The VLDB Journal
https://doi.org/10.1007/s00778-021-00683-4

REGULAR PAPER

Efficient distributed discovery of bidirectional order dependencies

Sebastian Schmidl[1] · **Thorsten Papenbrock**[1]

Received: 27 August 2020 / Revised: 16 April 2021 / Accepted: 1 July 2021

Abstract
Bidirectional order dependencies (bODs) capture order relationships between lists of attributes in a relational table. They can express that, for example, sorting books by *publication date* in ascending order also sorts them by *age* in descending order. The knowledge about order relationships is useful for many data management tasks, such as query optimization, data cleaning, or consistency checking. Because the bODs of a specific dataset are usually not explicitly given, they need to be discovered. The discovery of all minimal bODs (in set-based canonical form) is a task with exponential complexity in the number of attributes, though, which is why existing bOD discovery algorithms cannot process datasets of practically relevant size in a reasonable time. In this paper, we propose the *distributed* bOD discovery algorithm DISTOD, whose execution time scales with the available hardware. DISTOD is a scalable, robust, and elastic bOD discovery approach that combines efficient pruning techniques for bOD candidates in set-based canonical form with a novel, reactive, and distributed search strategy. Our evaluation on various datasets shows that DISTOD outperforms both single-threaded and distributed state-of-the-art bOD discovery algorithms by up to orders of magnitude; it can, in particular, process much larger datasets.

Keywords Bidirectional order dependencies · Distributed computing · Actor programming · Parallelization · Data profiling · Dependency discovery

1 Distributed discovery of order dependencies

Order is a fundamental concept in relational data because every attribute can be used to sort the records of a relation. Some sortings represent the natural ordering of attribute values by their domain (e. g., timestamps, salaries, or heights) and, hence, express meaningful statistical metadata; other sortings serve technical purposes, such as data compression (e. g., via run-length encoding [1]), index optimization (e. g., for sorted indexes [3]), or query optimization (e. g., when picking join strategies [23]).

Because a relational instance can follow only one sorting at a time, dependencies between different orders help to find optimal sortings; they also reveal meaningful correlations between attribute domains.

✉ Sebastian Schmidl
sebastian.schmidl@hpi.de

Thorsten Papenbrock
thorsten.papenbrock@hpi.de

[1] Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany

An order dependency (OD) expresses an order relationship between lists of attributes in a relational table. More specifically, an OD $\mathbf{X} \mapsto \mathbf{Y}$ specifies that when we order the tuples of a relational table based on the left-hand side attribute list $\mathbf{X}$, then the tuples are also ordered by the right-hand side attribute list $\mathbf{Y}$. The tuple order is lexicographical w. r. t. the attribute values selected by $\mathbf{X}$ and $\mathbf{Y}$, respectively. This means that ties in the order implied by the first attribute in the list are resolved by the next attribute in the list (and so forth). This resembles the ordering produced by the `ORDER BY`-clause in SQL. A *bidirectional* order dependency (bOD), such as $[\mathsf{A}\uparrow, \mathsf{B}\downarrow] \mapsto [\mathsf{C}\uparrow]$, lets us define the order direction of the individual attributes involved in the bOD; in this example: A in ascending order with ties resolved by B in descending order sorts C in ascending order. ODs are closely related to functional dependencies (FDs), which have been extensively studied in research [16], but due to their consideration of order, ODs subsume FDs [28].

The dataset shown in Table 1, for example, fulfills the bOD $[\texttt{ADelay}\uparrow] \mapsto [\texttt{ADGrp}\uparrow]$. In other words, when we sort the tuples by the `ADelay` attribute, then they are also ordered by the `ADGrp` attribute. Note that the inverse bOD $[\texttt{ADGrp}\uparrow] \mapsto [\texttt{ADelay}\uparrow]$ does not hold, because the value

Published online: 16 August 2021

in attribute ADGrp of the tuple t_5 is greater or equal to the value in ADGrp of tuple t_9, but t_5's value in ADelay is smaller than t_9's value in ADelay.

With order dependencies, we know how the ordering of tuples based on certain attributes translates to the ordering based on other attributes. This knowledge can be used in various situations: During *query planning*, for example, ODs help to derive additional orders that enable further optimizations, such as eliminating costly sort operations or selecting better join strategies [28]. In *database design*, ODs can be used to, for example, replace dense implementations of secondary indexes with sparse implementations if we know that the tuple ordering by the secondary index' attributes is determined by the ordering of the primary key attributes [6]. For *consistency maintenance* and *data cleaning*, ODs can be considered as integrity constraints (ICs). Like all other ICs, semantically meaningful ODs can describe business rules so that any violation of an ODs indicates an error in the dataset. In this way, ODs can guide automatic data cleaning [11].

Although certain ODs can be obtained manually, this process is very time-consuming and difficult. ODs are naturally expressed using lists of attributes, which leads to a search space that grows factorial with the number of attributes in a dataset. Fortunately, with the polynomial mapping of ODs to a set-based canonical form, which was presented by Szlichta et al. [25], we can construct a search space for ODs that grows only exponentially with the number of attributes. The search space is still too large for manual exploration, but it is small enough for automatic OD discovery algorithms. An example of a set-based OD is {AirTime, DayOfWeek, FlightNum, TailNum} : ArrTime↑ ∼ ADelay↑, which is valid in the flight dataset (see Sect. 9). The OD specifies that for flights flown by the same aircraft on the same route and day for the same flight time, the delay at the destination airport monotonically increases over the day. We define set-based ODs in more detail in Sect. 3.2.

To automate the discovery of ODs, researchers have proposed different order dependency [5,15,26] and bidirectional order dependency [25] discovery algorithms. Depending on the OD representation, these approaches have a factorial [5,15] or exponential [25,26] worst-case complexity in the number of attributes. Despite various clever pruning strategies, none of the existing OD discovery algorithms can process datasets of practically relevant size in a feasible time. The FASTOD-BID algorithm, for example, takes almost 5 h on the 700 KiB letter dataset with 17 attributes and 20K records, and it exceeds 58 GB of memory on the 70 MiB flight dataset with 21 attributes and 500K records (see Table 2 in Sect. 9).

To overcome existing algorithmic limitations, we propose DISTOD, a *scalable* (in number of cores and number of computers), *robust* (against limited memory), *elastic* (in adding and removing of computers at runtime), and *applicable* (i.e., with semantic pruning strategies equipped) bOD discovery algorithm.

DISTOD pursues a novel, reactive bOD search strategy that allows it to distribute both the discovery process and the validation of bODs on multiple machines in a compute cluster without a need for global parallelization barriers. The algorithm discovers all minimal bODs by deliberately centralizing the candidate generation and pruning; to maximize the efficiency and scalability of the discovery process, it dynamically parallelizes and distributes all other parts of the discovery via reactive programming strategies. DISTOD is based on the canonical representation of set-based bODs from Szlichta et al. [25], which allows it to traverse a relatively small set-containment lattice and to benefit from the known pruning rules.

The motivation for this research project is the observation that most distributed data profiling algorithms, including [14,21,22,33], are built on top of dataflow-based distributed computing frameworks, such as Apache Spark [31] or Apache Flink [30]. These frameworks force the discovery algorithms into batch processing, which is an unsuitable paradigm for all known dependency discovery approaches, because they rely on dynamic pruning and dynamic candidate generation tech-

Table 1 flight dataset excerpt. ⊥ denotes null values

ID	Month	Day	Code	Fips	State	DDelay	ADelay	ADGrp
t_0	12	3	CLE	39	Ohio	80	72	4
t_1	12	14	ORD	17	Illinois	−4	−10	−1
t_2	12	14	JFK	36	New York	−6	0	0
t_3	12	16	ORD	17	Illinois	13	59	3
t_4	12	20	CLE	39	Ohio	⊥	⊥	⊥
t_5	12	24	ORD	17	Illinois	5	−2	−1
t_6	12	27	BWI	24	Maryland	−5	−23	−2
t_7	12	28	ORD	17	Illinois	57	82	5
t_8	12	30	SGF	29	Missouri	2	0	0
t_9	12	30	ORD	17	Illinois	−5	−14	−1

niques. To implement the dynamic parts of the discovery, the algorithms split the search into multiple runs of batch processes and utilize the synchronization barriers in between the distributed runs for pruning and dynamic search decisions. For this reason, their performance implicitly suffers from idle times due to the synchronization barriers, unnecessary re-partitioning of data, and the inability to make dynamic search decisions within batch runs. We, therefore, advocate the use of a reactive computing paradigm, i. e., actor programming [8], for the implementation of distributed data profiling algorithms. At the cost of harder programming, we can thereby find superior search strategies, minimize idle times, avoid certain redundant work, optimize resource utilization, and support elasticity.

In this paper, we first introduce related work about the discovery of ODs (Sect. 2) and the formal foundations on the set-based canonical form for bODs (Sect. 3). We then make the following contributions:

Distributed, reactive search strategy We introduce a distributed, reactive bOD search strategy that breaks the strictly level-wise search approach of FASTOD-BID [25] up into fine-grained tasks that represent constant and order compatible bOD candidates separately. A reactive resolution strategy for intra-task dependencies allows a synchronization barrier-free work distribution for the candidate validation tasks (Sects. 4 to 6).

Parallel candidate generation We present a centralized, but highly parallel candidate generation algorithm. The algorithm guarantees that all minimal bODs are generated while traversing the candidate lattice (Sect. 5).

Revised validation algorithm We use a new index data structure, which we call inverted sorted partition, to improve the efficiency of the order compatible bOD validation algorithm from [25] (Sect. 6).

Hybrid, dynamic partition generation We contribute a hybrid and dynamic generation algorithm for stripped partitions that either uses a recursive partition generation scheme or a direct partition product to generate a stripped partition on-demand (Sect. 7).

Effective memory management We present a dynamic memory management strategy that caches intermediate and temporary results for as long as possible; freeing them as soon as memory runs short (Sect. 7).

Elasticity and semantic pruning We equip DISTOD with elasticity properties (Sect. 8.1) and semantic pruning strategies (Sect. 8.2) to enable the discovery of bODs in datasets of practically relevant size.

Evaluation We evaluate the runtime, memory usage, and scalability of DISTOD on various datasets and show that it outperforms both the single-threaded algorithm FASTOD-BID and the distributed algorithm DISTFASTOD-BID by up to orders of magnitude (Sect. 9).

2 Related work

In 1982, Ginsburg and Hull were the first to consider the ordering of records w.r.t. different lists of attributes in a relation as a kind of dependency [7]. Their work introduced *point-wise orders* with a complete set of inference rules and shows that the inference-problem in this formalism is co-NP-complete. In 2012, Szlichta et al. formally defined order dependencies as a dependency between lists of attributes such that if the relation is ordered by the values of the first attribute list, it is also ordered by the values of the second attribute list [29]. Like SQL `ORDER BY` operators, the formalism uses a lexicographical ordering of tuples by the attribute lists. Szlichta et al. also introduced a set of axioms and the proof that ODs properly subsume FDs. The list-based formalization was adopted by many following works on OD profiling [5,15,26]. Later, bidirectional order dependencies (bODs)—a combination of ascending and descending orders of attributes—have been introduced [28]. The authors of [29] and [28] show that the inference problem for both ODs and bODs is co-NP-complete. In this work, we discover bODs using the set-based formalism as defined in [28]. We now discuss existing OD and bOD discovery algorithms.

Order dependency discovery The first automatic OD discovery algorithm, called ORDER, was proposed by Langer and Naumann [15]. It traverses a list-containment lattice of OD candidates to find (all) valid, minimal dependencies in a given dataset. The algorithm has a factorial worst-case complexity in the number of attributes, is sound, but is intentionally incomplete as confirmed in [25,26].

Inspired by [4] and [18], Jin et al. proposed a hybrid OD discovery approach that discovers ODs by alternately comparing records on a sample of the dataset and validating candidates on the entire dataset [12]. Their approach can discover ODs as well as bODs. The authors show that their algorithm discovers the same set of ODs as ORDER; hence, the algorithm also produces incomplete results.

FASTOD proposed by Szlichta et al. is the first algorithm that discovers complete sets of minimal ODs [26]. By mapping ODs to a new set-based canonical representation, the algorithm has only exponential worst-case complexity in the number of attributes and linear complexity in the number of tuples. The authors also provide effective inference rules for the new OD representation. With the algorithm FASTOD-BID, the same authors later expanded their discovery approach to bODs [25]. They show that discovering bidirectional ODs does not take significantly longer than discovering unidirectional ODs. We base our algorithm on the same set-based canonical representation of bODs and the corresponding definition of minimality to also benefit from the reduced search space size and efficient pruning rules.

With OCDDISCOVER, Consonni et al. took another approach to the OD discovery task by exploiting order

compatibility dependencies (OCDs) [5]. An OCD is a special form of OD in which two lists of attributes order one another if they are concatenated [29]. Unfortunately, OCDDISCOVER uses an incorrect definition of minimality and, therefore, prunes the search space too aggressively; consequently, the results are incomplete [27].

Distributed order dependency discovery Because FASTOD-BID is the only complete and correct OD algorithm, not much research exists on distributed OD discovery. In [22], Saxena et al. proposed common map-reduce style primitives (based on Apache Spark) into which they could break down any existing data profiling algorithm. In this way, they presented distributed versions of different dependency discovery algorithms including FASTOD-BID—we call this implementation DIST-FASTOD-BID. Performance-wise, all these algorithms suffer from non-optimal resource utilization because batch-oriented algorithms frequently re-partition the data and contain (many) hard synchronization barriers when used for dynamic discovery algorithms. They also do not support elasticity, i.e., they struggle with flexible cluster sizes, where nodes enter and leave at runtime. For these reasons, we use the reactive actor-programming model for distribution and parallelization, which leads to a fundamentally different algorithm design. Our approach waives hard synchronization barriers, reactively optimizes the load balancing, and reduces data communication costs.

3 Foundations

In this paper, we use the following notational conventions:

R denotes a relation and r a specific instance of R.
A and B, C, etc., denote single attributes from R.
t and s denote tuples of a relational instance r.
t_A denotes the value of an attribute A in a tuple t.
X and Y, etc., are sets of attributes and X_i the ith element of X with $0 \leq i < |X|$. We use W_i to indicate subsets with $|W_i| = |X| - 1$ and Z_i to indicate supersets with $|Z_i| = |X| + 1$ for a given attribute set X.
$\mathbf{X}$ and $\mathbf{Y}$, $\mathbf{Z}$, etc., are lists of attributes and $\mathbf{X}_i$ the i^{th} element of $\mathbf{X}$ with $0 \leq i < |\mathbf{X}|$. [ ] is the empty list and [A | $\mathbf{X}$] denotes a list with head A and tail $\mathbf{X}$. Lists and sets with the same name reference the same attributes, i.e., set X, contains all distinct elements from list $\mathbf{X}$.

In this section, we first formally define bODs, then we recap the set-based canonical form for bODs [25], and finally, we describe the core concepts of actor programming our means to dynamically distribute the discovery process in a cluster.

3.1 Order dependencies

Following the definitions for bidirectional order dependencies given in [28], we first define *order specifications*, which specify how to sort the tuples of a dataset based on multiple attributes w.r.t. different order directions (ascending or descending). It corresponds to the `ORDER BY`-clause in SQL and produces a lexicographical ordering.

Definition 1 An ordering based on an attribute $A \in R$ can be either ascending or descending. To indicate the order direction of a *marked attribute* $\overline{A}$, we use $A\uparrow$ for ascending and $A\downarrow$ for descending. An *order specification* is a list of marked attributes denoted as $\overline{\mathbf{X}}$ with $\mathbf{X}_i \in R$. For attributes without an explicit order direction, we implicitly assume an ascending ($\uparrow$) order.

Using order specifications, we can now introduce bidirectional order dependencies [28].

Definition 2 A bidirectional order dependency (bOD) is a statement of the form $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ (read: $\overline{\mathbf{X}}$ orders $\overline{\mathbf{Y}}$) specifying that ordering a relation r by *order specification* $\overline{\mathbf{X}}$ also orders r by *order specification* $\overline{\mathbf{Y}}$, where $X \subset R$ and $Y \subset R$. We use the notation $\overline{\mathbf{X}} \leftrightarrow \overline{\mathbf{Y}}$ (read: $\overline{\mathbf{X}}$ and $\overline{\mathbf{Y}}$ are order equivalent) if $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ and $\overline{\mathbf{Y}} \mapsto \overline{\mathbf{X}}$. For $\overline{\mathbf{XY}} \leftrightarrow \overline{\mathbf{YX}}$, the two order specifications $\overline{\mathbf{X}}$ and $\overline{\mathbf{Y}}$ are *order compatible* and we write $\overline{\mathbf{X}} \sim \overline{\mathbf{Y}}$. Table r over R satisfies a bOD $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ if $\forall s, t \in r : s \preceq_{\overline{\mathbf{X}}} t \Rightarrow s \preceq_{\overline{\mathbf{Y}}} t$. The lexicographical order operator $\preceq_{\overline{\mathbf{Z}}}$ for an order specification $\overline{\mathbf{Z}}$ and the tuples $s, t \in r$ is defined as:

$$s \preceq_{\overline{\mathbf{Z}}} t = \begin{cases} \overline{\mathbf{Z}} = [\,] \\ \overline{\mathbf{Z}} = [A\uparrow \mid \overline{\mathbf{T}}] \wedge s_A < t_A \\ \overline{\mathbf{Z}} = [A\downarrow \mid \overline{\mathbf{T}}] \wedge s_A > t_A \\ \overline{\mathbf{Z}} = \left([A\uparrow \mid \overline{\mathbf{T}}] \vee [A\downarrow \mid \overline{\mathbf{T}}]\right) \wedge s_A = t_A \wedge s \preceq_{\overline{\mathbf{T}}} t \end{cases}$$

It is $s \prec_{\overline{\mathbf{Z}}} t$ if $s \preceq_{\overline{\mathbf{Z}}} t$ but $t \not\preceq_{\overline{\mathbf{Z}}} s$.

The lexicographical order operator $\preceq_{\overline{\mathbf{Z}}}$ defines a weak total order over a set of tuples. We assume that numbers are ordered numerically, strings are ordered lexicographically, and dates are ordered chronologically.

If $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$, then any ordering of tuples for any table r that satisfies $\overline{\mathbf{X}}$ also satisfies $\overline{\mathbf{Y}}$. Considering our example in Table 1, i.a., the following bODs hold: [`ADelay`] $\mapsto$ [`ADGrp`], [`Code`$\uparrow$] $\mapsto$ [`Month`$\downarrow$], [`ADelay`] $\mapsto$ [`ADGrp`, `DDelay`], [`ADGrp`, `DDelay`] $\mapsto$ [`ADelay`], and [`State`$\uparrow$, `Day`$\downarrow$] $\mapsto$ [`Fips`$\uparrow$]. Note that these bODs do hold in our example table and not necessarily in general.

3.2 Set-based canonical bODs

The search space of bODs in list-based form $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ (see Definition 2) grows factorial with the number of attributes [15].

Despite clever candidate pruning rules, this growth defines the complexity of all bOD discovery algorithms that use the list-based bOD formalization. Hence, we now introduce (and later use) a set-based bOD formalization, as defined in [25]. Set-based bODs span much smaller set-containment candidate lattices similar to, e. g., FD discovery algorithms like TANE [10], that grow only exponential with the number of attributes and, therefore, make an efficient discovery feasible. For space reasons, we do not repeat the mapping between set-based and list-based bODs, all proofs, and the axioms for set-based bODs and refer to [25] for details.

First, we introduce *equivalence classes* and *partitions* consistent with [9] and [25]:

Definition 3 An *equivalence class* w. r. t. a given attribute set is denoted as $\mathscr{E}(t_X)$. It groups tuples s and t together if their projection on X is equal: $\mathscr{E}(t_X) = \{s \in \mathrm{r} \mid s_X = t_X\}$ and $X \in \mathrm{R}$. The attribute set X is called *context*.

This means that all tuples in an equivalence class $\mathscr{E}(t_X)$ have the same value (or value combination) in X. Partitions group equivalence classes by a common attribute set:

Definition 4 A *partition* Π_X is a set of disjoint equivalence classes with the same set of attributes: $\Pi_X = \{\mathscr{E}(t_X) \mid t \in \mathrm{r}\}$.

From our example dataset in Table 1, we can extract, for example, the partition $\Pi_{\{\texttt{State}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_2\}, \{t_6\}, \{t_8\}\}$. With equivalence classes and partitions, we now define the two set-based canonical forms for bODs [25, Definition 9]: *constant bODs* and *order compatible bODs*.

Constant bODs

Definition 5 A *constant bOD* is a marked attribute $\overline{\mathsf{A}}$ that is constant within each equivalence class w. r. t. the set of attributes in the *context* X. It is denoted as $X : [\,] \mapsto \overline{\mathsf{A}}$. It can be mapped to the list-based bODs $\overline{\mathbf{X'}} \mapsto \mathbf{X'}\overline{\mathsf{A}}$ for all permutations $\overline{\mathbf{X'}}$ of $\overline{\mathbf{X}}$.

In our example dataset, i. a., the following constant bODs are valid: $\{\} : [\,] \mapsto \texttt{Month}$ and $\{\texttt{ADelay}\} : [\,] \mapsto \texttt{ADGrp}$. Constant bODs directly represent FDs [25]. They can be violated only by so-called *splits*.

Definition 6 A *split* w. r. t. a constant bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ is a pair of tuples s and t such that both tuples are part of the same equivalence class $\mathscr{E}(t_X)$ but $s_\mathsf{A} \neq t_\mathsf{A}$.

The bOD $\{\texttt{ADGrp}\} : [\,] \mapsto \texttt{ADelay}$ is not valid in our example dataset because it is invalidated by at least one *split* (e. g., tuple t_1 and t_5).

Order compatible bODs

Definition 7 An *order compatible bOD* is denoted as $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ and states that two marked attributes $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ are order compatible within each equivalence class w. r. t. the set of attributes in the *context* X. It can be mapped to a list-based bOD $\overline{\mathbf{X'}\mathsf{A}} \sim \overline{\mathbf{X'}\mathsf{B}}$ for any permutation $\overline{\mathbf{X'}}$ of $\overline{\mathbf{X}}$.

A valid order compatible bOD of our example dataset is $\{\} : \texttt{Fips}\uparrow \sim \texttt{State}\uparrow$. It tells us that when we order the dataset by `Fips` it is also ordered by `State`. Order compatible bODs are violated by so-called *swaps*:

Definition 8 A *swap* w. r. t. an order compatible bOD $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ is a pair of tuples s and t such that both tuples are part of the same equivalence class $\mathscr{E}(t_X)$ and $s \prec_{\overline{\mathsf{A}}} t$ but $t \prec_{\overline{\mathsf{B}}} s$.

The order compatible bOD $\{\} : \texttt{Fips}\uparrow \sim \texttt{State}\downarrow$, for example, is not valid in Table 1, because, i. a., t_9 and t_6 form a *swap*. Discovery algorithms for bODs in set-based form use both *splits* and *swaps* to validate constant and order compatible bOD candidates.

Mapping from list to set-based form List-based bODs can be mapped to a set of set-based bODs in polynomial time. This mapping is based on the fact that $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ is valid only if $\overline{\mathbf{X}} \mapsto \overline{\mathbf{XY}}$ and $\overline{\mathbf{X}} \sim \overline{\mathbf{Y}}$ are valid as well [25, Theorem 2]. $\overline{\mathbf{X}} \mapsto \overline{\mathbf{XY}}$ ensures that there are no *splits* and $\overline{\mathbf{X}} \sim \overline{\mathbf{Y}}$ ensures that there are no *swaps* that falsify the bOD. As we have seen in Definitions 6 and 8, the two set-based canonical forms for bODs enforce the same constraints.

Definition 9 A list-based bOD $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ is valid iff the two following statements are true: First, $\forall \overline{\mathsf{A}} \in \overline{\mathbf{Y}}$ the set-based bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ is valid and, second, $\forall i \in \{1, \ldots, |\overline{\mathbf{X}}|\}$, $j \in \{1, \ldots, |\overline{\mathbf{Y}}|\}$ the set-based bOD $\{\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}, \mathsf{Y}_1, \ldots, \mathsf{Y}_{j-1}\} : \overline{\mathsf{X}_i} \sim \overline{\mathsf{Y}_j}$ is valid.

3.3 Actor programming model

The actor model is a reactive programming paradigm for concurrent, parallel and distributed applications [8]. It helps to avoid blocking behavior via isolation and asynchronous message passing. The core primitive in this model are *actors*, which are objects with strictly private state and behavior. Actors are dynamically scheduled on threads by the actor runtime and, hence, can execute tasks in parallel. They communicate within and across process boundaries via asynchronous messages, which are immutable objects with arbitrary, but serializable content. Incoming messages to an actor are buffered in the actor's mailboxes and then processed sequentially, so all parallelization happens between actors but not within one actor.

The strong isolation of actors and their lock-free, reactive concurrency model supports the development of highly scalable, but still dynamic algorithms [32], which is needed for search tasks, such as dependency discovery. Batch processing frameworks for distributed computing, such as Apache Spark [31] or Apache Flink [30], impose stricter workflows that sacrifice algorithmic flexibility to ease the implementation. Therefore, we implement our algorithm with the Akka toolkit [24] for actor programming.

4 Efficient distributed bOD discovery

In this section, we give an overview of our scalable, robust, and elastic bOD discovery algorithm DISTOD. DISTOD is executed on a cluster of network-connected compute machines (nodes). The algorithm assumes an asynchronous, switched network model, in which messages can be arbitrarily dropped, delayed, and reordered. We now first introduce the DISTOD algorithm and, then, describe its architecture.

4.1 DISTOD algorithm

DISTOD is a discovery algorithm that traverses a bOD candidate lattice based on all possible sets of attributes reactively breadth-first. Figure 1 shows a snapshot of such a set-lattice for the attributes A, B, C, D, where some nodes have already been processed (bold nodes) and others have been pruned (dashed nodes). DISTOD starts the search with singleton sets of attributes and progresses to ever-larger sets of attributes in the lattice. When processing node X, it checks the following bODs: Constant bODs of the form $X \setminus \{A\} : [\,] \mapsto \overline{A}$, where $A \in X$, and order compatible bODs of the form $X \setminus \{A, B\} : \overline{A} \sim \overline{B}$, where $A, B \in X$ and $A \neq B$. Following FASTOD-BID's bottom-up search strategy [25], DISTOD can use the same minimality definitions and pruning rules that guarantee that only minimal and valid bODs are added to the result set (see Sect. 5). DISTOD produces exactly the same results as FASTOD-BID.

Although both DISTOD and FASTOD-BID use the same formalisms and pruning rules, DISTOD does not generate the candidate lattice level-wise, but instead uses a task-based approach that interleaves candidate generation, validation, and pruning. Hence, there are no synchronization barriers between the three steps and the algorithm can use the available resources in its distributed environment more effectively. In theory, DISTOD still follows the following high-level steps proposed by FASTOD-BID, but interleaved: (i) initialization and generation of the initial candidates (ii) candidate validation (iii) node pruning (iv) generation of the next candidates. The steps (ii) to (iv) repeat until all candidates have been processed. Step (ii) is explained in Sect. 6, while steps (i), (iii), and (iv) are subject of Sect. 5. Interleaving the four main algorithm steps means that they may occur concurrently for different nodes in the lattice. We do not strictly enforce that a level l_i has to be completed before the next level l_{i+1} is started. In our task-based approach, each node (attribute set X) in the candidate lattice represents a task, whose candidates have to be generated, validated, and pruned. DISTOD works on these tasks in parallel and a set of rules ensures that only minimal and non-pruned candidates are checked. Hence, a snapshot of the candidate lattice of a running instance of DISTOD might look like Fig. 1, where some high level nodes (e.g., {A, B, C}) have already been processed while lower level nodes (e.g., {A, D}) still need to be finished.

A single central master actor is responsible for maintaining a consistent view on the candidate lattice, performing minimality checks, and executing pruning decisions. Once the master has generated a candidate, the candidate can be validated independently of other candidates, which allows us to distribute these checks to different compute nodes. We describe the candidate generation in detail in Sect. 5 and the validation of candidates in Sect. 6. Section 7 explains how data are managed in DISTOD.

4.2 DISTOD architecture

The DISTOD system consists of a cluster with a single leader node and various follower nodes. The leader node is the initial connection point for the cluster setup and it is responsible for managing the cluster. The single leader hosts the master component that is responsible for the generation of minimal candidates and all pruning decisions. The leader distributes validation jobs to the follower nodes, which in turn send the results back to the leader. We assume that all input data physically resides on the leader node. On algorithm startup, the leader automatically replicates the input data to the other nodes in the system; during the discovery, it writes the final results to the leader node's disk. Each node of the DISTOD cluster is started individually either immediately or later at runtime if more compute power is needed. A common seed node configuration ensures that all nodes find each other to form the DISTOD cluster. Hence, the start of DISTOD is not synchronized across nodes and the algorithm accepts follower nodes for validation tasks until the leader ends the discovery. This reactive startup strategy enables elasticity (see Sect. 8.1) and improves resource usage because the candidate processing begins as soon as possible, i. e., it does not wait until all nodes are ready.

Following the actor programming model, DISTOD consists of different actors that communicate using message-passing. Each node in the DISTOD cluster runs a set of actors which are grouped into five modules (see Fig. 2):

Master module The master module consists of the master components. Its actors are tasked with input, output, and state management as well as candidate generation. The master module is available only on the leader node.
Worker module The worker module contains actors responsible for the validation of bOD candidates and sending them back to the master components. The actors in this module can be spawned on all nodes.
Partition management module The partition management module hosts the actors storing the partitions (see Definition 4 on Page 5), which are used to validate bOD candidates. This module is available on all cluster nodes.

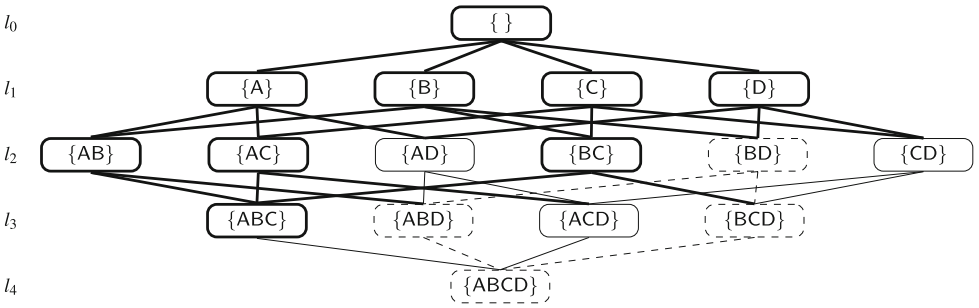


Fig. 1 Snapshot of a set lattice for the attributes A, B, C, D. Bold nodes have already been processed by DISTOD, thin nodes still await processing, and dashed nodes have been pruned from the lattice

Leader module The leader module contains actors controlling the shutdown procedure and the replication of the initial partitions. It is available on the leader node only.
Follower module The follower module contains puppet actors for the shutdown procedure and the partition replication. They are directly controlled by the corresponding actors in the leader module and steer the local parts of both processes on the follower nodes. They are placed only on the follower nodes.

The leader node hosts the actors from the master, partition management, and leader module; the follower nodes host the actors from the worker, partition management, and follower module. In this *passive leader* setup, the leader node does not host actors from the worker module and, therefore, does not perform expensive bOD candidate validations. The *active leader* setup, in contrast, hosts the worker module also on the leader node so that the leader can contribute spare resources to candidate validations; the active leader is also required for stand-alone executions on one node without follower nodes. In this setup, the master module actors are run on separate high-priority threads to ensure that the leader node remains reactive and can answer requests from the other nodes despite the hosting of worker actors. All our experiments use the active leader setup, but we recommend passive leader for particularly wide datasets with many bODs.

The actors of DISTOD are depicted with rounded corners in Fig. 2. The algorithm uses the master-worker pattern to work on the validation tasks in parallel. The `Master` module is responsible for creating and maintaining the candidate lattice. It generates the bOD candidates, creates validation jobs, and distributes them to the `Worker` actors via the dynamic work pulling pattern, which ensures a balanced load distribution. By passing all modifications through the `Master` actor, it maintains a consistent view on the candidate lattice. It also performs all pruning decisions and maintains the job queue. The `MasterHelper` actors support the `Master` actor by performing parallelizable tasks, such as the candidate generation or job-to-`Worker` dispatching. Section 5 describes how DISTOD ensures minimality and consistent pruning of bODs.

The `Worker` actors validate the bOD candidates, which is the most time-consuming part of the discovery. All `Workers` are supervised by a local `WorkerMgr` actor, which ensures that the system always operates at full capacity. The `Workers` emit valid bODs to the local `RCProxy` to immediately request a new validation job from the `Master` without waiting for result transmission. The `RCProxy` collects valid bODs from multiple `Workers` in a batch before reliably sending them to the single `ResultCollector` actor, which is responsible for formatting the results in a human-readable format and writing them to a file. Every batch from an `RCProxy` is immediately and asynchronously flushed to disk. This means that DISTOD outputs valid bODs progressively to a file on disk while the algorithm is still running. In this way, DISTOD can be stopped early, if the result set is already satisfactory.

The validation of bODs is performed using partitions (see Definition 4) of the original input dataset. Section 6 describes this approach in detail. At the start of the algorithm, the `DataReader` actor reads the input dataset, parses it, and uses multiple `Partitioner` actors to create the initial partitions, which are then sent to the `PartitionMgr` actor. The `PartitionMgr` stores the initial partitions and caches intermediate partitions. All requests for partitions from the local `Workers` are send to the `PartitionMgr`. If a requested partition is available in the cache, it is directly served to the `Worker`; otherwise, a partition generation job is sent to one of the `PartitionGen` actors. They perform the partition generation as described in Sect. 6 and return the partition to the `PartitionMgr`, which inserts the partition into the cache and forwards it to the requesting `Worker`(s).

If DISTOD runs on multiple nodes, some additional actors are needed for the cluster management. Figure 2 depicts them in the gray leader and follower modules. Since each node requires the initial partitions, we replicate the initial partitions from the leader's `PartitionMgr` to the `PartitionMgr` actors on all follower nodes via a side-channel implemented by the temporary `PartitionRepl` actors on the follower nodes and the corresponding `PMEndpoint` actors on the leader node (cf. Sect. 7.1). In addition to the partition replication, the algorithm also ensures that all nodes of the DISTOD cluster shut down cleanly at the end of the algo-

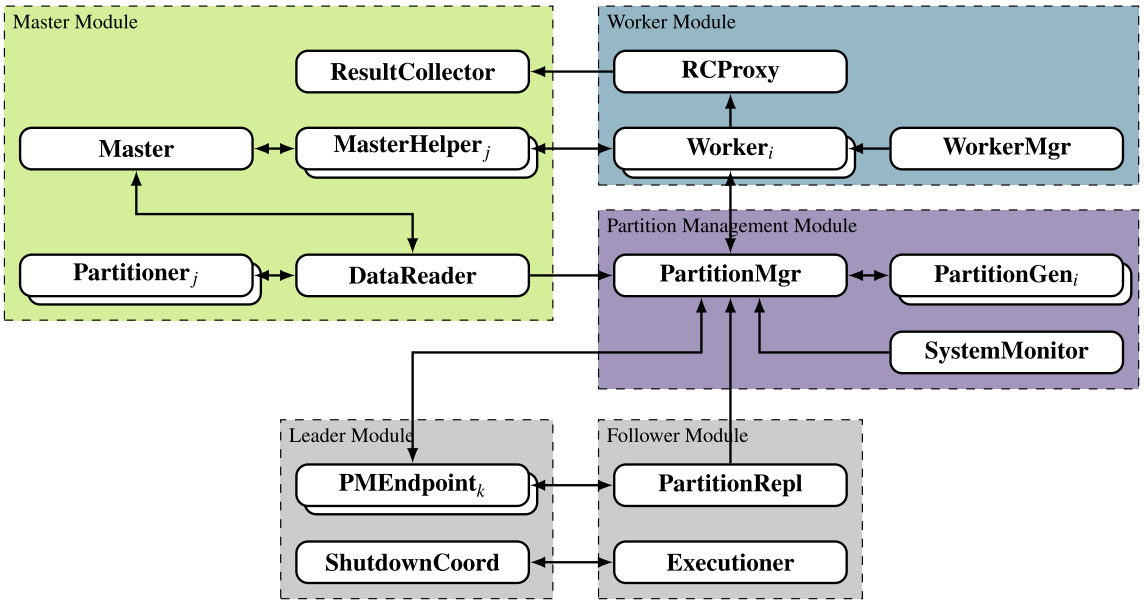


Fig. 2 DISTOD's architecture consisting of multiple actors grouped into logical modules. Multiple instances of the same actor type are indicated with indices i, j, and k, where i and j control the parallelism and $k \in [0 \ldots \#\texttt{nodes} - 1]$. Unidirectional arrows indicate unidirectional communication, i. e., asynchronous message sends, and bidirectional arrows indicate bidirectional communication, usually as request-response pairs

rithm and that all results are flushed to disk beforehand. This is handled by a coordinated shutdown protocol implemented using the `ShutdownCoord` actor on the leader node and the `Executioner` actors on the follower nodes. The `ShutdownCoord` actor implements a registry for follower nodes and drives the shutdown process.

5 Candidate generation

Like all existing algorithms, which are, i. a., [5,10,15,25,26], DISTOD uses a shared lattice data structure that tracks the results of the candidate validations to guarantee complete and correct results. This data structure is also used to check the minimality constraints during candidate generation. Since distributing this data structure causes a significant communication, i.e., synchronization, overhead that cannot be compensated by gains in parallelization, DISTOD sticks to a non-distributed, centralized candidate tracking and generation approach. A central component on the leader node, the `Master`, watches over intermediate results and ensures completeness and correctness of the algorithm. It generates bOD candidates, checks the candidates' minimality, and performs the candidate pruning because these three parts of the discovery algorithm rely on information about other nodes in the set lattice. All other parts of the algorithm can be executed independently of each other and, hence, they are distributed to the compute nodes. Intermediate results and pruning data are sent to the `Master`, which integrates them into its encapsulated state and considers them for the pruning decisions.

In this section, we first define minimality for bODs (Sect. 5.1) and then discuss how DISTOD ensures that only minimal bOD candidates are generated (Sect. 5.2). Afterward, we explain DISTOD's candidate generation algorithm (Sect. 5.3).

5.1 Trivial and minimal bODs

Like other dependency discovery algorithms, DISTOD outputs only minimal, non-trivial bODs. Non-minimal and trivial bODs can easily be inferred from the result set using the axioms for set-based bODs [25, Figure 5]. For triviality and minimality, we adopt the definition of FASTOD-BID [25] so that we can use the same highly effective pruning rules:

Definition 10 A constant bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ is *trivial* iff $\mathsf{A} \in X$. An order compatible bOD $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ is *trivial* iff $\mathsf{A} \in X$, $\mathsf{B} \in X$, or $\overline{\mathsf{A}} = \overline{\mathsf{B}}$.

Definition 11 A constant bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ is *minimal* iff it is not trivial and there is no context $Y \subset X$, such that $Y : [\,] \mapsto \overline{\mathsf{A}}$ holds in the instance r. An order compatible bOD $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ is *minimal* iff it is not trivial and there is no context $Y \subset X$, such that $Y : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$, $X : [\,] \mapsto \overline{\mathsf{A}}$, or $X : [\,] \mapsto \overline{\mathsf{B}}$ hold in r.

As [25] shows, we can further reduce the number of bODs that we have to consider in our discovery algorithm by elim-

inating bODs with similar semantics. Constant bODs of the form $X : [\,] \mapsto \mathsf{A}\uparrow$ and $X : [\,] \mapsto \mathsf{A}\downarrow$ are semantically equivalent (cf. [25, Revers-I in Figure 5]). Thus, we consider only constant bODs of the form $X : [\,] \mapsto \mathsf{A}\uparrow$. Order compatible bODs of the form $X : \mathsf{A}\uparrow \sim \mathsf{B}\uparrow$ and $X : \mathsf{A}\uparrow \sim \mathsf{B}\downarrow$ eliminate $X : \mathsf{A}\downarrow \sim \mathsf{B}\downarrow$ and $X : \mathsf{A}\downarrow \sim \mathsf{B}\uparrow$ respectively by Reverse-II [25, Figure 5]. Thus, we consider only order compatible bODs of the form $X : \mathsf{A}\uparrow \sim \mathsf{B}\uparrow$ and $X : \mathsf{A}\uparrow \sim \mathsf{B}\downarrow$. In summary, we use the following minimality pruning rules:

1. All relevant bOD candidates have the form $X : [\,] \mapsto \mathsf{A}\uparrow$, $X : \mathsf{A}\uparrow \sim \mathsf{B}\uparrow$, or $X : \mathsf{A}\uparrow \sim \mathsf{B}\downarrow$.
2. A constant bOD candidate $X : [\,] \mapsto \mathsf{A}\uparrow$ is not minimal if
   (a) it is trivial ($\mathsf{A} \in X$) or
   (b) there is a valid bOD $Y : [\,] \mapsto \mathsf{A}\uparrow$, where $Y \subset X$.
3. An order compatible bOD candidate $X : \overline{\mathsf{E}} \sim \overline{\mathsf{F}}$ with $(\overline{\mathsf{E}}, \overline{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\uparrow)$ or $(\overline{\mathsf{E}}, \overline{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\downarrow)$ is not minimal if
   (a) it is trivial ($\mathsf{A} \in X$, $\mathsf{B} \in X$, or $\mathsf{A} = \mathsf{B}$) or
   (b) there is a valid bOD $Y : \overline{\mathsf{E}} \sim \overline{\mathsf{F}}$, where $Y \subset X$, or
   (c) there is a valid bOD $X : [\,] \mapsto \mathsf{A}\uparrow$ or
   (d) there is a valid bOD $X : [\,] \mapsto \mathsf{B}\uparrow$.

Similar to TANE and just like FASTOD-BID, our candidate generation tracks all dependency candidates that either have not been tested yet or are known to be non-valid; these candidates will eventually lead to minimal bODs. Each node X in the candidate lattice, i. e., the candidate state $\mathscr{S}$, stores its untested/non-valid constant bOD candidates in the candidate set $\mathscr{S}(X).C_c$ and its untested/non-valid order compatible bOD candidates in the candidate set $\mathscr{S}(X).C_o$. An untested set of candidates may still result in valid minimal bODs. Removing valid bODs from the set after their validation enforces all pruning rules listed above. The sets then fulfill the following definitions:

Definition 12 $\mathscr{S}(X).C_c = \{\mathsf{A} \in \mathrm{R} \mid \forall \mathsf{B} \in X : X \setminus \{\mathsf{A}, \mathsf{B}\} : [\,] \mapsto \mathsf{B}\uparrow \text{ does not hold}\}$ [10, Lemma 3.3], [25, Definition 10]

If a constant bOD candidate $\mathsf{A} \in \mathscr{S}(X).C_c$ holds for a specific node X, then there was no valid constant bOD $Y \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A}\uparrow$ for any $Y \subset X$. Therefore, we can find minimal constant bODs by considering only candidates of the form $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A}\uparrow$, where $\mathsf{A} \in X$ (Rule 2a) and $\forall \mathsf{B} \in X : \mathsf{A} \in \mathscr{S}(X \setminus \{\mathsf{B}\}).C_c$ (Rule 2b). The same technique is used for order compatible bOD candidates:

Definition 13 $\mathscr{S}(X).C_o = \{(\overline{\mathsf{E}}, \overline{\mathsf{F}}) \mid (\overline{\mathsf{E}}, \overline{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\uparrow) \text{ or } (\mathsf{A}\uparrow, \mathsf{B}\downarrow), (\mathsf{A}, \mathsf{B}) \in X^2, \mathsf{A} \neq \mathsf{B} \text{ and } \forall \mathsf{C} \in X : X \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{C}\} : \overline{\mathsf{E}} \sim \overline{\mathsf{F}} \text{ does not hold, and } \forall \mathsf{C} \in X : X \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{C}\} : [\,] \mapsto \mathsf{C}\uparrow \text{ does not hold}\}$ [25, Definition 11].

If an order compatible bOD candidate $(\overline{\mathsf{E}}, \overline{\mathsf{F}}) \in \mathscr{S}(X).C_o$ for a specific node X with either $(\overline{\mathsf{E}}, \overline{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\uparrow)$ or $(\overline{\mathsf{E}}, \overline{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\downarrow)$, where $\mathsf{A} \in X$ and $\mathsf{B} \in X$ (Rule 3a), then there was no valid order compatible bOD $Y : \overline{\mathsf{E}} \sim \overline{\mathsf{F}}$ for any context $Y \subset X$ (Rule 3b) and both $X \setminus \{\mathsf{A}, \mathsf{B}\} : [\,] \mapsto \mathsf{A}\uparrow$ (Rule 3c) and $X \setminus \{\mathsf{A}, \mathsf{B}\} : [\,] \mapsto \mathsf{B}\uparrow$ (Rule 3d) do not hold.

In summary, storing only non-valid dependency candidates in each node and removing valid ones, automatically enforces all pruning rules listed above. All three minimality pruning rules eventually lead to [25, Lemma 12] (we call it "node pruning" rule), which states that if for a node X with $|X| \geq 2$ both candidate sets $\mathscr{S}(X).C_c$ and $\mathscr{S}(X).C_o$ are empty, all succeeding nodes' candidate sets $\mathscr{S}(Z).C_c$ and $\mathscr{S}(Z).C_o$ with $Z \supset X$ will be empty as well. This means that we can ignore all candidates from the succeeding nodes of a node, for which both candidate sets are empty. DISTOD prunes nodes from the candidate lattice by storing a flag $\mathscr{S}(X).p$ that indicates whether a node X should be considered or not. If a node X is pruned, all its successors in the candidate lattice are pruned as well; DISTOD does not generate bOD candidates for pruned nodes or their successors.

5.2 Generating minimal bOD candidates

In contrast to FASTOD-BID, DISTOD decouples the generation of candidates and their validation. The central `Master` component on the leader node drives the traversal of the candidate lattice. It performs the candidate generation, which includes the minimality checks and the pruning of nodes. The bOD candidates are encapsulated into jobs that are sent to `Worker` actors via a *work pulling* pattern. The distributed `Worker` actors then perform the validation of the minimal bOD candidates and send the results back to the `Master`.

DISTOD uses a task-based approach to bOD discovery. Each node in the candidate lattice (attribute set X) represents a task that is divided into five subtasks:

1. generation of minimal constant bOD candidates
2. generation of minimal order compatible bOD candidates
3. validation of minimal constant bOD candidates
4. validation of minimal order compatible bOD candidates
5. node pruning

Not all subtasks of a node can be executed concurrently because the subtasks depend on results of other subtasks of the node or its predecessors. Figure 3 depicts the inter- and intra-node dependencies for the subtasks of a node X with two predecessor nodes W_0 and W_1, where $|X| = 2 \wedge W_i = X \setminus \{\mathsf{X}_i\}$. Because **Gen**-tasks simply depend on all predecessor nodes' **Val**-tasks, Fig. 3 easily generalizes to $|X| > 2$.

The generation of the minimal constant bOD candidates of node X requires the checking of Rule 2b, i. e., it can be performed only after all constant bOD candidates of the set

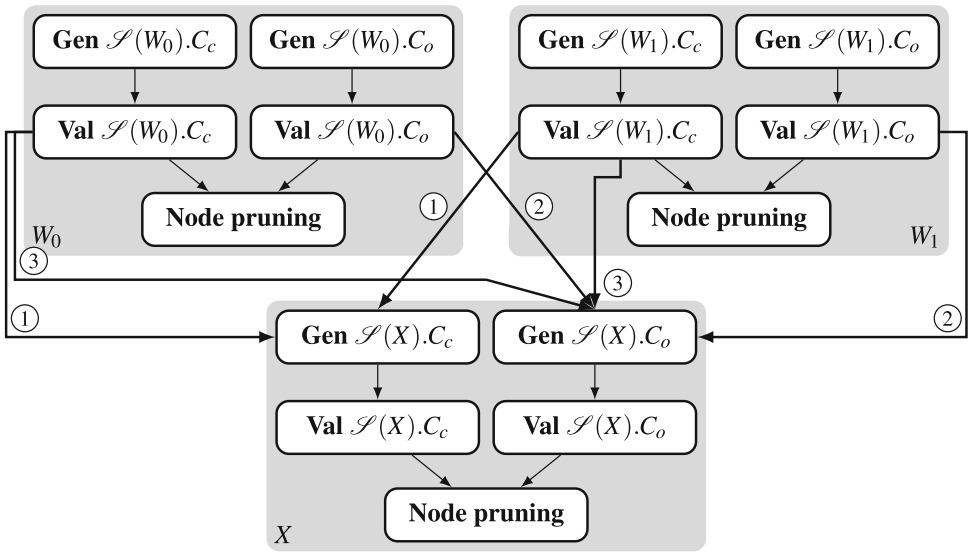


Fig. 3 Inter- and intra-node dependencies for the subtasks of a node X and its two predecessor nodes W_0 and W_1 with $W_i = X \setminus \{\mathsf{X}_i\}$. The arrow on an edge points to the subtask that depends on the subtask at the source of the edge

of predecessor nodes $W = \{W_i = X \setminus \{\mathsf{X}_i\}\}$ have been validated; ① indicates this dependency. To guarantee the minimality of the order compatible candidates $\mathscr{S}(X).C_o$ of node X, the `Master` module has to follow Rule 3b, Rule 3c, and Rule 3d. Rule 3b requires the results of the order compatible bOD validations of all W_i, which is expressed by ②. Rule 3c and Rule 3d require the results of the constant bOD validations of all W_i indicated by ③. $\mathscr{S}(X).C_o$ can thus be generated as soon as both constant and order compatible bOD candidates of all predecessor nodes W_i have been validated. The validations of constant bODs and order compatible bODs are independent of each other and can be performed concurrently as soon as the respective candidates are fully generated. If both validation checks for node X are finished, the `Master` module can check X's candidate states $\mathscr{S}(X).C_c$ and $\mathscr{S}(X).C_o$ to decide if the node and all its successors can be pruned from the lattice (node pruning).

In contrast to FASTOD-BID [25], where the generation of constant and order compatible bOD candidates happens after the node pruning subtasks of *all* previous level's nodes are done, DISTOD's candidate generation steps, which are the Gen $\mathscr{S}(X).C_c$ and Gen $\mathscr{S}(X).C_o$ boxes in Fig. 3, do not depend on the node pruning step. In this way, DISTOD proactively generates constant bOD validation jobs; some of which may be pruned once the order compatible validations of the predecessors are done. This does not violate the minimality of the discovered bODs, because the job cannot contain valid candidates anyway. We deliberately interlace the candidate generation subtasks and validate them independently from each other in the candidate validation steps, because this removes synchronization barriers and allows for a more fine-grained work distribution improving the resource utilization on all nodes; DISTOD distributes the candidate validations as encapsulated jobs, which are the Val $\mathscr{S}(X).C_c$ and Val $\mathscr{S}(X).C_o$ boxes in Fig. 3, which are pulled and processed by the `Worker` actors on the follower nodes. Node pruning is handled downstream.

5.3 Candidate generation algorithm

The candidate generation is handled by the `Master` module on the leader node (cf. Fig. 2). For better performance, the `Master` module is parallelized and consists of two types of actors: a single `Master` and a pool of `MasterHelpers`.

Lattice structure To ensure consistency, the `Master` is the sole actor that can manipulate the candidate state $\mathscr{S}$ and the validation job queue **Q**. The `MasterHelper` actors have read-only access to the candidate state $\mathscr{S}$, perform the actual generation of new bOD candidates and check the minimality pruning rules. This allows us to parallelize the candidate generation, which reduces the load on the central `Master`. For each node X in the candidate lattice, we store an entry $\{C_c, C_o, f_c, f_o, p, i_c, i_o\}$ in the candidate state $\mathscr{S}$: The entry's constant and order compatible bOD candidate sets $\mathscr{S}(X).C_c$ and $\mathscr{S}(X).C_o$ serve to track untested and invalid candidates as described in Sect. 5.2. The two flags $\mathscr{S}(X).f_c$ and $\mathscr{S}(X).f_o$ indicate whether or not $\mathscr{S}(X).C_c$ and $\mathscr{S}(X).C_o$ have already been validated by `Workers`. The flag $\mathscr{S}(X).p$ indicates whether the node as a whole is pruned. Lastly, the two counters $\mathscr{S}(X).i_c$ and $\mathscr{S}(X).i_o$ track the necessary preconditions for the candidate generation in node X, i. e., the number of predecessor nodes W_i, for which the constant ($\mathscr{S}(W_i).f_c = \text{TRUE}$) and order compatible ($\mathscr{S}(W_i).f_o = \text{TRUE}$) bOD validations have already been performed.

They allow the `MasterHelpers` to enforce the dependencies shown in Fig. 3, because they trigger the generation of candidates not before all preconditions are met, which is

Algorithm 1: `generateCandidates(`X`)`

Data: States $\mathscr{S}$, job queue **Q**

1 **if** $|X| \geq 2 \wedge \mathscr{S}(X).i_c = |X|$ **then**
2 $\quad c_c = \bigcap_{\mathsf{A} \in X} \mathscr{S}(X \setminus \{\mathsf{A}\}).C_c$
3 $\quad$ send job (X, c_c) to `Master`
$\quad$ `// Master:` $\mathscr{S}(X).C_c = c_c$ `and add` (X, c_c) `to` **Q**
4 **if** $|X| \geq 2 \wedge \mathscr{S}(X).i_c = |X| \wedge \mathscr{S}(X).i_o = |X|$ **then**
5 $\quad$ **if** $|X| = 2$ **then**
6 $\quad\quad c_o = \{(\mathsf{A}\uparrow, \mathsf{B}\uparrow), (\mathsf{A}\uparrow, \mathsf{B}\downarrow)\}$, where $X = \{\mathsf{A}, \mathsf{B}\}$
7 $\quad$ **else**
8 $\quad\quad c_o = \bigcup_{\mathsf{A} \in X} \mathscr{S}(X \setminus \{\mathsf{A}\}).C_o$
9 $\quad\quad$ **forall the** $(\bar{\mathsf{E}}, \bar{\mathsf{F}}) \in c_o$, *where* $(\bar{\mathsf{E}}, \bar{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\uparrow)$ *or* $(\mathsf{A}\uparrow, \mathsf{B}\downarrow)$ **do**
10 $\quad\quad\quad Y = X \setminus \{\mathsf{A}, \mathsf{B}\}$
11 $\quad\quad\quad$ **if** $\exists \mathsf{C} \in Y : (\bar{\mathsf{E}}, \bar{\mathsf{F}}) \notin \mathscr{S}(X \setminus \{\mathsf{C}\}).C_o$ **then**
12 $\quad\quad\quad\quad$ remove $(\bar{\mathsf{E}}, \bar{\mathsf{F}})$ from c_o
13 $\quad$ **forall the** $(\bar{\mathsf{E}}, \bar{\mathsf{F}}) \in c_o$, *where* $(\bar{\mathsf{E}}, \bar{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\uparrow)$ *or* $(\mathsf{A}\uparrow, \mathsf{B}\downarrow)$ **do**
14 $\quad\quad$ **if** $\mathsf{A} \notin \mathscr{S}(X \setminus \{\mathsf{B}\}).C_c \vee \mathsf{B} \notin \mathscr{S}(X \setminus \{\mathsf{A}\}).C_c$ **then**
15 $\quad\quad\quad$ remove $(\bar{\mathsf{E}}, \bar{\mathsf{F}})$ from c_o
16 $\quad$ send job (X, c_o) to `Master`
$\quad$ `// Master:` $\mathscr{S}(X).C_o = c_o$ `and add` (X, c_o) `to` **Q**

when $\mathscr{S}(X).i_c = |X|$ and $\mathscr{S}(X).i_o = |X|$, respectively. Every successful validation triggers i_c and i_o counter increments in all dependent nodes of X and with them the check if a node is ready to generate either C_c or C_o candidates. If $\mathscr{S}(X).i_c = |X|$, a `MasterHelper` reactively checks Rule 2b to initiate the generation of minimal constant bOD candidates for node X (Algorithm 1 Line 1f.); if $\mathscr{S}(X).i_o = |X|$ and $\mathscr{S}(X).i_c = |X|$, a `MasterHelper` checks Rule 3b, Rule 3c, and Rule 3d to generate minimal order compatible bOD candidates for node X (Algorithm 1 Line 4ff.). Although DISTOD checks whether candidates can be generated once per counter increment, the rule testing and actual generation of minimal bOD candidates is done only once per node. The job queue **Q** of the `Master` actor tracks the encapsulated validation jobs (the Val $\mathscr{S}(X).C_c$ and Val $\mathscr{S}(X).C_o$ boxes in Fig. 3) for the `Worker` actors.

Lattice initialization The state initialization is performed by the `Master` actor right after reading the input dataset. The state of the sole level l_0 node $\mathscr{S}(\{\})$ is initialized by setting $C_c = \mathrm{R}$, $C_o = \emptyset$, and $f_c = f_o = \mathrm{TRUE}$ (no validations to perform). Level l_1 is the first level that contains bOD candidates. For all $\mathsf{A} \in \mathrm{R}$, the `Master` sets $\mathscr{S}(\{\mathsf{A}\}).i_c = \mathscr{S}(\{\mathsf{A}\}).i_o = 1$, $\mathscr{S}(\{\mathsf{A}\}).C_c = \mathrm{R}$, $\mathscr{S}(\{\mathsf{A}\}).C_o = \emptyset$, $\mathscr{S}(\{\mathsf{A}\}).f_o = \mathrm{TRUE}$, and adds the initial validation jobs $(\{\mathsf{A}\}, \mathscr{S}(\{\mathsf{A}\}).C_c) \forall \mathsf{A} \in \mathrm{R}$ to **Q**, which effectively starts the discovery. Because l_1 includes only single-attribute nodes and, hence, no order compatible bOD candidates, the initialization also has to set the precondition counters $\mathscr{S}(X).i_o$ for all nodes in level l_2 to 2.

Validation job dispatching DISTOD uses work pulling to distribute validation jobs, which means that `Worker` actors once they finished a job immediately request a new job from the `Master`. For each request, the `Master` dequeues a validation job (X, C_c) (or (X, C_o)) from **Q**, removes trivial *constant* bOD candidates from C_c, and dispatches the job to the `Worker` for validation (Sect. 6). We cannot remove trivial candidates from $\mathscr{S}(X).C_c$ directly, because they might be required to generate candidates of succeeding nodes.

If **Q** is empty, the `Master` bookmarks the requesting `Worker`. As soon as new jobs are put into **Q**, bookmarked `Workers` are served with jobs again. Once all `Workers` are idle and **Q** is empty, there are no more nodes with minimal bOD candidates in the lattice and DISTOD is finished.

Validation result processing The `Worker` actors send the validation results for a candidate set C_c (or C_o respectively) back to the `Master` actor (via the `MasterHelpers`). The `Master` then updates the corresponding node's state $\mathscr{S}(X)$ by setting f_c (or f_o) to TRUE and removing all pruned candidates from C_c (or C_o). Once both validations have been performed ($f_c = f_o = \mathrm{TRUE}$), the `Master` checks if the node can be pruned from the candidate lattice (*node pruning*). If this is not the case, it updates the precondition counters of all successor nodes of X. For this, the `Master` iterates over all nodes $Z_i = X \cup \mathsf{S}_i$ with $\mathsf{S}_i \in \mathrm{R} \setminus X$ that have not been pruned yet and increments the precondition counter $\mathscr{S}(Z_i).i_c$ (or $\mathscr{S}(Z_i).i_c$). Non-existing node states $\mathscr{S}(Z_i)$ are dynamically created during this step and added to the lattice. After the counters of all successor nodes have been updated, the `Master` generates candidate generation jobs for all non-pruned successor nodes Z_i. These candidate generation jobs are sent to the `MasterHelper` actors in a round-robin fashion and are processed concurrently using Algorithm 1. Because the `MasterHelpers` cannot modify $\mathscr{S}$ and **Q** directly, they send the newly generated candidates and the new jobs back to the `Master` actor, which integrates them into $\mathscr{S}$ and **Q**.

Node pruning If a node X is prunable ($\mathscr{S}(X).C_c = \emptyset \wedge \mathscr{S}(X).C_o = \emptyset$), the `Master` actor marks the node X and all existing successors Z_i as pruned by setting their flag $p =$ TRUE. Then, it removes all related validation jobs from **Q**.

6 Candidate validation

DISTOD validates bOD candidates similar to FASTOD-BID [25] using data partitions. More specifically, it uses stripped partitions to validate constant bODs and a combination of stripped and sorted partitions to validate order compatible bOD. In contrast to FASTOD-BID, though, DISTOD uses a slightly optimized algorithm to validate order compatible bODs and faces an additional challenge in partition management, because the candidate validations are distributed across different nodes in the cluster.

In this section, we first introduce sorted and stripped partitions. We, then, explain how we validate constant and order compatible bOD candidates.

6.1 Sorted and stripped partitions

Partitions Π_X are sets of equivalence classes w. r. t. a context X (see Sect. 3.2 Definition 4). Similar to FASTOD-BID, DISTOD does not directly use these full partitions for the candidate validation checks, because they take a lot of memory and lack information about the order of the tuples in the dataset required to check order compatible bODs. Instead, DISTOD uses two variations of these partitions: *sorted partitions*, which capture the order of the tuples and, thus, enable the validation of order compatible bODs, and *stripped partitions*, which remove implicit information and, in this way, reduce the memory footprint needed to store partitions.

Sorted partitions Sorted partitions are necessary for the validation of order compatible bODs because they preserve the ordering information of the input dataset.

Definition 14 A sorted partition, denoted as $\widehat{\Pi}_X$, is a list of equivalence classes sorted by the ordering imposed to the tuples by X [25].

DISTOD's order compatible bOD validation algorithm performs only a single operation on sorted partitions. It looks up the positions of two given tuples to determine their order (see Sect. 6.2.2). We propose a reversed mapping of the sorted partitions, called *inverted sorted partition* Γ_X, to represent sorted partitions in DISTOD. Inverted sorted partitions allow us to lookup the position of a tuple identifier in a sorted partition in constant time.

Definition 15 An *inverted sorted partition* Γ_X is a mapping from tuple identifiers to the positions of their equivalence classes in a sorted partition $\widehat{\Pi}_X$.

To give an example, consider Table 1 and the partition $\Pi_{\{\texttt{Code}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_2\}, \{t_6\}, \{t_8\}\}$. The sorted partition for $X = \{\texttt{Code}\}$ is $\widehat{\Pi}_{\{\texttt{Code}\}} = [\{t_6\}, \{t_0, t_4\}, \{t_2\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_8\}]$. We store this sorted partition as inverted sorted partition $\Gamma_{\{\texttt{Code}\}} = \{t_0 \to 1,\ t_1 \to 4,\ t_2 \to 3, t_3 \to 4, t_4 \to 1, t_5 \to 4, t_6 \to 0, t_7 \to 4, t_8 \to 5, t_9 \to 4\}$.

DISTOD's validation checks require only inverted sorted partitions for the singleton attribute sets, where $|X| = 1$ (see Sect. 6.2.2). This means that DISTOD can compute the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ directly from the individual columns of the dataset. After the inverted sorted partitions have been generated, we work with tuple identifiers only. This has the advantage that we can discard the attribute type information and all concrete values, which saves memory and—because the computations effectively deal with integers only—makes the operations on partitions fast and simple.

Stripped partitions DISTOD requires only the sorted partitions for each $\{\mathsf{A}\} \in \mathrm{R}$ (level l_1 of the candidate lattice); for attribute sets with $|X| > 1$ (higher levels), it replaces sorted partitions with the smaller stripped partitions [5,10,12,22,25,26] (also known as *position list indexes* [2,15,19,20]).

Definition 16 Stripped partitions are partitions, where singleton equivalence classes with $|\mathscr{E}(t_X)| = 1$ are removed [10]. We denote them with Π^*_X.

Coming back to our example partition $\Pi_{\{\texttt{Code}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_2\}, \{t_6\}, \{t_8\}\}$, we can transform it into a stripped partition by removing all singleton equivalence classes such that $\Pi^*_{\{\texttt{Code}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}\}$. For the proposed bOD validation algorithms (see Sect. 6.2) using *stripped* instead of full partitions is sufficient because this also guarantees correctness [25].

DISTOD uses two different approaches to generate stripped partitions: one for attribute sets of level l_1 and another one for attribute sets of deeper levels. Stripped partitions $\Pi^*_{\{\mathsf{A}\}}$ for all single attribute sets $\{\mathsf{A}\} \in \mathrm{R}$ (level l_1) are generated from the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ by converting them back to sorted partitions $\widehat{\Pi}_{\{\mathsf{A}\}}$ and simply removing all singleton equivalence classes.

The partitions for larger attribute sets X, where $|X| \geq 2$, can efficiently be computed from two of their subsets using the product of refined partitions. The partition product is "the least refined partition $\Pi_{X \cup Y}$ that refines both Π_X and Π_Y" [10]. Stripped partition $\Pi^*_{\{\mathsf{A},\mathsf{B}\}}$ in level l_2, for example, is computed by the stripped partition product of $\Pi^*_{\{\mathsf{A}\}}$ and $\Pi^*_{\{\mathsf{B}\}}$: $\Pi^*_{\{\mathsf{A},\mathsf{B}\}} = \Pi^*_{\{\mathsf{A}\}} \cdot \Pi^*_{\{\mathsf{B}\}}$. Any two different subsets of size $|X| - 1$ of a stripped partition for X suffice for the stripped partition product. This fits well for our small-to-large search strategy and gives us flexibility in choosing the operands for the stripped partition product.

6.2 Validation algorithm

As discussed in Sect. 5, DISTOD generates only minimal and non-pruned bOD candidates. For each node X in the candidate lattice, the algorithm generates constant bOD candidates of the from $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A}\uparrow$ for all $\mathsf{A} \in X$ and order compatible bOD candidates of the form $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A}\uparrow \sim \mathsf{B}\uparrow$ and $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A}\uparrow \sim \mathsf{B}\downarrow$ for all $\mathsf{A}, \mathsf{B} \in X$, where $\mathsf{A} \neq \mathsf{B}$. For the validation, the constant bOD candidates and the order compatible bOD candidates of a node X are grouped together. Each group is distributed as a validation job to one of the `Worker` actors. The constant bOD candidates are validated using a partition refinement check on stripped partitions (see Sect. 6.2.1) and the order compatible bOD candidates are validated by comparing the ordering of tuples that is imposed by their first attribute (A) and their

second attribute (B) of the bOD as represented in their sorted partition (see Sect. 6.2.2).

6.2.1 Validating constant bODs

Constant bODs of the form $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A}\uparrow$ resemble FDs. Hence, they can be validated using a partition refinement check [10,25]. A partition Π refines another partition Π' if the equivalence classes in Π are all subsets of any of the equivalence classes in Π'. The partition refinement check on stripped partitions can efficiently be computed using the popular error measure $e(Y) = (\|\Pi_Y^*\| - |\Pi_Y^*|)/|r|$ from the TANE algorithm [10], where $|\Pi_Y^*|$ is the number of equivalence classes of the stripped partition and $\|\Pi_Y^*\|$ is the sum of the sizes of all equivalence classes in Π_Y^*. Partition Π_X refines partition $\Pi_{\{\mathsf{A}\}}$ if and only if $e(X) = e(X \cup \{\mathsf{A}\})$.

A constant bOD candidate of the from $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A}\uparrow$ is valid if the stripped partition $\Pi^*_{X\setminus\{\mathsf{A}\}}$ refines $\Pi^*_{\{\mathsf{A}\}}$ (no *split*; see Definition 6). We check this condition using the error measure: If $e(X \setminus \{\mathsf{A}\}) = e(X)$, the bOD is valid; if $e(X \setminus \{\mathsf{A}\}) \neq e(X)$, the bOD is invalid. Because computing the error measure on demand would require a scan over the stripped partition, i. e., a scan for each constant bOD candidate check, DISTOD stores for each stripped partition Π_X^* its number of equivalence classes $|\Pi_X^*|$ and its number of elements $\|\Pi_X^*\|$. The divisor $|r|$ is constant for all checks and can, thus, be removed. The algorithm calculates $|\Pi_X^*|$ and $\|\Pi_X^*\|$ during the generation of the respective stripped partition because it has to scan the partition product during this operation anyway. In this way, the candidate check consists of only three operations: two subtractions to compute the errors $e(X \setminus \{\mathsf{A}\})$ and $e(X)$, respectively, and a comparison of the two error values, i. e., $e(X \setminus \{\mathsf{A}\}) \stackrel{?}{=} e(X)$.

6.2.2 Validating order compatible bODs

Order compatible bOD candidates of the form $X \setminus \{\mathsf{A}, \mathsf{B}\} : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ are also validated using partitions. For this, we slightly change the validation algorithm from FASTOD-BID [25] to improve its efficiency: To verify whether there is no *swap* over the attributes A and B, FASTOD-BID's validation algorithm scans over the (large) sorted partition $\widehat{\Pi}_{\{\mathsf{A}\}}$ and over the (small) stripped context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$; our changed version scans only the (small) stripped context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$ twice. Sorted partitions always contain all tuples of the input dataset, while stripped partitions get smaller for larger attribute sets due to the partition refinement, i. e., the number of singleton equivalence classes in Π_X grows with the number of attributes in X and stripped partitions Π_X^* omit these classes. Figure 4 illustrates the rapid size reduction of stripped partitions over the levels of seven datasets.

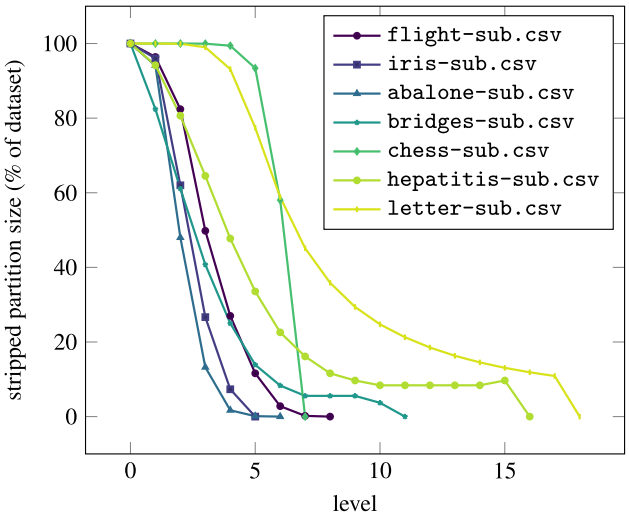


Fig. 4 Average relative size of stripped partitions $\left(\frac{\|\Pi_X^*\|}{|r|}\right)$ per level l_i ($|X| = i$) for different datasets

Algorithm 2 shows DISTOD's steps to validate an order compatible bOD candidate. It first sorts the equivalence classes of the stripped context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$ by the first attribute A of the bOD (Line 1) and then compares this order to the order imposed by the second attribute B (Line 3-16). The algorithm checks for candidates of the from $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A}\uparrow \sim \mathsf{B}\uparrow$ and $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A}\uparrow \sim \mathsf{B}\downarrow$ simultaneously. If it finds no swap in the data, the order compatible bOD $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A}\uparrow \sim \mathsf{B}\uparrow$ is valid (Line 17f). Analogously, if it finds no reverse swap in the data, the order compatible bOD $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A}\uparrow \sim \mathsf{B}\downarrow$ is valid (Line 17f). If neither a swap nor a reverse swap is found, both order compatible bOD forms are valid.

In Line 1 of Algorithm 2, we call Algorithm 3 to sort the tuples in the equivalence classes of the stripped context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$ by the first attribute A. Algorithm 3 first iterates over all equivalence classes $\mathscr{E}$ of the context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$ (Line 2). For each $\mathscr{E}$, it creates a new temporary list to store the sorted equivalence classes (Line 3). The algorithm then iterates over all tuples of the equivalence class $\mathscr{E}$ and retrieves their positions when being sorted by attribute A using the inverted sorted partition $\Gamma_{\{\mathsf{A}\}}$ (Line 4f). We store the tuple identifiers in the sorted map $\mathscr{M}$ with their position pos_t as the key. $\mathscr{M}$ stores key-value pairs and allows us to traverse the values in key order. DISTOD uses this property to add the tuple sets in sorted order to their new sorted equivalence class **E** in Line 8f. Afterward, it stores the sorted equivalence class in the output set γ (Line 10) and clears $\mathscr{M}$ to process the next equivalence class $\mathscr{E}$ of $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$ (Line 11). As an example, consider the stripped context partition $\Pi^*_{\{\texttt{Code}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}\}$ and the inverted sorted partition $\Gamma_{\{\texttt{ADGrp}\}} = \{t_0 \to 5,\ t_1 \to 2,\ t_2 \to 3,\ t_3 \to 4,\ t_4 \to 0,\ t_5 \to 2,\ t_6 \to 1,\ t_7 \to 6,\ t_8 \to 3,\ t_9 \to 2\}$.

Algorithm 2: Validate order compatible bOD

Input : bOD candidate $X \setminus \{A, B\} : \overline{A} \sim \overline{B}$, context partition $\Pi^*_{X\setminus\{A,B\}}$, inverted sorted partitions $\Gamma_{\{A\}}$ and $\Gamma_{\{B\}}$

$\gamma_A = \texttt{sortECs}(\Pi^*_{X\setminus\{A,B\}}, \Gamma_{\{A\}})$
$swap = rSwap =$ FALSE
forall the $\mathbf{E} \in \gamma_A$, *where* $|\mathbf{E}| \geq 2$ *if* $swap = rSwap =$ FALSE **do**
  for $i = 0$ *until* $|\mathbf{E}| - 1$ *if* $swap = rSwap =$ FALSE **do**
    $F = \mathbf{E}[i]$
    $G = \mathbf{E}[i+1]$
    $max_F = max_G = 0$
    $min_F = min_G =$ MAX_INT
    forall the $t \in F$ **do**
      if $\Gamma_{\{B\}}(t) > max_F$ **then** $max_F = \Gamma_{\{B\}}(t)$
      if $\Gamma_{\{B\}}(t) < min_F$ **then** $min_F = \Gamma_{\{B\}}(t)$
    forall the $t \in G$ **do**
      if $\Gamma_{\{B\}}(t) > max_G$ **then** $max_G = \Gamma_{\{B\}}(t)$
      if $\Gamma_{\{B\}}(t) < min_G$ **then** $min_G = \Gamma_{\{B\}}(t)$
    if $max_F > min_G$ **then** $swap =$ TRUE
    if $max_G > min_F$ **then** $rSwap =$ TRUE
if $swap =$ FALSE **then**
  emit $X \setminus \{A, B\} : A\uparrow \sim B\uparrow$ as valid bOD
if $rSwap =$ FALSE **then**
  emit $X \setminus \{A, B\} : A\uparrow \sim B\downarrow$ as valid bOD

Algorithm 3: $\texttt{sortECs}(\Pi^*_{X\setminus\{A,B\}}, \Gamma_{\{A\}})$

Data: sorted map (e. g., red-black tree) $\mathcal{M}$

$\gamma = \emptyset$
forall the $\mathscr{E} \in \Pi^*_X$ **do**
  $\mathbf{E} = []$
  forall the $t \in \mathscr{E}$ **do**
    $pos_t = \Gamma_{\{B\}}(t)$
    if $\mathcal{M}(pos_t) = null$ **then** $\mathcal{M}(pos_t) = \{\}$
    add t to $\mathcal{M}(pos_t)$
  forall the $\mathscr{E}_{new} \in \mathcal{M}$ **do** /* traversal in key order */
    add $\mathscr{E}_{new}$ to $\mathbf{E}$
  add $\mathbf{E}$ to γ
  clear $\mathcal{M}$
return γ

Sorting the equivalence classes in $\Pi^*_{\{Code\}}$ by ADGrp using Algorithm 3 results in $\gamma = \{[\{t_4\}, \{t_0\}], [\{t_1, t_5, t_9\}, \{t_3\}, \{t_7\}]\}$.

7 Distributed partition management

DISTOD manages data in different parts of the system. The Master actor on the leader node keeps track of all checked and unchecked bOD candidates and the search progress. This includes the status of specific bOD candidates, pending tasks, and pruning information. The Master actor also stores work queues to track waiting and pending validation jobs. It sends out the candidates as validation jobs to the nodes in the cluster and receives the intermediate results and pruning information back to integrate them into its state. The ResultCollector actor on the leader node manages the results of the discovery algorithm; more specifically, it receives all valid bODs from all nodes in the DISTOD cluster and writes them to disk for persistence. It is also responsible for removing duplicate results, which could occur if a follower node is removed from the DISTOD cluster and its unfinished tasks are dispatched to another node. Managing the candidate data and search state is easy because it is centralized on the leader node. The handling of the sorted and stripped partitions, however, is a challenge because the partitions are required on different follower nodes to perform the candidate validations and the set of stripped partitions grows exponentially while the algorithm generates ever-more partitions for further candidate validations. For this reason, we focus on distributed partition management in this section.

7.1 Partition handling

DISTOD distributes the bOD candidate validations as jobs to the Worker actors across the nodes in the cluster on a "first come, first served" basis. This helps to facilitate elasticity (see Sect. 8.1) because every node in the cluster can receive any validation job and we do not have to adapt our distribution strategy when nodes join or leave the cluster. However, different validation candidates require different partitions and, due to the dynamic distribution, we do not know a priori which node will receive which validation candidates. Therefore, DISTOD manages the partitions for each node individually and locally. Each DISTOD node stores and generates its own partitions and no node must hold all partitions because only the partitions relevant to the locally performed checks are generated. Based on an initial set of partitions, which is replicated across the nodes in the cluster, each node can generate all other stripped partitions on demand.

The *initial partitions* are generated directly from the input dataset. They include the inverted sorted partitions $\Gamma_{\{A\}}$ for each $A \in R$ and the stripped partition $\Pi^*_{\{\}}$. In DISTOD, only the leader node reads the input dataset and, thus, the leader node creates the initial partitions (i. e., the DataReader actors). After a follower node connects to the leader node, it replicates the initial partitions once and generates all other stripped partitions locally on demand.

Depending on the characteristics of the input dataset, the initial partitions can get quite large. If we would send these amounts of data using the default message channel, they may hinder or delay other messages from being sent and received. This could include time-critical messages, such as heartbeats, or other important messages, such as cluster membership updates and gossip. To prevent such message collisions, DISTOD uses message side-channels between all nodes in the DISTOD cluster for any large message transfers. A side-channel handles the streaming of the chunked

initial partitions over a separate, low-priority back-pressured communication channel.

The partition management and partition generation are implemented in the partition management module. Each node in the cluster has its own `PartitionMgr` actor, which stores the inverted sorted partitions $\Gamma_{\{A\}}$ for each $A \in R$, the l_1 stripped partitions and the stripped partition for the empty candidate set $\Pi^*_{\{\}}$. The `PartitionMgr` also serves as a cache for the temporary stripped partitions Π^*_X for $|X| \geq 2$. We cache the intermediate stripped partitions in the `PartitionMgr` because they are used to generate stripped partitions for larger attribute sets and different candidate validation checks can rely on the same partitions. The local `Workers` can start requesting partitions from their local `PartitionMgr` as soon as all initial partitions have been replicated; missing stripped partitions are generated from the existing ones on demand.

All partitions stored in the `PartitionMgr` are immutable and, thus, `Workers` can safely access the same partition concurrently. If a `Worker` has to manipulate a partition, e. g., to sort its equivalence classes, it uses a private working copy of the stripped partition.

Algorithm 4: Recursive partition generation job calculation

Input : target attribute set X
partition map $\mathscr{P}$
Output: sequence of partition generation jobs **J**
Data: store depth $d = 3$, job chain **J** = [ ]

```
1  Function calcJobChainRecursive(Y = X)
2      if Y in J then return
3      s = |Y| >= |X| − d
4      W = [W_i | W_i = X \ {A_i}]
5      sort W
6      partition predecessor attribute sets W_i ∈ W into
       W_hit where W_i ∈ P and
       W_miss where W_i ∉ P
7      if |W_hit| = 0 then                    /* no Π*_Wi in P */
8          calcJobChainRecursive (W_miss[0])
9          calcJobChainRecursive (W_miss[1])
10         Add Y → (W_miss[0], W_miss[1], s) to J
11     else if |W_hit| = 1 then               /* one Π*_Wi in P */
12         calcJobChainRecursive (W_miss[0])
13         Π*_y = P(W_hit[0])
14         Add Y → (Π*_y, W_miss[0], s) to J
15     else                                   /* at least two Π*_Wi in P */
16         Π*_y0 = P(W_hit[0])
17         Π*_y1 = P(W_hit[1])
18         Add Y → (Π*_y0, Π*_y1, s) to J
```

7.2 On demand stripped partition generation

DISTOD uses two alternative strategies for the local generation of stripped partitions: *recursive generation* and *direct partition product*. This section introduces both strategies.

7.2.1 Recursive partition generation

The node-local and on-demand generation of stripped partitions and DISTOD's distribution of the candidate validation checks to different nodes entails an irregular generation of stripped partitions: It is possible that a `Worker` receives a task where it requires a stripped partition, for which not all preceding partitions have been generated by the local `PartitionMgr`. Thus, the requested partition cannot be computed using the partition product of two of its subsets. This effect is amplified by the regular partition cleanup of the `PartitionMgr` and by partition eviction processes in the case of memory shortage (cf. Sect. 7.3). We overcome the issue of missing subset partitions by recursively computing partitions in a way that makes the best use of already available intermediate partitions.

For each partition request that cannot be served by the partition cache, the `PartitionMgr` recursively generates a chain of partition generation jobs for the `PartitionGen` actors. This job chain records the order of the partition generation jobs and the particular inputs for each job in the chain. A job chain is sent to a single `PartitionGen` actor, which processes it from the beginning to the end. The `PartitionGen` actor temporarily stores the generated partitions and can use them as input for the next partition generation job. If a newly generated partition should be stored in the partition cache, the `PartitionGen` actor sends the partition to its local `PartitionMgr`, which also forwards the partition to the requesting `Worker`.

Algorithm 4 shows the recursive function that creates a chain of partition generation jobs. It takes the target attribute set X and all stored partitions $\mathscr{P}$ as input and returns a list of partition generation jobs **J**. A partition generation job (e. g., in Line 10) consists of the attribute set for the target partition to generate, the two input partitions for the partition product and a flag that tells the `PartitionGen` actor whether this partition should be stored or not s. We store all partitions up to a depth of three. This is a compromise between not storing predecessors and storing all intermediate partitions. By storing all intermediate partitions, partition generations can be computed much faster, but the cache size would quickly outgrow any main memory capacity due to its exponential growth. By storing only the target partition, on the other hand, we would use a lot less memory, but had to compute common intermediate partitions for the following partition requests as well.

The two input partitions of a partition generation job are either the stripped partitions Π^*_X themselves or the identifiers X of the partitions. If only the identifiers are specified, the `PartitionGen` actor looks up the stripped partitions in its temporary partition state and uses the looked up partitions as input for the partition product. The partition generation job

chain ensures that all necessary stripped partitions are computed before they are used as input for the partition product and that no partition is generated multiple times by the same `PartitionGen` actor (see Line 2 in Algorithm 4).

Algorithm 4 generates a minimal and deterministic number of jobs. This is because the algorithm consistently chooses the partition product factors from the candidate set's predecessors in a left-oriented way utilizing the cached partitions optimally. For each recursion step, the predecessors of the current candidate set are sorted lexicographically (see Line 5 in Algorithm 4). If no predecessors are available (see Line 7), the algorithm generates the first two predecessor partitions recursively for the partition product; if only one predecessor is available (see Line 11), this predecessor partition is taken and the first non-available predecessor partitions are generated recursively; if more predecessor partitions exist (see Line 15), the algorithm takes the two first predecessor partitions. Any following partition generation run also uses this left-orientation and, in this way, automatically re-use the previously generated partitions. This cuts down the number of generation steps and, thus, reduces the time spent generating partitions.

7.2.2 Direct partition product

The recursive generation of stripped partitions is, in general, the fastest way of computing new partitions because the operation re-uses existing stripped partitions that usually become smaller for larger attribute sets. It also caches intermediate results to accelerate later partition retrieval and generation operations. However, if DISTOD's heap memory usage exceeds a certain threshold, the algorithm's partition eviction mechanism (see Sect. 7.3) removes all intermediate stripped partitions from the partition cache of the `PartitionMgr` so that later partitions have to be recursively generated from the partitions in level l_1. This is very costly because if there are no intermediate partitions the number of partition generation jobs needed for the recursive generation grows exponentially with the level in which a partition is requested. In addition, storing the intermediate partitions again would lead to a repeated memory exhaustion. For these reasons, we dynamically switch to a different partition generation method, which is *direct partition product*, whenever the chain of recursive partition generation jobs becomes too long. This strategy computes the partition product for a stripped partition not only from its immediate predecessors, but also from other subsets. In our case, we use the single attribute set partitions from level l_1 because they are always available.

As an example, consider the request for partition $\Pi^*_{\{A,B,C\}}$ and an empty partition cache. Instead of computing the partition product recursively via $\Pi^*_{\{A,B,C\}} = \left(\Pi^*_{\{A\}} \cdot \Pi^*_{\{B\}}\right) \cdot \left(\Pi^*_{\{A\}} \cdot \Pi^*_{\{C\}}\right)$, the direct partition product uses the persistent l_1 stripped partitions to directly, i. e., without intermediate results, compute the requested stripped partition: $\Pi^*_{\{A,B,C\}} = \Pi^*_A \cdot \Pi^*_B \cdot \Pi^*_C$. In this example, the recursive partition product would compute three partition products in three jobs with two intermediate partitions, namely $\Pi^*_{\{A,B\}}$ and $\Pi^*_{\{A,C\}}$. The direct partition product would compute only two partition products, but within one job, with no intermediate partitions.

Because the recursive generation of stripped partitions is much faster if the memory can hold the intermediate partitions and the partition product chains are not too long, DISTOD uses this strategy by default. The algorithm dynamically switches to the direct partition product if it has to generate a lot of intermediate stripped partitions. The threshold at which DISTOD switches from the recursive to the direct strategy is exposed as a parameter with 15 as default value.

7.3 Dealing with limited memory

The main memory resources in a cluster are usually limited. Hence, DISTOD has to effectively manage its memory consumption: If DISTOD allocates memory too aggressively, i. e., up to the limit, the Java garbage collector takes up most of the processing time slowing down the actual algorithm; if it exceeds the memory limit, the discovery fails and terminates. Working close to the memory limit cannot be prevented on large datasets, which is why we need to handle DISTOD's memory consumption carefully.

For the majority of datasets and especially for long ones, the partitions of the `PartitionMgr` take up most of the memory. As the discovery progresses, the recursive partition generation expands the size of the partition cache exponentially by adding ever more stripped partitions to it. Because any required partitions in level l_i, where $i \geq 2$, can always be computed on-demand from the stripped partitions in level l_1, we can optimize the time that intermediate stripped partitions are kept in the cache. For this, we propose two strategies: *periodic partition cleanup* and *partition eviction*.

7.3.1 Periodic partition cleanup

Most stripped partitions are needed for only a short period of time, e. g., before their successors make them obsolete. Only the initial partitions need to be preserved to enable the validation of order compatible bOD candidates and the regeneration of all other stripped partitions. The intermediate stripped partitions in deeper levels can be deleted when they are not needed anymore. However, the point in time, at which the partitions are no longer needed, is hard to predict because a single partition might be involved in different bOD candidate validations and only the `Master` actor on the leader node knows which candidates have already been processed and which candidates are next in the work queue. For this

reason, our PartitionMgr tracks the number of accesses for each partition to estimate the relevance of any locally cached partition. It then periodically removes not-recently-used partitions from its partition cache.

The periodic partition cleanup protocol is run by each PartitionMgr in the cluster: Every PartitionMgr tracks the accesses to its cached stripped partitions. The scheduler of the local actor system then periodically sends a tick message to the PartitionMgr. When the PartitionMgr receives a tick message, it removes all partitions from the partition cache that have not been accessed since the last tick was received and resets its internal access statistics.

In this protocol, the tick frequency defines the minimum lifetime of a partition because the tick interval of the partition's creation does not see the partition and the initial generation of the partition is triggered by accessing it so that its initial access counter is 1. Therefore, short intervals cause a more aggressive removal behavior and long intervals keep partitions in the cache for longer. Overall, the tick frequency trades off memory consumption and runtime because the removal of stripped partitions slows DISTOD down. For maximum performance, the periodic partition cleanup can be turned off completely, but this increases the algorithm's memory usage significantly. We propose a default partition cleanup interval of 40 s, which showed to be a good compromise between runtime and memory consumption in our experiments; the interval can be configured with a parameter.

7.3.2 Partition eviction

The periodic partition cleanup is a valuable technique to control the memory consumption for normal, non-critical discovery periods. Due to the dynamic nature of the discovery process, DISTOD's memory consumption is bursty at times and the periodic partition cleanups might not be able to remove enough partitions from the cache in critical situations. For this reason, we propose a second protocol, called partition eviction, that tries to prevent out-of-memory situations by carefully monitoring the memory consumption.

The heap size monitoring is implemented by a dedicated SystemMonitor actor on each node in the DISTOD cluster. This actor makes partition eviction decisions for its host independently of other SystemMonitors on remote nodes. This means that one node reaching its individual heap memory limit does not impact the performance or memory usage of other nodes.

The SystemMonitor monitors the local memory usage of DISTOD and compares it to a certain threshold. If the local memory usage exceeds the threshold, the SystemMonitor instructs the local PartitionMgr to remove *all* intermediate partitions from the partition cache. In this way, DISTOD frees all expendable memory at the cost of (re-)calculating all later requested partitions from scratch. Every such later generated partition is again stored in the PartitionMgr's partition cache to be utilized for further partition generations and the validations. This re-populates the cache with relevant partitions. If DISTOD then hits the heap threshold again, the partition eviction is triggered once more. Because the partition eviction costs a lot of performance, the protocol is triggered only if it is inevitable. We recommend a partition eviction threshold of 90%[1] so that the eviction process has enough scope for action.

8 Complexity control

Despite DISTOD's novel search strategy, the exponentially growing search space still limits the applicability of the algorithm: On the one hand, the candidate checks may take unexpectedly long although DISTOD aggressively parallelizes and distributes them; on the other hand, the search space might exhaust the memory of the leader node. As countermeasures, DISTOD supports *elasticity*, i. e., it can incorporate additional compute nodes at runtime if the discovery runs unexpectedly long, and it supports *semantic pruning* to narrow down the search space and, in this way, reduce both the required memory and the runtime. In this section, we briefly describe these two features.

8.1 Elastic bOD discovery

DISTOD's runtime is hard to predict. It depends not only on the size of the input dataset but also on the dataset's structure as well as the number and placement of valid bODs in the search space because these factors also determine the effectiveness of the pruning strategies and the number of candidate validations. Therefore, we designed DISTOD in a way that we can dynamically add more follower nodes to or remove existing follower nodes from a running DISTOD cluster. The idea is to increase DISTOD's capabilities and speed up the processing by elastically adding nodes to the cluster on demand. Removing nodes frees up the compute resources of the cluster for other tasks without impacting the correctness or completeness of the discovered bODs sets.

Adding follower nodes Because all nodes in the DISTOD cluster are started individually, the procedures of starting the initial DISTOD cluster or starting a new node are the same. To connect to an already running DISTOD cluster, a freshly started follower node requires only the address of one of the nodes in the cluster. It then joins the cluster by (i) connecting to the specified seed node, (ii) retrieving the addresses

[1] If the G1 garbage collector is used, we recommend running it with `-XX:G1ReservePercent=(1 - heap-eviction-threshold)`; defaults to 10%.

of all other nodes from it, (iii) fetching the initial partitions to its local `PartitionMgr` actor, (iv) connecting its local `RCProxy` to the `ResultCollector` actor on the leader node, and finally (v) registering its local `Workers` at the cluster's `Master` actor. DISTOD treats all connected follower nodes in the same way.

Removing follower nodes Any follower node can be removed from the DISTOD cluster by gracefully shutting it down. Only the leader node cannot be removed from the cluster, because it holds the central candidate state and orchestrates the discovery process. The shutdown of a single node is handled by the same coordinated shutdown protocol than the cluster shutdown, but only the node-local parts are executed. The termination procedure is executed by the local `Executioner` actor. It supervises the termination and makes sure that the following steps are executed in the correct order: (i) Stop local `Workers` and abort their jobs at the `Master` actor. The `Master` re-enqueues the jobs into the work queue so that they at some point get dispatched to the `Workers` of another node. (ii) Flush buffered results of the local `RCProxy` to the `ResultCollector` on the leader node. (iii) Leave the DISTOD cluster. (iv) Stop all remaining actors and cleanly terminate the Java Virtual Machine (JVM).

8.2 Semantic pruning strategies

In this section, we adapt two semantic pruning strategies for our distributed bOD discovery algorithm: *interestingness pruning* [25] and *size limitation* [19]. Both strategies discard candidates with certain characteristics that mark them as less practically relevant than other candidates. In this way, we reduce the result size at the cost of losing completeness of the discovered bOD sets. Therefore, semantic pruning is implemented as optional features that can be turned on and off.

Interestingness pruning The interestingness pruning strategy calculates a score for each bOD candidate and compares it to a threshold. If the score is too low, the candidate is not *interesting* enough and it is pruned. The interestingness score is defined as $\sum_{\mathscr{E}(t_X)\in\Pi_X} |\mathscr{E}(t_X)|^2$ for a bOD with the context X and indicates coverage and succinctness of a bOD candidate [25].

To facilitate interestingness pruning in DISTOD's distributed setting, we calculate and use the interestingness score directly before validating a bOD candidate in the `Worker` actors. The interestingness pruning decision of a single bOD candidate is independent of other candidates because it involves only the calculation of the interestingness score. Thus, the calculation and testing can be distributed similarly to the candidate validations. This allows us to parallelize the score calculation making use of the already distributed partitions.

Size limitation Limiting the maximum size of the dependency candidates to some fixed value is a technique that can reduce the size of the search space significantly improving both runtime and memory consumption. A size limitation has also been argued to be semantically meaningful because large dependencies are statistically more likely to appear by chance [19]. We, therefore, allow the user to restrict the number of attributes that can be involved in a bOD (size of a bOD) to a certain threshold. This threshold then directly corresponds to the maximum depth of a node in the candidate lattice (e. g., bOD candidates in level l_4 have exactly four attributes). To implement the size limitation pruning strategy in DISTOD, we simply design the `Master` actor to not generate bOD candidates of levels deeper than a specified size limit.

9 Evaluation

In this section, we evaluate DISTOD's performance in different settings and on various datasets. We compare its runtime with all existing complete OD discovery algorithms, which are FASTOD-BID [25] and its distributed variant DIST-FASTOD-BID [22]. Note that ORDER [15], its hybrid variant [12], and OCDDISCOVER [5] produce incomplete results and are, therefore, not comparable to our approach. We published the source code for DISTOD, additional technical documentation and the datasets for our evaluation on our repeatability website;[2] The source code for FASTOD-BID[3] and DIST-FASTOD-BID[4] is publicly available on Github.

After the performance evaluation (Sect. 9.2), we evaluate DISTOD's scalability w. r. t. the number of CPU cores of a single node (Sect. 9.3), the number of nodes in the cluster (Sect. 9.4), the number of tuples in a dataset (Sect. 9.5), and the number of attributes in a dataset (Sect. 9.6). To evaluate the robustness of DISTOD, we measure the runtime of our algorithm with different memory limits (Sect. 9.7). Our final experiments demonstrate the impact of the partition caching on DISTOD's runtime (Sect. 9.8).

9.1 Experimental setup

Hardware We perform our experiments on a cluster with twelve bare-metal nodes. The machines are equipped with an Intel Xeon E5-2630 CPU at 2.2 GHz (boost to 3.1 GHz) with 10 cores and hyper-threading. Eight nodes have 64 GB of main memory and four nodes have 32 GB of main memory. All nodes run an AdoptOpenJDK version 11.0.8 64-bit server

[2] https://hpi.de/naumann/projects/repeatability/algorithms/distod.html.

[3] https://git.io/fastodbid (Accessed 2020-08-26).

[4] https://git.io/dist-fastodbid (Accessed 2020-08-26).

JVM and Spark 2.4.4 on Ubuntu 18.04 LTS. We run our base experiments (see Table 2) three times and report the average runtime and the relative standard deviation (RSD).

Memory restriction Java's performance does not scale linearly with the used heap size, i. e., using a smaller heap might reduce not only the memory consumption but also the execution time of an algorithm. We observed this behavior in our experiments when on the `letter-sub.csv` dataset, for example, a single DISTOD node with 31 GB of memory was about 20% faster than the same DISTOD node with 58 GB of memory ($\sim$ 40 min compared to $\sim$ 49 min). This is because Java uses a JVM-internal performance optimization called compressed ordinary object pointers (OOPs) when the heap size is smaller than 32 GB. This reduces the size of object pointers to 32 bit instead of 64 bit, even on 64 bit architectures. As a consequence, less memory is used by the Java process so that the processor cache usage, as well as the memory bandwidth usage, is improved. This speeds up the algorithm execution significantly. For more details about compressed OOPs, we refer to Oracle's Java documentation.[5] For this reason, if not stated otherwise, we limit the Java heap size for all experiments, all nodes and all algorithms: The leader nodes, i. e., DISTOD's leader node and DIST-FASTOD-BID's driver process, limit their heap to 31 GB; the follower nodes, i. e., DISTOD's follower and DIST-FASTOD-BID's executors, limit their heap to 28 GB, which leaves 4 GB for stacks and the operating system.

Data characteristics For our experiments, we use several synthetic and real-world datasets from different domains, most of which have previously been used to evaluate FD and OD discovery algorithms. The datasets can be found on our repeatability website (see Footnote 2). We list all relevant details about the datasets in Table 2.

The implementation of DIST-FASTOD-BID does not support data types other than integers, but our datasets contain strings, dates, and decimals. For this reason, we had to preprocess all datasets to run DIST-FASTOD-BID on them: We removed all headers and substituted all values with their hash value so that each value is mapped to an integer representation. This transformation keeps all FDs intact, but may change bODs. Datasets with the suffix `-sub` in their name have been transformed using this method. Regardless of the fact that DISTOD can handle `NULL` values, text strings, decimal numbers, and date values, the mentioned datasets do not contain any of these and consist of only integer numbers. DISTOD follows the `NULLS FIRST` principle and infers the data type of a column during input parsing.

[5] https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#compressedOop (Accessed 2020-02-14).

9.2 Varying the datasets

In this first experiment, we compare the runtime of DISTOD, FASTOD-BID and DIST-FASTOD-BID in their most powerful configuration on various datasets. In Table 2, we report the measured runtimes in seconds and list the number of valid constant bODs (reported as #FDs) and the number of valid order compatible bODs (reported as #bODs). For the number of valid bODs, we count the actual results that have been written to disk. Note that all three algorithms produce the same results in all experiments. Furthermore, the total (incoming and outgoing) average network traffic for the leader node of DISTOD varies between 162 kB/s and 11 MB/s for the different datasets. The peak total activity varies between 445 kB/s and 16 MB/s including the initial dataset replication phase. This is significantly below the usual maximum network bandwidth; hence, DISTOD's performance is not bound by network.

The experiment uses the following rules: We execute the single-threaded algorithm FASTOD-BID on a single node of our compute cluster. Both the Akka cluster of DISTOD and the Spark cluster of DIST-FASTOD-BID are configured over the same twelve machines. The Spark master runs on the same machine as the driver process and we configured Spark to put one executor with 20 cores on each of the remaining eleven nodes. For DISTOD, we use an active leader configuration, where the leader node spawns 10 workers and each of the eleven follower nodes spawn 20 workers; we set the partition cleanup interval to 40 s (cf. Sect. 7.3) and *turn all semantic pruning strategies off* (cf. Sect. 8.2). Whenever an execution hits the memory limit of 31 GB, we increase the heap limit to 58 GB. In all these cases but one, which is the `letter-sub.csv` dataset for FASTOD-BID, increasing the memory limit did not enable the algorithms to process the dataset.

Table 2 shows that DISTOD is an order of magnitude faster than FASTOD-BID for datasets with a lot of rows. On the `adult-sub.csv` dataset with 15 columns and over 30k rows, DISTOD finishes slightly before 1 min and FASTOD-BID takes almost 1 h to complete. A similar observation can be made for the `letter-sub.csv` and the `imdb-sub.csv` dataset. DISTOD can finish the task for `letter-sub.csv` over 60× and for `imdb-sub.csv` nearly 20× faster than FASTOD-BID. For very small datasets with under 1k rows, FASTOD-BID is slightly faster than DISTOD. This is expected because DISTOD deals with the overhead of multiple parallel components and cluster management. Due to the active leader configuration and the reactive start procedure, DISTOD can start processing the dataset very early on, even before all follower nodes have connected to the leader. This allows DISTOD to process even small datasets very fast without the need to

Table 2 Runtimes in seconds of FASTOD-BID, DIST-FASTOD-BID, and DISTOD on different datasets in their most powerful configuration finding the complete set of minimal bODs (*semantic pruning turned off*)

Dataset	|R|	|r|	Size[a]	#FDs	#bODs	FASTOD-BID	DIST-FASTOD-BID	DISTOD
iris-sub.csv	5	150	4	4	7	**0.2** (2.9%)	21 (2.8%)	**0.5** (24%)
chess-sub.csv	7	28,056	519	1	0	8 (2.8%)	25 (2.3%)	**3** (15%)
imdb-sub.csv	8	7,734,683	628,014	8	26	26,010 (2.3%)	ML	**1264** (5.5%)
abalone-sub.csv	9	4177	187	137	318	4 (10.5%)	26 (2.2%)	**2** (7.5%)
bridges-sub.csv	13	108	6	142	1097	**2** (4.2%)	37 (0.0%)	3 (3.5%)
dblp-sub.csv	13	8 224 309	1,237,373	31	664	ML	ML	**27,122** (0.6%)
adult-sub.csv	15	32,561	3527	78	1140	3697 (3.0%)	326 (1.6%)	**54** (0.6%)
tpch-sub.csv	16	6001 215	736,193	3984	13,760	ML	ML	**60,010** (0.5%)
letter-sub.csv	17	20,000	696	61	2202	16,996 (0.6%)[b]	1287 (0.8%)	**247** (0.7%)
ncvoter-sub.csv	19	999,999	112,578	572	4362	ML	TL	**34,658** (0.2%)
hepatitis-sub.csv	20	155	7	8 250	54,766	149 (4.0%)	1404 (1.2%)	**95** (2.2%)
flight-long-sub.csv	21	499,999	71,032	290	2253	ML	906 (0.0%)	**215** (0.8%)
horse-sub.csv	29	300	25	128,727	2,231,321	ML	TL	**24,294** (0.4%)
flight-sub.csv	30	1000	135	5976	40,740	97 (5.8%)	463 (1.7%)	**55** (0.7%)
fd-reduced-sub.csv	30	250,000	69,580	89,571	742	2720 (1.7%)	1312 (0.6%)	**95** (1.1%)
plista-narrow.csv	35	1001	193	4467	54,921	–	–	872 (0.0%)
plista-sub.csv	63	1001	575	≥ 32,404	≥ 313,296	ML	TL	ML

We report the RSD in braces. Experiments that exceeded the *time limit* of 24 h are marked as TL and experiments that exceeded the *memory limit* of 31 GB are marked as ML—all ML experiments actually also exceeded 58 GB memory on the leader node. The lowest runtime for the corresponding datasets are marked in bold.

[a] Size of the original dataset without the value substitution on disk in KiB.

[b] Experiment ran with 58 GB heap space instead of 31 GB (lower heap space hit ML)

wait for a complete cluster startup and shutdown (e. g., for `abalone-sub.csv` or `chess-sub.csv`).

Compared to DIST-FASTOD-BID, DISTOD is at least 4× faster on all tested datasets. On short and wide datasets, such as `bridges-sub.csv` or `hepatitis-sub.csv`, DISTOD is even an order of magnitude faster than DIST-FASTOD-BID. This shows that DISTOD does not only gain its performance from scaling with the number of rows but also from scaling with the number of columns. On the small datasets in Table 2, DIST-FASTOD-BID is an order of magnitude slower than both FASTOD-BID and DISTOD. This is due to the synchronized cluster startup and shutdown procedure of the Spark implementation, which causes a significant runtime overhead.

DISTOD is the only approach that is able to process the `dblp-sub.csv`, the `tpch-sub.csv`, the `ncvoter-sub.csv`, and the `horse-sub.csv` datasets within our time and memory constraints. FASTOD-BID cannot process any of the four dataset because it hits the memory limit even when it uses 58 GB of heap memory. DIST-FASTOD-BID cannot process `dblp-sub.csv` and `tpch-sub.csv`, because its executors hit the memory limit in level three of the candidate lattice for both datasets. The `ncvoter-sub.csv` and `horse-sub.csv` datasets cannot be processed by DIST-FASTOD-BID, because it hits the time limit of 24 h. While DIST-FASTOD-BID did not finish level nine of the candidate lattice within the time limit for the `ncvoter-sub.csv` dataset, DISTOD explored all 15 levels of the candidate lattice in nearly 10 h validating more than 736k bOD candidates. Similarly for the `horse-sub.csv` dataset: While DIST-FASTOD-BID cannot finish processing level eight of the candidate lattice within the time limit, DISTOD explored all 18 levels within 7 h validating over 95m bOD candidates.

In summary, the experiment demonstrates that DISTOD competes well with FASTOD-BID on datasets with low numbers of bODs. It outperforms FASTOD-BID on harder datasets by about a factor that is proportional to the number of machines in the cluster, demonstrating that it distributes the workload effectively; it's reactive and dynamic search strategy, in particular, distributes the workload significantly better than DIST-FASTOD-BID's MapReduce-style distribution approach. Our novel partition caching strategies and the distributed setting also enable DISTOD to process much larger datasets before running into memory limits.

9.3 Scaling the cores

In our second experiment, we evaluate DISTOD's scalability with the number of cores in one node. Because the performance of a system cannot be judged based on its scalability behavior alone, i. e., good scalability can simply be the result of an improper implementation, McSherry et al. introduced a metric, called *configuration that outperforms a single thread* (COST), that puts the scalability of a system in relation to the performance of a competent single-threaded implementation [17]. The COST of a parallel/distributed system is the hardware configuration required to outperform the single-threaded variant. To judge the performance of DISTOD, the following scalability experiments, therefore, also evaluate COST.

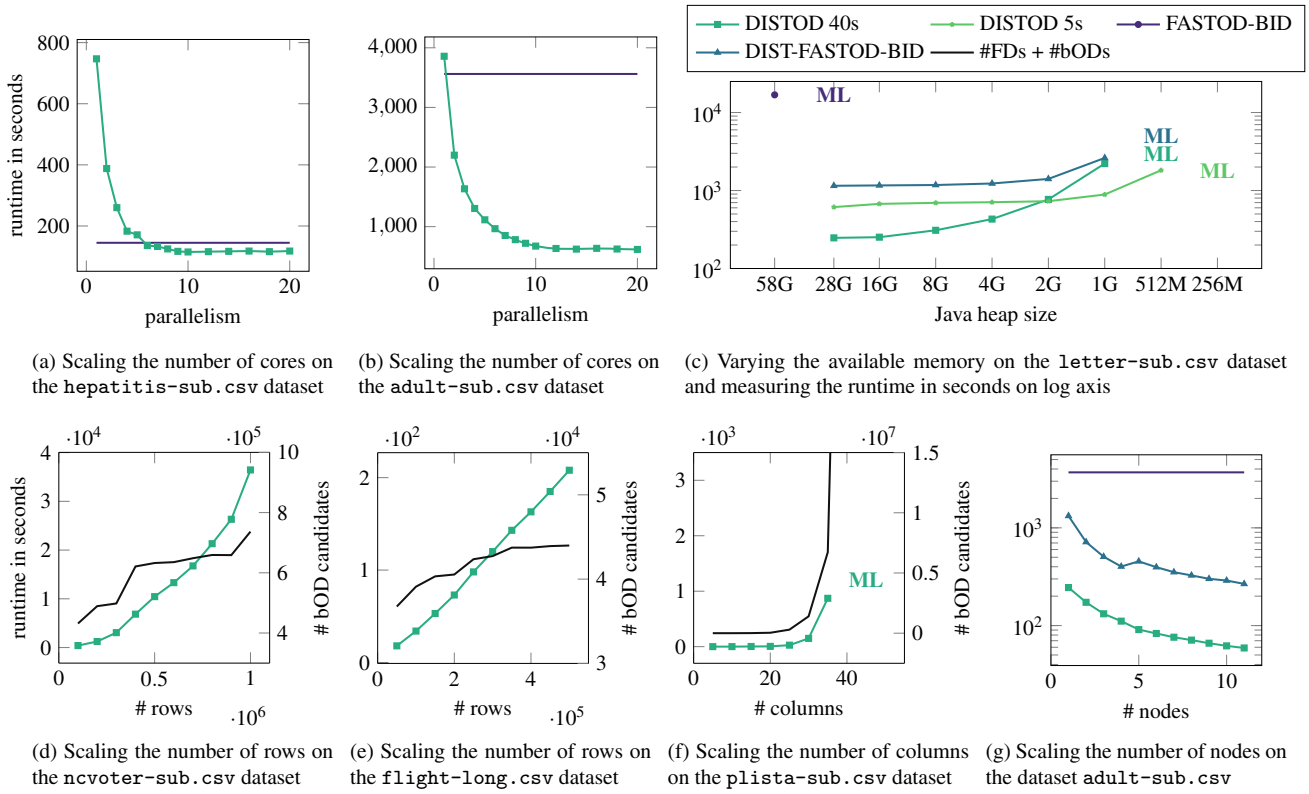
To evaluate the COST of DISTOD, we compare its runtime with different hardware configurations to the efficient single-threaded bOD discovery algorithm FASTOD-BID. We perform the experiments on a single node of our cluster and scale the number of cores from 1 to 20 (with DISTOD's parameter `max-parallelism`). Technically, we restrict the parallelism of the various parallel components of DISTOD by limiting DISTOD's actor system to a specific number of execution threads. We use two datasets to evaluate our COST metric: `hepatitis-sub.csv` as an example for a wide but short dataset and `adult-sub.csv` as an example for a narrow but long dataset.

Figure 5a shows the runtimes of DISTOD and FASTOD-BID for the `hepatitis-sub.csv` dataset in seconds. Since the `hepatitis-sub.csv` dataset is very short, there is not a big potential for parallelizing the candidate validations. Each validation is finished very fast and dispatching the validation jobs to different actors may introduce additional overhead. Despite that, DISTOD is able to outperform FASTOD-BID with a parallelism of six or more. Thus, DISTOD's COST for the `hepatitis-sub.csv` dataset is a single node with six cores. DISTOD's elastic task distribution strategy introduces only a low overhead and the parallelized candidate generation step improves its scalability even for short datasets.

Figure 5b shows the runtimes of DISTOD and FASTOD-BID for the `adult-sub.csv` dataset in seconds. The `adult-sub.csv` dataset with 15 columns is narrower than the `hepatitis-sub.csv` dataset with 20 columns, but it has more than 200× more rows. As expected, DISTOD scales very well on this dataset and can outperform FASTOD-BID already with a parallelism of two. DISTOD with a parallelism of three is already twice as fast as the single-threaded algorithm FASTOD-BID.

9.4 Scaling the nodes

DISTOD is a distributed algorithm that does not only scale vertically by utilizing all available cores of a single machine, but also horizontally by forming a cluster of multiple compute nodes. In Fig. 5g, we compare the runtimes of DISTOD, FASTOD-BID and DIST-FASTOD-BID when scaling them horizontally. Because FASTOD-BID is a single-threaded bOD discovery algorithm, its runtime is constant and serves as a reference. Note that we report the runtime of the



(a) Scaling the number of cores on the `hepatitis-sub.csv` dataset

(b) Scaling the number of cores on the `adult-sub.csv` dataset

(c) Varying the available memory on the `letter-sub.csv` dataset and measuring the runtime in seconds on log axis

(d) Scaling the number of rows on the `ncvoter-sub.csv` dataset

(e) Scaling the number of rows on the `flight-long.csv` dataset

(f) Scaling the number of columns on the `plista-sub.csv` dataset

(g) Scaling the number of nodes on the dataset `adult-sub.csv`

Fig. 5 Scaling experiments

approaches in seconds on a log axis. We ran the experiment on the `adult-sub.csv` dataset with 15 columns and 32,561 rows.

As the measurements in Fig. 5g show, DISTOD is 4× faster than DIST-FASTOD-BID and 14× faster than FASTOD-BID on a single node. On all twelve nodes, DISTOD is still more than 4× faster than DIST-FASTOD-BID, but its lead has shrunk, which is because the algorithm reaches the maximum parallelization for the parallelizable part of the bOD discovery for this specific dataset. Note that DISTOD uses the active leader configuration with ten `Worker` actors on the leader node while DIST-FASTOD-BID's Spark driver process utilizes only a single core on the leader node. Another observation that we make on this `adult-sub.csv` dataset and on many other datasets as well is that DISTOD on only one node is already faster than DIST-FASTOD-BID on all twelve nodes.

9.5 Scaling the rows

To perform the experiments on DISTOD's scalability in the number of rows |r|, we use two long datasets with a mid-range number of columns. Figure 5d plots DISTOD's runtime in seconds when scaling the number of rows of the `ncvoter-sub.csv` dataset from 100k to about 1m rows, and Fig. 5e plots the runtime of DISTOD for the `flight-long.csv` dataset with 50k to 500k rows.

The measurements in Fig. 5d show a tendency toward linear runtime growth and the measurements in Fig. 5e show almost perfectly linear scalability with the number of rows. This is because the computation time is dominated by the generation of partitions and the validation of bOD candidates, which both are linear in |r|. The deviation from a perfectly linear growth is due to the increasing number of bOD candidates: Additional records in the input data invalidate ever more candidates, which leads to a growth in the amount of candidates that need to be validated. If fewer bOD candidates are invalidated, the valid bODs are shorter and, hence, detected early in the lower lattice levels; our pruning strategies can, then, prune a lot of the candidates in higher levels from the search space. If many invalid candidates occupy the lower lattice levels, DISTOD cannot prune these candidates but needs to validate them.

In Fig. 5d, we see that DISTOD scales about linearly with the number of rows if the number of candidates does not change significantly (400k to 900k rows) and it scales worse than linearly if the number of candidates grows stronger (100k to 400k and 900k to 1m rows).

In Fig. 5e, we see that DISTOD scales almost perfectly linear with the number of rows because the increase in candidates is small and even flattens out in the end.

9.6 Scaling the columns

To evaluate DISTOD's ability to scale with the number of columns in a dataset, we perform an experiment on our widest dataset, which is `plista-sub.csv` with 63 columns. In the experiment, we scale the dataset's number of columns from five to 60 by increments of five and vary the number of columns by taking random projections of the `plista-sub.csv` dataset. Figure 5f shows that the runtime of DISTOD grows exponentially with the number of columns. This is expected because the number of bOD candidates (and minimal bODs) in the set containment lattice grows exponentially with the number of columns in the worst case. The increasing amount of bOD candidates depicted in Fig. 5f confirms this theoretical complexity. The candidate lattice for the `plista-sub.csv` dataset with 40 or more columns outgrows DISTOD's memory limit on our leader node because DISTOD is not able to free up any more memory without giving up the completeness or minimality of the results—in practice, the algorithm then sacrifices completeness by terminating early without finding all minimal bODs. For this reason, we report the runtimes only up to 35 columns.

As the result size grows exponentially with the number of columns in a dataset, DISTOD's runtime and memory consumption grow exponentially as well. To overcome this limitation, we introduced two semantic pruning strategies in Sect. 8.2: *interestingness pruning* and *size limitation*. Both reduce the number of valid bODs by restricting the search space to interesting bODs only. This improves the performance of DISTOD by orders of magnitude and allows it to mine larger datasets. By enabling the interestingness pruning, DISTOD can mine the `plista-sub.csv` dataset with 45 columns in 64 s (60 interesting bODs) and the entire dataset with 63 columns in 4.5 min (98 interesting bODs). Limiting the size of the bODs to a maximum of 6 columns achieves a similar result: For the `plista-sub.csv` dataset with 45 columns, DISTOD takes 89 s (532 bODs) and for the entire dataset with 63 columns, it takes just under 5 min (809 bODs).

9.7 Memory consumption

Current bOD discovery algorithms demand a lot of memory to store intermediate data structures. For DISTOD, this includes the candidate state and job queue on the leader node and the partitions on all other nodes. In the following experiment, we, therefore, compare DISTOD's performance with limited memory and its memory consumption to our baseline algorithms FASTOD-BID and DIST-FASTOD-BID. To measure the memory consumption, we limit the available memory in logarithmic steps starting from 28 GB. We stop reducing the memory limit when the algorithm experiences memory issues for the first time. We execute the single-threaded algorithm FASTOD-BID on a single node of our cluster. DISTOD and DIST-FASTOD-BID utilize all nodes of the cluster. For DISTOD, we limit the available memory of the leader node as well as the memory for all follower nodes to the same value. We still use the active leader configuration, where the leader node spawns ten local `Worker` actors. The experiment runs DISTOD in two configurations: one with a 40 s partition cleanup interval, which we also used in all previous experiments, and one with a more aggressive interval of 5 s (see Sect. 7.3). A shorter interval causes DISTOD to free up memory quicker, but it also influences DISTOD's runtime negatively. For DIST-FASTOD-BID, we gradually reduce the available memory for the Spark driver process as well as for all executors.

Figure 5c shows the runtimes of DISTOD, DIST-FASTOD-BID, and FASTOD-BID on the `letter-sub.csv` dataset when we reduce the available memory from 28 GB to 256 MB. Because FASTOD-BID already hit the memory limit (denoted with ML) with 28 GB memory, we included its runtime with 58 GB. FASTOD-BID uses a lot of memory because it's level-wise search strategy stores all partitions of the current and the next level while generating a level. In addition, it also stores the sorted partitions for all attributes of the dataset and the current intermediate candidate states. The partitions of the previous level are freed up not before the transition from one level to the next is completed. With the 58 GB of memory, FASTOD-BID takes more than 4.5 h to process the `letter-sub.csv` dataset. As a reference, DISTOD finishes the discovery on a single node with only 28 GB of memory within 38 min; this is 7× faster than FASTOD-BID while using only half of the memory.

DIST-FASTOD-BID can process the `letter-sub.csv` dataset with at least 1 GB available main memory. If we limit the memory to 512 MB or less, then the Spark executors fail, which is marked in Fig. 5c as reaching the memory limit (ML). However, the diminishing memory capacity already becomes noticeable with 2 GB as DIST-FASTOD-BID's runtime starts to increase because the Spark framework starts to spend extra cycles on data management.

Our algorithm DISTOD is able to process the `letter-sub.csv` dataset with a pessimistic partition cleanup interval of 5 s and only 512 MB of available memory. Even with only 512 MB memory and the 5 s partition cleanup interval, DISTOD is still 1.4× faster than DIST-FASTOD-BID with 1 GB of memory. For the experiment with 512 MB, DISTOD uses most of its memory and triggers partition evictions, i. e., it frequently frees up memory by removing all stored stripped partitions from the cache. This allows DISTOD to continue processing with less memory but increased processing time, which is from 888 s (1024 MB) to 1817 s (512 MB). However,

for the experiment with only 256 MB of memory, the candidate states outgrow the memory limit (ML) on the master node. In this case, freeing up stripped partitions does not help anymore and the algorithm becomes output bound. To still process the dataset, we recommend enabling the semantic pruning mechanism of DISTOD.

Figure 5c also shows the runtimes of DISTOD when we keep the partition cleanup interval at 40 s. With a limit of 28 GB of memory, DISTOD takes 613 s to process the `letter-sub.csv` dataset with a 5 s partition cleanup interval and but only 247 s with a 40 s partition cleanup interval (see Table 2 in Sect. 9.2). This shows that a small partition cleanup interval negatively impacts DISTOD's runtime when the available memory is adequately sized. A smaller partition cleanup interval allows DISTOD to efficiently run with lower memory bounds though. In the case of a 40 s partition cleanup interval, DISTOD's memory consumption peaks higher due to the less frequent partition cleanups, which causes DISTOD to hit the memory limit sooner than with a 5 s partition cleanup interval, which is already at 512 MB. As Fig. 5c also shows, DISTOD with a 40 s partition cleanup interval is slower than with a 5 s interval when using lower memory limits, such as 2 GB or 1 GB, because the JVM's garbage collector already starts fighting for memory, which is more costly and less effective than DISTOD's own memory management. Thus, for environments with limited memory, a small partition cleanup interval is preferable. It allows DISTOD to process the dataset more efficiently and, thus, faster.

9.8 Partition caching

In this section, we study the impact of DISTOD's partition caching mechanism (see Sect. 7.1) on the runtime for various datasets. For this, we measured the runtime of DISTOD with partition caching enabled and disabled in milliseconds and report the results in Table 3. For the experiment, DISTOD uses all twelve nodes of our testing cluster.

DISTOD with partition caching enabled is on average 26% faster than with partition caching disabled. For the datasets `adult-sub.csv` and `letter-sub.csv`, partition caching decreases DISTOD's runtime even by 55% and 70% respectively. The runtime increase with caching on the `iris-sub.csv` dataset is due to the overall small runtime on this dataset and the runtime fluctuations in the startup process that impact particularly such small measurements. If partition caching is disabled, DISTOD computes all stripped partitions from the initial partitions using the direct partition product (see Sect. 7.2.2) and does not cache stripped partitions in the `PartitionMgr` actor. The direct partition product is slower than computing a stripped partition from two of its predecessors and, thus, increases DISTOD's runtime. Since DISTOD works on constant and order compatible bOD candidate validations in parallel and checks from different validation jobs may require the same stripped partition, DISTOD may even compute stripped partitions multiple times on each node. If partition caching is enabled, DISTOD's `PartitionMgr` makes sure that `Workers` on the same node can reuse existing stripped partitions and that DISTOD can benefit from the faster recursive generation of stripped partitions (see Sect. 7.2.1). In summary, the experiment showed that the cached partitions might be superfluous in some discovery runs, but they can increase DISTOD's performance significantly in other runs.

Table 3 Runtimes of DISTOD in milliseconds when partition caching is off or on. Column *Diff* reports the runtime increase or decrease when partition caching is on w. r. t. the runtime when partition caching is off

Dataset	Cache off	Cache on	Diff
`iris-sub.csv`	375	414	+10%
`chess-sub.csv`	2873	2393	−17%
`abalone-sub.csv`	1951	1396	−28%
`bridges-sub.csv`	2979	2871	−4%
`adult-sub.csv`	127,889	58,135	−55%
`letter-sub.csv`	828,082	249,417	−70%
`hepatitis-sub.csv`	108,094	102,466	−5%
`flight-long-sub.csv`	407,593	214,334	−47%
`flight-sub.csv`	70,355	70,572	−1%
`fd-reduced-sub.csv`	200,402	101,846	−49%

10 Conclusion

In this paper, we proposed DISTOD, a novel, scalable, robust, and elastic bOD discovery algorithm that uses the actor programming model to distribute the discovery and the validation of bODs to multiple machines in a compute cluster. DISTOD discovers all minimal bODs w. r. t. the minimality definition of [25] in set-based canonical form with an exponential worst-case runtime complexity in the number of attributes and a linear complexity in the number of tuples. In our evaluation, DISTOD outperformed both the single-threaded bOD discovery algorithm FASTOD-BID [25] and the distributed algorithm DIST-FASTOD-BID [22] by orders of magnitude. The superior performance is the result of DISTOD's optimized search strategy and its improved validation techniques, which are both enabled by the reactive, actor-based distribution approach. With DISTOD's elasticity property and the semantic pruning strategies, we can discover bODs in datasets of practically relevant size, such as the `plista-sub.csv` dataset with 61 columns and 1k rows, which can now be mined in under 5 min.

Topics for future work are investigating strategies that can reduce the memory consumption and growth of the candidate

states in the central data structure on the leader node because this is the memory limiting factor in DISTOD at the moment; adopting hybrid data profiling approaches, i. a., the ideas of [4,12,18], into a distributed bOD discovery algorithm; or enhancing our approach to the discovery of approximate bODs [13].

Acknowledgements We sincerely thank Lukasz Golab and Jarek Szlichta for their support and advice, all authors for generously publishing or sharing their code, and the reviewers for the numerous constructive comments.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 671–682 (2006)
2. Abedjan, Z., Golab, L., Naumann, F., Papenbrock, T.: Data Profiling. Morgan & Claypool Publishers. ISBN: 978-1-68173-447-7 (2018)
3. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: Proceedings of the Workshop on Data Description (SIGFIDET, Now SIGMOD), pp. 107–141 (1970)
4. Bleifuß, T., Kruse, S., Naumann, F.: Efficient denial constraint discovery with hydra. Proc. VLDB Endow. **11**(3), 311–323 (2017)
5. Consonni, C., Montresor, A., Sottovia, P., Velegrakis, Y.: Discovering order dependencies through order compatibility. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 409–420 (2019)
6. Dong, J., Hull, R.: Applying approximate order dependency to reduce indexing space. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 119–127 (1982)
7. Ginsburg, S., Hull, R.: Order dependency in the relational model. Theor. Comput. Sci. **26**(1), 149–195 (1983)
8. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 235–245 (1973)
9. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: Efficient discovery of functional and approximate dependencies using partitions. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 392–401 (1998)
10. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: Tane: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100–111 (1999)
11. Ilyas, I.F., Chu, X.: Trends in cleaning relational data: consistency and deduplication. Found. Trends Databases **5**(4), 281–393 (2015)
12. Jin, Y., Zhu, L., Tan, Z.: Efficient bidirectional order dependency discovery. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 61–73 (2020)
13. Karegar, R., Godfrey, P., Golab, L., Kargar, M., Srivastava, D., Szlichta, J.: Efficient discovery of approximate order dependencies. arXiv: 2101.02174 [cs] (2021)
14. Kruse, S., Papenbrock, T., Naumann, F.: Scaling out the discovery of inclusion dependencies. In: Proceedings of the Conference on Datenbanksysteme in Business, Technologie Und Web (BTW), pp. 445–454 (2015)
15. Langer, P., Naumann, F.: Efficient order dependency detection. VLDB J **25**(2), 223–241 (2016)
16. Liu, J., Li, J., Liu, C., Chen, Y.: Discover dependencies from data—a review. IEEE Trans. Knowl. Data Eng. **24**(2), 251–264 (2012)
17. McSherry, F., Isard, M., Murray, D.G.: Scalability! But at what cost? In: Proceedings of the USENIX Conference on Hot Topics in Operating Systems (HotOS), p. 14 (2015)
18. Papenbrock, T., Naumann, F.: A hybrid approach for efficient unique column combination discovery. In: Proceedings of the Conference Datenbanksysteme in Business, Technologie Und Web (BTW), pp. 195–204 (2017)
19. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 821–833 (2016)
20. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J.-P.: Functional dependency discovery: an experimental evaluation of seven algorithms. Proc. VLDB Endow. **8**(10), 12 (2015)
21. Saxena, H., Golab, L., Ilyas, I.F.: Distributed discovery of functional dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1590–1593 (2019)
22. Saxena, H., Golab, L., Ilyas, I.F.: Distributed implementations of dependency discovery algorithms. Proc. VLDB Endow. **12**(11), 1624–1636 (2019)
23. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 23–34 (1979)
24. Lightbend Inc. 2020. Akka: build powerful reactive, concurrent, and distributed applications more easily. Version 2.6.3 (2020)
25. Szlichta, J., Godfrey, P., Golab, L., Kargar, M., Srivastava, D.: Effective and complete discovery of bidirectional order dependencies via set-based axioms. VLDB J. **27**(4), 573–591 (2018)
26. Szlichta, J., Godfrey, P., Golab, L., Kargar, M., Srivastava, D.: Effective and complete discovery of order dependencies via set-based axiomatization. Proc. VLDB Endow. **10**(7), 721–732 (2017)
27. Szlichta, J., Godfrey, P., Golab, L., Kargar, M., Srivastava, D.: Erratum for discovering order dependencies through order compatibility (EDBT 2019). In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 659–663 (2020)
28. Szlichta, J., Godfrey, P., Gryz, J., Zuzarte, C.: Expressiveness and complexity of order dependencies. Proc. VLDB Endow. **6**(14), 1858–1869 (2013)
29. Szlichta, J., Godfrey, P., Gryz, J.: Fundamentals of order dependencies. Proc. VLDB Endow. **5**(11), 1220–1231 (2012)

30. The Apache Software Foundation: Apache Flink - Stateful Computations over Data Streams (2019). Retrieved 08/03/2020 from https://flink.apache.org/
31. The Apache Software Foundation: Apache Spark—unified analytics engine for big data (2018). Retrieved 08/03/2020 from https://spark.apache.org/
32. Vernon, V.: (2015). Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional. ISBN: 978-0-13-384690-4
33. Zhu, G., Wang, Q., Tang, Q., Rong, G., Yuan, C., Huang, Y.: Efficient and Scalable Functional Dependency Discovery on Distributed Data-Parallel Platforms. IEEE Trans. Parallel Distrib. Syst. **30**(12), 2663–2676 (2019)