Order in Desbordante: Techniques for Efficient Implementation of Order Dependency Discovery Algorithms

Yakov Kuzin, Dmitriy Shcheka, Michael Polyntsov, Kirill Stupakov, Mikhail Firsov, George Chernishev
Saint-Petersburg University
Saint-Petersburg, Russia
{yakov.s.kuzin, dmitriy.v.shcheka, polyntsov.m, kirill.v.stupakov, mikhail.a.firsov, chernishev}@gmail.com

***Abstract*—Science-intensive data profiling focuses on discovery and validation of various patterns in datasets. This study considers discovery of one such pattern — order dependency (OD). Simply put, OD states that some list of columns is ordered according to another one. It is of use for database query optimization, data cleaning and deduplication, anomaly detection, and much more.**

Existing discovery methods have approached this problem solely from the algorithmic standpoint, without focusing on the implementation side. At the same time, this problem is very computationally intensive, and therefore this part should not be ignored, as it brings ODs closer to industrial use.

In this paper, we study two algorithms for OD discovery which target different OD axiomatizations — FASTOD and ORDER. We start by reimplementing these algorithms in C++ in order to speed them up and lower their memory consumption. We then analyze their bottlenecks and propose several techniques which improve their performance even further.

To perform evaluation, we have implemented these algorithms inside Desbordante — a science-intensive, high-performance, and open-source data profiling tool developed in C++. Experiments have demonstrated a performance improvement of up to 3x obtained by reimplemented versions, and, with the application of our techniques, up to 10x. Memory consumption has been lowered by up to 2.9x.

I. Introduction

Currently, there exists a considerable scholarly interest in the analysis of extensive datasets. These datasets often exhibit various inconsistencies, including missing values, duplicates, and many other anomalies [1].

Data profiling [2] is a research field that aims to detect and characterize such inconsistencies in order to prepare them for further use (e.g., data cleaning). Data profiling can be divided into two kinds [3]: naive and science-intensive. The former aims to extract simple dataset characteristics such as number of rows and columns, number of nulls, mean and variance, etc. The science-intensive kind concerns the extraction of complex patterns represented by structures which we will refer to as *primitives*. Examples of such patterns are database dependencies (functional [4], inclusion [5]), association rules [6], algebraic constraints [7], inferred semantic data types [8], and others. These patterns are typically discovered through employing various algorithms, which are usually very costly, resulting in dataset size being a significant limiting factor. Thus, the development of novel efficient algorithms and improving the performance of existing ones are relevant problems.

One such primitive is the order dependency (OD). Informally, an OD states that some column is ordered according to another column. For example, an increase in salary in the IT department payroll table might be directly correlated with increases in programmer's grade. According to reference [9], ODs prove effective in improving data quality, as their violation may serve as an indicator of underlying data errors. Furthermore, as discussed in [10], these dependencies can be leveraged by different database query optimizers to fine-tune query performance.

Desbordante (Spanish for *boundless*) [3] is a *science-intensive*, *high-performance*, and *open-source* data profiling tool implemented in C++. To the best of our knowledge, Desbordante is currently the only profiler that possesses these three qualities. It is capable of discovering and validating many primitives, including functional dependencies (both exact and approximate), conditional functional dependencies, metric functional dependencies, and others. The full list can be found on the web-site (https://github.com/Mstrutov/Desbordante/). However, Desbordante currently lacks support for ODs, which we aim to add.

ODs have been known since 80es [11], and therefore this subject contains a vast body of work. In this paper we focus on two recent types of OD, which are based on different axiomatizations — list-based [12] and set-based [9]. Different formalisms effectively lead to different primitives, each having its own algorithm and resulting primitive instances (result set). The list-based axiomatization offers the ORDER algorithm, while the set-based one — FASTOD.

However, existing discovery approaches have considered this problem from the algorithmic standpoint only, without focusing on the implementation side. To evaluate their algorithms, authors have developed research prototypes implemented in Java. Firstly, our previous studies [13] demonstrated that merely reimplementing these algorithms in C++ can improve their run times up to 3.5 times and lower memory consumption up to 2.5 times. Secondly, applying various code-level optimizations [14] can improve run times even further, up to 8 times. As the result, existing implementations are slower than they could be. Since primitive discovery problem is very

computationally intensive, the engineering part should not be ignored as it brings ODs closer to industrial use.

In this paper, we develop a technical approach to the problem based on efficient implementations of algorithms for OD discovery. We start by reimplementing these algorithms in C++ in order to speed them up and lower their memory consumption. Then we analyze their bottlenecks and propose several techniques which improve their performance even further.

Overall, the contribution of this paper is the following:

- A comprehensive study of two recent formalizations of OD and a description of the algorithms for their discovery — ORDER and FASTOD.
- Several novel techniques for efficient implementation of both algorithms.
- Open-source C++ implementations of both the algorithms and proposed techniques.
- Experimental evaluation of the proposed techniques and discussion of the results.

This paper is organized as follows. In Section II we provide definitions for both axiomatizations, after which we discuss them with examples. Next, in Section III we present related work concerning ODs. In Section IV we describe existing algorithms and our improvements. Their evaluation is discussed in Section V, and Section VI concludes the paper.

II. Background

Currently, there exist two axiomatizations describing the notion of order dependency. These axiomatizations define different objects, which results in different algorithms for their discovery. The first one treats left hand side and right hand side of a dependency as lists, and the second one — as sets of attributes. In order to understand the respective algorithms and our modifications, it is necessary to grasp the basics of these axiomatizations. Therefore, in this section we present essential concepts and formal definitions, closely following [15] and [9] while presenting descriptive examples.

A. Basic Definitions

Relations. $\mathbf{R}$ denotes a *relation (schema)* and $\mathbf{r}$ denotes a specific *relation instance (table)*. A, B and C denote single *attributes*, s and t denote *tuples*, and t_{A} denotes the value of an attribute A in a tuple t.

Sets. $\mathcal{X}$ and $\mathcal{Y}$ denote *sets* of attributes. Let $t_{\mathcal{X}}$ denote the *projection* of tuple t on $\mathcal{X}$. $\mathcal{XY}$ is a shorthand for $\mathcal{X} \cup \mathcal{Y}$. The empty set is denoted as $\{\}$.

Lists. $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ denote *lists* of attributes. Empty list is denoted as $[\,]$. $[\mathsf{A}, \mathsf{B}, \mathsf{C}]$ denotes an explicit list. $[\mathsf{A} \mid \mathbf{T}]$ denotes a list with *head* A and *tail* $\mathbf{T}$. Let $\mathbf{XY}$ be a concatenation of lists $\mathbf{X}$ and $\mathbf{Y}$. Set $\mathcal{X}$ denotes the set of elements in list $\mathbf{X}$. Any place a set is expected but a list appears, the list is cast to a set; e.g., $t_{\mathbf{X}}$ denotes $t_{\mathcal{X}}$. Let $\mathbf{X}'$ denote some other permutation of elements of list $\mathbf{X}$.

Definition 2.1: Given a relational schema $\mathbf{R}$ and an instance $\mathbf{r}$ over $\mathbf{R}$ with attribute sets $\mathcal{X}, \mathcal{Y} \subset \mathbf{R}$, we say that a functional dependency (FD) $\mathcal{X} \rightarrow \mathcal{Y}$ holds iff for any $s, t \in \mathbf{r}$, the following is true: $s_{\mathcal{X}} = t_{\mathcal{X}} \Rightarrow s_{\mathcal{Y}} = t_{\mathcal{Y}}$.

Example 2.1: For instance, in Table I functional dependency {*"Shipment cost"*} $\rightarrow$ {*"Weight"*} holds, since in all tuples with equal values of attribute *"Shipment cost"* (t_1 and t_5), the values of "Weight" are equal as well.

TABLE I. A table with shipment information, adapted from [12]

t_{id}	Weight	Distance	Shipment cost	Days
1	10	40	17	3
2	15	80	48	7
3	8	60	13	5
4	15	90	28	8
5	10	40	17	4
6	25	100	43	9
7	9	60	18	6

With FDs, however, it is impossible to capture relationships among ordered attributes, such as timestamps or numbers, which are quite common in business data. Therefore, the concept of OD is introduced, generalizing FD by allowing comparison operators other than $=$.

Definition 2.2: Let $\mathbf{X}$ be a list of attributes and $\theta \in \{\leq, <, >, \geq\}$. For two tuples r and s, $\mathcal{X} \in \mathbf{R}$ we say that $r_{\mathbf{X}}\, \theta\, s_{\mathbf{X}}$ if

1) $\mathbf{X} = [\,]$, *or*
2) $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}] \wedge r_{\mathsf{A}}\, \theta\, s_{\mathsf{A}}$, *or*
3) $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}] \wedge r_{\mathsf{A}} = s_{\mathsf{A}} \wedge r_{\mathbf{T}}\, \theta\, s_{\mathbf{T}}$.

Unless otherwise specified, numbers are ordered numerically, strings are ordered lexicographically and dates are ordered chronologically.

Definition 2.3: The order dependency $\mathbf{X} \mapsto_{\theta} \mathbf{Y}$, where $\theta \in \{\leq, <, >, \geq\}$, is present in the instance $\mathbf{r}$ over the relation $\mathbf{R}$ iff for any $s, t \in \mathbf{r}$, the condition $s_{\mathbf{X}}\, \theta\, t_{\mathbf{X}} \Rightarrow s_{\mathbf{Y}}\, \theta\, t_{\mathbf{Y}}$ holds. If θ is omitted, it is implied that θ is $<$.

Essentially, the presence of OD $\mathbf{X} \mapsto \mathbf{Y}$ means that, when ordering the values by $\mathbf{X}$, the resulting list would also be ordered by $\mathbf{Y}$.

Discovered order dependencies have many applications, such as database query optimization, data cleaning and deduplication, anomaly detection, and much more.

Example 2.2: In Table I we can see that the order dependency [*"Distance"*, *"Weight"*] $\mapsto_{\leq}$ [*"Days"*] holds. Ordering by *"Distance"* and breaking ties by *"Weight"* ($t_1 \leq t_5 \leq t_3 \leq t_7 \leq t_2 \leq t_4 \leq t_6$) is the same as ordering by *"Days"*. Also note that although FD {*"Weight"*} $\rightarrow$ {*"Shipment cost"*} holds, as seen in previous example, OD [*"Weight"*] $\mapsto_{\leq}$ [*"Shipment cost"*] does not hold. We discuss reasons for this in Section II-C.

B. List-based definitions

Definition 2.4: Two attribute lists $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible* with respect to $\theta \in \{\leq, <, >, \geq\}$, denoted as $\mathbf{X} \sim_{\theta} \mathbf{Y}$,

iff $\mathbf{XY} \leftrightarrow_\theta \mathbf{YX}$ ($\mathbf{XY} \mapsto_\theta \mathbf{YX}$ and $\mathbf{YX} \mapsto_\theta \mathbf{XY}$). $[]$ is order compatible with any attribute list. ODs in the form of $\mathbf{X} \sim_\theta \mathbf{Y}$ are called order compatible dependencies (OCDs)

Example 2.3: In Table I OCD [*"Distance"*] $\sim$ [*"Days"*] is valid: sorting by *"Distance"* and breaking ties by *"Days"* is equivalent to sorting by *"Days"* and breaking ties by *"Distance"*.

Definition 2.5: An attribute list $\mathbf{X}$ is *minimal*, iff for any disjoint, contiguous sub-lists $\mathbf{V}$ and $\mathbf{W}$ in $\mathbf{X}$, such that $\mathbf{W}$ precedes (not necessarily directly) $\mathbf{V}$, OD $\mathbf{V} \mapsto \mathbf{W}$ does not hold.

Definition 2.6: The order dependency $\mathbf{X} \mapsto \mathbf{Y}$ is *minimal*, iff

1) there is no prefix $\mathbf{V}$ of $\mathbf{X}$, such that $\mathbf{V} \mapsto \mathbf{Y}$ does not hold, *and*
2) there is no prefix $\mathbf{W}$ of $\mathbf{Y}$, such that $\mathbf{X} \mapsto \mathbf{W}$ holds, *and*
3) $\mathbf{X}$ is minimal, *and*
4) $\mathbf{Y}$ is minimal.

C. Violations

ODs can be violated in two ways. We begin with the following theorem and then explain how the two conditions therein correspond to two possible sources of violations. Detailed proofs can be found in the original paper [9].

Theorem 2.1: For every instance $\mathbf{r}$ of relation $\mathbf{R}$ and $\theta \in \{\leq, <, >, \geq\}$, $\mathbf{X} \mapsto_\theta \mathbf{Y} \iff \mathbf{X} \mapsto_\theta \mathbf{XY} \wedge \mathbf{X} \sim_\theta \mathbf{Y}$.

Definition 2.7: Tuples s and t form a *split* with respect to a pair of attribute lists $(\mathbf{X}, \mathbf{Y})$, if $s_\mathbf{X} = t_\mathbf{X}$, but $s_\mathbf{Y} \neq t_\mathbf{Y}$.

Definition 2.8: Tuples s and t form a *merge* with respect to a pair of attribute lists $(\mathbf{X}, \mathbf{Y})$, if $s_\mathbf{X} \neq t_\mathbf{X}$, but $s_\mathbf{Y} = t_\mathbf{Y}$.

A *split* among $(\mathbf{X}, \mathbf{Y})$ implies a *merge* among $(\mathbf{Y}, \mathbf{X})$ and vice versa. Presence of *split* or *merge* implies $\mathbf{X} \mapsto_\theta \mathbf{XY}$ being violated. Introducing both as separate concepts facilitates the distinction between the two types of order dependencies: *splits* invalidate only order dependencies under $\leq$ and $\geq$, and *merges* invalidate only order dependencies under $<$ and $>$.

Definition 2.9: Tuples s and t form a *swap* with respect to pair of attribute lists $(\mathbf{X}, \mathbf{Y})$ and $\theta \in \{\leq, <, >, \geq\}$, if $s_\mathbf{X}\, \theta\, t_\mathbf{X}$, but $\neg(s_\mathbf{Y}\, \theta\, t_\mathbf{Y})$. Presence of a *swap* implies $\mathbf{X} \sim_\theta \mathbf{Y}$ being violated.

Example 2.4: Order dependency in Table I [*"Weight"*] $\mapsto_\leq$ [*"Shipment cost"*] does not hold because of a split (t_2, t_4). OD [*"Days"*] $\mapsto$ [*"Shipment cost"*] does not hold because of a swap (t_1, t_3).

Theorem 2.2: $\mathbf{X} \mapsto_< \mathbf{Y}$ iff $\mathbf{X} \mapsto_\leq \mathbf{Y}$

This theorem unifies dependencies under operators $<$ and $\leq$. One of the algorithms discussed in this paper, ORDER [12], uses this fact by discovering only dependencies under strict comparison operators.

D. Set-based definitions

In [9] a polynomial mapping from list-based representation to a set-based canonical form of ODs is presented, allowing the traversal of a much smaller set-containment lattice instead of a list-containment lattice.

Definition 2.10: Let $\mathbf{R}$ be a relation schema, and $\mathbf{r}$ be its instance. The *equivalence class* of tuple $t \in \mathbf{r}$ with respect to a given set of attributes $\mathcal{X}$ is defined as the set $\varepsilon(t_\mathcal{X}) = \{s \in \mathbf{r} \mid s_\mathcal{X} = t_\mathcal{X}\}$.

Definition 2.11: An attribute A is considered *constant* within each equivalence class concerning the set of attributes $\mathcal{X}$ (denoted as $\mathcal{X} : [] \rightarrow_{cst} \mathsf{A}$) if there exists an order dependency $\mathbf{X}' \mapsto \mathbf{X}'\mathsf{A}$ for any permutation $\mathbf{X}'$ of elements in $\mathcal{X}$.

Definition 2.12: Two attributes A and B are *order compatible* within each equivalence class regarding the set of attributes $\mathcal{X}$ (denoted as $\mathcal{X} : \mathsf{A} \sim \mathsf{B}$) if there exists a permutation $\mathbf{X}'$, such that $\mathbf{X}'\mathsf{A} \mapsto \mathbf{X}'\mathsf{B}$.

Definition 2.13: Dependencies of the form $\mathcal{X} : [] \rightarrow_{cst} \mathsf{A}$ and $\mathcal{X} : \mathsf{A} \sim \mathsf{B}$ are referred to as *canonical* order dependencies. $\mathcal{X}$ is called the context.

In [9] theorems are presented that show a way of mapping list-based OD representations to equivalent set-based canonical forms of ODs. Given a set of attributes $\mathcal{X}$, for brevity, we state $\forall j, \mathsf{Y}_j$ to mean $\forall j \in \{1, 2, \ldots, |\mathcal{Y}|\}, \mathsf{Y}_j$.

Theorem 2.3: $\mathbf{X} \mapsto \mathbf{Y}$ iff $\forall i, \mathcal{X} : [] \rightarrow_{cst} \mathsf{Y}_i$ and $\forall i, j, \{\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}, \mathsf{Y}_1, \ldots, \mathsf{Y}_{j-1}\} : \mathsf{X}_i \sim \mathsf{Y}_j$

Example 2.5: By theorem 2.3, an OD $\mathsf{AB} \mapsto \mathsf{CD}$ can be mapped into the following equivalent set of canonical ODs:

1) $\{\mathsf{A}, \mathsf{B}\} : [] \rightarrow_{cst} \mathsf{C}$,
2) $\{\mathsf{A}, \mathsf{B}\} : [] \rightarrow_{cst} \mathsf{D}$,
3) $\{\} : \mathsf{A} \sim \mathsf{C}$,
4) $\{\mathsf{A}\} : \mathsf{B} \sim \mathsf{C}$,
5) $\{\mathsf{C}\} : \mathsf{A} \sim \mathsf{D}$,
6) $\{\mathsf{A}, \mathsf{C}\} : \mathsf{B} \sim \mathsf{D}$.

Definition 2.14: A canonical OD $\mathcal{X} : [] \rightarrow_{cst} \mathsf{A}$ is *trivial*, if $\mathsf{A} \in \mathcal{X}$. A canonical OD $\mathcal{X} : \mathsf{A} \sim \mathsf{B}$ is *trivial* if

1) $\mathsf{A} \in \mathcal{X}$, *or*
2) $\mathsf{B} \in \mathcal{X}$, *or*
3) $\mathsf{A} = \mathsf{B}$.

Definition 2.15: A canonical OD $\mathcal{X} : [] \rightarrow_{cst} \mathsf{A}$ is *minimal* if it is not *trivial* and there is no context $\mathcal{Y} \subset \mathcal{X}$, such that $\mathcal{Y} : [] \rightarrow_{cst} \mathsf{A}$ holds. A canonical OD $\mathcal{X} : \mathsf{A} \sim \mathsf{B}$ is *minimal* if it is not *trivial* and

1) there is no context $\mathcal{Y} \subset \mathcal{X}$, such that $\mathcal{Y} : \mathsf{A} \sim \mathsf{B}$ holds, *or*

2) $\mathcal{X} : [\,] \rightarrow_{cst} \mathsf{A}$, *or*
3) $\mathcal{X} : [\,] \rightarrow_{cst} \mathsf{B}$.

Violations for set-based axiomatization are defined similarly.

E. OD discovery algorithms

In this paper we consider the following OD discovery algorithms:

1) ORDER [12], which operates with *list-based* order dependencies,
2) FASTOD [9], which works with equivalent *set-based* mapping of *list-based* dependencies.

The set of dependencies produced by FASTOD is proven to be complete [9]. ORDER, however, uses excessively aggressive pruning rules, which leads to the resulting set of dependencies being incomplete in the following ways:

1) The algorithm skips dependencies in the form of $\mathbf{X} \mapsto \mathbf{XY}$,
2) if $\mathbf{X} \mapsto \mathbf{Y}$ is invalidated by a *swap*, then $\mathbf{XA} \mapsto \mathbf{YB}$ is not considered, leading to ODs in the form of $\mathbf{XA} \mapsto \mathbf{XAYB}$ being missed,
3) if $\mathbf{XA} \mapsto \mathbf{YB}$ is invalidated by a *split*, then $\mathbf{XA} \sim \mathbf{YB}$ is not considered, which maps to set-based canonical OD $\mathcal{XY} : \mathsf{A} \sim \mathsf{B}$.

FASTOD also has a more concise way of representing constants, producing only one *set-based* canonical OD $\{\} : [\,] \rightarrow_{cst} \mathsf{B}$ for each constant, while ORDER produces ODs $[\mathsf{A}] \mapsto [\mathsf{B}]$ for all attributes A and all constants B.

The fact that ORDER yields incomplete results, however, does not mean that it should not be used for dependency mining. ORDER's aggressive pruning rules allow it to perform better, especially on large datasets, while still being able to give out potentially useful dependencies.

III. Related Work

According to [15], order dependencies had first been proposed in [11]. Since then, a lot of dependency types differing from the initial ones had been discovered. Furthermore, alternative axiomatizations of initial concepts has been proposed: some were characterized by lists of attributes, while others — by sets. Pointwise Order Dependencies (PODs) and Lexicographical Order Dependencies (LODs) are examples of dependency classes with different axiomatizations. PODs — set-based dependencies — were presented by Seymour Ginsburg and Richard Hull [11] in the context of databases. LODs — list-based dependencies and a more useful alternative to PODs due to their applications in query optimization, were studied in [16]. Jaroslaw Szlichta et al. [15] presented the set of list-based inference rules which defines this class. Based on [15], inference problem was investigated both in theory and in practice [10], which led to the proof of its co-NP-completeness.

Various papers had proposed different mappings that connect those axiomatizations. A paper [10] presents an example of a mapping from LOD to PODs, which indicates that PODs generalize LODs. This generalization is strict, since LODs themselves don't in turn generalize PODs. Authors of [17] propose a polynomial mapping of list-based OD into an equivalent set-based canonical OD, which allows their algorithm to efficiently search for order dependencies. A paper [18] claims that PODs strictly generalize canonical ODs, which, combined with the previously mentioned mappings signifies that PODs represent a class of dependencies that is quite generic in nature. An even broader class of dependencies would be Denial Constraints (DCs), which could be found in [1], [19]. PODs are a subset of DCs (LODs, by extension, can also be classified as DCs, since they are a special case of PODs).

Order dependencies can also be generalized via Bidirectional Order Dependencies (BODs), which were analyzed in [17], [20]–[22]. They allow users to specify the order of sorting for both sides of an OD [10]. Papers [17] and [20] had delved into mining those dependencies, while works [22] and [21] had dealt with distributed search. This generalization is primarily useful due to emulating order-by clauses in SQL, which allows for an efficient optimization of such queries [20].

There have also been quite a number of papers researching less generalized dependencies. One example of such dependencies would be Functional Dependencies (FDs), explored in [4], [23]. According to [12], FDs are a special case of order dependencies. Furthermore, authors of the paper claim that their algorithm ORDER can be used to mine FDs, altough not very efficiently. Better performance can be achieved by algorithms such as FastFD [24] and Tane [25] due to them being designed specifically for mining FDs.

Order Compatibility Dependencies (OCDs), researched in [17], [26], are a more specific form of OD. This fact has been put to great use in the work [27] of Cristian Consonni et al. They used the idea of separating ODs into FDs and OCDs to propose a new approach to OD discovery. However, according to the paper [20], this pruning technique can lead to an incomplete set of dependencies being found.

To the best of out knowledge, there exist only two algorithms of exact order dependency discovery. Philipp Langer and Felix Naumann proposed the algorithm ORDER [12], which uses list-based inference rules to traverse list-containment lattice with worst-case time complexity of $O(|\mathbf{R}|!)$. Jaroslaw Szlichta et al. [9] presented algorithm FASTOD and the set-based inference rules, allowing for faster traversal of set-containment lattice, rather than list-containment one. FASTOD, an improvement of ORDER, is based on a polynomial mapping to a canonical forms of ODs with worst-case time complexity of $O(2^{|\mathbf{R}|})$. They prove the completeness of their approach and provide scenarios in which ORDER yields incomplete results. Cristian Consonni et al. [27] have made an attempt to improve the existing theory related to set-based ODs, proposing a new algorithm called OCDDISCOVER and showing that it has a significant speedup over FASTOD. Their claims have later been proven to be incorrect [28].

The two major algorithms (FASTOD and ORDER) al-

low its users to find exact order dependencies adhering to their modern definitions. Implementing these algorithms in an efficient manner and adding a new primitive would allow Desbordante to add yet another concept to its toolkit. On top of that, these algorithms rely on two different axiomatizations, which inspired us to research the differences arising from the distinction in their definitions.

IV. Algorithms

A. The General Scheme of Both Algorithms

FASTOD and ORDER are both lattice-based algorithms for dependency discovery. They initiate the search with either an attribute set or a list (depending on axiomatization) consisting of a single element, and progressively move to larger sets or lists through the lattice, traversing levels one by one. Dependency candidates obtained at a given level are checked for minimality based on the previous levels. Dependencies that pass that check — as well as an additional candidate verification check — are added to the resulting set of dependencies.

Thus, both FASTOD and ORDER employ a dependency search strategy from small to large. This approach enables the identification of minimal dependencies and efficiently reduces the search space. Partitions may be utilized for a more efficient dependency detection, enabling candidate verification checks in linear time.

Throughout the algorithms' execution, the following stages are repeated, as detailed in the article [9]: dependency search, pruning of the current level, and computation of the next level. This process can be described by the following blocks:

1) The first level includes all attributes from the original relation. In the initial iteration, this is the current level.
2) While the current level is not empty, steps 3-5 are executed.
3) All dependencies on the current level are identified.
4) The search space for dependencies is reduced by pruning the current level.
5) The next level is computed and becomes the current one.

In the following sections, we will describe some of the implementation details that are important for our proposed optimizations. We will also describe details that give a better understanding of the algorithms' approaches to dependency search, with emphasis on pruning in particular.

B. ORDER description

Algorithm ORDER discovers all *minimal n-ary lexicographical* order dependencies under the operator "$<$" (and by Theorem 2.2, all dependencies under "$\leq$").

First, the algorithm determines the columns that can be sorted. These columns are sorted so that sorted partitions can be created according to them, which the algorithm will then work with, without having to access the source data anymore.

Let's say we sorted some attribute A. *SortedPartition*(A) would then contain equivalence classes that retain the information regarding their indexes prior to sorting. If the values are equal, then their indexes would wind up in the same equivalence class. Sorted partitions are used during validation and allow it to be performed in linear time. It is possible to get sorted partitions for any list of attributes by calculating several products of sorted partitions for single attributes. Sorted partition production is a hash-join-like procedure, which you can learn more about in [12].

Next, the algorithm works with a lattice. All lists (permutations) of attributes of length i can be found at the i-th level of the lattice. Those permutations supply the algorithm with various candidates, dividing these lists into right and left parts. The algorithm starts from the first level and makes its way down the lattice, increasing value of i. Since the algorithm considers lists instead of sets of attributes, there can be a lot of candidates, so an aggressive candidate pruning is applied.

The pruning rules allow the algorithm to immediately establish the validity of a candidate, depending on the candidates that had already been verified. The rules are as follows:

1) $\mathbf{X} \not\mapsto_< \mathbf{Y} \Rightarrow \mathbf{XV} \not\mapsto_< \mathbf{Y}$;
2) $\mathbf{X} \mapsto_< \mathbf{Y}$ valid $\Rightarrow \mathbf{X} \mapsto_< \mathbf{YW}$ valid;
3) $\mathbf{X} \not\mapsto_< \mathbf{Y} \Rightarrow \mathbf{XV} \not\mapsto_< \mathbf{YW}$, where $\mathbf{X} \not\mapsto_< \mathbf{Y}$ is an invalidation by *swap*.
4) $\mathbf{X} \mapsto_< \mathbf{Y}$ is valid $\Rightarrow \mathbf{XV} \mapsto_< \mathbf{YW}$ is valid, if $\mathbf{X}$ contains only unique values.

The attribute lists $\mathbf{X}, \mathbf{Y}, \mathbf{V}, \mathbf{W}$ do not overlap, and only $\mathbf{V}, \mathbf{W}$ can be empty.

C. FASTOD description

FASTOD is an algorithm for efficient discovery of complete and minimal set of set-based canonical ODs. While ORDER traverses a lattice of all lists of attributes, FASTOD traverses a lattice of all sets of attributes. The idea of the algorithm is to utilize polynomial mapping of order dependencies to canonical forms, which allows it to achieve greater performance: its worst-case time complexity is $O(2^{|\mathbf{R}|})$.

Instead of regular partitions, FASTOD uses StrippedPartitions, in which equivalence classes with cardinality of 1 are excluded. This sort of compression allows for additional efficiency, but does not interfere with correctness of the algorithm. After calculating the partitions for individual attributes, the FASTOD evaluates partitions for subsequent levels, consisting of several attributes, in linear time using the product of partitions. So, partitions are not calculated from scratch, but are instead derived from previous levels: $\Pi_{\mathsf{A}\cup\mathsf{B}} = \Pi_{\mathsf{A}} * \Pi_{\mathsf{B}}$. This dramatically improves the performance of the algorithm.

FASTOD also employs a specific method for storing candidates. They are stored in $C_c^+(\mathcal{X}) = \{\mathsf{A} \in \mathbf{R} : \forall \mathsf{B} \in \mathcal{X}\,\mathcal{X}\backslash\{\mathsf{A},\mathsf{B}\} : [\,] \rightarrow_{cst} \mathsf{B}$ does not hold$\}$ and $C_s^+(\mathcal{X}) = \{\{\mathsf{A},\mathsf{B}\} \in \mathcal{X}^2 : \mathsf{A} \neq \mathsf{B}$ and $\forall \mathsf{C} \in \mathcal{X}\,\mathcal{X}\backslash\{\mathsf{A},\mathsf{B},\mathsf{C}\} : \mathsf{A} \sim \mathsf{B}$ does not hold, and $\forall \mathsf{C} \in \mathcal{X}\,\mathcal{X}\backslash\{\mathsf{A},\mathsf{B},\mathsf{C}\} : [\,] \rightarrow_{cst} \mathsf{C}$ does not hold $\}$, where $\mathbf{R}$ denotes the original relation. This approach prevents candidate sets from growing in size during algorithm's execution. Another benefit of using this representation lies in simplicity of pruning: a set of attributes X is deleted from a level (for all levels above 1) if both sets $C_c^+(\mathcal{X})$ and $C_s^+(\mathcal{X})$ are empty.

D. ORDER and FASTOD baselines

Both baselines are established by reimplementing the corresponding Java algorithm in C++. These adaptations involve minimal changes, addressing the absence of certain Java language features and specific data structures. Furthermore, both C++ implementations incorporate all data types supported by Desbordante, whereas the original FASTOD algorithm was tailored exclusively to integer columns. The C++ implementations abstain from utilizing specialized third-party libraries, such as libraries designed for specific memory management (allocators). Instead, they employ data structures from the standard library and incorporate Boost (https://www.boost.org).

E. FASTOD optimizations

Internally, the algorithm employs a specialized data structure called stripped partition (further referred as "partition"), which stores information about the partitioning of the dataset into equivalence classes. The FASTOD algorithm relies on numerous computationally intensive operations involving these partitions (represented by StrippedPartition class in the code).

Through the analysis of real datasets, it has been observed that a considerable portion of them contains columns predominantly composed of blocks of identical values. Some datasets are mainly comprised of such columns. Consequently, a decision was made to optimize the internal representation of partitions.

The standard approach involves storing the indices of all values within each equivalence class. However, this approach proves to be wasteful when encountering a large number of consecutive values falling into the same equivalence class. This results in excessive memory usage and the need for repeated copying of a substantial amount of data. For instance, consider the attribute $I = (1, \ldots, 1, 3, 1, \ldots, 1)$ and the equivalence class representation for the value of 1. In this case, it would appear as follows: $[1] = (0, 1, \ldots, N, N + 2, N + 3, \ldots, M)$, where N is the index of the 1 located before 3, and M is the index of the last 1.

The first optimization involves storing data not as a list of values, but as a list of ranges instead. Revisiting the same attribute I, the equivalence class representation for 1 would now be: $[1] = (0\text{–}N, (N + 2)\text{–}M)$. In this case, it is evident that memory is used much more efficiently, and less data needs to be copied. We call such approach a **range-based partition representation** (represented by RangeBasedStrippedPartition class in the code).

However, such representation would be inefficient for attributes that do not have a sufficiently large sequences of identical values. In such cases, many small or even degenerate ranges would be observed, not only occupying a significant amount of memory, but also slowing down partition operations.

The second optimization addresses this issue. It utilizes knowledge accumulated during dataset preprocessing. At this stage, values in each column are analyzed. If the proportion of values forming ranges is greater than a constant of 0.001, the corresponding attribute is flagged. The optimization involves mindful selection of the representation for the initial partition. If it is constructed based on an attribute flagged as having a sufficiently large proportion of range-forming values, the range-based partition representation is selected. Otherwise, the algorithm uses the standard partition representation. This achieves increased performance on attributes of a specific type without sacrificing performance on other attributes.

Furthermore, it has been observed that the size of value ranges does not increase as a result of partition operations; it usually decreases gradually down to a degenerate range containing a single value. Therefore, starting from a certain point, the representation based on ranges begins to slow down partition operations and expend unnecessary memory. The third optimization addresses this issue by dynamically switching the representation from range-based to the standard version. During partition operations, the percentage of small ranges (those with a size less than 40) relative to all ranges in its new state is calculated. If this ratio exceeds a constant of 0.5, the partition representation is switched to a standard one. This achieves the following effect: speed up during the initial stages when dealing with a large portion of range-forming values, and, when the ranges exhaust themselves (mostly turning into small ones) and start slowing down partition operations, the representation is switched to the standard one, retaining the performance characteristics of a baseline approach.

The new representation of partitions allows for performance gains. In particular, this is achieved through a special algorithm for computing range-based partition product. It uses the idea of fast range intersections, which is the basis of partition representation.

For each attribute, a list of "value-range" pairs is built, where range means the range of indices in the attribute followed by the corresponding value. By sorting this list by the second element of the pair, we obtain a new list, which we will call SI. If we sequentially expand the ranges of each of its pairs, an ordered sequence of indices spanning from zero to the number of table rows minus one is formed. Let's call this sequence SEQ. Next, for each such list we create a correspondence table T for each element a in SEQ with an index in SI. This index points to a pair whose range contains a. This table, together with the ordering condition of the SI list, will allow the algorithm to intersect an arbitrary range r with a list SI of the corresponding attribute in a short time. Instead of sequentially intersecting r with each range in SI (as would be the case if we represented the attribute as an arbitrary list of pairs), we first find two indices in constant time (these indices point to the ranges that contain the beginning and the end of r). Then we intersect r with ranges whose indices lie between the two found indices (including the ends). In this case, the ranges obtained as a result of intersection are matched to the same values that correspond to the ranges in SI. Combining the resulting "value-range" pairs will give us a list that will be the desired result of the intersection of the range r with SI.

It is also possible to intersect a list of ranges with SI. To do this, you need to intersect each range of this list with SI

and combine the results.

Our partition representation involves storing a list of ranges for each equivalence class. This way it can be easily mapped to a list of "value-range" pairs by adding a corresponding value for each range. The product of an existing partition with another one, built by some attribute A, can be calculated by intersecting this list with a list built by A, according to the considerations described earlier.

To better understand the described principle, consider the following example. Let us have an attribute $\mathsf{A} = (5, 5, 6, 6, 5, 5, 8)$. The list of "value-range" pairs for it will look like $I = \{(5, [0–1]); (5, [4–5]); (6, [2–3]); (8, [6–6])\}$. Sorting it by the second element of the pair (that is, by ranges) gives us $SI_{\mathsf{A}} = \{(5, [0–1]); (6, [2–3]); (5, [4–5]); (8, [6–6])\}$. Next, we create a correspondence table: $T_{\mathsf{A}} = \{(0 \to 0); (1 \to 0); (2 \to 1); (3 \to 1); (4 \to 2); (5 \to 2); (6 \to 3)\}$. Suppose that we have a partition $\Pi_{\mathcal{X}}$ and it's range-based representation $\{[0–1], [5–6], [2–4]\}$, where ranges $[0–1], [5–6]$ form the first equivalence class C_1, and range $[2–4]$ forms the second one, C_2. Lets say we want to calculate $\Pi_{\mathcal{X}\cup\mathsf{A}} = \Pi_{\mathcal{X}} * \Pi_{\mathsf{A}}$, where Π_{A} is a partition built for the attribute A. To do this, we need to intersect each equivalence class from $\Pi_{\mathcal{X}}$ with the list formed by the attribute A, that is, with SI_{A}. As an example, consider the intersection of the second equivalence class containing only a single range $d = [2–4]$ with SI_{A}. We calculate the indices of the ranges containing the beginning and the end of d: $T_{\mathsf{A}}[2] = 1$, $T_{\mathsf{A}}[4] = 2$. In the list SI_{A}, there are two ranges between indices 1 and 2: $[2–3]$ and $[4–5]$. We intersect them with d, which gives us $[2–4] \cap [2–3] = [2–3]$, $[2–4] \cap [4–5] = [4–4]$. In this case, the resulting ranges matched to the same values that corresponded to the ranges in SI_{A}. So $M = C_2 \cap SI_{\mathsf{A}} = \{(6, [2–3]); (5, [4–4])\}$. If the equivalence class consisted of several ranges, we would intersect each of them with SI_{A}, and then combine the results into a single list. Now we sort M by the first element of the pair and extract equivalence classes from it. This gives us two equivalence classes, the representation of which using indexes is as follows: $[4–4]$ and $[2–3]$. We exclude the equivalence classes with cardinality of 1, so only $[2–3]$ is added to the resulting list. After performing similar operations with each equivalence class from $\Pi_{\mathcal{X}}$, we obtain the final list of equivalence classes, which will be the result of the product of partitions.

A description of the algorithm in a more general form is presented in the Algorithm 1.

Data: T_{A} — correspondence table for attribute A, $\Pi_{\mathcal{X}}$ — first partition, SI_{Y} — sorted value-range representation of second partition Π_{A} of A
Result: O — partition $\Pi_{\mathcal{X}\cup\mathsf{A}}$

```
O ← ∅;
for C ∈ Π_X do
    M ← ∅;
    for [d_s; d_e] ∈ C do
        s ← T_A[d_s];
        e ← T_A[d_e];
        R ← SI_A[s ... e];
        for (v, [r_s; r_e]) ∈ R do
            r ← [d_s; d_e] ∩ [r_s; r_e];
            add (v, r) to M;
        end
    end
    sort M;
    E ← extract equivalence classes from M;
    exclude degenerate classes from E;
    add each element from E to O
end
return O;
```

Algorithm 1: Range-based partition product

F. ORDER optimizations

Unlike FASTOD, optimizations for ORDER are based on using more efficient data structures and sorting algorithms from the Boost library where it is most needed.

After studying the performance of the algorithm, we concluded that in the vast majority of cases, the following operations take the most time: sorting attribute values when creating sorted partitions, searching for elements of equivalence classes during validity checks, and calculating the product of sorted partitions.

Sorting. Sorting of values occurs at the start of the algorithm in order to obtain efficient data representation — sorted partitions. As a result, the source data is not used on the next steps of the algorithm.

This step is the primary target for optimization, since it will result in performance gains on all datasets, unlike calculation of the product of sorted partitions, which may not occur due to dependencies not necessarily being found.

In addition to the sorts offered in the C++ standard library, we have the opportunity to use Boost.Sort library, which offers a set of different sorts, both parallel and serial. For single-threaded execution, we select flat_stable_sort because it offers decent performance and low memory consumption. block_indirect_sort was chosen for multi-threaded execution, for the same reasons. Replacing the sorting algorithm from the standard C++ library can bring improvements in performance and memory consumption.

Candidate validation. Validation is performed using a pair of sorted partitions. Algorithm goes through pairs of equivalence classes, in which identical elements are searched. The most efficient structure for a large number of searches is unordered_set, which has several implementations. In addition to the implementation from the C++ standard library, Boost offers its own implementation: unordered_flat_set, which has high performance as its main characteristic. Using unordered_flat_set can theoretically increase the performance of the algorithm.

Calculation of Sorted Partitions Product. Partition product is calculated for datasets that have dependencies, and the

more dependencies there are in the dataset, the more often that product occurs. Product is calculated using hash maps, so using a more efficient unordered_flat_map from the Boost instead of unordered_map from the standard C++ library can bring an increase in speed to datasets that contain a large number of dependencies.

V. Experiments and Discussion

A. General

To evaluate our techniques, we have developed our own implementations of both algorithms — FASTOD and ORDER — and experimentally compared them with the existing implementations written in Java. We present research questions and report experimental results for each algorithm in the next sections. Note that we do not compare FASTOD and ORDER with each other, as it is meaningless since they are designed to yield different results.

In our experiments, we have only considered run time of the algorithm itself, as parsing and preprocessing differ in Desbordante and Java implementations, and therefore could skew the final results. We leave the parsing and data preprocessing stages for future investigations.

Each discussed experiment was repeated three times, and the average of the results was calculated.

Experimental Setup. Experiments were performed using the following hardware and software configuration. Hardware: Intel® Core™ i7-11800H CPU @ 2.30GHz (8 cores), 16GB DDR4 3200MHz RAM, 512GB SSD SAMSUNG MZVL2512HCJQ-00BL2. Software: Kubuntu 23.10, Kernel 6.5.0-14-generic (64-bit), gcc 13.2.0, openjdk 11.0.21 2023-10-17, OpenJDK Runtime Environment (build 11.0.21+9-post-Ubuntu-0ubuntu123.10), OpenJDK 64-Bit Server VM (build 11.0.21+9-post-Ubuntu-0ubuntu123.10, mixed mode, sharing).

B. FASTOD

Methodology. The Java implementation of the FASTOD algorithm (https://github.com/leveretconey/cocoa/tree/master/src/main/java/leveretconey/fastod), unlike its C++ counterpart, is limited to datasets composed solely of integer values. Consequently, the original datasets were transformed to adhere to the specified format before running experiments. The resulting datasets can be found in the corresponding repository (https://github.com/Sched71/Desbordante-OD-Data).

For FASTOD we pose the following research questions:

RQ1 Is it possible to outperform existing implementation by simply reimplementing OD discovery algorithm in C++?

RQ2 What improvement does the proposed range-based partition representation offer on datasets with columns containing ranges?

RQ3 What is the overhead of the proposed range-based partition representation? It is true that using this representation does not compromise algorithm performance on regular datasets?

RQ4 How well does the performance of the C++ implementations scale with the increase in the number of columns?

RQ5 What are the memory savings of both C++ implementations?

To answer these questions, we have designed the following experiments for each RQ:

1) In the first experiment, we are comparing the vanilla C++ implementation of the FASTOD algorithm with its counterpart written in Java.
2) In the second experiment, we are comparing the baseline C++ implementation with the one containing the proposed technique — the range-based partition representation.
3) In the third experiment, we are comparing the same approaches as in the second experiment, but on datasets containing little range data. Our goal is to demonstrate that this optimization does not compromise performance on typical datasets (i.e., those that contain little range data).
4) In the fourth experiment, we study how well the C++ implementations scale with the number of columns in the dataset.
5) The fifth experiment evaluates memory savings for both C++ implementations compared to the Java implementation.

Evaluation. To conduct experiments, we used the datasets shown in Table IV. It includes a description of the datasets, as well as their short names, which we will use for brevity. The overall results are shown in Table V. We grayed out rows with special datasets containing columns with a large number of ranges.

Experiment 1. In this experiment, we compare our baseline implementation of the FASTOD algorithm with the Java implementation. The results of the conducted experiments are presented in Table V. The experiments unequivocally demonstrated that the C++ implementation of the algorithm exhibits higher performance compared to the Java implementation. We outperform it by a factor of up to 8, with 4 being the average.

Experiment 2. Our second experiment demonstrates possible performance increase on special datasets that contain numerous repeated values in the columns. The results are presented in Table V, where we have highlighted in gray the rows with special datasets. The experiments demonstrated that the new partition representation can contribute to achieving a speedup of 1.3x–1.8x.

Experiment 3. This experiment is similar to the first one, except the comparison is now conducted between the baseline and optimized versions of the C++ Desbordante implementation of the algorithm. The results of the conducted experiments are presented in Table V. As evident from the results, not only did the algorithm's runtime not increase on typical datasets after the application of optimizations, but it even showed a slight decrease. Thus, it can be concluded that our optimizations not only do not deteriorate the original implementation, but also enable significant improvements in processing time on datasets of a specific nature.

Experiment 4. In this experiment, we studied the dependence of the algorithm's running time on the number of

columns in the corresponding dataset. We used typical dataset A and special dataset D3 containing many columns with ranges as our initial datasets. We excluded a certain number of columns from each dataset, starting from the beginning of the dataset, thereby obtaining a dataset with the required number of columns.

The results of this experiment are demonstrated in Figures 1 and 2. They both show a clear superiority of our implementations compared to the Java version, as well as the speedup due to a special representation of partitions, which is clearly visible in Figure 1.

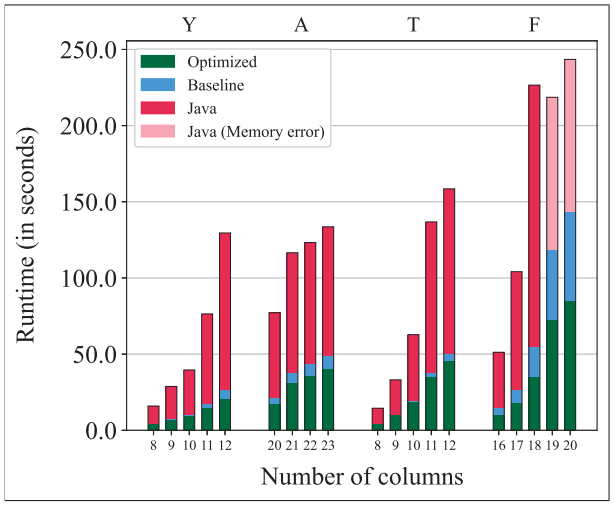

Fig. 1. Scalability in number of columns, part 1 (FASTOD)

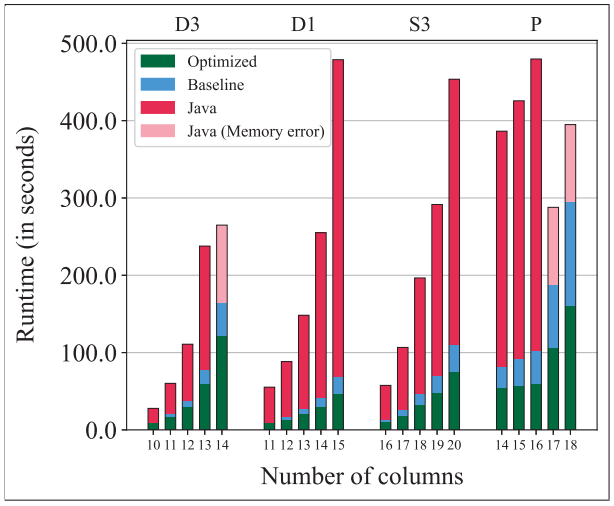

Fig. 2. Scalability in number of columns, part 2 (FASTOD)

Experiment 5. Our last experiment shows memory usage of all FASTOD implementations. The testing involved datasets G, A, S1, S2 and its results are presented in Figure 3.

We can observe a significant reduction in RAM consumption by both of our implementations of the algorithm. Baseline

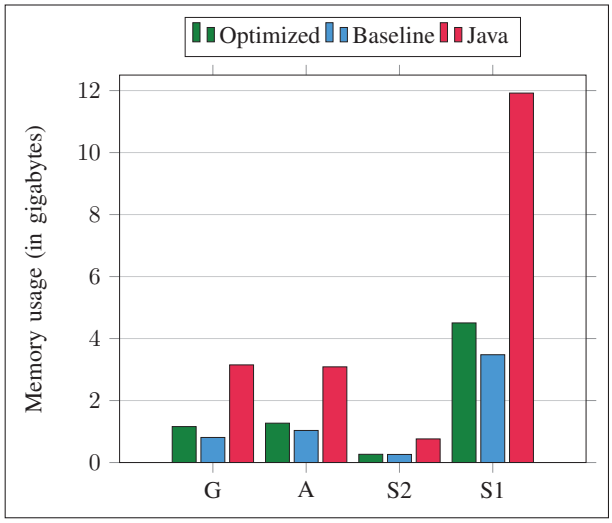

Fig. 3. Memory usage (FASTOD)

C++ implementation outperforms Java implementations by 2.9–3.9 times and the optimized implementation outperforms it by 2.4–2.9 times. It is also noticeable that the optimized implementation consumes more memory than the baseline one. This is a necessary price to pay for reducing execution time.

In addition, we found that the Java implementation consumes too much memory. For example, its execution on D3 — which includes 14 columns — ends with a memory error: there is not enough RAM on the computer on which the test was executed. This is demonstrated in Figure 1, where semi-transparent bar represents the memory error. The same situation is observed on some other datasets. We reflected this in the Table V using the same notation. At the same time, our implementations successfully tackle all these tasks.

C. ORDER

Methodology. For the ORDER algorithm, we pose the following research questions:

RQ1 Is it possible to outperform existing implementation by simply re-implementing OD discovery algorithm in C++?

RQ2 Which of the proposed optimizations can improve performance, and will their simultaneous application be effective?

RQ3 Can an optimized C++ implementation provide memory savings?

To answer these questions, we have designed the following experiments for each RQ:

1) In the first experiment, we are comparing the vanilla C++ implementation of the ORDER algorithm with its counterpart written in Java.
2) In the second experiment, we consider the optimization approaches both individually and simultaneously. We also make a comparison with the base version of the implementation.

3) In the third experiment, we are comparing memory usage of the optimized C++ implementation of the ORDER algorithm and implementation from Metanome.

Evaluation. To conduct experiments, we used the datasets shown in Table VII. It includes a description of the datasets, as well as their short names, which we will use for brevity. The overall results are shown in Table VIII. We had to select other datasets since ORDER algorithm is much faster than FASTOD. This results in sub second run times, which is not suitable for experiments, as the overall run time is a subject to measurement errors. Note that this does not mean that FASTOD is useless — ORDER misses some of the dependencies due to excessive pruning. On the other hand, ORDER can be useful for quick profiling aimed at obtaining a rough picture rather than striving for completeness of the set of discovered dependencies.

Experiment 1. In this experiment, we compared the base version of the C++ implementation with the Java implementation in Metanome.

The results of the conducted experiments are presented in Table VIII in columns Java, Base and Impr (base).

Experiments have shown that the base implementation in C++ is superior to the Java implementation in most cases. We outperform it by a factor of up to 9, with 4 being the average.

Due to the presence of datasets on which Java has higher performance, it became clear that additional optimizations were necessary.

Experiment 2. In this experiment, we compared the base C++ implementation with implementations where the proposed optimizations were applied. Optimizations were applied both individually and simultaneously.

Overall, we have compared boost::unordered_flat_map (flat_map), boost::unordered_flat_set (flat_set), and boost::block_indirect_sort (sort) to their standard counterparts — std::unordered_map, std::unordered_set, and std::sort. The resulting ratio is presented in Table II. The last column contains the results of three optimizations combined.

TABLE II. C++ OPTIMIZED IMPROVEMENTS IN COMPARISON TO BASE (ORDER)

Dataset	flat_map	flat_set	sort	Final
Diabetes	1.017x	4.954x	0.969x	6.253x
Pfw	1.019x	1.850x	1.015x	2.258x
Ditag	1.026x	1.267x	1.546x	2.189x
Credit	1.003x	1.128x	1.421x	1.727x
Epic	1.023x	1.480x	1.268x	2.128x
Modis	1.025x	1.592x	1.341x	2.550x
Bay	1.026x	1.360x	1.503x	2.785x

According to the results of the experiments, it can be concluded that the use of unordered_flat_map did not bring significant improvement, unlike the use of sort from Boost and the use of unordered_flat_set. In addition, applying all optimizations at the same time gives even bigger performance boost than the product of the performance increases obtained by testing individual optimizations.

Experiment 3. In the third experiment, we compared memory usage of the optimized C++ implementation with the Metanome version. The results of the conducted experiments are presented in Table III.

TABLE III. METANOME VS DESBORDANTE MEMORY CONSUMPTION (ORDER)

Dataset	C++ (MB)	Java (MB)	Improvement
Diabetes	177.94	429.37	2.413x
Pfw	150.69	265.62	1.762x
Ditag	3715.66	5951.69	1.601x
Credit	5607.87	7333.7	1.307x
Epic	662.78	1309.7	1.976x
Modis	2614.75	6240.43	2.386x
Bay	5690.77	7345.08	1.290x

Experiments have shown that our implementation uses less memory compared to the implementation from Metanome. To be specific, we achieved memory savings of 1.3–2.4x, depending on the dataset.

VI. CONCLUSION

In this paper, we have presented optimization techniques for two state-of-the-art OD discovery algorithms (ORDER and FASTOD), which allow us to add a new important primitive to Desbordante. Our experiments show a significant increase in algorithm performance (up to 10x), as well as a decrease in memory consumption (up to 2.9x). Described optimizations and the new representation of partitions can help optimize any other algorithms with a similar structure of components, which once again signifies the importance of our research.

Both C++ implementations will be of use for Desbordante's end-users, despite aiming at the same problem, namely discovery of ODs. ORDER is significantly faster than FASTOD, but misses some of the dependencies due to excessive pruning. ORDER, on the other hand, can be useful for quick profiling aimed at obtaining a rough picture rather than striving for completeness of the set of discovered dependencies. Finally, both implementations — ORDER and FASTOD — are open-source (https://github.com/Mstrutov/Desbordante/) and are merged (PRs 294, 355) into the Desbordante.

ACKNOWLEDGMENT

We would like to thank Vladislav Makeev for his help with the preparation of the paper.

REFERENCES

[1] X. Chu, I. F. Ilyas, and P. Papotti, "Holistic data cleaning: Putting violations into context," in *ICDE'13*, C. S. Jensen *et al.*, Eds. IEEE Computer Society, 2013, pp. 458–469.

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

[3] G. Chernishev *et al.*, "Desbordante: from benchmarking suite to high-performance science-intensive data profiler," *CoRR*, vol. abs/2301.05965, 2023.

[4] T. Papenbrock *et al.*, "Functional dependency discovery: an experimental evaluation of seven algorithms," *Proc. VLDB Endow.*, vol. 8, no. 10, p. 1082–1093, jun 2015.

[5] F. Dürsch *et al.*, "Inclusion dependency discovery: An experimental evaluation of thirteen algorithms," in *CIKM'19*, 2019, p. 219–228.

TABLE IV. DATASET DESCRIPTION
(FASTOD)

Dataset	Short name	Columns	Rows	Size (MB)	#OD	#FD	#OCD
graduation_dataset_norm_15c.csv	G	15	4424	0.14	333	1	332
Anonymize_norm.csv	A	23	38480	4.28	115400	7774	107626
PFW_2021_public_norm.csv	P	18	100000	7.51	1240	48	1192
Spotify_Dataset_V3_norm.csv	S1	11	651936	35.96	1645	171	1474
diabetes_binary_norm.csv	D3	14	67136	2.18	0	0	0
spotify-2023_norm.csv	S2	20	953	0.06	198327	17915	180412
diabetes_binary2_norm.csv	D4	12	236378	6.99	0	0	0
Dataset_norm.csv	D1	15	175028	13.64	858	68	790
file.csv	F	20	52955	7.08	3134	70	3064
DOSE_V2_norm.csv	D2	16	46797	5.18	5912	592	5320
merged_data_norm.csv	M	4	11509051	276.06	0	0	0
Test_norm.csv	T	12	89786	3.17	326	12	314
openpowerlifting_norm.csv	O	13	386414	33.50	1079	19	1060
youtube_norm.csv	Y	12	161470	13.35	1752	96	1656
superstore_norm.csv	S3	20	51290	6.47	45916	2098	43818

TABLE V. OVERALL RESULTS
(FASTOD)

Dataset	#Columns	#RB-columns	Java (seconds)	Base (seconds)	Optimized (seconds)	Impr (base)	Impr (opt)	Impr (total)
G	15	8	50.326	17.083	14.184	2.946x	1.204x	3.548x
A	23	3	135.004	49.309	40.137	2.738x	1.229x	3.364x
P	18	8	ME	294.900	160.967	∞	1.832x	∞
S1	11	2	724.101	93.216	71.644	7.768x	1.301x	10.107x
D3	14	11	ME	166.503	121.866	∞	1.366x	∞
S2	20	2	7.177	5.187	4.994	1.384x	1.039x	1.437x
D4	12	8	ME	189.312	142.254	∞	1.331x	∞
D1	15	3	478.885	69.236	47.129	6.917x	1.469x	10.161x
F	20	15	ME	143.485	84.933	∞	1.689x	∞
D2	16	2	23.348	7.176	6.676	3.254x	1.075x	3.497x
M	4	4	49.791	16.723	10.438	2.977x	1.602x	4.770x
T	12	6	158.503	50.214	45.864	3.157x	1.095x	3.456x
O	13	7	ME	264.255	182.810	∞	1.446x	∞
Y	12	5	129.516	26.755	20.810	4.841x	1.287x	6.224x
S3	20	10	453.583	110.795	75.537	4.094x	1.467x	6.005x

TABLE VI. OVERALL MEMORY USAGE RESULTS
(FASTOD)

Dataset	Java (GB)	Base (GB)	Optimized (GB)	Java vs Base	Java vs Optimized
G	3.150	0.809	1.160	3.894x	2.716x
A	3.087	1.035	1.270	2.983x	2.431x
P	ME	9.321	14.304	∞	∞
S1	11.919	3.478	4.504	3.427x	2.646x
D3	ME	8.296	14.124	∞	∞
S2	0.762	0.261	0.266	2.920x	2.865x
D4	ME	7.233	12.062	∞	∞
D1	6.878	2.374	3.556	2.897x	1.934x
F	ME	6.031	8.941	∞	∞
D2	1.062	0.071	0.126	14.958x	8.423x
M	9.508	2.519	2.595	3.775x	3.664x
T	2.537	0.760	0.994	3.338x	2.552x
O	ME	8.109	11.215	∞	∞
Y	3.168	0.839	1.248	3.776x	2.538x
S3	7.142	2.698	3.915	2.647x	1.824x

TABLE VII. DATASET DESCRIPTION (ORDER)

Dataset	Short name	Columns	Rows	Size (MB)	#OD
diabetes_binary_BRFSS2021.csv	Diabetes	22	236379	17.0	0
PFW_2021_public.csv	Pfw	22	100001	14.7	17
DITAG.csv	Ditag	5	4339917	299.0	0
creditcard_2023.csv	Credit	31	568631	324.8	0
EpicMeds.csv	Epic	10	1281732	56.8	9
modis_2000-2019_Australia.csv	Modis	15	5081220	410.4	7
bay_wheels_data_wrangled.csv	Bay	8	5022834	660.1	0

TABLE VIII. OVERALL RESULTS (ORDER)

Dataset	Java (seconds)	Base (seconds)	Optimized (seconds)	Impr (base)	Impr (opt)	Impr (total)
Diabetes	3.331	3.696	0.591	0.901x	6.253x	5.636x
Pfw	1.361	0.472	0.209	2.883x	2.258x	6.511x
Ditag	16.833	7.219	3.297	2.331x	2.189x	5.105x
Credit	32.585	5.279	3.056	6.172x	1.727x	10.662x
Epic	8.649	3.438	1.615	2.515x	2.128x	5.355x
Modis	88.069	9.339	3.661	9.430x	2.550x	24.055x
Bay	33.158	18.466	6.630	1.795x	2.785x	5.001x

[6] C. C. Aggarwal and J. Han, *Frequent Pattern Mining*. Springer Publishing Company, Incorporated, 2014.

[7] P. G. Brown and P. J. Hass, "Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data," in *VLDB'03*. VLDB Endowment, 2003, p. 668–679.

[8] M. Hulsebos *et al.*, "Sherlock: A deep learning approach to semantic data type detection," in *SIGKDD'19*, 2019, p. 1500–1508.

[9] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava, "Effective and complete discovery of order dependencies via set-based axiomatization," *Proc. VLDB Endow.*, vol. 10, no. 7, pp. 721–732, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p721-szlichta.pdf

[10] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte, "Expressiveness and complexity of order dependencies," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1858–1869, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1858-szlichta.pdf

[11] S. Ginsburg and R. Hull, "Order dependency in the relational model," *Theoretical Computer Science*, vol. 26, no. 1, pp. 149–195, 1983. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397583900841

[12] P. Langer and F. Naumann, "Efficient order dependency detection," *VLDB J.*, vol. 25, no. 2, pp. 223–241, 2016. [Online]. Available: https://doi.org/10.1007/s00778-015-0412-3

[13] 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.

[14] 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.

[15] J. Szlichta, P. Godfrey, and J. Gryz, "Fundamentals of order dependencies," *Proc. VLDB Endow.*, vol. 5, no. 11, p. 1220–1231, jul 2012. [Online]. Available: https://doi.org/10.14778/2350229.2350241

[16] W. Ng, "An extension of the relational data model to incorporate ordered

[18] Z. Tan, A. Ran, S. Ma, and S. Qin, "Fast incremental discovery of pointwise order dependencies," *Proc. VLDB Endow.*, vol. 13, no. 10, p. 1669–1681, jun 2020. [Online]. Available: https://doi.org/10.14778/3401960.3401965

domains," *ACM Trans. Database Syst.*, vol. 26, no. 3, p. 344–383, sep 2001. [Online]. Available: https://doi.org/10.1145/502030.502033

[17] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava, "Effective and complete discovery of bidirectional order dependencies via set-based axioms," *The VLDB Journal*, vol. 27, no. 4, p. 573–591, aug 2018. [Online]. Available: https://doi.org/10.1007/s00778-018-0510-0

[19] X. Chu, I. F. Ilyas, and P. Papotti, "Discovering denial constraints," *Proc. VLDB Endow.*, vol. 6, no. 13, p. 1498–1509, aug 2013.

[20] Y. Jin, L. Zhu, and Z. Tan, "Efficient bidirectional order dependency discovery," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 61–72.

[21] H. Saxena, L. Golab, and I. F. Ilyas, "Distributed implementations of dependency discovery algorithms," *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1624–1636, jul 2019.

[22] S. Schmidl and T. Papenbrock, "Efficient distributed discovery of bidirectional order dependencies," *The VLDB Journal*, vol. 31, no. 1, p. 49–74, aug 2021.

[23] H. Yao and H. J. Hamilton, "Mining functional dependencies from data," *Data Min. Knowl. Discov.*, vol. 16, no. 2, p. 197–219, apr 2008.

[24] C. Wyss, C. Giannella, and E. Robertson, "Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract," in *DaWaK*, Y. Kambayashi *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 101–110.

[25] 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.

[26] R. Karegar, M. Mirsafian, P. Godfrey, L. Golab, M. Kargar, D. Srivastava, and J. Szlichta, "Discovering domain orders via order dependencies," in *ICDE'22*, 2022, pp. 1098–1110.

[27] C. Consonni *et al.*, "Discovering order dependencies through order compatibility," in *EDBT'19*, M. Herschel *et al.*, Eds. OpenProceedings.org, 2019, pp. 409–420.

[28] P. Godfrey, L. Golab, M. Kargar, D. Srivastava, and J. Szlichta, "Errata note: Discovering order dependencies through order compatibility," *CoRR*, vol. abs/1905.02010, 2019.