JDK compilation options (HotSpot, except AOT)

Method	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
AOT (GraalVM)	8525 ± 39	119 ± 6	54108 ± 1329	163724 ± 385	5761 ± 121	27417 ± 1864	1105 ± 51	147 ± 5
Client	8229 ± 116	282 ± 2	44681 ± 597	45006 ± 1686	3138 ± 136	26602 ± 469	842 ± 17	233 ± 9
Server	8205 ± 52	282 ± 3	44691 ± 552	46962 ± 3588	3160 ± 154	27033 ± 320	848 ± 23	252 ± 11
Tiered (default)	7703 ± 126	114 ± 2	41527 ± 243	51261 ± 3480	3312 ± 116	27098 ± 345	822 ± 20	149 ± 5
T3=2K, T4=15K	8029 ± 78	174 ± 2	42747 ± 231	47883 ± 5175	3331 ± 146	28335 ± 210	887 ± 27	159 ± 8
T3=0.5K, T4=5K	7845 ± 140	120 ± 2	41660 ± 252	50770 ± 4484	3401 ± 104	27802 ± 617	821 ± 32	153 ± 7
T3=6K, T4=30K	7953 ± 89	176 ± 2	42560 ± 339	47740 ± 5113	3389 ± 144	27859 ± 293	892 ± 26	162 ± 11

TABLE VI. RQ2: Varying garbage collector

Method	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
G1	7684 ± 114	113 ± 3	41635 ± 317	50468 ± 4065	3359 ± 126	26352 ± 485	818 ± 15	150 ± 9
MarkSweep	7717 ± 88	120 ± 4	42226 ± 504	66501 ± 2543	3513 ± 58	26447 ± 318	873 ± 8	143 ± 5
Parallel	7723 ± 89	114 ± 2	42448 ± 372	56927 ± 344	3291 ± 36	25829 ± 265	779 ± 17	139 ± 3
Serial	7911 ± 200	122 ± 23	43022 ± 922	55055 ± 2945	3390 ± 80	25814 ± 586	845 ± 15	148 ± 6

TABLE VII. RQ2: Varying heap size

Method	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
Xms256	7972 ± 85	113 ± 3	ML	53301 ± 3989	3392 ± 87	ML	823 ± 27	150 ± 7
Xms512	7502 ± 45	128 ± 4	ML	54054 ± 4278	3207 ± 94	27343 ± 272	818 ± 35	147 ± 7
Xms1024	7483 ± 114	119 ± 3	41149 ± 436	53099 ± 4206	3265 ± 53	26026 ± 278	835 ± 40	139 ± 7
Xms2048	7639 ± 115	120 ± 4	41022 ± 267	50358 ± 4390	3290 ± 124	24889 ± 191	830 ± 25	149 ± 4

the third one is an option that allows to select a compiler on a per-method basis. This option is guided by a heuristic that is roughly an invocation counter that cools down over time. This counter has two thresholds which we name T3 and T4 ("-XX:Tier3InvocationThreshold" and "-XX:Tier4InvocationThreshold" respectively). If the first threshold has been reached, then the C1 compiler is run. Next, if the second is reached, then the C2 is used to compile the method. The results that are presented in Table V demonstrate that Ahead-Of-Time compilation is an unreliable choice which may be suitable only for tiny datasets and yet it fails to beat the JIT approach. Next, using only Client or Server compiler is also a poor choice. Overall, JIT compilation that comes out-of-the-box (Tiered with T3=200, T4=5000) appears to be the best option.

The next option package concerned the impact of the garbage collector. It can be observed (see Table VI) that for regular datasets there is no real difference since the confidence intervals intersect in a large number of cases. The default option — ParallelGC — looks like a decent, safe choice.

Next, the effects of the "Xms" option are presented in Table VII. It can be seen that having more memory at the start improves performance on large datasets (see Table III).

Finally, we can conclude that it is not possible to improve performance of Metanome using "straightforward" techniques, i.e. by tinkering with run time options or switching to another JVM. Additionally, we can recommend to utilize default options and the LTS JDK when using Metanome.

C. RQ3: What are the reasons behind the differences in these metrics?

Let us attempt to address why exactly Desbordante is faster than Metanome. First of all, Desbordante uses less memory, thus probably requiring fewer allocations (see Table III). Second, we have decided to check the number of various hardware and software events, like cache misses and page faults. For this, we run perf both for Desbordante and Metanome. The results are displayed in Table VIII. For the presentation reasons we have included the performance improvement ratio of Desbordante over Metanome as the first line. It was calculated using the data taken from Table II. The next lines contain event statistics, which were counted for Desbordante (D) and Metanome (M) for each of the considered datasets.

We can divide our datasets into two groups: high-improvement and low to no improvement. The former contains

TABLE VIII. RQ3:
VARIOUS EVENTS

Event	P	breast_cancer	LegacyPayors	adult	EpicVitals	Iowa1KK	Neighbors100K	CIPublicHighway	EpicMeds
improvement	—	3.44	2.27	0.98	1.31	1.19	3.49	1.34	2.26
instructions	D	368.98M	7.59G	52.15G	15.05G	130.64G	1.18G	199.43G	48.67G
	M	4.80G	37.24G	106.32G	53.70G	275.52G	12.22G	489.63G	590.22G
L1-dcache-load-misses	D	1.75M	131.21M	716.73M	742.94M	3.56G	10.96M	2.65G	17.26G
	M	94.27M	468.57M	1.29G	975.52M	4.07G	155.90M	6.10G	17.24G
L1-icache-load-misses	D	8.72M	2.92M	73.72M	3.19M	49.65M	1.01M	34.61M	13.47M
	M	89.09M	113.13M	258.55M	131.81M	267.78M	89.57M	265.14M	186.29M
L2-misses	D	2.45M	165.54M	662.08M	786.95M	6.28G	14.41M	3.45G	9.37G
	M	139.69M	565.78M	1.42G	1.15G	7.43G	189.89M	6.38G	4.64G
LLC-load-misses	D	11.53K	13.12M	8.92M	26.77M	114.54M	761.36K	492.79M	77.43M
	M	1.97M	16.47M	32.46M	34.33M	217.43M	4.26M	879.14M	94.71M
LLC-store-misses	D	32.19K	28.14M	6.73M	10.62M	76.16M	746.36K	105.85M	19.45M
	M	2.66M	31.22M	65.53M	35.72M	207.50M	6.92M	263.19M	70.00M
cache-misses	D	203.48K	117.83M	105.77M	154.55M	877.17M	6.96M	1.95G	531.16M
	M	31.56M	255.98M	527.59M	332.53M	2.05G	64.09M	4.82G	918.86M
branch-instructions	D	79.17M	1.47G	10.77G	3.07G	26.09G	253.11M	40.97G	10.08G
	M	838.41M	7.13G	18.62G	11.99G	47.91G	2.24G	88.28G	170.86G
branch-misses	D	947.56K	8.64M	186.83M	29.47M	358.19M	1.60M	272.52M	143.19M
	M	37.00M	153.30M	362.65M	157.66M	832.07M	59.18M	493.29M	393.31M
context-switches	D	16.00	106.00	760.00	207.00	5.34K	3.00	243.00	3.17K
	M	2.50K	3.48K	4.23K	3.50K	11.36K	2.68K	13.90K	6.17K
cpu-migrations	D	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
	M	119.00	343.00	286.00	273.00	1.24K	217.00	1.46K	477.00
dTLB-load-misses	D	5.31K	5.75M	3.13M	16.45M	76.73M	328.80K	504.91M	36.52M
	M	827.63K	15.00M	9.44M	18.90M	91.24M	1.61M	839.22M	39.69M
iTLB-load-misses	D	3.05K	137.62K	673.54K	187.71K	1.46M	23.64K	1.44M	447.86K
	M	608.40K	1.76M	3.09M	2.09M	3.96M	790.17K	4.59M	2.62M
page-faults	D	508.00	65.19K	90.31K	67.32K	269.40K	7.60K	365.63K	123.22K
	M	17.89K	234.75K	302.52K	237.15K	379.81K	46.43K	550.19K	362.13K

TABLE IX. RQ3: VARYING GCC
OPTIMIZATION LEVEL

Method	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
O0	73702 ± 177	258 ± 2	420799 ± 672	73522 ± 99	14214 ± 21	220619 ± 85	2483 ± 8	589 ± 0
O1	8805 ± 7	37 ± 0	47053 ± 96	25500 ± 70	2758 ± 13	25439 ± 65	397 ± 0	54 ± 0
O2	8319 ± 9	37 ± 0	42809 ± 61	24952 ± 44	2565 ± 8	23755 ± 42	364 ± 2	48 ± 0
O3	8209 ± 28	36 ± 0	42897 ± 71	24799 ± 101	2567 ± 8	23169 ± 43	363 ± 1	47 ± 0

EpicMeds, LegacyPayors, Neighbors, and breast_cancer. The latter consists of adult, Iowa1KK, and EpicVitals. One can see that there is no single factor which can be used to predict the resulting performance improvement. Instead, one can note several performance-affecting factors:

1) The most important one is the number of instructions. Recall RQ2, where close examination revealed that the number of instructions directly affected the performance of Metanome. However, as we can see from this table, the number of instructions does not translate into improvement directly. For example, adult and Iowa1KK datasets have roughly the same ratio of instructions, but show different improvement. However, if this ratio is at least 5, then there is a noticeable speedup.
2) The next factors are L1/L2/L3 cache misses, both for data and instructions. Adult and Iowa1KK datasets are among the lowest L1 data cache miss ratio in the whole dataset collecion. At the same time we can see that adult has worse or the same cache miss ratio statistics over the whole cache hierarchy than Iowa1KK. This fact suggests that there is another factor that defines the performance.
3) Number of branch instructions and branch misses. First of all, one can note that branching instructions make up about 20% total, regardless of dataset and implementation. Therefore, similarly to the overall instruction number, the number of branching instructions is important as well.
   Turning to branch misses, one can say that they are much more important. The table shows that Desbordante improves by 19% on Iowa1KK in terms of the number of branch misses. Given the fact that other listed events have better or comparable statistics for the adult dataset, we conclude that the number of branch misses is also one of the defining factors.

Some final remarks:

1) CPU migrations and context switches. Metanome, being a Java program, is a multi-threaded application. Therefore, it is prone to CPU migrations which may happen due to a thread waking up, for example. These operations are much more costly than cache misses and therefore

they may create a performance bottleneck for Metanome. It will be an interesting future work direction to study its effects, for example, by attaching Metanome to a particular core. However, the outcome is unclear since such an approach will restrict garbage collection and the JIT compilation process.
2) Moving to C++ drastically reduces the number of TLB misses, sometimes by several orders of magnitude. However, adult and Iowa1KK show that they are less important that the reduction of branch misses.
3) The number of page faults is also reduced for Desbordante. However, its contribution is unclear on this step, since at the moment Desbordante is not optimized at all, and page faults are likely shadowed by cache and TLB misses.

We have also separately checked the garbage collection time. For all datasets, this overhead amounted for 0.5%–2% of the overall run time.

Finally, our guess was the impact of the C++ code optimizer. To check this, we have re-run our experiments with all four available compilation options that control code optimizer aggressiveness. The “-O0” is the default optimization level that g++ offers. It has the fastest compile time, and is used to generate an executable for the “debug” mode, thus not optimizing code at all. The next options “-O1”, “-O2”, and “-O3” extend compilation time and produce increasingly faster binaries. Table IX demonstrates that the default optimization level is almost ten times slower than any of the alternatives, and it also loses to Metanome. Increasing the optimization level leads to speed up of the executable. Switching from “-O1” to “-O2” adds 10% to performance on some datasets. However, increasing the level further does not add any improvement. It is also interesting to observe that performance improves over all datasets relatively uniformly.

D. RQ4: What are the reasons that stand behind Java runtime outliers? Is it possible to get rid of them? Can we guarantee stable run times?

During the early stages of this project, we have noticed that the confidence intervals in the preliminary versions of Figure 1 were abnormally large. Having looked into the data itself, we have found out that results sometimes vary up to 20 times.

Running our experiments, we have ensured a “clean” environment: freshly-booted systems, no extra processes run, system updates turned off, etc. This was done due to three reasons:

1) First of all, we need to ensure the fairness of comparisons. A sudden spike in load may impact the performance of the currently tested implementation and lead to unreliable conclusions.
2) Next, we need to make understanding implementation behavior easier by eliminating external factors to the maximum possible extent.
3) Finally, such a “clean” environment reflects the real use-case scenario, since the FD discovery task is high-performance. Therefore, it is reasonable to run it on a dedicated system, allocating maximum computational resources.

Therefore, interference of an outside process had nothing to do with such variability of the obtained results. An extensive study has revealed several other factors:

1) First of all, Pyro, as well as many other FD discovery algorithms, is RNG-sensitive. Such algorithms rely on random sampling which is used to control the discovery process. Therefore, if an algorithm samples an ill-fitting data fragment, it may lead to bad choices and the resulting performance may be low. Our experiments demonstrate that such choices have a significant on performance: it may differ up to ten times. Therefore, in order to run fair experiments, we had to ensure that random number sequences are the same. For this, we implemented custom random number generators in both Metanome and Desbordante.
2) The next issue that we noticed is the change in the CPU frequency during program runs. Having investigated the issue closer, we found out that modern CPUs adjust [39] their frequency depending on the load. Therefore, we had to address this issue via locking the CPU frequency. For this, we have used the `cpufreq-set` command for each core.
3) Finally, we have noticed that if a table has a large number of `NULL` values (see Table I) it also negatively impacts the stability of results. `NULL`s force Pyro to resample data.

Having eliminated the first two of these issues, we have obtained the current confidence intervals. The third issue, being a data quality problem, was left as is. At the same time, our experiments demonstrate that it seriously impacts the stability of the run times. For example, consider CIPublicHighway, which has 46% of `NULL`s (see Table I). For both implementations this leads to almost two orders of magnitude result variability.

E. RQ5: How is the performance impacted if maxLHS parameter is set?

Almost all dependency discovery algorithms can be parameterized by the maximum desired size of the left hand side (LHS) of dependency. In this case, the algorithm ceases lattice traversal early, thus saving time. It allows to mine a subset of all dependencies on slower systems.

In this experiment, we have studied the performance of Desbordante and Metanome in such scenarios. For this, we have set a memory limit of 1GB (using the `ulimit` command) and run a series of experiments while increasing the LHS parameter. Tables X and XI describe the results. First of all, we can see that our datasets can be classified into three groups:

1) Adult: run time grows with increasing LHS;
2) EpicVitals, LegacyPayors: run times rapidly grow, then stabilize;
3) CIPublicHighway, Iowa1KK: heavy datasets that run into the memory limit fast;
4) SG_Bioentry: this type has a “hump” at the start which then stabilizes.

TABLE X. RQ5: VARYING MAXLHS, DESBORDANTE, CPU TIME

maxLHS	adult	CIPublicHighway	EpicVitals	Iowa1KK	LegacyPayors	SG_Bioentry
1	119	762	1458	4265	417	61
2	740	5789	2554	19073	550	171
3	2587	ML	2270	29707	372	216
4	5694	ML	2672	28802	372	228
5	9876	ML	2631	25629	369	100
10	8368	ML	2671	23498	369	100
100	8331	ML	2676	23678	370	99

TABLE XI. RQ5: VARYING MAXLHS, METANOME, CPU TIME

maxLHS	adult	CIPublicHighway	EpicVitals	Iowa1KK	LegacyPayors	SG_Bioentry
1	194	737	1652	4841	617	203
2	923	6276	3363	21875	960	344
3	2300	ML	3051	36150	768	507
4	4990	ML	2979	ML	936	419
5	9243	ML	3078	31186	848	253
10	8541	ML	3023	28998	785	301
100	8257	ML	2939	28713	795	252

Such behavior is the property of the algorithm — its sampling and lattice traversal policy which depend on the exact table that it works with.

Regarding the performance of Desbordante and Metanome, one can make the following assumptions based on these tables:

- For most individual values of maxLHS, Desbordante offers either the same or better performance than Metanome.
- If the overall performance (see Table IV) is the same for both implementations, it continues to be the same even for individual values of maxLHS. This fact looks promising for understanding the reasons standing behind such ties.
- In some cases Desbordante can finish the task without hitting the memory limit (IOWA1KK, maxLHS=4).

V. CONCLUSION

In this paper we have presented Desbordante — the first tool intended specifically for high-performance dependency discovery. It is written completely in C++, extendable, and fully open-source.

Experimental evaluation has demonstrated that our tool provides the superior speed of dependency discovery (2x on average), lower memory requirements (almost 2x on average), and in general pushes the limits in terms of datasets that can be processed. At the same time, its tuning potential is not yet exhausted: we have not employed custom data structures and libraries, custom memory managers, special compilation options, and we have not tried SIMD-enabled algorithms.

ACKNOWLEDGMENT

We would like to thank Ilya Vologin for his help with the implementation. We would also like to thank Anna Smirnova for her help with the preparation of the article.

REFERENCES

[1] I. F. Ilyas and X. Chu, *Data Cleaning*. New York, NY, USA: Association for Computing Machinery, 2019.

[2] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock, *Data Profiling*. Morgan & Claypool Publishers, 2018.

[3] 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

[4] F. Dürsch, A. Stebner, F. Windheuser, M. Fischer, T. Friedrich, N. Strelow, T. Bleifuß, H. Harmouch, L. Jiang, T. Papenbrock, and F. Naumann, “Inclusion dependency discovery: An experimental evaluation of thirteen algorithms,” in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 219–228. [Online]. Available: https://doi.org/10.1145/3357384.3357916

[5] T. Bleifuß, S. Bülow, J. Frohnhofen, J. Risch, G. Wiese, S. Kruse, T. Papenbrock, and F. Naumann, “Approximate discovery of functional dependencies for large datasets,” in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1803–1812. [Online]. Available: https://doi.org/10.1145/2983323.2983781

[6] T. Papenbrock and F. Naumann, “A hybrid approach to functional dependency discovery,” in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 821–833. [Online]. Available: https://doi.org/10.1145/2882903.2915203

[7] P. Schirmer, T. Papenbrock, S. Kruse, F. Naumann, D. Hempfing, T. Mayer, and D. Neuschäfer-Rube, “DynFD: Functional dependency discovery in dynamic datasets,” in *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi, Eds. OpenProceedings.org, 2019, pp. 253–264. [Online]. Available: https://doi.org/10.5441/002/edbt.2019.23

[8] S. Kruse and F. Naumann, “Efficient discovery of approximate dependencies,” *Proc. VLDB Endow.*, vol. 11, no. 7, p. 759–772, Mar. 2018. [Online]. Available: https://doi.org/10.14778/3192965.3192968

[9] P. Schirmer, T. Papenbrock, I. Koumarelas, and F. Naumann, “Efficient discovery of matching dependencies,” *ACM Trans. Database Syst.*, vol. 45, no. 3, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3392778

[10] 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

[11] P. Charles, “Java Benchmark Harness,” 2013. [Online]. Available: https://github.com/openjdk/jmh

[12] N. Piotr, “JAVA and SIMD.” [Online]. Available: https://web.archive.org/web/20201112030104/https://prestodb.rocks/code/simd/

[13] “Vector API.” [Online]. Available: https://openjdk.java.net/jeps/338

[14] Oracle, “JDK 16,” 2021. [Online]. Available: https://openjdk.java.net/projects/jdk/16/

[15] Oracle, “JNI,” 2021. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[16] N. A. Halli, H.-P. Charles, and J.-F. Mehaut, “Performance comparison between Java and JNI for optimal implementation of computational micro-kernels,” in *Proceedings of the 5th International Workshop on Adaptive Self-tuning Computing Systems 2015*, ser. ADAPT’15, 2014.

[17] D. Kurzyniec and V. Sunderam, “Efficient cooperation between Java and native codes – JNI performance benchmark,” in *In The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[18] “Desbordante project,” 2021. [Online]. Available: https://github.com/Mstrutov/Desbordante

[19] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, “Tane: An efficient algorithm for discovering functional and approximate dependencies,” *The Computer Journal*, vol. 42, no. 2, pp. 100–111, 1999.

[20] P. A. Flach and I. Savnik, “Database dependency discovery: A machine learning approach,” *AI Commun.*, vol. 12, no. 3, p. 139–160, Aug. 1999.

[21] Z. Abedjan, P. Schulze, and F. Naumann, “DFD: Efficient functional dependency discovery,” in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, ser. CIKM ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 949–958. [Online]. Available: https://doi.org/10.1145/2661829.2661884

[22] H. Yao, H. J. Hamilton, and C. J. Butz, “FD_Mine: Discovering functional dependencies in a database using equivalences,” in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ser. ICDM ’02. USA: IEEE Computer Society, 2002, p. 729.

[23] S. Lopes, J.-M. Petit, and L. Lakhal, “Efficient discovery of functional dependencies and armstrong relations,” vol. 1777, 03 2000, pp. 350–364.

[24] C. Wyss, C. Giannella, and E. Robertson, “FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract,” 01 2001, pp. 101–110.

[25] L. Caruccio and S. Cirillo, “Incremental discovery of imprecise functional dependencies,” *J. Data and Information Quality*, vol. 12, no. 4, Oct. 2020. [Online]. Available: https://doi.org/10.1145/3397462

[26] L. Caruccio, S. Cirillo, V. Deufemia, and G. Polese, “Incremental discovery of functional dependencies with a bit-vector algorithm,” in *Proceedings of the 27th Italian Symposium on Advanced Database Systems, Castiglione della Pescaia (Grosseto), Italy, June 16-19, 2019*, ser. CEUR Workshop Proceedings, M. Mecella, G. Amato, and C. Gennaro, Eds., vol. 2400. CEUR-WS.org, 2019. [Online]. Available: http://ceur-ws.org/Vol-2400/paper-21.pdf

[27] P. Mandros, M. Boley, and J. Vreeken, “Discovering reliable approximate functional dependencies,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 355–363. [Online]. Available: https://doi.org/10.1145/3097983.3098062

[28] Y. Zhang, Z. Guo, and T. Rekatsinas, “A statistical perspective on discovering functional dependencies in noisy data,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 861–876. [Online]. Available: https://doi.org/10.1145/3318464.3389749

[29] S. Wu, S. Sanghavi, and A. G. Dimakis, “Sparse logistic regression learns all discrete pairwise graphical models,” *CoRR*, vol. abs/1810.11905, 2018. [Online]. Available: http://arxiv.org/abs/1810.11905

[30] “UC Irvine Machine Learning Repository,” 2021. [Online]. Available: http://archive.ics.uci.edu/ml/index.php

[31] “South Australian Government Data Directory,” 2021. [Online]. Available: https://data.sa.gov.au/

[32] “U.S. Environmental Protection Agency,” 2021. [Online]. Available: https://www.epa.gov/

[33] “Iowa Liquor Sales,” 2021. [Online]. Available: https://www.opendatanetwork.com/dataset/mydata.iowa.gov/m3tr-qhgy

[34] “Sloan Digital Sky Survey Data Release 13,” 2015. [Online]. Available: https://www.sdss.org/dr13/

[35] “BioSQL,” 2008. [Online]. Available: https://biosql.org/wiki/Downloads

[36] A. Birillo and N. Bobrov, “Smart caching for efficient functional dependency discovery,” in *New Trends in Databases and Information Systems*, T. Welzer, J. Eder, V. Podgorelec, R. Wrembel, M. Ivanović, J. Gamper, M. Morzy, T. Tzouramanis, J. Darmont, and A. Kamišalić Latifić, Eds. Cham: Springer International Publishing, 2019, pp. 52–59.

[37] “No Bugs” Hare, “Testing Memory Allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc While Trying to Simulate Real-World Loads,” 2018. [Online]. Available: http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/

[38] S. Oaks, *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*, 1st ed. Sebastopol, CA: O’Reilly Media, May 2014.

[39] “CPU frequency scaling,” 2021. [Online]. Available: https://wiki.archlinux.org/index.php/CPU_frequency_scaling