Master's Thesis

Efficient Distributed Discovery of Bidirectional Order Dependencies

Effizientes und verteiltes Suchen von bidirektionalen Sortierungsabhängigkeiten

by

Sebastian Schmidl, B.Sc.

sebastian.schmidl@student.hpi.de

Supervisor

Dr. Thorsten Papenbrock

Chair for Information Systems

Faculty of Digital Engineering at the University of Potsdam

Potsdam, May 29th, 2020

Abstract

Bidirectional order dependencies (bODs) capture order relationships between lists of attributes in a relational table closely following the semantic of `ORDER BY`-clauses in SQL. They can express, for example, the relationship between year of birth and age: *year of birth ascending* orders *age descending*. The knowledge about order relationships is useful for many data management tasks, such as to optimize database query execution, to cleanse data from errors, or to maintain consistency in relational datasets. The bODs of a specific dataset are, however, usually not explicitly given so that we have to discover them. Unfortunately, mining a dataset for bODs is very difficult, because bODs are naturally expressed using lists of attributes, which leads to a search space that grows factorial in the number of attributes. Using the recently published set-based canonical form for bODs, the search complexity can be reduced from factorial to exponential in the number of attributes. Despite this complexity reduction, existing bOD discovery algorithms still fail in finding all minimal bODs in a reasonable time.

In this thesis, we propose the *distributed* bOD discovery algorithm DISTOD that scales with the available hardware to much larger datasets than state-of-the-art approaches. DISTOD is a scalable, robust and elastic bOD discovery approach that discovers complete sets of minimal, valid bODs. It combines highly efficient pruning techniques based on the set-based canonical form for bODs with a novel, reactive and distributed search strategy that takes advantage of the actor programming model. DISTOD's elasticity property and the semantic pruning strategies allow it to discover bODs in datasets of practically relevant size. Our evaluation on various real-life and synthetic datasets shows that DISTOD can outperform both single-threaded and distributed state-of-the-art bOD discovery algorithms by orders of magnitude.

Zusammenfassung

Bidirektionale Sortierungsabhängigkeiten (bODs) erfassen die Sortierungsbeziehungen zwischen Listen von Attributen in einem relationalen Datensatz in enger Anlehnung an die Semantik der `ORDER BY`-Bedingungen in SQL. Sie können z.B. die Beziehung zwischen Geburtsjahr und Alter ausdrücken: *Geburtsjahr aufsteigend* sortiert *Alter absteigend*. Das Wissen über Ordnungsbeziehungen ist für viele Datenverwaltungsaufgaben nützlich, z.B. zur Optimierung von Datenbankabfragen, zur Fehler-Bereinigung von Daten oder zur Aufrechterhaltung der Konsistenz in relationalen Datensätzen. Die bODs eines bestimmten Datensatzes werden jedoch in der Regel nicht explizit angegeben, so dass wir sie erst entdecken müssen. Leider ist das Mining eines Datensatzes nach bODs sehr schwierig, da bODs natürlicherweise durch Attributlisten ausgedrückt werden, was zu einem Suchraum führt, der in der Anzahl der Attribute faktoriell wächst. Mit der kürzlich veröffentlichten set-basierten kanonischen Form für bODs kann diese Suchkomplexität von faktoriell auf exponentiell reduziert werden. Trotz dieser Komplexitätsreduktion sind die bestehenden Algorithmen zur Entdeckung von bODs immer noch nicht in der Lage, alle minimalen bODs eines Datensatzes in angemessener Zeit zu finden.

In dieser Arbeit stellen wir den *verteilten* bOD Discovery Algorithmus DISTOD vor, der sich mit der eingesetzen Hardware auf sehr viel grössere Datensätze skalieren lässt als state-of-the-art Ansätze. DISTOD ist ein skalierbarer, robuster und elastischer bOD Discovery Ansatz, der alle minimalen und validen bODs entdeckt. Er kombiniert hocheffiziente Pruning-Techniken, die auf der set-basierten kanonischen Form für bODs basieren, mit einer neuartigen, reaktiven und verteilten Suchstrategie, die sich das Aktor-Programmiermodell zunutze macht. Die Elastizitätseigenschaft von DISTOD und die semantischen Pruning-Strategien ermöglichen die Entdeckung von bODs in Datensätzen von praktisch relevanter Größe. Unsere Evaluierung an verschiedenen realen und synthetischen Datensätzen zeigt, dass DISTOD sowohl single-threaded als auch verteilte state-of-the-art bOD Discovery Algorithmen um Größenordnungen übertreffen kann.

Contents

1 Distributed discovery of order dependencies

Order is a fundamental concept in relational data and database systems. Most of the attribute domains in relational data are naturally ordered, such as timestamp, numbers, or strings. This order property of the data is used to improve database systems in many ways. One example is the optimization of the database-internal physical storage format. Here, order is used to reduce the storage space requirements of index structures (e. g. sorted-string tables and log-structured merge-trees (LSM trees)) or the data itself (e. g. run-length encoding in columnar data) [1]. Another example is the optimization of database queries [6, 29, 24, 28]. There are three ways how order can be introduced to data: (i) The user can explicitly sort the data using the `ORDER BY` operator in SQL. (ii) Some database operations, such as sort-merge joins or sort-based aggregates, implicitly introduce order into the result set, because their implementation requires sorted data. (iii) Tables in a database system may be naturally ordered when they already include a tuple order during ingestion, e. g. through timestamps or auto-incremented keys. The order property is always tied to a list of attributes of the relational table. This knowledge about the order of tuples in result sets can, for example, be exploited to pick better join strategies [22].

Based on the order property of relational data, we can define another concept, called order dependency (OD), that captures an order relationship between lists of attributes in a relational table. 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 formal definition for ODs follows in Section 3.

ID	Month	Day	OCode	OFips	OState	DepDelay	ArrDelay	ArrDelGrp
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	$\perp$	$\perp$	$\perp$
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

Table 1: Excerpt of the flight dataset with eight columns and ten rows. $\perp$ denotes `null` values.

For now, consider the example dataset in Table 1 and the OD `ArrDelay` *ascending orders* `ArrDelGrp` *ascending*. It specifies that when we order tuples by the `ArrDelay` attribute, then they are also ordered by the `ArrDelGrp` attribute. This is true for the tuples in Table 1, which means that the OD is valid for this table. Note that the inverse OD `ArrDelGrp`

ascending orders `ArrDelay` *ascending* does not hold, because the value in attribute `ArrDelGrp` of the tuple t_5 is greater or equal to the value in `ArrDelGrp` of tuple t_9, but t_5's value in `ArrDelay` is smaller than t_9's value in `ArrDelay`. This invalidates the OD `ArrDelay` *ascending orders* `ArrDelGrp` *ascending* for our example dataset.

With the concept of order dependencies in a relational table, we know that, given an OD, the decision to order a set of tuples on a specific list of attributes also determines the ordering by another list of attributes. This knowledge can be used to optimize database query execution or to improve the database system design even further. During query planning, ODs help to derive additional order information, which enables further optimization techniques, e. g. eliminating costly operators, such as join or sort operations [28]. One example, where ODs can be used to improve database system design is the following: We can replace a dense implementation of a secondary index with a sparse implementation if we know that the tuple ordering by the secondary index' attributes is determined by the ordering from the key attributes [6]. This reduces the space requirement of the index structure. ODs are not only useful for database design and database query optimization but also for maintaining and improving the consistency of relational data (data quality and data integration) [2]. ODs are integrity constraints (ICs). Like all other ICs, ODs can describe business rules and any violation of an OD shows an error in the dataset. A set of ICs consisting of ODs, functional dependencies (FDs), denial constraints (DCs), or any other ICs can even guide automatic data cleaning [11]. ODs are closely related to FDs that have been extensively studied in research [15]. Compared to FDs, however, ODs capture the order of values in a column as well. Therefore, ODs are more general and they subsume FDs [28].

But before we can use them effectively, we need to know which ODs hold for a specific dataset. While ODs can be obtained manually, this process is very time-consuming and difficult. ODs are naturally expressed using lists of attributes. This leads to a search space that grows factorial with the number of attributes in a dataset. Fortunately, Szlichta et al. presented a polynomial mapping of ODs to a set-based canonical form in [25]. It allows us to explore a much smaller search space that grows exponentially with the number of attributes. But, we can still not expect a human to explore the whole search space to find valid ODs. Even if domain experts use their knowledge about the dataset to specify ODs, it remains a difficult and slow task. ODs tend to get too complex to achieve a big advantage by using domain knowledge in the OD discovery process. Simple ODs, such as the reverse age orders the date of birth in a table about people, are easy to find, but this is not true for more complex and interesting ODs. In the flight dataset for example, the following complex and interesting set-based OD is valid: {`AirTime`, `DayOfWeek`, `FlightNum`, `TailNum`} : `ArrTime` $\uparrow\sim$ `ArrDelay` $\uparrow$. We define set-based ODs in Section 3.2. The set-based OD tells us that for specific flights that are flown by the same aircraft (`TailNum`) on the same route (`FlightNum`) at the same day of the week and that take the same time for the flight, the delay at the destination airport (`ArrDelay`) monotonically increases over the course of the day (`ArrTime`).

As a consequence of the complexity of OD discovery, multiple order dependency [5, 13, 26] and bidirectional order dependency [25] discovery algorithms have been proposed. In the worst-case, these OD discovery algorithms have to explore the entire search space

to find all ODs in a dataset. Consequently, these approaches have a factorial [5, 13] or exponential [25, 26] worst-case complexity in the number of attributes depending on the OD representation they use. Despite Szlichta et al.'s effort to reduce the column-complexity and clever pruning strategies, state-of-the-art OD discovery algorithms still fail for larger datasets. Szlichta et al.'s algorithm FASTOD-BID, for example, takes almost five hours to terminate for the datasets *letter* with 17 attributes and 20K tuples and it hits the memory limit of 58 GB for the dataset *flight* with 21 attributes and 500K tuples (Table 3). The *letter* dataset's size is only 800 KiB, while the *flight* dataset takes up 70 MiB. This shows the need for more efficient OD discovery approaches.

Therefore, we propose DISTOD, a scalable, robust and elastic OD discovery approach. DISTOD pursues a novel, reactive OD search strategy that allows us to distribute both the discovery process and the validation of ODs on multiple machines forming a compute cluster. We decompose the OD discovery process into self-contained parts that can be executed independently from each other. This allows us to parallelize and distribute the processing of these parts while still being able to use effective pruning strategies to reduce the search space. DISTOD guarantees minimality and completeness of the discovered ODs by moving the decision points for candidate generation and pruning to a central component. Other parts of the algorithm are decoupled so that they can be parallelized efficiently and scaled out dynamically according to DISTOD's current environment. We employ custom-tailored strategies for the management of input data, output data, and intermediate results. These strategies reduce data communication costs between the compute machines in the cluster to a minimum.

Unlike Saxena, Golab, and Ilyas [21], we do not build upon a batch-oriented distributed system, such as Spark, to distribute our computations. When used for dynamic algorithms, batch-oriented distributed systems cannot utilize the available resources of the connected compute nodes in an optimal way, because they frequently repartition the data and contain hard synchronization barriers. They also do not support true elasticity, i. e., they can hardly deal with flexible cluster sizes, where nodes enter and leave at runtime. Spark admittedly can use resources dynamically, but the scaling is tied to the load on the system and can not be controlled from the outside. It is for example not possible to remove a node from the Spark cluster when a high load job is using executors on this node. Instead, we distribute DISTOD using the actor programming model. It allows us to use the same abstractions to parallelize components on one node and to distribute components on multiple nodes. Its shared-nothing approach helps us to enforce consistent pruning decisions and to balance the load on the cluster elastically. We base our OD discovery algorithm on the canonical representation of bidirectional order dependencies (bODs) from Szlichta et al. [25]. It allows DISTOD to traverse a much smaller set-containment lattice instead of a list-containment lattice and to benefit from the effective pruning rules, which reduce the overall algorithm runtime. DISTOD implements the following viable properties that enable the algorithm to discover ODs in larger datasets than all existing approaches:

Scalability DISTOD scales vertically by utilizing all available cores on a single machine and horizontally by forming a cluster of multiple computing machines. The workload is balanced across all participating nodes using the work pulling principle. This allows

DISTOD to perform more OD candidate checks in shorter time, which enables the processing of larger datasets that would otherwise not terminate in a reasonable amount of time.

Robustness DISTOD effectively deals with limited memory. If the memory usage grows too high, it frees up memory by removing intermediate and temporary results that can be re-computed at any time. Valid ODs are progressively spilled to disk, so that none gets lost in the case of node failures and memory pressure is reduced. If the candidate state grows too large, i. e., the result size at some point outgrows the memory capacity and we cannot free up memory without breaking the minimality and correctness guarantees, DISTOD terminates safely preserving all results collected to that point. This strategy provides best possible results, because DISTOD discovers small ODs first and small dependencies are usually considered as the most relevant dependencies [18]. Note that all existing algorithms do not produce any results when they fail. Especially algorithms that deal with large search spaces need to use the available memory conservatively to increase the size of datasets that they can analyze.

Elasticity DISTOD supports elasticity in the number of compute nodes, which allows the user to add more compute resources to the cluster or to free up nodes for other tasks at runtime. Because the overall runtime of an OD discovery algorithm is hard to predict and the capabilities of an execution environment might change, for example, due to the submission of additional other jobs to the cluster, DISTOD allows adding or removing nodes from a running DISTOD cluster.

Applicability DISTOD implements two semantic pruning strategies, which are interestingness pruning (see [25]) and size limitation (see [18]) to let the user effectively restrict the size of the OD search space. Semantic pruning helps to decrease both the algorithm runtime and the size of the result sets, which is necessary in order to discover ODs in large datasets. To facilitate the pruning strategies in DISTOD's distributed setting, we scale out the calculation of the interestingness score alongside the candidate validations, while the pruning decisions are still taken by the central component that is also responsible for guaranteeing minimality and completeness of the discovered ODs.

We make the following technical contributions: (i) We introduce a novel, reactive search strategy that breaks up the levels of the candidates lattice into individual tasks. In this task-based approach, we can handle constant and order compatible bOD candidates independently from each other, which allows a fine-grained work distribution of our candidate validation jobs. (ii) We present a centralized, but parallel, candidate generation algorithm that guarantees consistency and correctness of the outputted bODs. (iii) We revise the validation algorithm for order compatible bOD candidates from [25] to improve its efficiency. We use a new index data structure, which we call inverted sorted partition. (iv) We contribute a hybrid and dynamic generation algorithm for stripped partitions that either uses a recursive partition generation scheme (see Section 7.3.1) or a direct partition product (see Section 7.3.2) to generate a stripped partition on-demand.

In our evaluation, we show that our scalable and elastic implementation DISTOD outperforms the single-threaded algorithm FASTOD-BID by orders of magnitude and the distributed, primitive-driven implementation DIST-FASTOD-BID by Saxena, Golab, and Ilyas by factors of five to eight; for wider datasets even by an order of magnitude. With its elasticity property and semantic pruning strategies, DISTOD is able to discover ODs in datasets of practically relevant size.

We first introduce related work about ODs and the discovery of ODs in Section 2. In Section 3, we then provide the theoretical foundations on bODs and the set-based canonical form for bODs by Szlichta et al. [25], on which our approach is based on. Section 3.4 also introduces the actor programming model, which allows us to develop a highly scalable, dynamic and distributed bOD discovery algorithm. Section 4 gives a high-level algorithmic and technical overview about our novel distributed bOD discovery approach, before we subsequently explain the algorithm's components in more detail. Section 5 explains how DISTOD ensures minimal bODs and how it reactively generates the bOD candidates of the candidate lattice. Section 6 explains how DISTOD uses inverted sorted partitions and stripped partitions, also called *position list indexes*, to efficiently validate bOD candidates. Section 7 explains how DISTOD manages data in the different parts of the system. Since managing partitions is specifically challenging, this section focuses on the distributed management of inverted sorted partitions and stripped partitions, the reactive generation of stripped partitions, and partition-related memory management. DISTOD's elasticity features are described in Section 8 and its semantic pruning strategies are described in Section 9. We evaluate the runtime, memory consumption, and scalability of our algorithm on various datasets in Section 10, before we summarize our findings and discuss possible future work in Section 11.

2 Related work

In this section, we first introduce work covering the notion and formalisms of ODs and bODs. We then present all previous OD discovery approaches and conclude this section with distributed approaches on FD and OD discovery.

Order dependencies (ODs) The idea of considering the ordering of attributes in a relation as a kind of dependency was first introduced in 1982 by Ginsburg and Hull. Their work formalizes the so-called *point-wise ordering* with a complete set of inference rules and shows that the inference-problem in their formalism is co-NP-complete [8].

In 2012, Szlichta, Godfrey, and Gryz presented the definition of ODs [23] used throughout all following work in this area [13, 5, 26]. This definition using lists of attributes for the left and right-hand side of ODs leads to a lexicographical ordering of tuples as it is generated by the `ORDER BY` operator in SQL. Their work also introduced a set of axioms and the proof that ODs properly subsume FDs. In [28], Szlichta et al. introduce bidirectional order dependencies (bODs), a combination of ascending and descending orders of attributes. The authors of [23] and [28] show that the inference problem for both ODs and bODs is co-NP-complete. However, FDs can be inferred from ODs in linear time [28].

Order dependency discovery In contrast to the related field of FD discovery, literature on OD discovery is scarce. The first algorithm for OD discovery, called ORDER, was proposed by Langer and Naumann [13]. It loops over a list-containment lattice of OD candidates to find the ones that hold in the dataset. It has a factorial worst-case complexity in the number of attributes and establishes a baseline for the following discovery algorithms. ORDER is sound, but was shown to be incomplete [26, 25].

Jin, Zhu, and Tan propose a novel hybrid OD discovery approach inspired by [4] and [17] that iteratively discovers ODs on a sample of the dataset and validates them on the complete dataset [12]. Their approach can discover ODs as well as bODs. The authors use the same minimality definition as Langer and Naumann. In their experimental evaluation in [12], they show that their algorithm discovers the same set of ODs as ORDER. Unfortunately, ORDER's result sets are incomplete, which renders Jin, Zhu, and Tan's new approach to bOD discovery incomplete as well.

Szlichta et al. present the first approach, called FASTOD, that is able to find a complete and minimal set of ODs [26]. Their algorithm has exponential worst-case complexity in the number of attributes and linear complexity in the number of tuples. This is achieved by mapping ODs to a new canonical representation in polynomial time. Their work provides inference rules for the new canonical ODs. In [25], the authors expand this approach to bODs. They show that discovering bODs does not take significantly longer than discovering unidirectional ODs. We base our algorithm on their canonical representation for bODs and the corresponding definition of minimal bODs to benefit from the reduced search space size and efficient pruning rules.

Consonni et al. take another approach to OD discovery using order compatibility dependencies (OCDs), called OCDDISCOVER [5]. An OCDs is a special form of OD in which two lists of attributes order each other when they are concatenated [23]. Their approach was shown to use an incorrect definition of minimality [27]. OCDDISCOVER misses valid ODs because it prunes the search space too aggressively. This renders the results of OCDDISCOVER incomplete [27].

Distributed order dependency discovery Because there exists only one complete and correct OD discovery algorithm, there is not much research in the distribution of OD discovery algorithms. In [21], Saxena, Golab, and Ilyas present distributed versions of different dependency discovery algorithms using a map-reduce style framework (Spark). They introduce common programming primitives for those algorithms and compare their runtime. This work also evaluates a distributed implementation of FASTOD-BID. We call this implementation DIST-FASTOD-BID. Saxena, Golab, and Ilyas focus on shaping the shared programming primitives and providing distributed implementations for them. We use more sophisticated distribution techniques that are not necessarily applicable to other dependency discovery strategies. But we show that we can increase the overall performance by removing synchronization barriers and reducing data communication costs. DIST-FASTOD-BID serves as a baseline for our evaluation.

In the related field of FD discovery multiple distributed discovery approaches were published [32, 33, 20]. They focus on discovering FDs on already partitioned data. We assume that the data and results reside on a single node. Our algorithm then replicates the data to all our compute nodes in order to use more resources to find all minimal bODs.

3 Foundations

Throughout this thesis, we use the following notational conventions:

- R denotes a relation and r a specific instance of R.
- A denotes a single attribute from R.
- t denotes tuples of a relational instance r.
- t_{A} denotes the value of an attribute A in a tuple t.
- X is a set of attributes and X_i is the i^{th} element of X, where $0 < i < |X|$.
- $\mathbf{Y}$ is a list of attributes, Y_i is the i^{th} element of $\mathbf{Y}$, where $0 < i < |\mathbf{Y}|$. $[\,]$ is the empty list and $[\mathsf{H} \mid \mathbf{T}]$ denotes a list with head H and tail $\mathbf{T}$. Lists and sets with the same name reference the same data in a different representation. This means that set X contains all distinct elements from list $\mathbf{X}$.

We begin this section by formally defining ODs and bODs. In Section 3.2, we recap the set-based canonical form for bODs from Szlichta et al. [25], because it is the theoretical foundation for our discovery algorithm. Section 3.3 then describes how we can transform bODs (in their list-based form) to set-based bODs and vice versa. DISTOD uses the actor programming model as a means to dynamically distribute the discovery of bODs in a cluster. Thus, we give a short overview of the core concepts of actor programming in Section 3.4.

3.1 Order dependencies

In this section, we give a definition for bidirectional order dependencies and (unidirectional) order dependencies. We use the definitions that were introduced by Szlichta et al. in [28]. First, we define an *order specification*, which is a guideline to sort the tuples of a dataset based on multiple attributes with different order directions (ascending or descending). It corresponds to the `ORDER BY`-clause in SQL.

Definition 1 *An ordering based on an attribute* $\mathsf{A} \in \mathrm{R}$ *can have one of two directions: ascending or descending. To indicate the order direction of a marked attribute* $\overline{\mathsf{A}}$*, we use* $\mathsf{A}\uparrow$ *for ascending and* $\mathsf{A}\downarrow$ *for descending. An order specification is a list of marked attributes denoted as* $\overline{\mathbf{X}} = \left[\overline{\mathsf{A}} \mid \mathsf{A} \in \mathrm{R}\right]$*. For attributes without an order direction, the default order direction is implicitly assumed. The default order direction is ascending (*$\uparrow$*). This is also the default direction in the SQL standard.*

An example for an *order specification* is $\overline{\mathbf{X}} = [\texttt{OCode}\uparrow, \texttt{Day}\downarrow]$. This means that tuples are first sorted by `OCode` in ascending order and then within each value of `OCode` by `Day` in descending order. The result of this *order specification* applied on Table 1 is shown in Table 2. *Order specifications* produce a lexicographical ordering, which is also called nested sort.

ID	Month	Day	OCode	OFips	OState	DepDelay	ArrDelay	ArrDelGrp
t_6	12	27	BWI	24	Maryland	-5	-23	-2
t_4	12	20	CLE	39	Ohio	$\perp$	$\perp$	$\perp$
t_0	12	3	CLE	39	Ohio	80	72	4
t_2	12	14	JFK	36	New York	-6	0	0
t_9	12	30	ORD	17	Illinois	-5	-14	-1
t_7	12	28	ORD	17	Illinois	57	82	5
t_5	12	24	ORD	17	Illinois	5	-2	-1
t_3	12	16	ORD	17	Illinois	13	59	3
t_1	12	14	ORD	17	Illinois	-4	-10	-1
t_8	12	30	SGF	29	Missouri	2	0	0

Table 2: Table 1 sorted by the *order specification* [OCode ↑, Day ↓]. $\perp$ denotes null values.

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 instance* r *by order specification* $\overline{\mathbf{X}}$ *also orders* r *by order specification* $\overline{\mathbf{Y}}$*, where* $X \subset \mathrm{R}$ *and* $Y \subset \mathrm{R}$*. We use the notation* $\overline{\mathbf{X}} \leftrightarrow \overline{\mathbf{Y}}$ *(read:* $\overline{\mathbf{X}}$ *and* $\overline{\mathbf{Y}}$ *are order equivalent) when* $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ *and* $\overline{\mathbf{Y}} \mapsto \overline{\mathbf{X}}$*. When* $\overline{\mathbf{XY}} \leftrightarrow \overline{\mathbf{YX}}$*, 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* $\forall s, t \in \mathrm{r} : s \preceq_{\overline{\mathbf{X}}} t \Rightarrow s \preceq_{\overline{\mathbf{Y}}} t$*. The lexicographical order operator* $\preceq_{\overline{\mathbf{Z}}}$ *for an order specification* $\overline{\mathbf{Z}} = [\overline{\mathsf{A}} \mid \mathsf{A} \in \mathrm{R}]$ *and the tuples* $p, q \in \mathrm{r}$ *is defined as:*

$$p \preceq_{\overline{\mathbf{Z}}} q = \begin{cases} \overline{\mathbf{Z}} = [\,] \\ \overline{\mathbf{Z}} = [\mathsf{A} \uparrow \mid \overline{\mathbf{T}}] \wedge p_{\mathsf{A}} < q_{\mathsf{A}} \\ \overline{\mathbf{Z}} = [\mathsf{A} \downarrow \mid \overline{\mathbf{T}}] \wedge p_{\mathsf{A}} > q_{\mathsf{A}} \\ \overline{\mathbf{Z}} = \big([\mathsf{A} \uparrow \mid \overline{\mathbf{T}}] \vee [\mathsf{A} \downarrow \mid \overline{\mathbf{T}}]\big) \wedge p_{\mathsf{A}} = q_{\mathsf{A}} \wedge p \preceq_{\overline{\mathbf{T}}} q \end{cases}$$

Let $s \prec_{\overline{\mathbf{Z}}} t$ *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 we know that $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$, than we know that 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: [ArrDelay] ↦ [ArrDelGrp], [OCode ↑] ↦ [Month ↓], [ArrDelay] ↦ [ArrDelGrp, DepDelay], [ArrDelGrp, DepDelay] ↦ [ArrDelay], and [OState ↑, Day ↓] ↦ [OFips ↑]. Note that those bODs do hold in our example table and not necessarily in general. A less complex version of bODs are unidirectional ODs [23]. We simply call them ODs.

Definition 3 *An (unidirectional) order dependency (OD) is a bOD for which all attributes are marked with the same order direction; either all attributes are marked ascending or*

all attributes are marked descending. Note that the default order direction is ascending. $\mathbf{X} \mapsto \mathbf{Y}$ *is an OD specifying that ordering a relation instance* r *by* $\mathbf{X}$ *also orders* r *by* $\mathbf{Y}$*, where* $X \subset \mathrm{R}$ *and* $Y \subset \mathrm{R}$*. Table* r *over* R *satisfies* $\mathbf{X} \mapsto \mathbf{Y}$ *if* $\forall s, t \in \mathrm{r} : s \preceq_{\mathbf{X}} t \Rightarrow s \preceq_{\mathbf{Y}} t$*. The operator* $\preceq_{\mathbf{Z}}$ *over a list of attributes* $\mathbf{Z} = [\mathsf{A} \mid \mathsf{A} \in \mathrm{R}]$ *and the tuples* $p, q \in \mathrm{r}$ *is defined as:*

$$p \preceq_{\mathbf{Z}} q = \begin{cases} \mathbf{Z} = [\,] \\ \mathbf{Z} = [\mathsf{A} \mid \mathbf{T}] \wedge p_{\mathsf{A}} < q_{\mathsf{A}} \\ \mathbf{Z} = [\mathsf{A} \mid \mathbf{T}] \wedge p_{\mathsf{A}} = q_{\mathsf{A}} \wedge p \preceq_{\mathbf{T}} q \end{cases}$$

Unidirectional ODs do not allow attributes to have different order directions in the order specifications. An OD $\mathbf{X} \mapsto \mathbf{Y}$ means that $\mathbf{Y}$'s values are monotonically non-decreasing with respect to the values in $\mathbf{X}$. Unidirectional ODs with all attributes marked ascending and unidirectional ODs with all attributes marked descending are equivalent in their meaning, because reversing one gives the other. This can be shown in Table 1 with the OD [`ArrDelay` $\uparrow$] $\mapsto$ [`ArrDelGrp` $\uparrow$]. If we reverse the order of all columns in the table resulting in the tuples (top to bottom) t_9, t_8, …, t_0, the OD [`ArrDelay` $\downarrow$] $\mapsto$ [`ArrDelGrp` $\downarrow$] becomes valid. Most existing OD discovery approaches assume an ascending order for all attributes in the order specifications [5, 13, 26].

BODs are a generalization of ODs and for this reason, discovering all bODs also discovers all ODs. A bOD discovery algorithm can provide more information about a dataset than an OD discovery algorithm and Szlichta et al. experimentally showed in [25] that the discovery of bODs does incur only a low runtime overhead of about 5 % compared to the discovery of (unidirectional) ODs, despite the increased number of results (by up to 50 %).

3.2 Set-based canonical bODs

As represented in their formal definition $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$, bODs are naturally expressed by lists of attributes. The bOD candidates of a relation form a lattice with $\binom{n}{k} \cdot k!$ nodes in level k. Each node in level k consists of k attributes and represents $k - 1$ bOD candidates. The lattice contains $\lfloor n! \cdot \mathrm{e} \rfloor$ nodes [13]. This, however, means that the candidate space for bODs over a relation grows factorial to the number of attributes n.

Dependency discovery algorithms have to efficiently test all bODs candidates of a dataset to determine which bODs are valid for the selected dataset. For most of the existing dependency discovery algorithms that traverse a candidate lattice the worst-case complexity directly depends on the number of candidates. Therefore, having a factorial search space indicates that the complexity of the discovery problem is factorial as well. But Szlichta et al. show in [25] that we can express bODs using sets of attributes instead of lists. This allows bOD discovery algorithms to traverse a much smaller set-containment lattice instead of the list-containment lattice, reducing the worst-case complexity to be exponential in the

number of attributes of the dataset. This is comparable to other dependency discovery algorithms, such as FD discovery algorithms based on the idea of TANE [10].

We make use of this idea because it drastically reduces the search space and, therefore, also the overall runtime of the algorithm. For this reason, we now recap the set-based canonical form for bODs [25]. The mapping between set-based and list-based bODs is explained in Section 3.3. All proofs and the axioms for set-based bODs can be found in the work from Szlichta et al. [25]. We will omit repeating these foundations in this paper for space reasons. Firstly, we introduce the notion of *equivalence class* and *partition* consistent with [9] and [25]:

Definition 4 *An equivalence class w. r. t. a given attribute set is noted as* $\mathcal{E}(t_X)$*. It groups tuples* s *and* t *together when their projection on* X *is equal:* $\mathcal{E}(t_X) = \{s \in \mathrm{r} \mid s_X = t_X\}$ *and* $X \in \mathrm{R}$*.*

This means that all tuples in an equivalence class $\mathcal{E}(t_\mathsf{A})$ have the same value in A and all other tuples have different values in A. Partitions group equivalence classes w. r. t. a common attribute set together. They are defined as follows:

Definition 5 *A partition is a set of disjoint equivalence classes with the same set of attributes:* $\Pi_X = \{\mathcal{E}(t_X) \mid t \in \mathrm{r}\}$*.*

When we consider our example dataset from Table 1, we can extract, for example, the following partition: $\Pi_{\{\texttt{OState}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_2\}, \{t_6\}, \{t_8\}\}$. It contains the equivalence classes $\mathcal{E}(\text{Ohio}) = \{t_0, t_4\}$, $\mathcal{E}(\text{Illinois}) = \{t_1, t_3, t_5, t_7, t_9\}$, $\mathcal{E}(\text{New York}) = \{t_2\}$, $\mathcal{E}(\text{Maryland}) = \{t_6\}$, and $\mathcal{E}(\text{Missouri}) = \{t_8\}$. Using this information about equivalence classes and partitions, we can define the two set-based canonical forms for bODs (see Definition 9 in [25]). We start with *constant bODs*:

Definition 6 *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* $\overline{\mathbf{X'}} \mapsto \overline{\mathbf{X'A}}$ *for all permutations* $\overline{\mathbf{X'}}$ *of* $\overline{\mathbf{X}}$*.*

In our example dataset (Table 1) i. a. the following constant bODs are valid: $\{\} : [\,] \mapsto$ `Month` and $\{\texttt{ArrDelay}\} : [\,] \mapsto$ `ArrDelGrp`. Szlichta et al. show that constant bODs directly represent FDs. They can be violated only by so-called *splits*.

Definition 7 *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* $\mathcal{E}(t_X)$ *but* $s_\mathsf{A} \neq t_\mathsf{A}$*.*

The bOD $\{\texttt{ArrDelGrp}\} : [\,] \mapsto \texttt{ArrDelay}$ is not valid for our example dataset because it is invalidated by at least one *split*. There are three *splits* in total. One *split* occurs between tuple t_1 and t_5. Admittedly both tuples belong to the same equivalence class $\mathcal{E}$(-1) but their value in `ArrDelay` differs. This one *split* in the dataset is enough to invalidate the mentioned constant bOD. The second set-based canonical form for bODs are *order compatible bODs*:

Definition 8 *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 $\{\,\} : \texttt{OFips} \uparrow\sim \texttt{OState} \uparrow$. It tells us that when we order the dataset by `OFips` it is also ordered by `OState`. The FIPS state codes are increasing numbers assigned to the lexicographically ordered state names [1]. This fact is represented in the above bOD and because there is a one-to-one mapping of code and name, the constant bODs $\{\texttt{OState}\} : [\,] \mapsto \texttt{OFips}$ and $\{\texttt{OFips}\} : [\,] \mapsto \texttt{OState}$ are also valid. If we know the state name then we also know the FIPS and the other way around. Order compatible bODs are violated by so-called *swaps*:

Definition 9 *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* $\mathcal{E}(t_X)$ *and* $s \prec_{\overline{\mathsf{A}}} t$ *but* $t \prec_{\overline{\mathsf{B}}} s$*.*

We can show this using the opposite bOD to our former example where `OState`'s order is descending instead of ascending: $\{\,\} : \texttt{OFips} \uparrow\sim \texttt{OState} \downarrow$. This order compatible bOD is not valid for Table 1, because i. a. t_9 and t_6 form a *swap*. When considering the order specification $[\texttt{OFips} \uparrow]$, t_9 must precede t_6 but when we use $[\texttt{OState} \downarrow]$ then t_6 must come before t_9. We use both forms of violations (*splits* and *swaps*) to check which bODs are valid for a specific dataset.

3.3 Mapping between set-based and list-based bODs

BODs in the set-based canonical form and list-based bODs are equally expressive. Both bOD forms can be mapped to each other. Unfortunately, there is no one-to-one mapping. We first explain, how we can get from list-based to set-based bODs and then the other way around. Each section shortly discusses the implications of the mappings.

[1] All FIPS codes: `https://www.census.gov/geographies/reference-files/2018/demo/popest/2018-fips.html`

3.3.1 Mapping from list-based to set-based bODs

List-based bODs can be mapped to 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* that falsify the bOD and $\overline{\mathbf{X}} \sim \overline{\mathbf{Y}}$ ensures that there are no *swaps* that falsify the bOD. The two set-based canonical forms for bODs are defined over their violations as well. Constant bODs $X : [\,] \mapsto \overline{\mathsf{A}}$ are valid if there is no *split* and order compatible bODs $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ are valid if there is no *swap*.

A list-based bOD of the form $\overline{\mathbf{X}} \mapsto \overline{\mathbf{XY}}$ is valid if $\forall \overline{\mathsf{A}} \in \overline{\mathbf{Y}}$ the set-based bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ is true. This guarantees the absence of *splits*. A list-based bOD of the form $\overline{\mathbf{X}} \sim \overline{\mathbf{Y}}$ is valid if $\forall i \in \{1, \ldots, |\overline{\mathbf{X}}|\}$ and $\forall j \in \{1, \ldots, |\overline{\mathbf{Y}}|\}$ the set-based bOD $\{\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}, \mathsf{Y}_j, \ldots, \mathsf{Y}_{j-1}\} : \overline{\mathsf{X}_i} \sim \overline{\mathsf{Y}_j}$ is true. This ensures that no *swaps* violate the bOD. A list-based bOD $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ can, therefore, be mapped to all of the above set-based bODs [25, Theorem 6]:

Definition 10 *A list-based bOD $\overline{\mathbf{X}} \mapsto \overline{\mathbf{Y}}$ is valid if $\forall \overline{\mathsf{A}} \in \overline{\mathbf{Y}}$ the set-based bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ and $\forall i \in \{1, \ldots, |\overline{\mathbf{X}}|\}, j \in \{1, \ldots, |\overline{\mathbf{Y}}|\}$ the set-based bOD $\{\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}, \mathsf{Y}_j, \ldots, \mathsf{Y}_{j-1}\} : \overline{\mathsf{X}_i} \sim \overline{\mathsf{Y}_j}$ are valid.*

We illustrate this mapping with an example. Consider the list-based bOD $[\mathsf{A}\uparrow, \mathsf{B}\downarrow] \mapsto [\mathsf{C}\uparrow, \mathsf{D}\uparrow]$. It is mapped to the following set-based bODs: (i) $\{\mathsf{A}, \mathsf{B}\} : [\,] \mapsto \mathsf{C}\uparrow$, (ii) $\{\mathsf{A}, \mathsf{B}\} : [\,] \mapsto \mathsf{D}\uparrow$, (iii) $\{\} : \mathsf{A}\uparrow \sim \mathsf{C}\uparrow$, (iv) $\{\mathsf{A}\} : \mathsf{B}\downarrow \sim \mathsf{C}\uparrow$, and (v) $\{\mathsf{A}, \mathsf{C}\} : \mathsf{B}\downarrow \sim \mathsf{D}\uparrow$. In this example, a single list-based bOD is mapped to five set-based bODs. Note that different list-based bODs can map to similar set-based bODs. The list-based bOD $[\mathsf{B}\downarrow, \mathsf{A}\uparrow] \mapsto [\mathsf{C}\uparrow]$ for example shares the set-based bOD (i) and the list-based bOD $[\mathsf{A}\uparrow, \mathsf{B}\downarrow] \mapsto [\mathsf{D}\uparrow, \mathsf{C}\uparrow]$ shares the set-based bODs (i) and (ii) with the example bOD $[\mathsf{A}\uparrow, \mathsf{B}\downarrow] \mapsto [\mathsf{C}\uparrow, \mathsf{D}\uparrow]$. This means that the number of set-based bODs is not necessarily larger than the number of list-based bODs from which the set-based bODs have been generated from. The mapping procedure for a single list-based bOD, however, has polynomial complexity (in the size $|\overline{\mathbf{X}}| \times |\overline{\mathbf{Y}}|$) [25].

3.3.2 Mapping from set-based to list-based bODs

DISTOD's results are set-based bODs. If we want to work with list-based bODs, we have to revert the mapping. Set-based bODs can be mapped to list-based bODs as we have already shown in the definitions for the two set-based canonical forms for bODs. According to Definition 6, a constant bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ can be mapped to the list-based bODs $\overline{\mathbf{X'}} \mapsto \overline{\mathbf{X'}\mathsf{A}}$ for all permutations $\overline{\mathbf{X'}}$ of $\overline{\mathbf{X}}$. According to Definition 8, an order compatible bOD $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ can be mapped to a list-based bOD $\overline{\mathbf{X'}\mathsf{A}} \sim \overline{\mathbf{X'}\mathsf{B}}$ for all permutations $\overline{\mathbf{X'}}$ of $\overline{\mathbf{X}}$. If we want to list all valid list-based bODs we have to generate the list-based bODs for all permutations $\overline{\mathbf{X'}}$ of $\overline{\mathbf{X}}$ for each valid set-based bOD. The complexity of listing all permutations of a set is factorial $O(n!)$, where $n = |X|$. This means that the mapping from set-based to list-based bODs is factorial as well.

If we consider the set-based bOD $\{A, B\} : [\,] \mapsto C \uparrow$ to be valid, the list-based bODs $[A \uparrow, B \uparrow] \mapsto [A \uparrow, B \uparrow, C \uparrow]$, $[A \uparrow, B \downarrow] \mapsto [A \uparrow, B \downarrow, C \uparrow]$, $[A \downarrow, B \uparrow] \mapsto [A \downarrow, B \uparrow, C \uparrow]$, $[A \downarrow, B \downarrow] \mapsto [A \downarrow, B \downarrow, C \uparrow]$, $[B \uparrow, A \uparrow] \mapsto [B \uparrow, A \uparrow, C \uparrow]$, $[B \uparrow, A \downarrow] \mapsto [B \uparrow, A \downarrow, C \uparrow]$, $[B \downarrow, A \uparrow] \mapsto [B \downarrow, A \uparrow, C \uparrow]$, and $[B \downarrow, A \downarrow] \mapsto [B \downarrow, A \downarrow, C \uparrow]$ are valid as well. Because C is a constant w. r. t. the context $\{A, B\}$, all permutations of A and B and all combinations of order directions must be considered for the list-based bODs. It does not matter if the values of the attributes are sorted ascending or descending. For any unique combination of the values in A and B the values in C are the same, otherwise there would be a *split* and the set-based bOD $\{A, B\} : [\,] \mapsto C \uparrow$ would not be valid. Set-based bODs are thus more concise in their representation. We do not have to list all possible attribute and order direction combinations for the left hand-side of the bODs. This also shows that the order direction of C is neglectable as well. $\{A, B\} : [\,] \mapsto C \downarrow$ is equivalent to $\{A, B\} : [\,] \mapsto C \uparrow$ [25, Axiom 9 Reverse-I in Figure 5].

The order compatible bOD $\{A\} : B \uparrow\sim C \downarrow$ is mapped to the two list-based bODs $[A \uparrow, B \uparrow] \mapsto [A \uparrow, C \downarrow]$ and $[A \downarrow, B \uparrow] \mapsto [A \downarrow, C \downarrow]$. For the list-based bODs, we have to consider both order directions for the attribute A. This makes sense, because the relation between B and C stays the same regardless of whether we sort the relation by $A \uparrow$ breaking ties with $B \uparrow$ and compare it with sorting the relation by $A \uparrow$ breaking ties with $C \downarrow$ or sort the relation by $A \downarrow$ breaking ties with $B \uparrow$ and compare it with sorting the relation by $A \downarrow$ breaking ties with $C \downarrow$.

While the mapping from set-based to list-based bODs is factorial, it is performed on a much smaller scale. We have to perform the mapping for only those results of the discovery algorithm that are interesting to us (which are usually results that contain relevant attributes). In general, we can assume that only a subset of all valid bODs is interesting and most interesting bODs are small. Hence, the factorial mapping is performed on only small bODs with few involved attributes. This means that our n for the mapping will in practice be much smaller compared to the number of all attributes in the relation.

3.4 Actor programming model

Discovering bODs in a relational dataset is hard due to the large search space of bODs. Even if we use the set-based canonical representation for bODs, the number of candidates grows exponential with the number of attributes in the dataset [25]. In addition, an algorithm that discovers bODs takes considerably more time to validate candidates than an FD algorithm because it has to check not only for *splits* (Definition 7) but also for *swaps* (Definition 9). Current state-of-the-art bOD discovery approaches, therefore, employ effective pruning strategies that shrink the search space to avoid unnecessary expensive checks. This improves the algorithm runtime by orders of magnitude [25].

In a distributed setting, however, consistently pruning the search space and exchanging intermediate results is challenging. Depending on the dataset at hand the checks take a different amount of time, pruning might be more or less effective, and data exchanged between the nodes might have different sizes. A naive distribution approach that traverses

the whole search space would in praxis not be able to compete with the state-of-the-art bOD discovery approaches. A distributed algorithm must, therefore, be able to efficiently prune the search space and to deal with the dynamic behavior while fully using all available compute resources.

Given the dynamic and explorative nature of our bOD discovery task, data processing tools, such as Apache Spark [31] or Apache Flink [30], cannot exploit the available resources well, because for dynamic workloads they need to introduce synchronization barriers and shuffling operations. This means that they do not use the available compute resources in an optimal way. At synchronization barriers, for example, the processing continues only if all previous parallel tasks have finished. If one task is significantly slower than the other ones, most of the computing resources are not used until the last task finished. Shuffling operations transfer data between nodes and depending on the data sizes this can take a significant amount of time. To deal with the necessary dynamic control flow and to reduce the data communication costs to the bare minimum, we use the actor programming model. It is a reactive concurrency model and abstraction that avoids blocking and locking via isolation and asynchronous message passing. The core primitive in this model are actors, which are objects comprised of state and behavior. Actors can execute tasks in parallel and their only way to communicate are asynchronous messages. Incoming messages are stored in a mailbox, which allows their independent processing. An actor's state is accessible only by its own behavior and any state manipulation follows single-threaded semantics. This encourages shared-nothing system architectures. The strong isolation of actors and their lock-free concurrency model allow us to develop a highly scalable and dynamic algorithm. We use the Akka toolkit [14] for actor programming on the Java Virtual Machine (JVM) and implement our algorithm in the Scala programming language [7].

4 Efficient distributed bOD discovery

We implement our novel, scalable, robust and elastic bOD discovery approach as the algorithm DISTOD. DISTOD is executed on a number of compute machines, also called nodes, which are connected via a network forming the DISTOD cluster. Throughout the thesis, we assume that all involved machines are connected to each other via a network and that each machine can connect to all other machines. DISTOD comes with its own discovery service that DISTOD nodes can use to connect to the DISTOD cluster. We assume an asynchronous network model in which messages can be arbitrarily dropped, delayed, and reordered. We further assume that machines do not fail and that the network is always available. DISTOD does not tolerate node or network failures, but can deal with dropped or delayed messages. If a node disconnects from the cluster, DISTOD tries to reconnect the node for 30 seconds. After that period, the node is removed from the cluster and the current tasks of the `Worker`s on this node are lost. The rest of the DISTOD cluster will continue discovering bODs and all other results are safely stored on disk. In the case of node or network failures, DISTOD does not output complete results.

We introduce the DISTOD algorithm in Section 4.1 and its architecture in Section 4.2. Section 4.3 briefly describes DISTOD's startup and shutdown procedures and important configuration options.

4.1 DISTOD algorithm

We present an efficient distributed bOD discovery algorithm, DISTOD, that traverses a lattice of all possible sets of attributes breadth-first. Figure 1 shows a snapshot of such a set-lattice for the attributes $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$, where some nodes have already been processed by DISTOD (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 candidate lattice. When processing node X in the lattice, the following bODs are checked: Constant bODs of the form $X \setminus \{\mathsf{A}\} : [\,] \mapsto \overline{\mathsf{A}}$, where $\mathsf{A} \in X$ and order compatible bODs of the form $X \setminus \{\mathsf{A}, \mathsf{B}\} : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$, where $\mathsf{A}, \mathsf{B} \in X$ and $\mathsf{A} \neq \mathsf{B}$. DISTOD's small-to-large search strategy is due to our decision to reuse the minimality definition and pruning strategies from FASTOD-BID [25]. Only minimal and valid bODs in terms of the definitions from [25] are added to the result set (see Section 5).

Despite that DISTOD is similar to the FASTOD-BID algorithm by Szlichta et al. [25], it does not build the candidate lattice level-wise. It uses a task-based approach, where candidate generation, validation, and pruning are interleaved. This breaks up the synchronization barriers between the three steps and allows us to use the available resources in our distributed environment in an optimal way. In theory, our algorithm still follows the following high-level steps also used by FASTOD-BID: (i) DISTOD initializes all data structures and generates the initial candidates of the first level directly from the dataset. (ii) It performs the validation of the current bODs candidates. (iii) DISTOD prunes nodes from the candidate lattice. (iv) Finally, it generates the next candidates using the information

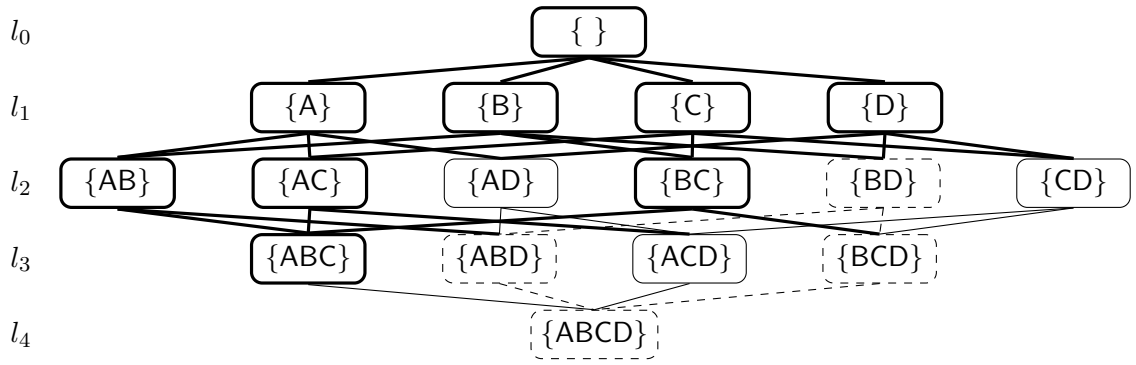
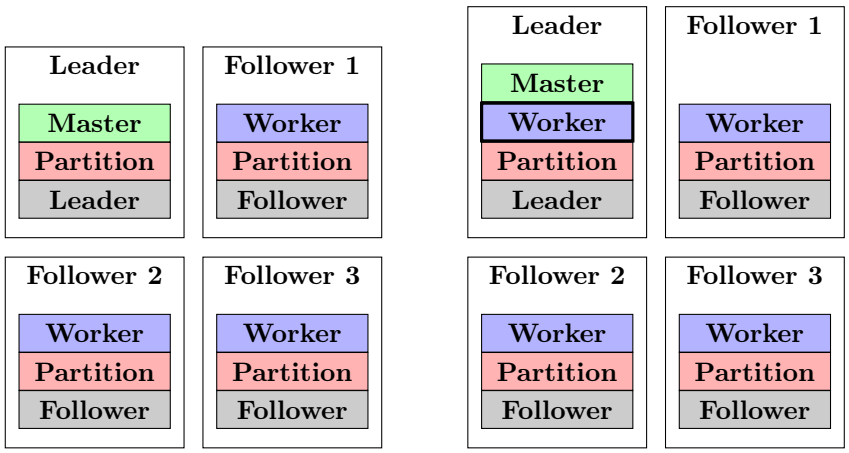


Figure 1: Snapshot of a set lattice for the attributes $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$. Bold nodes have already been processed by DISTOD, thin nodes still await processing, and dashed nodes have been pruned from the lattice.

from the previous levels and starts over from step (ii) until all candidates have been processed. However, in our implementation these steps occur concurrently for different nodes in the lattice and we do not strictly enforce that a level l_i has been completely checked before the checks in the next level l_{i+1} can be 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. This means, if we take a snapshot of the candidate lattice of a running instance of DISTOD, it could look like Figure 1. Note that node $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ of level l_3 has already been processed besides that three nodes of level l_2 still await processing (nodes $\{\mathsf{A}, \mathsf{D}\}$, $\{\mathsf{B}, \mathsf{D}\}$, and $\{\mathsf{C}, \mathsf{D}\}$). In the example, DISTOD's rules ensured that only node $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ from level l_3 was generated and checked, because only the preconditions for this node were met. All other nodes in l_3 could not be processed yet. In contrast to FASTOD-BID, DISTOD performs the minimality check and takes the pruning decision already during the candidate generation step. A single central actor, the master actor, is responsible for maintaining a consistent view on the candidate lattice and it takes these decisions. After the candidates have been generated, they can be checked independently from each other, which allows us to distribute these checks to different nodes. We describe the candidate generation procedure in Section 5 and the validation of the candidates in Section 6. Section 7 explains how data is managed in DISTOD.

4.2 DISTOD architecture

The DISTOD algorithm is implemented in a distributed system that runs concurrently on a set of compute nodes, called cluster. We first describe the two roles of the cluster nodes and the different cluster setups before we introduce the different system components. The system components are represented using actors, because we use the actor programming model to implement concurrency and distribution.



(a) *Passive leader* (b) *Active leader*

Figure 2: Exemplary *passive leader* and *active leader* cluster setups for DISTOD.

The DISTOD system consists of a cluster with a variable number of nodes that have two different roles. One of the nodes is selected as the cluster leader. It is the initial connection point for the cluster setup and it is responsible for managing the cluster. If the leader is stopped or dies, all other nodes will stop as well and the DISTOD system terminates. The other nodes in the DISTOD cluster are called followers. The single leader hosts the master component that is responsible for the generation of minimal candidates and all pruning decisions. It maintains consistency in those decisions and holds the state of the algorithm, which contains intermediate results and the bOD candidates. The master component on the leader node generates the bOD candidates and distributes validation jobs to the follower nodes. The results of the validation jobs are sent back to the leader node by the follower nodes. This approach requires the user to select one node of the cluster to be the leader of the DISTOD system. We assume that all input data physically resides on the leader node. On algorithm startup, DISTOD automatically replicates the necessary data to the other nodes in the system and writes the final results back to the leader node's disk.

DISTOD follows the actor programming model. The DISTOD system, therefore, consists of different actors that communicate using message-passing to achieve the common goal to find all valid and minimal bODs in a given dataset. Each node in the DISTOD cluster runs a set of actors which are grouped into five planes:

Master plane The master plane consists of the master components. Its actors are tasked with input, output, and state management as well as candidate generation. The master plane is available only on the leader node.

Worker plane The worker plane contains actors responsible for the validation of bOD candidates and sending them back to the master components. The actors in this plane can be spawned on all nodes.

Partition management plane This plane hosts the actors storing the partitions (see Definition 5 on Page 11), which are used to validate bOD candidates. This plane is available on all nodes in the cluster.

Leader plane The leader plane contains actors controlling the shutdown procedure and the replication of the initial partitions. It is available on the leader node only.

Follower plane The follower plane contains puppet actors for the shutdown procedure and the partition replication. They are directly controlled by the corresponding actors in the leader plane and steer the local parts of both processes on the follower nodes. They are placed only on the follower nodes.

The leader node usually runs actors from the master, the partition management, and the leader plane and the follower nodes run actors from the worker, the partition management, and the follower planes. Figure 2a shows an exemplary DISTOD cluster consisting of a single leader node and three follower nodes. In this setup – we call it *passive leader* setup – the leader node does not include actors from the worker plane and therefore does not perform expensive bOD candidate validations. If the leader node is a powerful machine and we use the *passive leader* setup, we would not use the available resources in the cluster in an optimal way, because most of the cores on the leader node would be idle. We, therefore, propose a second cluster setup, called *active leader* setup, where the leader node not only consists of the master, partition management, and leader planes but also includes the worker plane. This setup is shown in Figure 2b. The leader node can now contribute its resources to the expensive candidate validation tasks, which allows us to utilize more compute resources to finish the overall task of finding all minimal bODs in a dataset even faster. The master actors are run on separate threads with a higher thread priority than the worker actors and the number of worker actors can be configured separately for each node. This ensures that the leader node remains reactive and can answer requests from the other nodes besides the higher CPU utilization caused by the additional worker actors.

Figure 3 shows DISTOD's architecture consisting of the actor planes and their actors. The actors are depicted with rounded corners. If some actors work together on the same logical function, they are grouped together under a name. This is the case for example for the `Master` actor and the `MasterHelper` actors under the name `Master`. We refer to the group by the name `Master` and to the actual actor by the name `Master` *actor* to distinguish between them. The same applies to the `DataReader` and the `PartitionManager` groups.

DISTOD uses the master-worker pattern to work on the validation tasks in parallel. The `Master` is responsible for creating and maintaining the candidate lattice. It generates the bOD candidates, creates validation jobs, and distributes them to the `Worker` actors. The `Master` actor maintains a consistent view on the lattice, because all modifications have to pass through the `Master` actor. 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 dispatching jobs to the `Worker`s. Section 5 describes how we ensure minimality and consistent pruning of bODs.

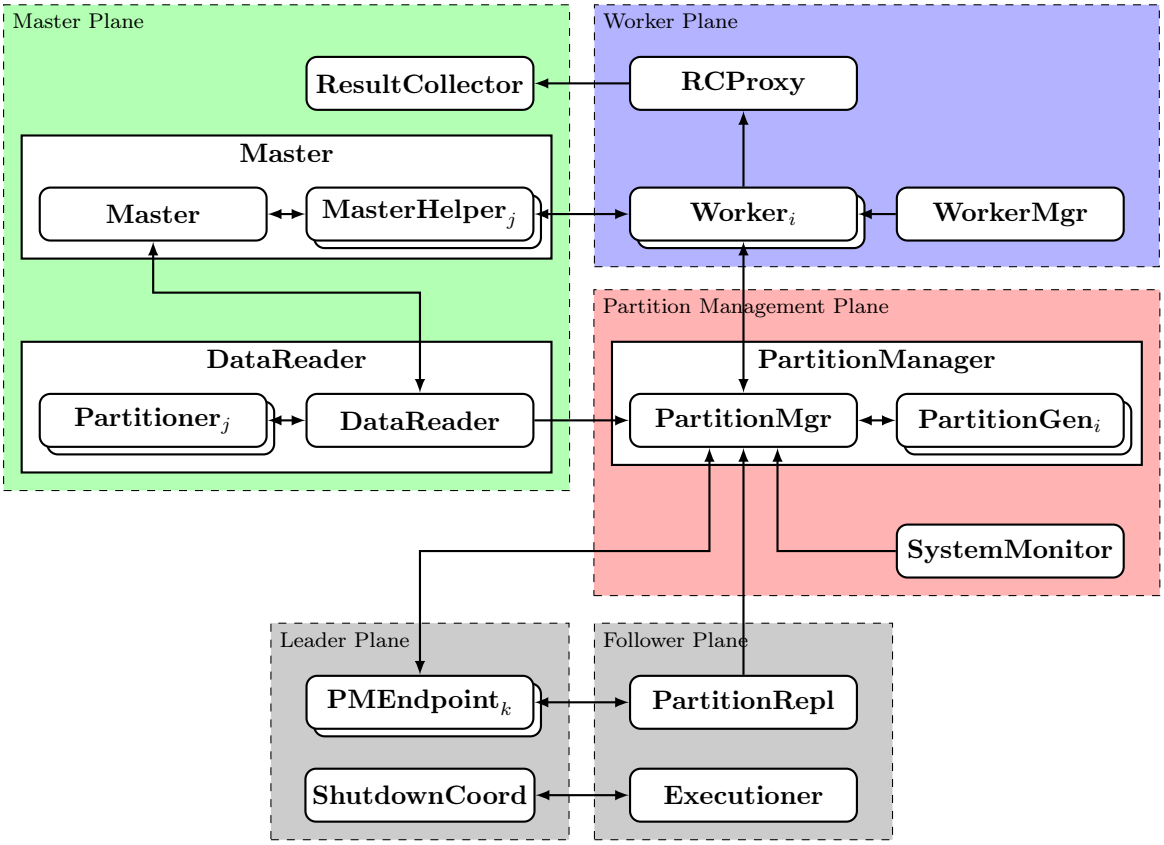


Figure 3: DISTOD's architecture consisting of multiple actors grouped into logical planes. Multiple incarnations of the same actor type are indicated with indices i, j, and k, where $i \in [0 \dots \min(\texttt{max-workers}, \texttt{max-parallelism})]$, $j \in [0 \dots \texttt{max-parallelism}]$, and $k \in [0 \dots \texttt{\#nodes}]$. The variables `max-workers` and `max-parallelism` are configuration options that can be set during deployment. They default to the number of available cores of the machine where DISTOD is executed on. Unidirectional arrows indicate that messages are send only from the source of the arrow to the target. Bidirectional arrows, however, indicate that both actors send messages to each other; usually as request-response pairs.

The `Worker` actors perform the validation of the bOD candidates, because this is the part of the algorithm that requires most of the processing time. The `Worker` actors are supervised by a `WorkerMgr` actor. If a `Worker` fails, the `WorkerMgr` stops it and starts a new one, so that our system operates at full capacity at all time. The `Worker`s emit valid bODs to the local `RCProxy`. This very fast operation allows the `Worker`s to start with a new validation job without waiting for result transmission. The `RCProxy` collects valid bODs from multiple `Worker`s 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 a `RCProxy` is immediately flushed to disk in an asynchronous way. This means that DISTOD outputs valid bODs progressively to a file on disk while the algorithm is running. If the result set is satisfactory, DISTOD can be stopped early.

The validation of bODs is performed using partitions (see Definition 5) 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 for each column in the dataset. The initial partitions are sent to the `PartitionManager`, which stores the initial partitions and generates refined partitions. The `PartitionMgr` actor acts as a cache for intermediate partitions. All requests for partitions from the `Worker`s first reach the `PartitionMgr` actor. If the 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 refinement as described in Section 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 in a cluster setup, some additional actors for the cluster management are needed. In Figure 3, they are depicted in the grey leader and follower planes at the bottom of the figure. The bOD validation checks performed by the `Worker` actors require access to the partitions, which are generated from the initial partitions. We, therefore, replicate the initial partitions to all nodes in the DISTOD cluster. Each follower node has a temporary `PartitionRepl` actor that connects to the corresponding `PMEndpoint` actor on the leader node to replicate the initial partitions from the primary `PartitionMgr` actor on the leader node to the `PartitionMgr` actor on the follower node. The `PartitionRepl` actor and the `PMEndpoint` actor implement a side-channel for partition replication, which is described in Section 7.2. In addition to the partition replication at the start of the algorithm, we also have to make sure that all nodes of the DISTOD cluster shut down at the end of the algorithm together. This is handled by a coordinated shutdown protocol that is implemented using the `ShutdownCoord` actor on the leader node and the `Executioner` actors on the follower nodes. The `ShutdownCoord` initiates and supervises the shutdown procedure and the `Executioner`s ensure that the local nodes, where they run on, shut down cleanly before reporting back to the `ShutdownCoord`. Section 4.3 describes the startup and shutdown of the DISTOD system in more detail, including the mentioned coordinated shutdown protocol.

4.3 DISTOD startup and shutdown

In this section, we quickly describe how DISTOD starts up and shuts down. We highlight important configuration options and show user interaction points. All configuration options can be set for each node individually.

DISTOD can be run on a single node or multiple nodes in a cluster. DISTOD requires all nodes in the cluster to be connected to the same network. Since DISTOD is written in Scala, it requires a JVM to be installed on the nodes. To use DISTOD most effectively, a cluster-setup with an active leader is preferred. If DISTOD is used on a single node, we must be sure that we use the *active leader* configuration as well, otherwise no work will be performed by DISTOD. We use the configuration option `max-workers` on the leader node to specify the maximum number of `Worker` actors that will be spawned there. If this option is set to 0, the leader will have no `Worker` actors and switch to the *passive leader* mode. The actual number of `Worker` actors is determined by the function min(`max-workers`, `max-parallelism`). The option `max-parallelism` restricts the parallelism of the other components, such as the `MasterHelper` actors or the `PartitionGen` actors. The number of `Worker` actors can never exceed `max-parallelism`. Both options default to the number of available CPU cores of the current node.

4.3.1 Startup

Before we can start the algorithm, the DISTOD binary and the configuration files must be available on all nodes of the DISTOD cluster and the input dataset must be physically available on the leader node. We start DISTOD on each node individually. The master components on the leader node then immediately read the input dataset, parse the columns, infer their types, and create the initial partitions for the first two levels (l_0 and l_1) of the candidate lattice (see Figure 1). The partitions are sent to the primary `PartitionMgr` on the leader node. They are now available to be replicated to the follower nodes. At the same time, the `Master` actor initializes the first two levels of the candidate state and generates the initial candidates for level l_1. They are ready to be validated and put into the work queue. This marks the point from which on the leader node is ready to distribute validation jobs to the `Worker` actors on the local or remote nodes. If it runs in *active leader* mode, it immediately starts working on the first few candidate validation jobs. In the meantime, all nodes running DISTOD connect to each other and exchange their addresses to form the DISTOD cluster. As soon as a follower node connects to the leader node, it starts replicating the initial partitions. After this process has finished, the follower is ready to validate candidates as well. It starts the validation process by requesting candidate validation jobs from the `Master`. The start of the discovery algorithm is not synchronized across nodes. As long as the leader is ready and as soon as a specific follower node connects to it, this follower can start validating candidates. This reactivity improves resource usage, because we begin processing candidates as soon as possible and we do not have to wait until all nodes are ready. Section 8 follows up on this aspect of DISTOD by introducing DISTOD's elastic properties.

4.3.2 Shutdown

DISTOD implements a coordinated shutdown protocol that ensures that all results are flushed to disk and that all nodes terminate when the algorithm is finished. This protocol is implemented in the leader and follower plane using the `ShutdownCoord` actor on the leader node and the `Executioner` actors on the follower nodes. If the `Master` actor on the leader node determines that the algorithm is finished or the leader node is stopped using an operating system signal (by the user or the system), the coordinated shutdown protocol is triggered. The `ShutdownCoord` actor starts the protocol by notifying all follower nodes' `Executioner` actors about the shutdown and its reason. For this to work, all followers register themselves at the `ShutdownCoord` actor on the leader node during startup. The `Executioner`s first tell their local `Worker` actors to stop. The `Worker`s immediately stop requesting more work from the `Master`, finish their current subtask, and cancel their candidate validation jobs at the `Master` by putting them back into the `Master`'s work queue. When all `Worker`'s have stopped, the `RCProxy` is told to flush its buffered results to the `ResultCollector` actor on the leader node, where they are written to disk. When the `Worker` actors and the `RCProxy` actor have shut down, the `Executioner` reports back to the `ShutdownCoord` and shuts down the local node. If all follower nodes have shut down, the `ShutdownCoord` actor orders the `ResultCollector` to write all results to disk and shuts down the leader node as well. The leader node finishes its shutdown sequence not before the `ShutdownCoord` received the notification of successful shutdown of all follower nodes or after a user-defined timeout.

5 Candidate generation

To guarantee complete and correct results, existing algorithms (i. a. [5], [10], [13], [25], [26]) employ a shared data structure that tracks the results of the candidate validations. This data structure is also used to check the minimality constrains during candidate generation. To completely distribute our bOD discovery approach, we would need to distribute the information from this data structure across the nodes in our cluster to allow a distributed candidate generation. However, synchronizing the local data across the compute nodes would introduce a high communication overhead between the nodes (the candidate data and intermediate results change very frequently). This drastically reduces the algorithm's efficiency. Instead, DISTOD also uses a centralized approach to track the results of the candidate validations and to generate the bOD candidates. A central component on the leader node, the `Master`, watches over intermediate results and ensures completeness and correctness of the algorithm. It performs the generation of bOD candidates, the minimality checks, and the node 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 precalculated pruning data are sent to the `Master`, which integrates them into its encapsulated state and considers them for the pruning decisions.

In summary, DISTOD's candidate generation is centralized and generates only minimal and non-prunable bOD candidates. We introduce the definition for minimal bODs in Section 5.1. Section 5.2 discusses how DISTOD ensures that only minimal bOD candidates are generated. In Section 5.3, we then explain DISTOD's candidate generation algorithm in detail.

5.1 Minimal bODs

Like other dependency discovery algorithms, DISTOD outputs only the minimal bODs of a dataset. Non-minimal bODs are not considered, but can later be inferred from the result set using the axioms for set-based bODs introduced by Szlichta et al. [25, Figure 5]. We use the same minimality definition as the FASTOD-BID algorithm [25]. It allows us to benefit from their highly effective pruning rules.

A constant bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ is minimal if (i) it is not trivial, (ii) $X \setminus \{\mathsf{B}\} : [\,] \mapsto \overline{\mathsf{B}}$ is not valid, and (iii) there is no context $Y \subset X$, such that $Y : [\,] \mapsto \overline{\mathsf{A}}$ holds in the relational instance r. A constant bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ is trivial if $\mathsf{A} \in X$. An order compatible bOD $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ is minimal if 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. An order compatible bOD $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ is trivial if $\mathsf{A} \in X$, $\mathsf{B} \in X$, or $\overline{\mathsf{A}} = \overline{\mathsf{B}}$ [25].

We can further reduce the number of bODs that we have to consider in our discovery algorithm by eliminating bODs with similar semantics [25]. Constant bODs of the form $X : [\,] \mapsto \mathsf{A} \uparrow$ and $X : [\,] \mapsto \mathsf{A} \downarrow$ are semantically equivalent, because sorting a constant

sequence in either direction always yields the same sequence (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]. Therefore, we just have to find 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, this leads us to the following minimality pruning rules:

1. We check only bOD candidates of the forms $X : [\,] \mapsto \mathsf{A}\uparrow$, $X : \mathsf{A}\uparrow \sim \mathsf{B}\uparrow$ and $X : \mathsf{A}\uparrow \sim \mathsf{B}\downarrow$.

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 $X : [\,] \mapsto \mathsf{B}\uparrow$.

To ensure that we consider only minimal bODs, we have to check the candidates of a node X for minimality using all the above listed minimality pruning rules. This means that we have to track the results of the candidate validations. Similar to FASTOD-BID [25], we track information about minimal bODs by storing non-valid candidates. Each node X in the candidate lattice stores its non-valid constant bOD candidates in the candidate set $\mathcal{S}(X).C_c$ and non-valid order compatible bOD candidates in the candidate set $\mathcal{S}(X).C_o$.

Definition 11 $\mathcal{S}(X).C_c = \{\mathsf{A} \in \mathrm{R} \quad | \forall \mathsf{B} \in X : X \setminus \{\mathsf{A}, \mathsf{B}\} : [\,] \mapsto \mathsf{B}\uparrow$ *does not hold*$\}$ *[10, Lemma 3.3].*

If a constant bOD candidate $\mathsf{A} \in \mathcal{S}(X).C_c$ 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 $\forall \mathsf{B} \in X : \mathsf{A} \in \mathcal{S}(X \setminus \{\mathsf{B}\}).C_c$ (Rule 2b). The same technique is used for order compatible bOD candidates:

Definition 12 $\mathcal{S}(X).C_o = \{(\overline{\mathsf{E}}, \overline{\mathsf{F}}) \mid (\overline{\mathsf{E}}, \overline{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\uparrow)$ *or* $(\mathsf{A}\uparrow, \mathsf{B}\downarrow), (\mathsf{A}, \mathsf{B}) \in X^2, \mathsf{A} \neq \mathsf{B}$ *and* $\forall \mathsf{C} \in X : X \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{C}\} : \overline{\mathsf{E}} \sim \overline{\mathsf{F}}$ *does not hold, and* $\forall \mathsf{C} \in X : X \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{C}\} : [\,] \mapsto \mathsf{C}\uparrow$ *does not hold*$\}$ *[25, Definition 11].*

If an order compatible bOD candidate $(\overline{\mathsf{E}}, \overline{\mathsf{F}}) \in \mathcal{S}(X).C_o$ for a specific node X, where $(\overline{\mathsf{E}}, \overline{\mathsf{F}}) = (\mathsf{A}\uparrow, \mathsf{B}\uparrow)$ or $(\mathsf{A}\uparrow, \mathsf{B}\downarrow)$, $\mathsf{A} \in X$, 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.

All three minimality pruning rules eventually lead to [25, Lemma 12], which states that if for a node X, where $|X| \geq 2$, both candidate sets $\mathcal{S}(X).C_c$ and $\mathcal{S}(X).C_o$ are empty, all succeeding nodes' candidate sets $\mathcal{S}(Z).C_c$ and $\mathcal{S}(Z).C_o$, where $Z \supset X$, 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 these nodes from the candidate lattice by storing whether a node X should be considered or not in the a flag $\mathcal{S}(X).p$. If a node X gets pruned, all the successors in the candidate lattice are marked 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 the `Worker` actors. The distributed `Worker` actors 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 the node's predecessors. For example, Figure 4 depicts the inter- and intra-node dependencies for the subtasks of a node X and its two predecessor nodes W_0 and W_1, where $|X| = 2$ and $W_i = X \setminus \{\mathsf{X}_i\}$. We discuss the

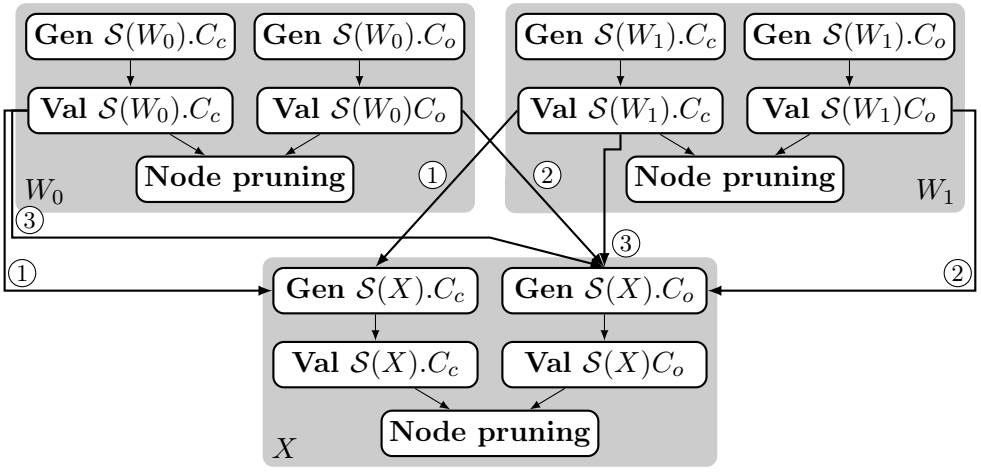


Figure 4: Inter- and intra-node dependencies for the subtasks of a node X and its two predecessor nodes W_0 and W_1 with $W_i = X \setminus \{\mathsf{X}_i\}$. In this example, $|X| = 2$, but the dependency graph generalizes to $i > 2$ by adding further W_i. The arrow on an edge points to the subtask that depends on the subtask at the source of the edge. A subtask can only be executed when all incoming dependencies are fulfilled. Regarding the minimality pruning rules for bOD candidates, ① labelled edges enforce Rule 2b; ② labelled edges enforce Rule 3b; ③ labelled edges enforce Rule 3c and Rule 3d.

dependencies between the subtasks by the example shown in Figure 4, but the graph established by the dependencies generalizes to $i > 2$. The generation of the minimal constant bOD candidates of node X can be performed after all constant bOD candidates of the set of predecessor nodes $W = \{W_i = X \setminus \{\mathsf{X}_i\}\}$ have been validated ① to allow checking Rule 2b. To guarantee the minimality of the order compatible candidates $\mathcal{S}(X).C_o$ of node X, we have to follow Rule 3b, Rule 3c, and Rule 3d. Rule 3b requires the results of the order compatible bOD validations of all W_i ②. Rule 3c and Rule 3d require the results of the constant bOD validations of all W_i ③. $\mathcal{S}(X).C_o$ 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, we can check its candidate states $\mathcal{S}(X).C_c$ and $\mathcal{S}(X).C_o$ 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 predecessor nodes are done, DISTOD's candidate generation steps (i.e., the Gen $\mathcal{S}(X).C_c$ and Gen $\mathcal{S}(X).C_o$ boxes in Figure 4) do not depend on the node pruning step. Therefore, the candidate generation subtask may occur before the node pruning subtasks of the previous nodes have finished. This means that DISTOD may perform needless work, i. e., DISTOD would generate and validate an empty constant bOD candidate set, which does not change the algorithm's results. We intentionally unravel the dependencies of the candidate generation subtasks

in order to validate them independently from each other in the candidate validation steps. This allows for a more fine-grained work distribution improving the resource utilization on all nodes because DISTOD distributes the validation of candidates in encapsulated jobs (i.e., the Val $\mathcal{S}(X).C_c$ and Val $\mathcal{S}(X).C_o$ boxes in Figure 4) to the `Worker` actors on the follower nodes. Node pruning is handled downstream.

The dependencies between nodes in our lattice restrict the order in which we can generate minimal bOD candidates. If we would not follow this order, we would not be able to guarantee the minimality of our results. In order to enforce the dependencies shown in Figure 4, each node in the candidate lattice has two counters that track the number of predecessor nodes, for which the constant and order compatible bOD validations respectively have already been performed. The counter $\mathcal{S}(X).i_c$ for node X in the candidate lattice stores the number of predecessor nodes W_i, for which the constant bOD candidates have already been validated ($\mathcal{S}(W_i).f_c$ is TRUE). The counter $\mathcal{S}(X).i_o$ for node X in the candidate lattice stores the number of predecessor nodes W_i, for which the order compatible bOD candidates have already been validated ($\mathcal{S}(W_i).f_o$ is TRUE). Each node X in the candidate lattice has $|X|$ predecessor nodes. For example, the node $X = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, has the three predecessors $W_1 = \{\mathsf{A}, \mathsf{B}\}$, $W_2 = \{\mathsf{A}, \mathsf{C}\}$, and $W_3 = \{\mathsf{B}, \mathsf{C}\}$ (cf. Figure 1). The counters allow the algorithm to check if it can start generating the minimal bOD candidates for a node X without checking all the dependencies to the other nodes W_0 to $W_{|X|-1}$. Instead, every successful validation triggers counter increments in all its dependent nodes and with them the check if a node is ready to generate either C_c or C_o candidates. If $\mathcal{S}(X).i_c = |X|$ ①, we can check Rule 2b to generate minimal constant bOD candidates for node X. If $\mathcal{S}(X).i_o = |X|$ ② and $\mathcal{S}(X).i_c = |X|$ ③, we can check Rule 3b, Rule 3c, and Rule 3d in order to generate minimal order compatible bOD candidates for node X. Note that the counter $\mathcal{S}(X).i_c$ is reused for ① and ②. The actual generation of the candidates then follows the dependencies to the predecessor nodes W_i to access the candidate sets $\mathcal{S}(W_i).C_c$ and $\mathcal{S}(W_i).C_o$ for the minimality checks (see Algorithm 6). In DISTOD, the actual generation of the minimal bOD candidates is done only once per node, but we need to check if we are able to generate minimal candidates more often. DISTOD does not materialize the whole candidate lattice a priori, but creates a node in the lattice upon the node's first usage; usually this is the increment of one of the two counters.

5.3 Candidate generation algorithm

The candidate generation is handled by the `Master` component on the leader node. To increase the performance of DISTOD, the `Master` component is parallelized and consists of two types of actors: the single `Master` actor and a pool of multiple `MasterHelper` actors. The `Master` actor performs the state manipulations and, thus, ensures consistency. All other operations, such as preparing the state updates and generating the bOD candidates, are executed by the `MasterHelper` actors. The pool of `MasterHelper` actors is interposed between the `Master` actor and the `Worker` actors (Figure 5). All messages to and from the `Worker`s are handled by the `MasterHelper`s. The `Master` actor communicates only with the `MasterHelper` actors and, during the initialization, with the `DataReader` actor. The messages to the `MasterHelper` actors are scheduled in a round robin fashion.

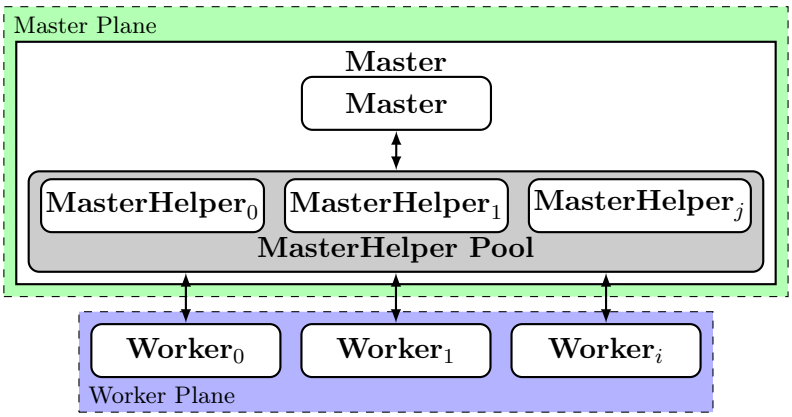


Figure 5: `Master` component on the leader node consisting of the single `Master` actor and multiple `MasterHelper` actors that proxy requests from the `Worker` actors to the `Master` actor.

The `Master` actor is the sole entity that can manipulate the candidate state $\mathcal{S}$ and the job queue $\mathbf{Q}$. However, the `MasterHelper` actors have read-only access to the candidate state $\mathcal{S}$. This allows us to concurrently pre- and post-process the messages from the `Worker`s and to parallelize the candidate generation procedure. We intentionally break the actor paradigm by sharing $\mathcal{S}$ between the `Master` and `MasterHelper` actors for performance reasons. The `Master` component becomes the performance bottleneck in DISTOD for wide datasets, because it has to access each node in the candidate lattice multiple times and the number of nodes grows exponentially with the number of attributes in the dataset. To control the accesses to the shared state $\mathcal{S}$, we use concurrent hashmaps to implement $\mathcal{S}$ in a thread-safe way[2]. They allow any number of parallel reads to $\mathcal{S}$ without locking. Synchronized writes to $\mathcal{S}$ are only performed by the single `Master` actor and do not lock an entire map, but only a segment of it, which allows other actors to read from the other segments concurrently. We use a separate concurrent hashmap for each level in the candidate lattice. This limits the size of the maps and the scope of their write-locks.

For each node X in the candidate lattice, we store an entry in $\mathcal{S}$. The entry consists of the node's constant bOD candidate set $\mathcal{S}(X).C_c$, its order compatible bOD candidate set $\mathcal{S}(X).C_o$, whether its constant bOD candidates have been checked $\mathcal{S}(X).f_c$, whether its order compatible bOD candidates have been checked $\mathcal{S}(X).f_o$, if the node is pruned $\mathcal{S}(X).p$, and the counters for the constant bOD preconditions $\mathcal{S}(X).i_c$ and the order compatible preconditions $\mathcal{S}(X).i_o$. If not otherwise specified, flags (f_c, f_o, and p) are FALSE per default and the counters (i_c and i_o) start with 0.

At the beginning of the algorithm, the `Master` component initializes its data structures from the input dataset using Algorithm 1, which places the constant bOD candidate validation jobs for the nodes in level l_1 into the job queue $\mathbf{Q}$. The candidates of the following nodes, where $|X| \geq 2$, cannot yet be generated because they require the validation results of the previous nodes – starting with the results from the nodes with $|X| = 1$ – to perform the minimality checks. The candidates for the other nodes are generated dynamically while

[2] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html`

validation results arrive at the leader node (Algorithm 6). After the initialization, the `Worker` actors request validation jobs from the `Master` component. For each request, the `Master` component dequeues a job from **Q** and sends the job to the `Worker` (Algorithm 2). The `Worker` actors perform the validation of the candidates, prepare the pruned candidates, and send the pruned candidates back to the `Master` (Algorithm 3). For each incoming response, the `Master` integrates the results into its state (Algorithm 4). It checks the node pruning condition and the states of the succeeding nodes in the candidate lattice. If the node needs to be pruned, the `Master` performs the node pruning (Algorithm 5). If the preconditions for a succeeding node are met, DISTOD generates the candidates for the succeeding node and puts an according validation job into **Q** (Algorithm 6).

5.3.1 Initialization

Algorithm 1 shows the initialization procedure for the state and the job queue of the `Master` component. The initialization is performed by the `Master` actor. We use the input dataset to initialize the first two levels of the candidates lattice l_0 and l_1 (Line 1-9).

Algorithm 1: Initialize state

Input: Relational instance r of schema R
Data: States $\mathcal{S}$, job queue **Q**

```
   /* Initialize level l_0                                          */
 1 S({}).C_c = R
 2 S({}).C_o = ∅
 3 S({}).f_c = S({}).f_o = TRUE
   /* Initialize level l_1                                          */
 4 forall A ∈ R do
 5 |  S({A}).i_c = S({A}).i_o = 1
 6 |  S({A}).C_c = R
 7 |  add ({A}, CONST) to Q
 8 |  S({A}).C_o = ∅
 9 |  S({A}).f_o = TRUE
   /* Initialize level l_2                                          */
10 forall (A, B) ∈ R^2, where A ≠ B do
11 |  S({A, B}).i_o = 2
```

The first level l_0 contains the single node for the empty attribute set $\{\}$. Its constant bOD candidate set $\mathcal{S}(\{\}).C_c$ is set to R (Line 1) because there are no valid candidates yet. $\mathcal{S}(\{\}).C_o$ is set to $\emptyset$ (Line 2). DISTOD initializes the order compatible bOD candidates not until the candidates for level l_2 are generated, because order compatible bOD candidates require at least two attributes and they depend on the validation results of the previous level's constant bOD validation results. For level l_0, there are no validations to perform, so we set both validation flags to TRUE (Line 3). This means that the generation preconditions in the next level l_1 are already met. We set $\mathcal{S}(X).i_c$ and $\mathcal{S}(X).i_o$, where $|X| = 1$, to 1 (Line 5).

The nodes in level l_1 have a single attribute in their attribute set (Line 4). For these nodes, the constant bOD candidates are ready to be validated. They are the initial validation jobs in the job queue **Q** (Line 7). We still cannot generate any order compatible bOD candidates for nodes with a single attribute. Hence, $\mathcal{S}(X).C_o$ is set to $\emptyset$ (Line 8) and $\mathcal{S}(X).f_o$ is set to TRUE (Line 9), where $|X| = 1$. By implication, no order compatible bOD validations are performed for level l_1, which means that the precondition counters $\mathcal{S}(X).i_o$, where $|X| = 2$, must be set to 2 (Line 10f). As soon as the constant bOD candidates for level l_1 have been validated, we can generate the constant and order compatible bOD candidates for level l_2.

5.3.2 Dispatch validation job

Idle `Worker` actors request work from the `Master` component, which dispatches new jobs to the `Worker`s. This procedure is shown in Algorithm 2. The incoming work requests first arrive at the `MasterHelper` actors, which forward them to the `Master` actor. For each work request, the `Master` actor dequeues a validation job from the job queue **Q** (Line 1) and sends it to one of the `MasterHelper` actors, which prepares the information necessary to process the job (Line 2ff.) before sending it to the `Worker`.

Algorithm 2: Dispatch job (X, ϕ)

Output: Job with node ID X, job type $\phi \in \{\text{CONST}, \text{COMP}\}$, and candidates C
Data: States $\mathcal{S}$, job queue **Q**

```
1 (X, φ) ← Q
2 if φ = CONST then
3 |   C = S(X).C_c ∩ X /* Rule 2a                                  */
4 else
5 |   C = S(X).C_o
6 return X, φ, C
```

For each job, the particular `MasterHelper` actor looks up the job's corresponding validation candidates in the state $\mathcal{S}$ (Line 3 or Line 5). The validation job consists of the node identifier X, a marker for the type of bOD candidates to validate $\phi \in \{\text{CONST}, \text{COMP}\}$, and the respective bOD candidates themselves C (Line 6). If constant bOD candidates should be validated (marker $\phi = \text{CONST}$), trivial constant bOD candidates are removed from the candidate set C in Line 3 before dispatching the job to the worker.

The Algorithm 2 is not executed if the job queue **Q** is empty. In this case, the `Master` actor bookmarks requesting `Worker` actors and does not immediately dispatch jobs to them. As soon as new jobs are put in **Q**, bookmarked `Worker` actors are served with jobs using Algorithm 2 until no jobs are left in **Q** or all bookmarked `Worker`s are busy again. The job queue **Q** is filled by generating new candidates (Algorithm 6). If no `Worker` actor is busy and no jobs are left in **Q**, there are no more nodes with minimal bOD candidates in the lattice and DISTOD is finished.

5.3.3 Validate and prune candidates

Minimality checking and node pruning are performed by the central `Master` component, but the candidate pruning information is pre-computed by the `Worker` actors and sent back to the `Master` component. Algorithm 3 shows the overall candidate validation procedure executed by the `Worker` actors. Section 6 explains how DISTOD validates bOD candidates of the form $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A}\uparrow$ and $X \setminus \{\mathsf{A},\mathsf{B}\} : \overline{\mathsf{E}} \sim \overline{\mathsf{F}}$ in more detail. The `Worker` actors validate the candidates C received with the validation job (X, ϕ) (Line 4, 10) and store invalid and pruned bOD candidates in C_r (Line 6, 7, 12). Valid bODs are emitted to the local `RCProxy`s (Line 5, 11), which forward them to the `ResultCollector` on the leader node. However, the invalid and pruned candidates in C_r are sent back to the `Master` component along with the acknowledgement message for the job (X, ϕ) (Line 13). The acknowledgement message triggers Algorithm 4, in which the `Master` component uses the candidates in C_r to update the candidate states $\mathcal{S}(X).C_c$ (Definition 11) and $\mathcal{S}(X).C_o$ (Definition 12).

Algorithm 3: Validate and prune candidates of job (X, ϕ)

Input : Job with node ID X, job type $\phi \in \{\text{CONST}, \text{COMP}\}$, and candidates C
Output: Node ID X, job type ϕ, and pruned candidates C_r

```
1  C_r = ∅
2  if φ = CONST then
3  |  forall A ∈ C do
4  |  |  if X \ {A} : [ ] ↦ A↑ then
5  |  |  |  emit X \ {A} : [ ] ↦ A↑ as valid bOD
6  |  |  |  add A to C_r
7  |  |  |  add all B ∈ R \ X to C_r
8  else
9  |  forall (Ē, F̄) ∈ C, where (Ē, F̄) = (A↑, B↑) or (A↑, B↓) do
10 |  |  if X \ {A, B} : Ē ∼ F̄ then
11 |  |  |  emit X \ {A, B} : Ē ∼ F̄ as valid bOD
12 |  |  |  add (Ē, F̄) to C_r
13 return X, φ, C_r
```

5.3.4 Update state, perform pruning and generate candidates

Algorithm 4 is executed for each validation job acknowledgement message received by the `Master` component from the `Worker`s. This procedure updates the search state using the new results of the validation job (X, ϕ) in Line 1-6, calls the `pruneNode`(X) function (Algorithm 5) in Line 7f, increments the respective precondition counters of X's successor nodes in Line 15 and Line 17, and performs the candidate generation with a call to `generateCandidates`(X, ϕ) (Algorithm 6) in Line 19.

Algorithm 4: Update state

Input : Node ID X, job type $\phi \in \{\text{CONST}, \text{COMP}\}$, pruned candidates C_r
Data: States $\mathcal{S}$, job queue $\mathbf{Q}$

```
1  if φ = CONST then
2  |  S(X).f_c = TRUE
3  |  remove all A ∈ C_r from S(X).C_c
4  else
5  |  S(X).f_o = TRUE
6  |  remove all (Ē, F̄) ∈ C_r from S(X).C_o
7  if |X| ≥ 2 ∧ S(X).f_c = TRUE ∧ S(X).f_o = TRUE then
8  |  pruneNode(X)
9  else
10 |  W = {W_i | W_i = X \ {X_i}}
11 |  forall W_i ∈ W, where W_i ∉ S or S(W_i).p = FALSE do
12 |  |  if W_i ∉ S then
13 |  |  |  create state S(W_i)
14 |  |  if φ = CONST then
15 |  |  |  inc S(W_i).i_c
16 |  |  else
17 |  |  |  inc S(W_i).i_o
18 if S(X).p = FALSE then
19 |  generateCandidates(X, φ)
```

Algorithm 4 is executed by the `MasterHelper`s, but the modifications to $\mathcal{S}$ and $\mathbf{Q}$ are not performed directly. The `MasterHelper`s record the changes to the state variables up until Line 18 and send them to the `Master` actor. The `Master` actor applies the changes to $\mathcal{S}$ and $\mathbf{Q}$ in a batch operation and checks if the node was pruned (Line 18). If it was not pruned, the `Master` actor instructs another `MasterHelper` to check if it can generate candidates and, if this is the case, to perform the generation (Line 19). The candidate generation procedure is shown in Algorithm 6. The execution of this procedure is split between the `Master` actor and the `MasterHelper`s in the same way. The `MasterHelper`s perform the checks and computations in Algorithm 6, but only record the changes to the state variables. The recorded changes are sent back to the `Master` actor, which applies them to $\mathcal{S}$ and $\mathbf{Q}$.

At the beginning of Algorithm 4, the `Master` component integrates the validation results into its state by marking the specific check of node X as completed (Line 2, 5) and removing the invalid and pruned candidates in C_r from the candidate set $\mathcal{S}(X).C_c$ (Line 3) or $\mathcal{S}(X).C_o$ (Line 6) respectively. If the node X is at least in level l_2 and both validations have been performed (Line 7), we can check whether the node can be pruned from the candidate lattice using Algorithm 5.

Algorithm 5: `pruneNode(`X`)`

Data: States $\mathcal{S}$, job queue $\mathbf{Q}$

```
1 if |S(X).C_c| = 0 ∧ |S(X).C_o| = 0 then
2 |   S(X).p = TRUE
3 |   forall Z_i|Z_i = X ∪ S_i ∧ S = R \ X do
4 |   |   S(Z_i).p = TRUE
5 |   |   remove (Z_i, CONST) from Q
```

Algorithm 5 shows the node pruning procedure. If the node X is prunable (Line 1), the node X itself (Line 2) and all its successors $Z_i = X \cup \mathsf{S}_i$, where $S = \mathrm{R} \setminus X$, are marked pruned (Line 4) and all related validation jobs are removed from $\mathbf{Q}$ (Line 5). It is possible that DISTOD generates the constant bOD candidates for a node X that would later be pruned, because one of the node's predecessors $W_i | W_i = X \setminus \{\mathsf{X}_i\}$ has no remaining candidates in both candidate sets $\mathcal{S}(W_i).C_c$ and $\mathcal{S}(W_i).C_o$. This leads to a validation job (X, CONST) in $\mathbf{Q}$. This validation job does not contain any constant bOD candidates that would be valid. Thus, we can safely remove it from the job queue. If DISTOD is fast enough, it is possible that this validation job has already been dispatched to a `Worker` actor when the `Master` actor tries to remove it from $\mathbf{Q}$ due to node pruning. In this case, DISTOD may perform some needless work, but the correctness of the result set is still guaranteed. Order compatible bOD candidates always get generated after the node pruning check. This means that we do not have to check for redundant order compatible bOD validation jobs in $\mathbf{Q}$.

If there is at least one candidate in any of the two candidate sets $\mathcal{S}(X).C_c$ and $\mathcal{S}(X).C_o$, the node X cannot be pruned and we need to update the precondition counters of the successor nodes of X (Line 9ff in Algorithm 4). The `Master` iterates over all successor nodes $Z_i = X \cup \mathsf{S}_i$, where $S = \mathrm{R} \setminus X$, in the candidate lattice that have not been pruned

yet and increments the precondition counter $\mathcal{S}(Z_i).i_c$ (Line 15 in Algorithm 4) or $\mathcal{S}(Z_i).i_c$ (Line 17 in Algorithm 4) depending on the validation job type ϕ. If the successor node Z_i does not already exist in the state $\mathcal{S}$, the `Master` creates the state object $\mathcal{S}(Z_i)$ using the default values for all its attributes (Line 12f in Algorithm 4). After the counters of all successor nodes have been updated, the `Master` generates new candidates for all non-pruned successor nodes Z_i using Algorithm 6 (`generateCandidates`(X, ϕ)) if possible (Line 18f in Algorithm 4).

Algorithm 6: `generateCandidates`$(X,\ \phi \in \{\text{CONST}, \text{COMP}\})$

Data: States $\mathcal{S}$, job queue $\mathbf{Q}$

```
1  Z = {Z_i | Z_i = X ∪ S_i ∧ S = R \ X}
2  forall Z_i ∈ Z, where S(Z_i).p = FALSE do
3      if φ = CONST ∧ S(Z_i).i_c = |Z_i| then
4          S(Z_i).C_c = ⋂_{A∈Z_i} S(Z_i \ {A}).C_c /* Rule 2b                    */
5          add (Z_i, CONST) to Q
6      if S(Z_i).i_c = |Z_i| ∧ S(Z_i).i_o = |Z_i| then
7          if |Z_i| = 2 then
8              A, B = Z_i(0), Z_i(1)
9              c_o = {(A ↑, B ↑), (A ↑, B ↓)} /* Rule 3a                      */
10         else
11             c_o = ⋃_{A∈Z_i} S(Z_i \ {A}).C_o
               /* Check Rule 3b                                                */
12             forall (Ē, F̄) ∈ S(Z_i).C_o, where (Ē, F̄) = (A ↑, B ↑) or (A ↑, B ↓) do
13                 Y = Z_i \ {A, B}
14                 if ∃C ∈ Y : (Ē, F̄) ∉ S(Z_i \ {C}).C_o then
15                     remove (Ē, F̄) from c_o
           /* Check Rule 3c and Rule 3d                                        */
16         forall (Ē, F̄) ∈ S(Z_i).C_o, where (Ē, F̄) = (A ↑, B ↑) or (A ↑, B ↓) do
17             if A ∉ S(Z_i \ {B}).C_c ∨ B ∉ S(Z_i \ {A}).C_c then
18                 remove (Ē, F̄) from c_o
19         S(Z_i).C_o = c_o
20         add (Z_i, COMP) to Q
```

Algorithm 6 shows the candidate generation function. The `Master` component uses the precondition counters $\mathcal{S}(Z_i).i_c$ and $\mathcal{S}(Z_i).i_o$ to determine whether it can generate new constant bOD candidates (Line 3) or new order compatible bOD candidates (Line 6) for each successor Z_i of the previously checked node X. DISTOD generates constant bOD candidates only if the validation job that triggered the candidate generation validated constant bOD candidates (indicated by $\phi = \text{CONST}$ in Line 3) to prevent generating the constant bOD candidates for a node twice. If $\mathcal{S}(X).i_c = |X|$, the constant bOD candidates of all predecessor nodes of X have been validated and we can generate the minimal constant bOD candidates for X by intersecting the candidate sets $\mathcal{S}(W_i).C_c$ of the predecessor nodes $W_i | W_i = X \setminus \{\mathsf{X}_i\}$ (Line 4). To let a `Worker` validate the new candidates, we add a new constant bOD validation job for node X to $\mathbf{Q}$ (Line 5). The preconditions for the generation of order compatible bOD candidates are checked for both values of ϕ (Line 6), because they

depend on the results of both the constant and order compatible bOD validations of the predecessor nodes. The nodes X, where $|X| = 2$, are the first nodes in the candidate lattice with order compatible bOD candidates. We initialize $\mathcal{S}(X).C_o$ for those nodes with the only two non-trivial candidates $(\mathsf{A} \uparrow, \mathsf{B} \uparrow)$ and $(\mathsf{A} \uparrow, \mathsf{B} \downarrow)$, where $\mathsf{A}, \mathsf{B} \in X$ and $\mathsf{A} \neq \mathsf{B}$ (Line 8f). For those candidates, we do not have to check Rule 3b because there are no order compatible bOD candidates in preceding nodes that could be valid. But Rule 3c and Rule 3d are checked in Line 16ff. For all other nodes X, where $|X| > 2$, the new order compatible bOD candidates are the union of $\mathcal{S}(X \setminus \{\mathsf{A}\}).C_o$ for all $\mathsf{A} \in X$ (Line 11). Those candidates are first checked to obey the minimality Rule 3b in Line 12ff and then to obey Rule 3c and Rule 3d in Line 16ff. At the end of the order compatible bOD candidate generation procedure, the `Master` component adds a new order compatible bOD validation job (X, COMP) to $\mathbf{Q}$ (Line 20).

6 Candidate validation

DISTOD validates bOD candidates using partitions similar to FASTOD-BID [25]. 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 handling and management because the candidate validations are distributed across different nodes in the cluster.

In Section 6.1, we introduce sorted and stripped partitions. Section 6.2 explains how we generate sorted and stripped partitions from the input dataset. With the knowledge about these two partition variations, we describe how we use them to validate constant and order compatible bOD candidates in Section 6.3.

6.1 Sorted and stripped partitions

We already formally introduced partitions Π_X in Section 3.2 Definition 5. They are sets of equivalence classes w. r. t. a context X. DISTOD does not directly use these full partitions for the candidate validation checks, because they need 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 capture the order of the tuples and, thus, enable the validation of order compatible bODs; we describe them in Section 6.2.1. Stripped partitions remove implicit information and, in this way, reduce the memory footprint needed to store partitions; we introduce them in Section 6.1.2.

6.1.1 Sorted partitions

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

Definition 13 *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].*

We have to sort only the equivalence classes themselves, because all tuples within an equivalence class correspond to the same value when projected to X. If we, for example, take Table 1 and the partition $\Pi_{\{\texttt{OCode}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_2\}, \{t_6\}, \{t_8\}\}$ as an example, the sorted partition for $X = \{\texttt{OCode}\}$ is $\widehat{\Pi}_{\{\texttt{OCode}\}} = [\{t_6\}, \{t_0, t_4\}, \{t_2\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_8\}]$.

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

(cf. Section 6.3.2). Sets of tuple identifiers are unsuitable for this operation because they require a scan over the elements of the list. Hence, we propose a reversed mapping of the sorted partitions to represent sorted partitions in DISTOD. We call this new data structure *inverted sorted partition* Γ_X and it is a mapping from the tuple identifiers to the positions of their equivalence classes in the sorted partition. It allows us to lookup the position of a tuple identifier in the sorted partition in constant time.

Recap the sorted partition for the attribute `OCode`: $\widehat{\Pi}_{\{\texttt{OCode}\}} = [\{t_6\},\ \{t_0,\ t_4\},\ \{t_2\},\ \{t_1,\ t_3,\ t_5,\ t_7,\ t_9\},\ \{t_8\}]$. We store this sorted partition as inverted sorted partition $\Gamma_{\{\texttt{OCode}\}} = \{t_0 \to 1, t_1 \to 4, t_2 \to 3, t_3 \to 4, t_4 \to 1, t_5 \to 4, t_6 \to 0, t_7 \to 4, t_8 \to 5, t_9 \to 4\}$. Inverted sorted partitions are directly computed from the input dataset. We explain this step in-depth in Section 6.2.1.

6.1.2 Stripped partitions

Similar to FASTOD-BID, we do not use sorted partitions for all attribute sets, because this would take up a lot of memory, would increase the runtime, and is not necessary for candidate validation checks. As we will see in Section 6.3, we need only the sorted partitions for each $\{\mathsf{A}\} \in \mathrm{R}$ (level l_1 of the candidate lattice). For attribute sets, where $|X| > 1$ (levels l_i where $i > 1$), we replace sorted partitions by stripped partitions [5, 10, 12, 21, 25, 26] (also known as *position list indexes* [3, 13, 19, 18]).

Definition 14 *Stripped partitions are partitions, where singleton equivalence classes (* $|\mathcal{E}(t_X)| = 1$ *) are removed [10]. We denote them with* Π^*_X.

Coming back to our example partition $\Pi_{\{\texttt{OCode}\}} = \{\{t_0, t_4\}, \{t_1, t_3, t_5, t_7, t_9\}, \{t_2\}, \{t_6\}, \{t_8\}\}$, we can transform it into a stripped partition by removing all singleton equivalence classes from it: $\Pi^*_{\{\texttt{OCode}\}} = \{\{t_0,\ t_4\},\ \{t_1,\ t_3,\ t_5,\ t_7,\ t_9\}\}$. Stripped partitions contain the same information as full partitions and the refinement relations are the same. As partitions become more and more refined for larger attribute sets, the number of singleton equivalence classes increases. By omitting these classes, we can save a lot of memory and speed up the scans over the partitions. Szlichta et al. proof that for the proposed bOD validation algorithms (Section 6.3) using *stripped* partitions over full partitions is sufficient, i. e. guarantees correctness [25]. Stripped partitions are used in the validation checks for constant and order compatible bODs. They are directly generated from the input dataset or from previously computed partitions using the partition product. The partition product operation is explained in Section 6.2.2.

6.2 Partition generation

Partitions resemble the input dataset and are thus generated directly or indirectly from the dataset. Sorted partitions capture the order of tuples. They need to be computed form

the input dataset directly. We explain how DISTOD generates sorted partitions from the individual columns of the input dataset in Section 6.2.1. To efficiently generate stripped partitions, DISTOD uses two different approaches: one for attribute sets of level l_1 and another one for attribute sets of deeper levels. Stripped partitions for the single attribute sets in level l_1 are generated from the sorted partitions by simply removing all singleton equivalence classes. All other stripped partitions are generated with the partition product operation from two (possibly already refined) stripped partitions that belong to direct attribute subsets of the current lattice node. Section 6.2.2 discusses both approaches for generating stripped partitions.

6.2.1 Generating sorted partitions

DISTOD's validation checks require only inverted sorted partitions for the singleton attribute sets, where $|X| = 1$ (cf. Section 6.3.2). 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. First, we infer the datatype of the attribute values. We distinguish between different standard and custom date types, such as ISO-8601 or the US date format `MM/dd/yyyy`, numbers, strings, and a `NULL` type with decreasing specificity. We then create the equivalence classes by mapping the tuples to their attribute value. Subsequently, we sort the equivalence classes according to the ordering imposed by the datatype of the attribute and discard the attribute values. The result is a sorted partition $\widehat{\Pi}_{\{\mathsf{A}\}}$ for each singleton attribute set $\mathsf{A} \in \mathrm{R}$. To compute the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each singleton attribute set $\mathsf{A} \in \mathrm{R}$, we iterate over all equivalence classes of the sorted partition $\widehat{\Pi}_{\{\mathsf{A}\}}$ and assign each tuple identifier in the equivalence class $\mathcal{E}_i$ the index number of the equivalence class $\mathcal{E}_i$. 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 the concrete values. This saves memory and – because the computations effectively deal with integers only – makes the operations on partitions fast and simple.

6.2.2 Generating stripped partitions

As explained in the previous section, we generate inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ directly from the dataset. To create the stripped partitions $\Pi^*_{\{\mathsf{A}\}}$ for all $\mathsf{A} \in \mathrm{R}$, we first reverse the mapping of $\Gamma_{\{\mathsf{A}\}}$ to get the original sorted partition $\widehat{\Pi}_{\{\mathsf{A}\}}$ back. We then remove all singleton equivalence classes from $\widehat{\Pi}_{\{\mathsf{A}\}}$ and store the result in a set-form as $\Pi^*_{\{\mathsf{A}\}}$. We have to reconstruct the original sorted partitions $\widehat{\Pi}_X$ ($|X| = 1$) because we store and replicate only the inverse sorted partitions to the nodes (cf. Section 7). This reduces our memory consumption and the required network bandwidth.

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 was originally defined for full partitions ($\Pi_{X \cup Y} = \Pi_X \cdot \Pi_Y$) by Huhtala et al. and is "the least

refined partition $\Pi_{X \cup Y}$ that refines both Π_X and Π_Y" [10]. However, we deal only with stripped partitions. This means that we have to compute the stripped partitions for the following levels from stripped partitions instead of full partitions.

We use the original partition product to define our own stripped partition product as: $\Pi^*_{X \cup Y} = \Pi^*_X \bullet \Pi^*_Y = strip(\Pi^*_X \cdot \Pi^*_Y)$. We can reuse the original partition product on stripped partitions, because singleton equivalence classes are self-preserving. If a singleton equivalence class is present in one of the source partitions, it will be present in the result partition as well. Consequently, removing the singleton equivalence class from the source partition and computing the partition product will lead to the singleton equivalence class missing in the result partition. This is negligible as long as we want to have a stripped partition as result, because we would have to remove the singleton equivalence class from the result partition anyway. However, the result is not necessarily free of singleton equivalence classes. The equivalence class refinement during the partition product may produce some additional singleton equivalence classes. This is why we need a second operation after computing the partition product of the two stripped partitions called *strip*(). This step removes all remaining singleton equivalence classes from the result partition to make it obey the stripped partition definition.

We can use this stripped partition product to efficiently compute all stripped partitions for the attribute sets X, where $|X| \geq 2$. 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 $\Pi^*_{\{\mathsf{B}\}}$: $\Pi^*_{\{\mathsf{A},\mathsf{B}\}} = \Pi^*_{\{\mathsf{A}\}} \bullet \Pi^*_{\{\mathsf{B}\}}$. 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.

6.3 Validation algorithm

As discussed in Section 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 $\mathsf{A} \neq \mathsf{B}$. In this section, we introduce an efficient validation algorithm for these candidates that uses the previously introduced sorted and stripped partitions. 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 (Section 6.3.1) 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 (Section 6.3.2).

6.3.1 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]. A partition Π refines another partition Π' if the equivalence classes in Π are all subsets of any of the equivalence classes in Π'. Huhtala et al. show in [10] that the partition refinement check on stripped partitions can efficiently be computed using a simple error measure $e(Y)$. We define $e(Y)$ to be the minimum number of tuples that have to be removed from the stripped partition Π^*_Y so that the partition would consist of singleton equivalence classes only – meaning it would be empty: $e(Y) = \|\Pi^*_Y\| - |\Pi^*_Y|$, 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.

We can use the error measure $e(Y)$ to check if one stripped partition refines another stripped partition and, therefore, if a constant bOD candidate of the from $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A} \uparrow$ is valid or not. If the refinement check $e(X \setminus \{\mathsf{A}\}) = e(X)$ is true, we cannot find any *split*s (Definition 7) in the dataset r and the constant bOD $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A} \uparrow$ is valid. If $e(X \setminus \{\mathsf{A}\}) \neq e(X)$, the stripped partition $\Pi^*_{X\setminus\{\mathsf{A}\}}$ does not refine $\Pi^*_{\{\mathsf{A}\}}$, we found a *split*, and the constant bOD $X \setminus \{\mathsf{A}\} : [\,] \mapsto \mathsf{A} \uparrow$ is not valid in r. Computing the error measure would require a scan over the stripped partition. Instead of scanning the stripped partition for each constant bOD candidate check separately, each stripped partition Π^*_X stores its number of equivalence classes $|\Pi^*_X|$ and the number of all elements $\|\Pi^*_X\|$. We calculate $|\Pi^*_X|$ and $\|\Pi^*_X\|$ during the generation of the respective stripped partition, because we have to scan the partition product result during the *strip*() operation anyway. This reduces the candidate check itself to three operations – two subtractions to compute the errors ($e(X \setminus \{\mathsf{A}\}) = \|\Pi^*_{X\setminus\{\mathsf{A}\}}\| - |\Pi^*_{X\setminus\{\mathsf{A}\}}|$ and $e(X) = \|\Pi^*_X\| - |\Pi^*_X|$) and the comparison of the two error values ($e(X \setminus \{\mathsf{A}\}) \stackrel{?}{=} e(X)$).

6.3.2 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 validated using partitions as well. We slightly change the validation algorithm from [25] to improve its efficiency. This allows us to verify whether there is no *swap* over the attributes A and B via two scans over the tuples of the stripped context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$. Szlichta et al.'s validation algorithm requires one scan over the sorted partition $\widehat{\Pi}_{\{\mathsf{A}\}}$ and one scan over the stripped context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$. Sorted partitions always contain all tuples of the input dataset. Stripped partitions, however, get smaller for larger attribute sets due to partition refinement. The number of singleton equivalence classes in Π_X grows with the number of attributes in X and the stripped partition Π^*_X omits those classes. Figure 6 plots the average size of stripped partitions over the levels. It clearly shows that the size of stripped partitions drops rapidly for larger attribute sets (l_i, where $|X| = i$). This means that our validation algorithm has to iterate over less values compared to the one from [25]. In the worst case, the stripped context partition also includes all tuples of the input dataset and our algorithm has to iterate over the same number of tuples.

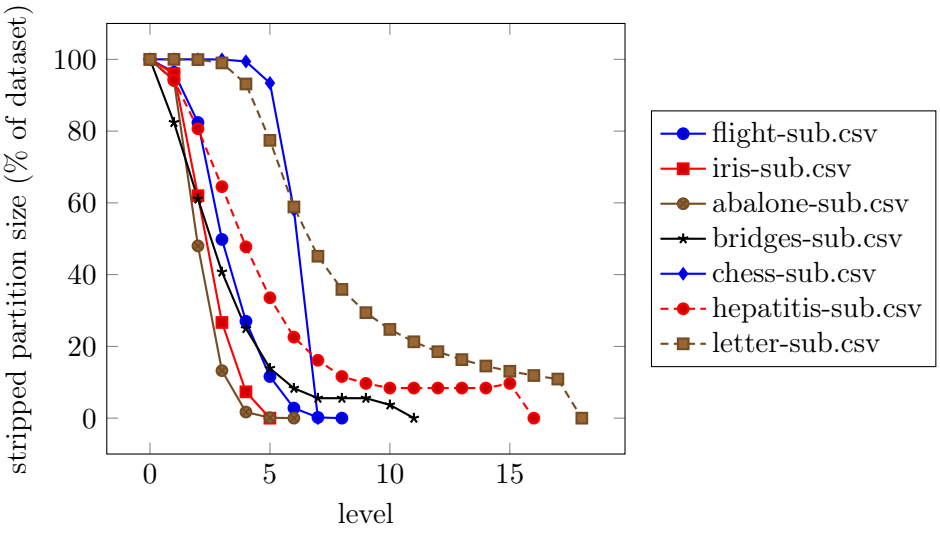


Figure 6: Average relative size of stripped partitions ($\frac{\|\Pi^*_X\|}{|r|}$) per level l_i ($|X| = i$) for different datasets. We report the relative size of the stripped partitions to the original input dataset in percent.

Algorithm 7: Validate order compatible bOD

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

Output: *swap* and *rSwap*

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

Algorithm 7 shows the steps used to validate an order compatible bOD candidate. We first sort 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 we compare this order to the order imposed by the second attribute B (Line 3-16). We check for candidates of the from $X \setminus \{\mathsf{A},\ \mathsf{B}\} : \mathsf{A} \uparrow\sim \mathsf{B} \uparrow$ and $X \setminus \{\mathsf{A},\ \mathsf{B}\} : \mathsf{A} \uparrow\sim \mathsf{B} \downarrow$ at the same time. If we find 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 we find no reverse swap in the data, the order compatible bOD $X \setminus \{\mathsf{A},\ \mathsf{B}\} : \mathsf{A} \uparrow\sim \mathsf{B} \downarrow$ is true (Line 17f). If we can find neither a swap nor a reverse swap, both order compatible bOD forms are valid.

Algorithm 8: `sortEquivClasses`$(\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}, \Gamma_{\{\mathsf{A}\}})$

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

```
1  γ = ∅
2  forall E ∈ Π*_X do
3  |  E = []
4  |  forall t ∈ E do
5  |  |  pos_t = Γ_{B}(t)
6  |  |  if M(pos_t) = null then M(pos_t) = {}
7  |  |  add t to M(pos_t)
8  |  forall E_new ∈ M do /* traversal in key order                    */
9  |  |  add E_new to E
10 |  add E to γ
11 |  clear M
12 return γ
```

In Line 1 (Algorithm 7), we use Algorithm 8 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 8 first iterates over all equivalence classes $\mathcal{E}$ of the context partition $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$ (Line 2). For each $\mathcal{E}$, we create a new temporary list to store the sorted equivalence classes (Line 3). We then iterate over all tuples of the equivalence class $\mathcal{E}$ and lookup their positions when sorting by attribute A using the inverted sorted partition $\Gamma_{\{\mathsf{A}\}}$ (Line 4f). We store the tuple identifiers in the sorted map $\mathcal{M}$ with their position pos_t as the key. $\mathcal{M}$ stores key value pairs and allows us to traverse the values in key order. We use this property to add the tuple sets in sorted order to their new sorted equivalence class E in Line 8f. Afterwards, we store the sorted equivalence class in the output set γ (Line 10) and clear $\mathcal{M}$ to process the next equivalence class $\mathcal{E}$ of $\Pi^*_{X\setminus\{\mathsf{A},\mathsf{B}\}}$ (Line 11). As an example consider the stripped context partition $\Pi^*_{\{\texttt{OCode}\}} = \{\{t_0,\ t_4\},\ \{t_1,\ t_3,\ t_5,\ t_7,\ t_9\}\}$ and the inverted sorted partition $\Gamma_{\{\texttt{ArrDelGrp}\}} = \{t_0 \to 5,\ t_1 \to 2,\ t_2 \to 3,\ t_3 \to 4,\ t_4 \to 0,\ t_5 \to 2,\ t_6 \to 1,\ t_7 \to 6,\ t_8 \to 3,\ t_9 \to 2\}$. Sorting the equivalence classes in $\Pi^*_{\{\texttt{OCode}\}}$ by `ArrDelGrp` using Algorithm 8 results in $\gamma = \{[\{t_4\}, \{t_0\}], [\{t_1, t_5, t_9\}, \{t_3\}, \{t_7\}]\}$.

After the sort operation, each equivalence class of the context partition is a sorted list of tuple sets. The order is defined by $\Gamma_{\{\mathsf{A}\}}$. In Algorithm 7, we store the sorted equivalence classes in γ_{A} (Line 1). In the second step, we iterate over the sorted equivalence classes of the context partition and look for *swaps* ($s \prec_{\mathsf{A}\uparrow} t$ and $t \prec_{\mathsf{B}\uparrow} s$) or *reverse swaps* ($s \prec_{\mathsf{A}\uparrow} t$

and $t \prec_{\mathsf{B}\downarrow} s$) using the inverted sorted partition $\Gamma_{\{\mathsf{B}\}}$ (Line 3-16). For each iteration, we look at two consecutive tuple sets F and G in each equivalence class $\mathbf{E}$ of γ_{A} (Line 5f). For both F and G, we iterate over all their tuples and lookup the tuples' positions in $\Gamma_{\{\mathsf{B}\}}$ to find their maximum and minimum position (Line 7-14). If the highest position of the tuples in F is larger than the lowest position of the tuples in G, we found a swap (Line 15). If the highest position of the tuples in G is larger than the lowest position of the tuples in F, we found a reverse swap (Line 16). If both *swap* and $rSwap$ are TRUE, we can terminate the loops in Line 3f early, because a single swap *and* a single reverse swap are enough to invalidate both order equivalent bOD candidates. An order compatible bOD $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A} \uparrow\sim \mathsf{B} \uparrow$ is valid only if there is no swap in the dataset (Line 17) and an order compatible bOD $X \setminus \{\mathsf{A}, \mathsf{B}\} : \mathsf{A} \uparrow\sim \mathsf{B} \downarrow$ is valid only if there is no reverse swap in the dataset (Line 19).

7 Data 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 that take care of tracking the waiting and pending validation jobs. The `Master` actor 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 (cf. the description of elastic bOD discovery in Section 8). 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.

In Section 7.1, we explain how DISTOD efficiently manages the storage of partitions and how it coordinates their usage between the numerous distributed `Worker` actors. Each node in the DISTOD cluster generates its own stripped partitions using an initial set of partitions, which resemble the input dataset. In Section 7.2, we describe how DISTOD replicates these initial partitions to all nodes in the cluster using a side-channel. Section 7.3 then describes how DISTOD uses the product of refined partitions (cf. Section 6.2.2) to efficiently generate only the required stripped partitions from the initial set of partitions while reusing intermediate partitions that have been cached. Throughout the discovery process, DISTOD carefully manages its memory consumption to process datasets effectively, even with limited memory. In Section 7.4, we describe how DISTOD can trade a higher processing time for a reduced memory consumption by selectively dropping stripped partitions from the cache.

7.1 Partition handling

DISTOD distributes the bOD candidate validations as jobs to the `Worker` actors across the nodes in the cluster in a first come first served manner. This helps to facilitate elasticity, which is described in Section 8, 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, this also means that we do not know which node will receive which validation candidates a priori. Different validation candidates require different partitions. DISTOD, therefore, fully replicates the initial set of inverted sorted partitions $\Gamma_{\{A\}}$ for each $A \in R$ to all nodes in the cluster. The implementation of this side-channel is described in

the next section (Section 7.2). The partitions for the bOD validation checks on a node are computed directly on the node where the checks are performed on. On receiving the initial set of partitions, each node generates the stripped partitions $\Pi^*_{\{\mathsf{A}\}}$ for all $\mathsf{A} \in \mathrm{R}$ (l_1 *stripped partitions*) from the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$. All the other stripped partitions Π^*_X for $|X| \geq 2$ are computed on demand.

Each DISTOD node manages its own partitions and no node must hold all partitions, because only the partitions relevant to the locally performed checks are generated. We do not employ a partition exchange protocol, because individual partition generations are relatively fast compared to the overall runtime of the algorithm and exchanging the partitions over the network would introduce a significant latency and partition tracking overhead (which node has which partitions). Partition management and partition generation together account for only about 3.35 % of the total CPU time[3]. The latency would be hard to hide, because the partitions are the requirement for the expensive candidate validations and we cannot predict which partitions are required next. The partition exchange protocol would not be worth the effort. The local partition management distributes the creation and storage of partitions over the nodes in the cluster and slightly improves the memory pressure on a single node. Admittedly, this also leads to some identical partitions being generated on several nodes independently. As previously mentioned, this is negligible because partition generation is cheap.

The partition management and partition generation is implemented in the partition management plane. Each node in the cluster has its own `PartitionMgr` actor, which stores the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$, 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 $|X| \geq 2$. The initial partitions consisting of the stripped partition for the empty candidate set $\Pi^*_{\{\}}$ and the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ are created by the `DataReader` actor on the leader node and replicated to all other nodes via the side-channel. When a `PartitionMgr` receives the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$, it immediately creates and stores the l_1 stripped partitions $\Pi^*_{\{\mathsf{A}\}}$ for all $\mathsf{A} \in \mathrm{R}$. The local `Worker`s can thus start requesting partitions from their local `PartitionMgr` as soon as all initial partitions have been replicated. The interaction between the `Worker`s, the `PartitionGen` actors and the `PartitionMgr` is depicted in Figure 7. If a `Worker` requests a partition that is already available, the `PartitionMgr` directly serves the stripped partition to the requesting `Worker` actor. If a `Worker` requests a partition that is not available (at startup, these are all stripped partitions Π^*_X, where $|X| \geq 2$), the `PartitionMgr` sends a partition generation task (depicted as $\mathbf{J}_X$ in Figure 7) to one of the `PartitionGen` actors. The partition generation task is either a chain of partition generation jobs calculated by Algorithm 9 on Page 51 (Section 7.3.1) or a task to directly compute the stripped partition from the l_1 stripped partitions $\Pi^*_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ using the direct partition product (Section 7.3.2). The `PartitionGen` actors perform the partition generation and return the partition to the `PartitionMgr`, which inserts the newly generated partition into the cache and forwards it to the requesting `Worker`s. Section 7.3

[3] Measured on scaling the number of workers from one to 20 for the flight dataset with 1 000 rows and 30 columns and taking the average CPU time spent in partition management actors and overall CPU time.

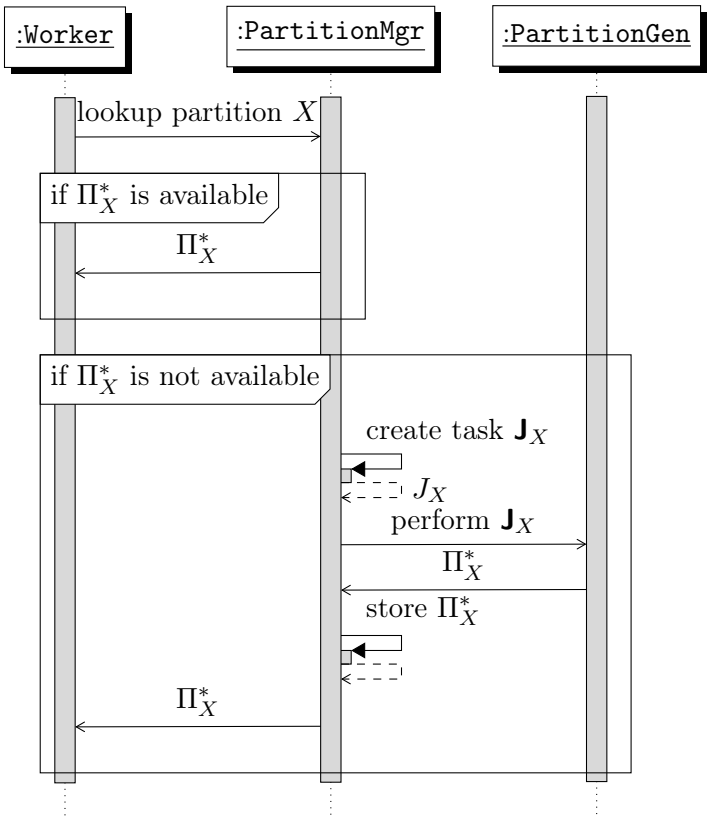


Figure 7: Message exchange between the `Worker` actor, the `PartitionMgr` actor and the `PartitionGen` actor to generate stripped partitions on-demand. Open arrows indicate asynchronous messages; we do not slant these arrows to preserve space.

describes DISTOD's techniques to efficiently generate stripped partitions while reusing the existing partitions of the `PartitionMgr`.

We store 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. If two nodes in our set lattice share a common predecessor node, they also require the same partition for the partition refinement check of their constant bOD candidates if no candidates have been pruned. Consider the nodes $X_1 = \{\mathsf{A},\ \mathsf{B},\ \mathsf{C}\}$ and $X_2 = \{\mathsf{A},\ \mathsf{B},\ \mathsf{D}\}$. Their common predecessor in the set lattice is the node $X_3 = \{\mathsf{A},\ \mathsf{B}\}$. Node X_1's constant bOD candidates include the candidate $\{\mathsf{A},\ \mathsf{B}\} : [\,] \mapsto \mathsf{C} \uparrow$. To check the candidate's validity, we need the stripped partitions $\Pi^*_{\{\mathsf{A},\mathsf{B},\mathsf{C}\}}$ and $\Pi^*_{\{\mathsf{A},\mathsf{B}\}}$. Node X_2 includes i. a. the constant bOD candidate $\{\mathsf{A},\ \mathsf{B}\} : [\,] \mapsto \mathsf{D} \uparrow$. This candidate's validity can be checked using the partitions $\Pi^*_{\{\mathsf{A},\mathsf{B},\mathsf{D}\}}$ and $\Pi^*_{\{\mathsf{A},\mathsf{B}\}}$. Consequently, the candidate validation checks for both the node X_1 and X_2 require the partition $\Pi^*_{\{\mathsf{A},\mathsf{B}\}}$. Each node's candidate validation checks are grouped into two independent tasks. One task performs the constant bOD validation and the other task the order compatible bOD validation. This means that the constant bOD validation checks for the two nodes could be dispatched to two different `Worker` actors. In this case the detour of the program flow over the `PartitionMgr` synchronizes the access on the shared partition $\Pi^*_{\{\mathsf{A},\mathsf{B}\}}$ and ensures that it is generated only once. If it was already generated, it is stored in the cache and can immediately be used by the `Worker`s. The same applies to the case when the tasks are dispatched to the same `Worker` actor. The first task will trigger the generation of the shared stripped partition $\Pi^*_{\{\mathsf{A},\mathsf{B}\}}$. It is then stored in the `PartitionMgr`. The `Worker` must merely retrieve it from the `PartitionMgr` to perform the second task. All partitions in the cache of the `PartitionMgr` are immutable and, thus, `Worker`s 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.

7.2 Partition exchange

DISTOD validates bOD candidates using sorted partitions and stripped partitions. The inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ are directly computed from the input dataset on the leader node and the stripped partitions $\Pi^*_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ (l_1 stripped partitions) are generated from their respective inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$. We explain this procedure in Section 6.2.

All stripped partitions Π^*_X with $|X| \geq 2$ can be computed from the l_1 stripped partitions using the recursive partition generation algorithm (see Section 7.3.1) or the direct partition product (see Section 7.3.2). There is one exception that we have not considered yet: The stripped partition for the empty candidate set $\Pi^*_{\{\}}$. It is the root of the partition lattice and always contains all tuple identifiers of the dataset in a single equivalence class. We can generate $\Pi^*_{\{\}}$ as soon as we know the size of the input dataset. We call the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ and the stripped partition $\Pi^*_{\{\}}$ *initial partitions*, because we compute them directly from the dataset.

In DISTOD, only the leader node reads the input dataset and creates the initial partitions. The follower nodes connect to the leader node and replicate the initial partitions once. There is an alternative approach, where every node reads the dataset on its own. In this case, every node could generate the initial partitions, but the user would have to copy the dataset to all nodes in the cluster by other means. This would not improve the overall runtime and it would decrease the usability of DISTOD. Thus, we do not consider this approach and integrate the replication of the initial partitions in DISTOD.

Depending on the characteristics of the input dataset, the initial partitions can get quite large. For example, the serialized initial partitions for the `letter-sub.csv` dataset have a cumulative size of 1 932 KiB (6 634 KiB on disk, original dataset without substitution has 834 KiB on disk) and the serialized initial partitions for the `ncvoter-sub.csv` dataset have a cumulative size of about 131 MiB compared to about 319 MiB of the dataset on disk. If we send these amounts of data using the default message channel, we may prevent 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. Another disadvantage of sending large messages over the default message channel is that if something goes wrong during the transmission of the message, we have to resend the entire message. To circumvent these disadvantages, we employ a streamed side-channel between the follower nodes and the leader node. It handles the streaming of the chunked initial partitions over a separate back-pressured message channel.

The side-channel is implemented by two types of actors: the `PMEndpoint` actors and the `PartitionRepl` actors. The `PartitionRepl` actors are the consumers and they reside on the follower nodes; One `PartitionRepl` actor per follower node. The `PartitionRepl` actors are created during the startup of the follower nodes. They connect to the primary `PartitionMgr` on the leader node via a `PMEndpoint` actor. Each `PartitionRepl` actor has a corresponding `PMEndpoint` actor on the leader node to handle the node-specific state of the side-channel. The `PMEndpoint` actor is the producer of the initial partitions and resides on the leader node. It has access to the initial partitions of the primary `PartitionMgr` actor on the leader node. The initial partitions are generated and sent to the `PartitionMgr` by the `DataReader` actors during the startup of the leader node. The primary `PartitionMgr` actor does not process incoming requests for partitions until all initial partitions have been received. When the primary `PartitionMgr` receives a request from a `PartitionRepl`, it spawns the corresponding `PMEndpoint` and gives it access to the initial partitions. The `PMEndpoint` continues the communication with the `PartitionRepl` actor and prepares the data messages before sending them out. The side-channel consists of a control channel implemented using the default actor messaging and a data channel implemented using Akka Streams[4].

The side-channel utilizes its own communication protocol. During startup of a follower node, it creates a temporary `PartitionRepl` actor. This actor's responsibility is the replication of the initial partitions to its local `PartitionMgr`. Therefore, it opens a side-channel by sending a message to the primary `PartitionMgr` on the leader node. When the primary `PartitionMgr` receives this message, it creates a new session-bound `PMEndpoint` and

[4] `https://doc.akka.io/docs/akka/current/stream/index.html`

enables it to access the initial partitions. The `PMEndpoint` then prepares the data channel by setting up the local part of a data stream pipeline. It completes the setup by replying to its corresponding `PartitionRepl` actor with a reference to the data stream. The `PartitionRepl` accepts the reference to the data stream and connects it to its local part of the pipeline. The data stream pipeline handles chunking, compression and serialization of the large data messages. The producer part on the leader node is responsible of serializing a data message, e. g. an initial partition, splitting it in smaller chunks, and terminating the chunks with a special termination marker. Each chunk is then send as a separate event through the stream. The consumer part on the follower nodes is responsible for aggregating the chunks until a termination marker arrives and deserializing the aggregated bytes back to a data message. When both the control channel between the two actors and the data channel have been connected, the side-channel is ready to replicate the initial partitions. The `PartitionRepl` first requests metadata about the input dataset from the `PMEndpoint`. After that, it requests one of the initial partitions after the other. The `PMEndpoint` looks up the requested partition in its state and sends it over to the `PartitionRepl`. The `PartitionRepl` receives the partition and forwards it to its local `PartitionMgr`. When the `PartitionRepl` received all initial partitions, it informs the `PMEndpoint` actor that all initial partitions have been replicated. The `PMEndpoint` then closes the data channel, removes all resources, and stops itself. The `PartitionRepl` actor terminates together with the data channel.

This stream-based side-channel allows us to send arbitrary large data messages between the nodes in the DISTOD cluster without impacting system messaging or the algorithm communication. We employ this side-channel at the beginning of the algorithm to fully replicate the initial partitions to all nodes in the cluster. If a follower node joins the DISTOD cluster late, it also uses the side-channel to retrieve all initial partitions from the leader node.

7.3 Managing the generation of stripped partitions

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 needs 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 can be amplified by the regular partition cleanup of the `PartitionMgr` or by a partition eviction in the case of memory shortage (cf. Section 7.4). We solve this by recursively computing partitions making use of already available intermediate partitions. Section 7.3.1 explains this recursive generation of stripped partitions.

The recursive partition generation method results in a chain of stripped partitions that get generated in order to reach a target partition. If this chain gets very long, the intermediate partitions take up valuable memory space and it takes a lot of time to generate all the intermediate partitions until we can compute the target partition. Therefore, we switch to a second partition generation method, called direct partition product, if the chain of partition

generation jobs exceed a certain threshold. The direct partition product is explained in Section 7.3.2.

7.3.1 Recursive generation of stripped partitions

For each request for a partition that is not already stored in 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 partition should be stored in the partition cache, the `PartitionGen` actor sends the newly generated partition to its local `PartitionMgr`, which in turn can forward the partition to the requesting `Worker`.

Algorithm 9: Recursive partition generation job calculation

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

```
1  Function calcJobChainRecursive(Y = X)
2      if Y in J then return
3      W = [W_i | W_i = X \ {X_i}]
4      sort W
5      partition predecessor attribute sets W_i ∈ W into
         W_hit  where W_i ∈ P and
         W_miss where W_i ∉ P
6      if |W_hit| = 0 then                          /* no predecessor part. in P */
7          calcJobChainRecursive (W_miss[0])
8          calcJobChainRecursive (W_miss[1])
9          Add Y → (W_miss[0], W_miss[1], |Y| >= |X| − d) to J
10     else if |W_hit| = 1 then                     /* one predecessor in P */
11         calcJobChainRecursive (W_miss[0])
12         Π*_y = P(W_hit[0])
13         Add Y → (Π_y, W_miss[0], |Y| >= |X| − d) to J
14     else                                         /* at least two predecessors in P */
15         Π*_y0 = P(W_hit[0])
16         Π*_y1 = P(W_hit[1])
17         Add Y → (Π_y0, Π_y1, |Y| >= |X| − d) to J
```

Algorithm 9 shows the recursive function to determine the chain of partition generation jobs. It has the target attribute set X and all stored partitions $\mathcal{P}$ as input and returns a list of partition generation jobs $\mathbf{J}$. A partition generation job (e. g. in Line 9) consists of the attribute set for the target partition to generate (Y), the two input partitions for the

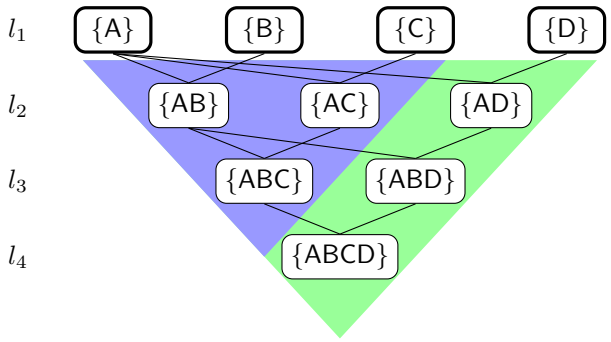


Figure 8: Partition lattice with marked partition generation job chains. Available partitions are depicted in bold. The blue job chain consisting of {A, B}, {A, C}, and {A, B, C} is calculated first. The green job chain consisting of {A, D}, {A, B, D}, and {A, B, C, D} is calculated second.

partition product and a flag that tells the `PartitionGen` actor whether this partition should be stored or not. We store all partitions up to a depth of three. This means that the target partition, up to two of its predecessors, and up to four of their predecessors are stored in the partition cache for each partition that is generated. This is a compromise between not storing all predecessors and storing all intermediate partitions. If we store all intermediate partitions, the following partition generations can be computed much faster, but we use more memory. If we store only the target partition, we use a lot less memory, but have to compute the intermediate partitions for each of the following requests as well. This trade-off is important in the case of partition evictions. Our observations showed that under normal circumstances, DISTOD generates only job chains ranging down at most three levels. But in the case of a partition eviction all intermediate partitions have been removed and the chains get longer. In this case, we are already short of memory and we do not want to put more pressure on the system by storing all intermediate results. We, therefore, limit the number of partitions of the recursive partition generation procedure that get stored. The two input partitions of a partition generation job are either the stripped partitions themselves (Π^*_X) or the identifiers of the partitions (X). If only the identifiers are specified, the `PartitionGen` actor looks up the stripped partitions in its temporary partition state and uses those 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 9).

Algorithm 9 generates a minimal and deterministic number of jobs. The algorithm chooses the partition product factors from the candidate set's predecessors in a left-oriented way. For each recursion step the predecessors of the current candidate set are sorted lexicographically (Algorithm 9, Line 4) and the first one or two available predecessors from the beginning are chosen for the partition product. Any following partition generation runs also use this left-orientation and automatically re-use the previously generated partitions. This cuts down the number of generation steps and thus reduces the time spent generating partitions.

We explain this procedure by an example, where the partition cache of the `PartitionMgr` is empty and it holds only the initial partitions and the l_1 stripped partitions. Remember, the initial partitions are the stripped partition for the empty candidate set $\Pi^*_{\{\}}$ and the inverted sorted partitions $\Gamma_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$. The l_1 stripped partitions are the stripped partitions $\Pi^*_{\{\mathsf{A}\}}$ for all $\mathsf{A} \in \mathrm{R}$. For this example, we use only the candidate set X to identify a stripped partition Π^*_X for brevity. The already existing partitions are reflected in Figure 8 by the bold nodes in level l_1. We omit {} and the inverted sorted partitions, because they are not relevant for our example. The `PartitionMgr` receives two requests for partitions one after the other. First, a `Worker` actor requests the partition $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. It is not in the partition cache, because our cache is empty. The `PartitionMgr`, therefore, uses the recursive Algorithm 9 to generate the job chain for the target candidate set $X = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. It calculates the predecessors of the candidate set and sorts it lexicographically (Algorithm 9, Line 4), which results in the list $\mathbf{W} = [\{\mathsf{A}, \mathsf{B}\}, \{\mathsf{A}, \mathsf{C}\}, \{\mathsf{B}, \mathsf{C}\}]$. The list is split into available and unavailable partitions. Since the partition cache is empty, none of the above partitions is available and the two left-most predecessors are selected as input for the partition product (see Algorithm 9, Line 6ff.). We must generate the partitions $\{\mathsf{A}, \mathsf{B}\}$ and $\{\mathsf{A}, \mathsf{C}\}$ first, before we can generate the target partition $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. Therefore, we recursively call `calcJobChainRecursive()` for each of the two selected predecessors. The predecessors for those two candidate sets are singleton candidate set partitions and they are available. We select the two left-most predecessors for each of the two candidate sets and lookup their partitions, so we can use them directly for the partition product (see Algorithm 9, Line 14ff.): $\Pi^*_{\{\mathsf{A},\mathsf{B}\}} = \Pi^*_{\{\mathsf{A}\}} \bullet \Pi^*_{\{\mathsf{B}\}}$ and $\Pi^*_{\{\mathsf{A},\mathsf{C}\}} = \Pi^*_{\{\mathsf{A}\}} \bullet \Pi^*_{\{\mathsf{C}\}}$. In this case it would not matter which predecessor we chose, because $\{\mathsf{A}, \mathsf{B}\}$ and $\{\mathsf{A}, \mathsf{C}\}$ have only two predecessors. This stops the recursion and results in the job chain for the target partition $X = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$: $\mathbf{J} = [\{\mathsf{A}, \mathsf{B}\}, \{\mathsf{A}, \mathsf{C}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}]$. Figure 8 shows this job chain marked in a blue triangle.

When the next request for partition $\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ hits the `PartitionMgr`, all previous partitions have been added to the partition cache and we can generate the next partition chain making use of those previously generated partitions. The sorted list of predecessors for the target candidate set $X = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ is $\mathbf{W} = [\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{D}\}, \{\mathsf{A}, \mathsf{C}, \mathsf{D}\}, \{\mathsf{B}, \mathsf{C}, \mathsf{D}\}]$. Only the partition $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ is available in the partition cache. We can use it as one factor for the partition product, but must also select a second predecessor to recurse further (see Algorithm 9, Line 10ff.). We take the first unavailable partition $\{\mathsf{A}, \mathsf{B}, \mathsf{D}\}$ and calculate its job chain in the recursive call to `calcJobChainRecursive()`. Its predecessors are $[\{\mathsf{A}, \mathsf{B}\}, \{\mathsf{A}, \mathsf{D}\}, \{\mathsf{B}, \mathsf{D}\}]$. Again, the first predecessor $\{\mathsf{A}, \mathsf{B}\}$ is already available and we can use it as the first factor (we are again in the branch starting at Line 10). We have to generate only the second predecessor $\{\mathsf{A}, \mathsf{D}\}$ from the l_1 stripped partitions. This eventually results in the job chain $\mathbf{J} = [\{\mathsf{A}, \mathsf{D}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{D}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}]$ for the target candidate set $X = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$, which is depicted in green in Figure 8. This shows that we have to generate only one of the two partitions for each level in the lattice, when we start with larger candidates and an empty partition cache, which would be the case for nodes that join the DISTOD cluster late.

7.3.2 Direct partition product

When DISTOD's heap memory usage exceeds a certain threshold, all temporary stripped partitions are removed from the partition cache of the `PartitionMgr` by the partition eviction mechanism (see Section 7.4). Partitions that are generated after such an event have to be recursively generated from the singleton attribute set partitions in level l_1. If there are no intermediate partitions, the number of partition generation jobs needed to generate a partition grows exponential with the level in which the partition is. Especially for wide datasets and late partition evictions, partitions for deep levels have to be computed from scratch. If the chain of partition generation jobs becomes this long, it takes a lot of time to generate all the intermediate partitions until we can compute the requested partition. In addition, we also have to store those intermediate partitions temporarily, so that we can use them for the following partition products. This puts additional pressure on the limited memory. For these reason, we switch to a different partition generation method if the chain of recursive partition generation jobs becomes too long. The threshold can be specified in the configuration options of DISTOD. We can compute 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. We call this a direct partition product, because we do not compute intermediate partitions.

Please consider the following example, where we assume a threshold of three jobs in the partition generation chain that would trigger a switch to the direct partition product. A partition eviction happened and the partition cache was cleared. In the next step, a `Worker` requests partition $\Pi^*_{\{\mathsf{A},\mathsf{B},\mathsf{C}\}}$. Instead of computing the partition product recursively via $\Pi^*_{\{\mathsf{A},\mathsf{B},\mathsf{C}\}} = \left(\Pi^*_{\{\mathsf{A}\}} \bullet \Pi^*_{\{B\}}\right) \bullet \left(\Pi^*_{\{\mathsf{A}\}} \bullet \Pi^*_{\{\mathsf{C}\}}\right)$, we use the direct partition product $\Pi^*_{\{\mathsf{A},\mathsf{B},\mathsf{C}\}} = \Pi^*_{\mathsf{A}} \bullet \Pi^*_{\mathsf{B}} \bullet \Pi^*_{\mathsf{C}}$ to compute the requested stripped partition from the persistent l_1 stripped partitions. The recursive partition product would take three partition products (three jobs) and would create two intermediate partitions, namely $\Pi^*_{\{\mathsf{A},\mathsf{B}\}}$ and $\Pi^*_{\{\mathsf{A},\mathsf{C}\}}$. It cannot benefit from existing partitions, because they were cleared. The direct partition product takes only two steps and generates the single target partition. If in a second step, a `Worker` requests the partition $\Pi^*_{\{\mathsf{A},\mathsf{B},\mathsf{D}\}}$, the recursive partition generation would be used again, because the partition chain would consist of only two jobs: $[\Pi^*_{\{\mathsf{A},\mathsf{D}\}}, \Pi^*_{\{\mathsf{A},\mathsf{B},\mathsf{D}\}}]$.

Generally, the recursive generation of stripped partitions is much faster, because it can use existing stripped partitions in the computation. Partitions for large attribute sets tend to be smaller than the original single attribute set partitions from level l_1. The partition refinement leads to a growing number of singleton equivalence classes in the partitions, which we do not store. Consequently, the partition product is faster to compute, because the two input stripped partitions contain less tuples that we have to consider in the product. Only in the rare case, where we have to generate a lot of intermediate stripped partitions in order to be able to compute the target partition, the direct partition product is faster. We expose a setting key for easy customization of the threshold at which DISTOD switches from the recursive partition generation to the direct partition product.

7.4 Dealing with limited memory

For situations with limited memory, DISTOD has to efficiently manage its memory consumption. If DISTOD pushes its memory limit, the Java garbage collector takes up most of the processing time slowing down the actual algorithm. If it exceeds the memory limit, the algorithm fails and terminates. These scenarios cannot be prevented completely, e. g. for large datasets that already take up most of the memory, it is unavoidable that DISTOD's memory usage is close to the limit. However, we can improve the memory consumption of DISTOD allowing it to process larger datasets with fewer memory. For most datasets and especially for long ones the partitions of the `PartitionMgr` take up most of the main memory. DISTOD already improves the memory consumption of single nodes in the cluster by distributing the load between them. Each node generates only those stripped partitions that are required locally. No node must hold all partitions. However, this improves the memory consumption just slightly. Stripped partitions are a good candidate to improve the memory consumption of DISTOD further. Stripped partitions get smaller during refinement. This means that stripped partitions in deeper levels use less memory than stripped partitions in the first couple of levels, because they contain more singleton equivalence classes that are omitted (cf. Figure 6 on Page 42). However, deeper levels contain considerably more stripped partitions. We can use these facts to our advantage, because we do not have to store all stripped partitions forever. Only the sorted partitions in level l_1 are required over and over again. The intermediate stripped partitions in the deeper levels can be deleted when they are not needed anymore. We can even go a step further and do not store the intermediate partitions altogether. This is possible because every stripped partition in level l_2 or below can be generated from the sorted partitions $\widehat{\Pi}_{\{\mathsf{A}\}}$ for each $\mathsf{A} \in \mathrm{R}$ in level l_1. These two ideas lead to the two methods how DISTOD manages partitions to deal with limited memory: Periodic partition cleanup and partition eviction.

7.4.1 Periodic partition cleanup

Stripped partitions are used only for a limited amount of time before they get obsolete by their successors. We, therefore, do not need to store all partitions forever. Only the inverted sorted partitions and the stripped partitions in level l_1 have to be preserved to allow the validation of order compatible bOD candidates and to allow the regeneration of all partitions. The intermediate stripped partitions in the deeper levels can be deleted when they are not needed anymore. The point in time, when the partitions are not needed anymore, is hard to predict, because only the `Master` actor on the leader node knows which candidates have already been processed and which candidates are next in the work queue. This information is necessary to determine if a partition is still required, because a single partition is involved in a varying number of bOD candidate validations. A partition in level l_i would be involved in $|\mathrm{R}| - i$ constant bOD candidate validations and in $|\mathrm{R}| - (i+1)$ order compatible bOD candidate validations if we disable DISTOD's pruning and minimality checking. Due to the facts that we cannot compute whether a partition will not be used in the future and that we can recompute any partition from the initial set of partitions, the `PartitionMgr` just keeps track of partition accesses and periodically removes unused

partitions from its partition cache. We do not use a standard least recently used (LRU) cache implementation, because we cannot determine the expected size of partitions to limit the number of slots in the cache. The size of the stripped partitions depends highly on the dataset and the number of distinct values in the attributes. If the number of distinct values for an attribute is equal to the number of tuples in the dataset, a stripped partition does not store any tuple identifiers, because it would consist of singleton equivalence sets only. Contrary to that, if an attribute of the dataset is constant and contains only a single value across all tuples, the stripped partition would include all tuple identifiers and consist of a single large equivalence class. This shows, that the stripped partitions in the partition cache can have varying size and we cannot predict their size easily. Instead of a LRU cache, we use a custom implementation that does not limit the maximum number of partitions and periodically removes unused ones. The partition cleanup interval can be configured manually. If set to 40 seconds, the `PartitionMgr` receives a tick message every 40 seconds. During the interval, the `PartitionMgr` tracks which partitions of the cache have been accessed. When the tick message arrives at the `PartitionMgr`, it removes all partitions from the partition cache that have not been accessed in the time frame and resets its internal access statistics. The interval time determines the minimum lifetime of a partition, because the initial generation of the partition is triggered by accessing it and does count towards the access statistic. A shorter interval corresponds to a more aggressive removal, because partitions that are not accessed a second time within the interval are already removed from the partition cache during the next tick and may need to be regenerated when they are required later. Even if DISTOD removes a stripped partition prematurely, it still can regenerate it from the l_1 stripped partitions using the direct partition product explained in Section 7.3.2 or from other cached partitions using the recursive partition generation algorithm explained in Section 7.3.1. The periodic partition cleanup can be turned off completely, but this increases memory usage. The removal of stripped partitions cannot break the algorithm, but a small interval may slow DISTOD down.

7.4.2 Partition eviction

The partition cache of the `PartitionMgr` is not limited in size and the periodic partition cleanup removes only unused partitions from the cache. This means that it is still possible for DISTOD to use up too much memory. For this scenario, there is a second mechanism, called partition eviction, that enables DISTOD to perform more work before running into out-of-memory errors. It is triggered only when it is inevitable, because it removes all intermediate partitions from the partition cache and, thus, costs a lot of performance.

Since we can regenerate all intermediate partitions, we do not need to store them entirely and can recompute them for every request. This implies a lot of duplicated work and decreases the algorithm's performance. We want to use this approach only if it is inevitable. The partition eviction mechanism, hence, monitors the heap usage of our algorithm and compares it to a certain threshold. If DISTOD's heap usage exceeds the threshold, all partitions in the partition cache are removed and the requested partitions have to be regenerated from scratch. The newly generated partitions are stored in the partition cache

again to utilize them for further partition generations and the validations. If we hit the heap threshold once more, the partition eviction is triggered again and we start over.

The heap size monitoring is implemented by the `SystemMonitor` actor on each node in the DISTOD cluster. It polls the heap status of the JVM and compares it to the specified heap threshold value. If the used heap surpasses the threshold, the `SystemMonitor` sends a message to the `PartitionMgr` that instructs it to clear its partition cache. The mechanism is triggered independently on the nodes in the cluster. This means that a single node reaching its heap memory limit does not impact the performance or memory usage of the other nodes. However, it is likely that all nodes run into the heap memory limit at roughly the same time if they use the same heap sizes, because DISTOD tries to distribute the candidate validations equally across the nodes in the cluster.

8 Elastic bOD discovery

DISTOD is designed to run on a cluster of compute nodes and during the execution of DISTOD, all its resources can not be used for other tasks. The overall runtime of a bOD discovery algorithm is hard to predict. This also applies to DISTOD. Its runtime not only depends on the size of the dataset but also on the structure and contents, because they determine the effectiveness of our pruning strategies and the number of valid bODs. Businesses are not always able to guarantee a separate execution environment for DISTOD and the capabilities of a selected execution environment might change, for example, due to the submission of additional other jobs to the cluster. Therefore, DISTOD supports elasticity in the number of compute nodes. We allow the user to add more follower nodes to a running DISTOD cluster to increase its capabilities and to speed up the processing. How DISTOD integrates new follower nodes in the cluster is described in Section 8.1. If the compute resources of the cluster are needed for other tasks, the user can remove follower nodes from a running DISTOD cluster without impacting correctness or completeness of the found bODs. However, DISTOD does not tolerate hard node failures or disconnects. Follower nodes must be shut down gracefully, otherwise the current tasks of the disconnected follower node will be lost. DISTOD's leader node can not be scaled. There is always only one leader node in the cluster and if the leader node is actively stopped or fails, the whole DISTOD cluster terminates. Section 8.2 describes the implementation of the graceful termination procedure of the follower nodes in order to remove them from a running DISTOD cluster.

8.1 Adding follower nodes

To add a new follower node to a running DISTOD cluster, we copy the DISTOD binary and the configuration files to a machine in the same network as all the other nodes and start it. The configuration of the new node must be compatible to the configuration of all the other nodes, especially regarding the various pruning options. Since 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 (cf. 4.3). To be able to connect to the already running DISTOD cluster, the newly started DISTOD follower node just requires the address of any of the nodes in the cluster. It joins the cluster by connecting to the specified seed node and exchanging the addresses of all other nodes with it. Each follower node is started with a temporary `PartitionRpl` actor that connects to the primary `PartitionMgr` on the leader node and replicates the initial partitions to its local `PartitionMgr`. The exchange of partitions between DISTOD nodes is epxlained in Section 7.2. Meanwhile, the `WorkerMgr` resolves the reference to the `Master` actor and starts the `Worker`s and the local `RCProxy` connects to the `ResultCollector` actor on the leader node. The `Worker`s immediately start requesting candidate validation tasks from the `Master` and process these tasks as soon as the initial partitions have been successfully replicated. The validation results are sent back to the leader node using the connection between the local `RCProxy` and the `ResultCollector` on the leader node. DISTOD treats all connected nodes in the same way.

8.2 Removing follower nodes

Any follower node can be removed from the DISTOD cluster. The leader node can not be removed from the cluster because it holds the central candidate state and it orchestrates the discovery process. If the leader node is instructed to terminate, the whole DISTOD cluster shuts down, as explained in Section 4.3. 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 DISTOD process on a follower node can be stopped by sending a SIGTERM signal to it (e. g. with `Ctrl-C`). DISTOD catches this signal and triggers the coordinated shutdown protocol only on the selected node. The termination procedure is executed by the local `Executioner` actor. It supervises the termination and makes sure that all steps are executed in the correct order. Gracefully terminating a DISTOD follower node involves the following four steps: (i) In the first step, the `Executioner` instructs all local `Worker` actors to terminate. They immediately quit requesting new work. But before the `Worker` actors can be completely stopped, they first need to finish their current subtasks of the validation jobs. If a validation job is completed, the results are sent back to the leader node. Otherwise, the job is aborted at the `Master` actor. The `Master` re-enqueues the job in the work queue in order to be able to dispatch it to another `Worker` on another node. If all jobs have been completed or aborted, the `Worker` actors terminate. (ii) In the second step, the buffered results of the local `RCProxy` are flushed to the `ResultCollector` on the leader node. This ensures that we do not lose any valid bODs when terminating the follower node. When the `ResultCollector` confirms the receipt, the local `RCProxy` terminates as well. (iii) When all `Worker`s and the `RCProxy` are stopped, the third step is triggered. In this step, the local follower node leaves the DISTOD cluster by informing all nodes about its leaving status. The cluster leader has to acknowledge the status change before the local follower node is considered down by all other nodes in the cluster. (iv) In the last step, the local follower node stops all remaining actors and the JVM process terminates cleanly.

9 Semantic pruning strategies

BODs properly subsume FDs [28]. This means that the search space and the number of valid bODs may be larger for bODs than for FDs, even with minimality pruning. The result size easily exceeds the size of the original dataset (i. a. for the hepatitis-sub.csv dataset or the horse-sub.csv dataset in Table 3 in Section 10.2). Interpreting a large number of minimal valid bODs is hard. In addition, a bOD discovery algorithm may not be able to mine a dataset with a large result size because storing the results in memory already exceeds its memory limit. Even if the results fit into memory, existing discovery algorithms still require a long time to explore the whole search space and to output all valid bODs.

If we shrink the search space of our discovery algorithm, we can greatly reduce its result size, memory consumption and runtime. We, therefore, adapt two optional semantic pruning strategies for our distributed bOD discovery algorithm. The first strategy is called *interestingness pruning* [25] and is described in Section 9.1. In [18], Papenbrock and Naumann introduce a pruning strategy that dynamically limits the size of resulting FDs of their FD discovery algorithm HYFD if these would exhaust the memory capacity. We take inspiration of this idea and adopt a static pruning strategy *size limitation* that limits the size of discovered bODs. It is described in Section 9.2. Both strategies reduce the result size at the cost of losing completeness of the algorithm. Thus, they are implemented as optional features that can be turned on and off.

9.1 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 is pruned. Only interesting bOD candidates are considered and validated for the dataset. The interestingness score is based on the measure proposed by Szlichta et al. in [25] and indicates coverage and succinctness of a bOD candidate. The authors briefly evaluate interestingness pruning for their algorithm FASTOD-BID with a carefully set interestingness score threshold that restricts the result size to the top 100 bODs and conclude that it provides a substantial performance improvement.

The interestingness score for a bOD is defined over the number of tuple pairs covered by it. We can calculate the score for a constant bOD $X : [\,] \mapsto \overline{\mathsf{A}}$ or an order compatible bOD $X : \overline{\mathsf{A}} \sim \overline{\mathsf{B}}$ by summing up the squares of the sizes of all equivalence classes of the context partition: $\sum_{\mathcal{E}(t_X) \in \Pi_X} |\mathcal{E}(t_X)|^2$ [25].

To facilitate interestingness pruning in DISTOD's distributed setting, we calculate and use the interestingness score directly before validating an OD 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 and, thus, it can be scaled out similarly to the candidate validations. This allows us to parallelize the score calculation making use of the already distributed partitions. If the

context partition is not already available on the node, it is generated on demand re-using the partition generation infrastructure also used for the validation of the candidates. If the score exceeds our interestingness threshold, the bOD candidate is interesting and has to be validated. Since the partitions are cached by the `PartitionMgr`, we can directly use the newly generated partition for the validation checks. If the score does not exceed the threshold, the bOD candidate is not validated, but added to the pruned candidate set C_r (cf. Section 5.3). The `Worker` actors report back the pruned candidate sets to the `Master` component, which integrates the pruning results into the candidate lattice, so that this information can be considered during the generation of the next candidates. If a candidate for a node X is pruned, it is removed from the candidate sets $\mathcal{S}(X).C_c$ or $\mathcal{S}(X).C_o$, so that candidates with context Y, where $Y \supset X$, are also considered pruned. BOD candidates with a larger context naturally have more unique equivalence classes and, therefore, a smaller interestingness score.

Note that interestingness pruning removes valid bOD candidates from the result set. Obviously, this gives up completeness of the bOD discovery algorithm. In DISTOD, interestingness pruning can be turned on or off. We can set the interestingness threshold as a configuration option before starting DISTOD.

9.2 Size limitation

Papenbrock and Naumann use a result size limitation strategy to improve the robustness of their algorithm HYFD [18]. They dynamically determine a threshold for the size of the FDs's left-hand sides based on memory consumption. If there is not enough free memory, the maximum FD size is reduced and existing FDs that are too large are removed from their state while new FDs that are too large are excluded. This improves the robustness of their algorithm w. r. t. the memory consumption because their algorithm can find more smaller FDs by removing larger ones.

We use a similar result size limitation strategy to be able to mine datasets with a large amount of valid bODs. In contrast to their approach, our constraint is not the size of individual bODs, but the sheer number of results. We, therefore, do not determine the size threshold dynamically during runtime, but allow the user to restrict the number of attributes that is involved in a bOD (size of a bOD) statically. This size of a bOD directly corresponds to the depth of the corresponding node in the candidate lattice. For example, bOD candidates of a node in level l_4 involve exactly four attributes. Our breadth-first search strategy on the candidate lattice guarantees that if we found all bODs up to a specific size (up to level l_i), all following bOD candidates will be larger, because they are generated in deeper levels. To implement size limitation in DISTOD, we limit the depth of the candidate lattice. The candidate lattice is managed by the central `Master` actor. If size limitation is turned on, the `Master` actor generates only bOD candidates with a size smaller or equal to a specified size limit set by the user. All levels deeper than the size limit are not generated. Size limitation is an optional feature of DISTOD that, if turned on, limits the size of considered bOD candidates and, thus, gives up completeness of the algorithm.

10 Evaluation

Our proposed algorithm DISTOD is an exact approach to OD discovery. Our sole performance measure is the overall algorithm runtime for a given dataset, because we limit the memory consumption in our experiments and we do not perform any approximation. We compare the runtime of our algorithm DISTOD with the FASTOD-BID algorithm [25] and its distributed variant DIST-FASTOD-BID from [21] in Section 10.2. The source code of DISTOD[5] and DIST-FASTOD-BID[6] is publicly available on Github. For FASTOD-BID, we use an implementation of the algorithm that can be found in the Github repository of the DIST-FASTOD-BID-project as well[7].

We especially evaluate if and with which configuration our distributed algorithm outperforms the efficient single-threaded algorithm FASTOD-BID [25] to ensure that DISTOD is evaluated not in isolation but in comparison with its most capable alternative. There is a metric that measures exactly this. It is called *configuration that outperforms a single thread* (COST) and was introduced by McSherry, Isard, and Murray in 2015 [16]. We report this metric in Section 10.3, which shows DISTOD's scalability in the number of CPU cores on a single node.

Because DISTOD is a distributed algorithm, we not only evaluate the scalability of our approach w. r. t. the number of CPU cores of a single node (Section 10.3), the number of tuples in a dataset (Section 10.5) and the number of attributes in a dataset (Section 10.6), but also w. r. t. the number of nodes in the cluster (Section 10.4). To evaluate the robustness of DISTOD, we measure the runtime of our algorithm with different memory limits. Besides, we perform experiments, where we show the impact of the partition caching on the runtime of DISTOD. Those in-depth experiments can be found in Section 10.7.

10.1 Experimental setup

Hardware We perform our experiments on a bare-metal machine cluster with twelve nodes. The machines are equipped with an Intel Xeon E5-2630 CPU at 2.20 GHz (boost up to 3.10 GHz) with 10 cores and hyper-threading (20 virtual cores). Eight nodes have 64 GB of main memory and four nodes have 32 GB of main memory. All nodes run an Oracle Java 1.8.0 64 bit server JVM on Ubuntu 18.04 LTS. We run each experiment three times and report the best (means fastest) runtime.

Java compressed *ordinary object pointer*s Java performance does not linearly scale with the used heap size. Using a smaller heap can improve memory usage and performance of an application. We observed this behavior in our experiments. For example for the letter-sub.csv dataset, a single DISTOD node with 31 GB of memory was about 20 % faster than

[5] `https://github.com/CodeLionX/distod`
[6] `https://github.com/hemant271990/distributed-dependency-discovery`
[7] `https://github.com/hemant271990/distributed-dependency-discovery/blob/master/distributed-fastod/src/main/java/fastod_impl/ODAlgorithm.java`

the same DISTOD node with 58 GB of memory ($\sim$ 40 minutes with 31 GB compared to $\sim$ 49 minutes with 58 GB). Java uses a JVM-internal performance optimization called compressed *ordinary object pointer*s (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 and the processor cache usage as well as the memory bandwidth usage is improved. This speeds up the overall algorithm runtime significantly. For more details about compressed OOPs, we refer to Oracle's Java documentation. The performance optimization was introduced with Java 7[8]. For this reason, we limit the Java heap size for all experiments to 31 GB or less. This includes our experiments with FASTOD-BID and DIST-FASTOD-BID to make the comparison as fair as possible. We run the DISTOD leader node with 31 GB heap and all follower nodes with 28 GB because some of our nodes have only 32 GB main memory and we need some space left for the process and thread stacks. DISTOD does not require homogeneous follower node configurations, but this allows us to compare the runtimes when scaling the number of nodes. FASTOD-BID runs with 31 GB heap. For DIST-FASTOD-BID, we use a similar configuration, where the Spark driver runs with 31 GB of heap and the eleven worker nodes run with 28 GB.

Data characteristics For our experiments, we use several synthetic and real-world datasets that have previously been used to evaluate FD and OD discovery algorithms. The datasets are available on the HPI website[9] or were published through the UCI Machine Learning Repository[10]. Details about the datasets can be found in Table 3.

The implementation of DIST-FASTOD-BID does not support data types other than integers, but our datasets contain strings, dates, or decimals. We must preprocess all datasets to be able to run DIST-FASTOD-BID on them. We remove the headers of all datasets and substitute all values with their hash value, so that we can map all values to an integer representation. This keeps all FDs intact, but may change bODs. Datasets with the suffix `-sub` in their name are transformed using this method. For DIST-FASTOD-BID, we also convert the existing CSV files to JSON. DISTOD and FASTOD-BID read the original CSV files. Hereafter, we use the names of the CSV files only. If DIST-FASTOD-BID is evaluated, we assume the corresponding JSON files are used implicitly. 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.

10.2 Varying the datasets

We compare the runtime of DISTOD, FASTOD-BID and DIST-FASTOD-BID in their most powerful configuration on various datasets in Table 3. We report the runtime in

[8] `https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#compressedOop`
[9] `http://metanome.de`
[10] `https://archive.ics.uci.edu/ml/index.php`

seconds and list the number of valid constant bODs (reported as #FDs) and the number of valid order compatible bODs (reported as #bODs) separately. FASTOD-BID and DIST-FASTOD-BID report different numbers of bODs than DISTOD. FASTOD-BID and DIST-FASTOD-BID count candidates for which both the bidirectional ($\{X\} : \mathsf{A} \uparrow\sim \mathsf{B} \downarrow$) and the normal check ($\{X\} : \mathsf{A} \uparrow\sim \mathsf{B} \uparrow$) are true only as a single bOD. We count them as two separate bODs. Despite the difference in reporting, all three algorithms write the same results to disk. For the number of results reported in Table 3, we count the actual results that have been written to disk and count bidirectional and normal bODs separately.

We execute the single-threaded algorithm FASTOD-BID on a single node of our compute cluster with 64 GB of main memory. It uses 31 GB heap memory. We increase the heap limit to 58 GB when the lower memory configuration was not enough to process the dataset. For all of those cases but one (dataset letter-sub.csv), increasing the memory limit did not improve the results and FASTOD-BID was still not able to process the dataset. The Spark cluster for DIST-FASTOD-BID is configured over the same twelve machine cluster as we use for the other experiments. The Spark master runs on the same machine as the driver process. This machine has 64 GB of main memory. The Spark driver process is started with a heap size of 31 GB. We configure Spark to put one executor with 20 cores on each of the remaining eleven nodes, which results in 220 executor cores in total. Each executor gets 28 GB of memory. DISTOD's leader node also runs on a node with 64 GB and gets 31 GB main memory for its heap. We use an active leader configuration, where the leader node spawns ten workers and all follower nodes spawn 20 workers each. The follower nodes are placed on the remaining eleven machines and get 28 GB of heap memory. All nodes share the same configuration and we set the partition cleanup interval to 40 seconds (cf. Section 7.4). All semantic pruning strategies are turned off (cf. Section 9).

Table 3 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 after one minute and FASTOD-BID takes almost an hour to complete. A similar observation can be made for the letter-sub.csv dataset with 17 columns and 20k rows. DISTOD can finish the task within five minutes while FASTOD-BID requires more than 5.5 hours. For this experiment, we had to increase the heap size of FASTOD-BID to 58 GB as well because it hit the memory limit with 31 GB. 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 is able to start processing the dataset very early on, even before all follower nodes are able to connect to the leader. This allows DISTOD to process even small datasets very fast without the need to wait for a complete cluster startup and shutdown.

Compared to DIST-FASTOD-BID, DISTOD is at least four times 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 first few small datasets in Table 3, DIST-FASTOD-BID is an order of magnitude slower than both FASTOD-BID and DIS-

Dataset	Columns	Rows	Size (KiB)*	#FDs	#bODs	FASTOD-BID	DIST-FASTOD-BID	DISTOD
iris-sub.csv	5	150	15	4	7	**0.3**	20	0.4
chess-sub.csv	7	28 056	520	1	0	8	25	**3**
abalone-sub.csv	9	4 177	753	137	318	4	26	**2**
bridges-sub.csv	13	108	28	142	1 097	**2**	37	3
adult-sub.csv	15	32 561	61	78	1 140	3 561	322	**67**
letter-sub.csv	17	20 000	834	61	2 202	16 846†	1 279	**251**
ncvoter-sub.csv	19	999 999	326 519	572	4 362	ML§	TL	**36 441**
hepatitis-sub.csv	20	155	6 635	8 250	54 766	145	1 386	**111**
flight-long-sub.csv	21	499 999	71 032	290	2 253	ML§	906	**212**
horse-sub.csv	29	300	168	≥ 113 007	≥ 1 820 293	ML§	TL	ML§
flight-sub.csv	30	1 000	597	5 976	40 740	91	455	**56**
fd-reduced-sub.csv	30	250 000	149 269	89 571	742	2 649	1 304	**203**
plista-narrow.csv	35	1 001		4 467	54 921			872
plista-sub.csv	63	1 001	576	≥ 32 404	≥ 313 296	ML§	TL	ML§

Table 3: Runtimes in seconds of FASTOD-BID, DIST-FASTOD-BID, and DISTOD on different datasets in their most powerful configuration. Experiments that ran out of memory are denoted with ML and experiments that did not finish within 24 hours are denoted with TL.
* Size of the original dataset without the value substitution on disk.
† FASTOD-BID ran with 58 GB heap space instead of 31 GB (lower heap space hit ML).
§ Experiment hit memory limit even with 58 GB heap memory (for the leader node).

TOD. This is due to the synchronized cluster startup and shutdown procedure of the Spark implementation, which causes a significant runtime overhead.

DISTOD is the sole approach that is able to process the ncvoter-sub.csv dataset within our time and memory constraints. FASTOD-BID can not process the dataset because it hits the memory limit, even when we allow it to use 58 GB of heap memory, and DIST-FASTOD-BID does not finish level nine of the candidate lattice until it hits the time limit of 24 hours. DISTOD explores all 15 levels of the candidate lattice in nearly ten hours validating more than 736k bOD candidates.

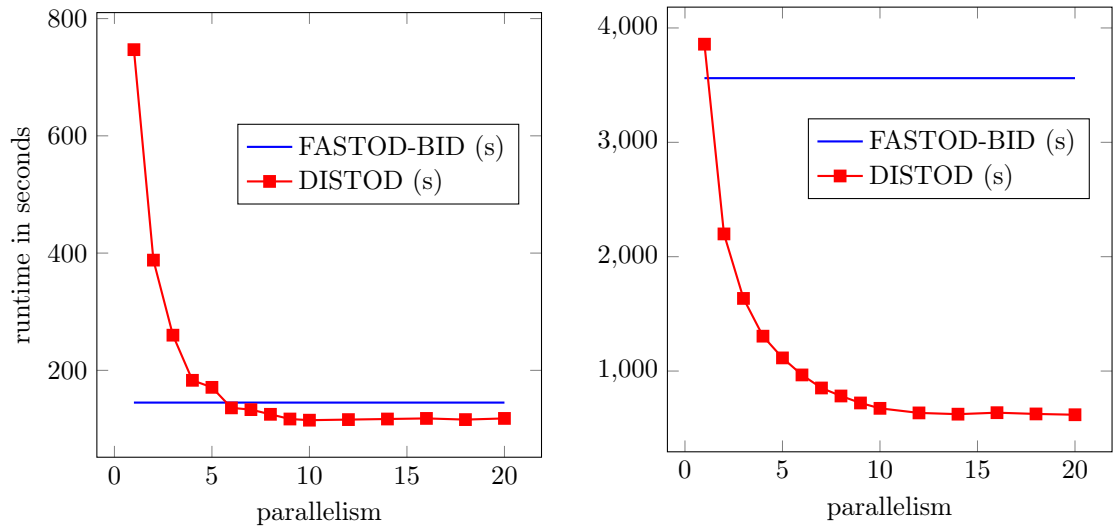
In summary, while FASTOD-BID is well suited for short datasets with varying number of attributes and DIST-FASTOD-BID is good at narrow but long datasets, DISTOD performs equally good on all datasets and can even process the ncvoter-sub.csv dataset, which could not be processed by FASTOD-BID or DIST-FASTOD-BID within our time and memory limits. If they terminate, all three approaches produce the same results.

10.3 Scaling the cores

Evaluating a system's performance only on its scalability is insufficient, because any system can scale arbitrary well with a sufficient lack of care in its implementation [16]. Introducing parallelizable runtime overheads into a system improves its apparent scalability, but its true performance gets worse. On the contrary, removing parallelizable overheads impairs the apparent scalability despite overall improved performance. McSherry, Isard, and Murray discuss this aspect of distributed system evaluation and introduce 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 [16]. The COST of a given system is the hardware configuration required before it outperforms the single-threaded variant.

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 value for the `max-parallelism` configuration option of DISTOD. This option restricts the parallelism of the various parallel components of DISTOD, such as the number of `MasterHelper` actors, the number of `Worker` actors, or the number of `PartitionGen` actors, and effectively limits DISTOD to the specified number of parallel execution threads (CPU cores). 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 (cf. Figure 9).

Figure 9a 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 an 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



(a) DISTOD compared to FASTOD-BID on hepatitis-sub.csv datasets with 20 columns and 155 rows.

(b) DISTOD compared to FASTOD-BID on adult-sub.csv dataset with 15 columns and 32 561 rows.

Figure 9: COST: Scaling the parallelism of a single DISTOD node and comparing its runtime in seconds to the runtime of the efficient single-threaded implementation FASTOD-BID.

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 9b 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.

The plots in Figure 9 both show that DISTOD's runtime does not improve further by increasing its parallelism beyond a value of ten. This is due to hardware restrictions. The test machine uses a CPU with hyper-threading, which increases the number of cores from ten real cores to 20 virtual cores. We cap the value for the `max-parallelism` option of DISTOD to the number of available cores of the machine, which results in a maximum value of 20. DISTOD does not benefit from hyper-threading.

10.4 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. DISTOD's vertical scalability is shown in Figure 9 of Section 10.3. In

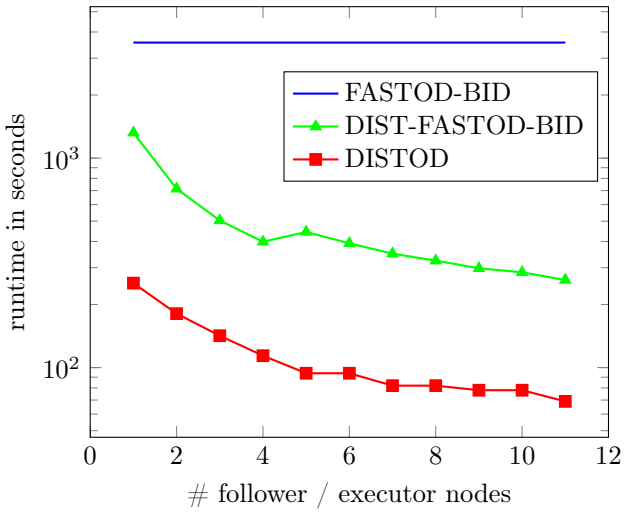
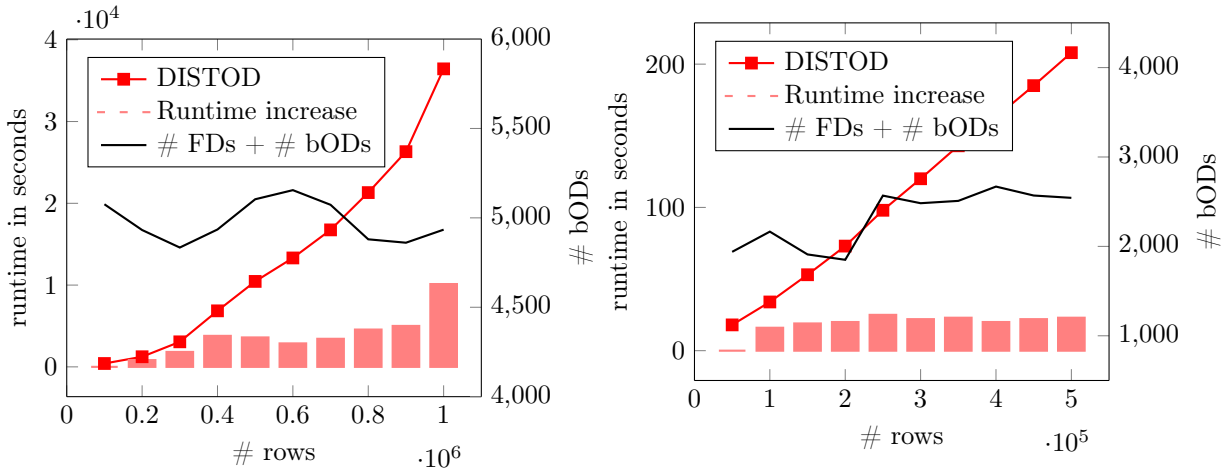


Figure 10: Scaling the number of follower nodes for DISTOD and the number of Spark executor nodes for DIST-FASTOD-BID with the dataset adult-sub.csv. We report the runtime of the three algorithms in seconds on a log axis.

Figure 10, we compare the runtimes of DISTOD, FASTOD-BID and DIST-FASTOD-BID when scaling them horizontally. FASTOD-BID is a single-threaded bOD discovery algorithm and we show its constant runtime as a reference. Note that we report the runtime of the approaches in seconds on a log axis. The experiments were performed on the adult-sub.csv dataset with 15 columns and 32 561 rows. Figure 10 shows that DISTOD is already five times faster than DIST-FASTOD-BID and 14 times faster than FASTOD-BID when a single node is used. When DISTOD is executed in its full configuration with one leader node and eleven follower nodes, it is still more than three times faster than DIST-FASTOD-BID with eleven executors that each use 20 cores. 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. Despite that DISTOD in its smallest configuration is already faster than all DIST-FASTOD-BID configurations it scales out well. The slowdown of non-parallelizable parts hit a faster, more optimized algorithm, such as DISTOD, harder than a slower one, because the non-parallelizable parts have a bigger share of the runtime. Nevertheless, DISTOD is able to scale out in a similar manner to DIST-FASTOD-BID.

10.5 Scaling the rows

We use the two longest datasets to perform our experiments on DISTOD's scalability in the number of tuples |r|. Figure 11a shows DISTOD's runtime in seconds when scaling the number of tuples of the ncvoter-sub.csv dataset with 999 999 rows and 19 columns. We limit the number of tuples to x by stopping the input parsing process after x (+1 if a header is present) rows. Figure 11a shows a nearly linear runtime growth, because the computation time is dominated by the generation of partitions and the validation of bOD candidates. The deviation from the linear growth is due to the different number of



(a) Scaling the number of tuples from 100 000 to 1 000 000 of the ncvoter-sub.csv dataset with 19 attributes.

(b) Scaling the number of tuples from 50 000 to 499 999 of the flight-long.csv dataset with 21 attributes.

Figure 11: Runtime of DISTOD in seconds when scaling the number of tuples. The bar plots show the absolute increase in runtime w. r. t. the previous measurement.

results that is directly correlated with the effectiveness of our pruning strategies. If valid bODs have a higher succinctness, they are detected early on in lower levels and a lot of nodes and candidates of higher levels can be pruned from the search space. This also leads to a smaller number of results and a shorter runtime of the algorithm. This effect can be observed in Figure 11a for the measurements with 400k and 500k rows. If there is a smaller number of valid bODs, pruning is more effective and the algorithm is faster. This case can be seen for the measurements with e. g. 200k or 300k rows.

The higher runtime increase of the experiment with 1m rows can be explained by looking at the results. Due to the additional rows in the dataset for the experiment with 1m rows, some small and general bODs get invalid and more involved bODs become valid. Small and general bODs are discovered very early on in the discovery process because they are located in the first few levels. BOD candidates with n attributes are checked in level l_n. If a general bOD does not hold, a lot of the larger candidates of the next levels can not be pruned from the candidate lattice, which increases the number of candidate checks and thus leads to a longer runtime. We can observe this behavior for the experiment on the ncvoter-sub.csv dataset with 1m rows in contrast to the ncvoter-sub.csv dataset with 900k rows. Figure 12 depicts the number of discovered bODs at different lattice levels for the ncvoter-sub.csv dataset with 900k and 1m rows. The experiment with 1m rows discovers less bODs in lower levels (i. e. l_3, l_5, l_6) but more in higher levels (i. e. l_7, l_8, l_9, l_{10}, l_{11}), which drastically increases the number of necessary candidate validations from 658 650 checks for the dataset with 900k rows to 737 518 checks for the dataset with 1m rows. This means that DISTOD has to perform about 12 % more checks for the experiment on the ncvoter-sub.csv dataset with 1m rows than with 900k rows.

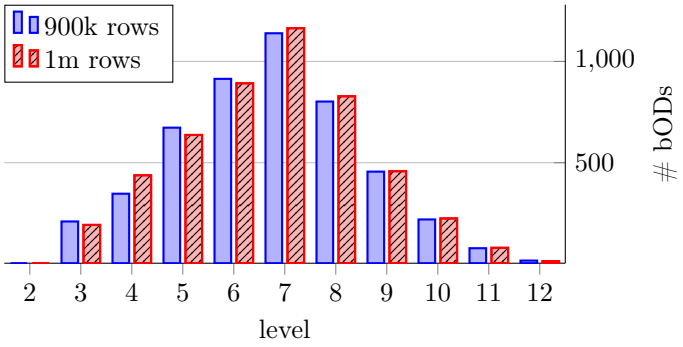


Figure 12: Number of bODs discovered at each level of the set lattice for the experiments with 900k rows and 1m rows of the ncvoter-sub.csv dataset.

We plot the runtime of DISTOD for the flight-long.csv dataset with 499 999 rows and 21 columns in Figure 11b. For this dataset, DISTOD scales nearly linear with the number of rows in the dataset. This is expected, because the complexity of bOD discovery using the set-based form is linear in the number of tuples. Figure 11b also shows that the algorithm runtime increase is influenced by the result size, as we can see for the measurements with 20k and 25k rows.

10.6 Scaling the columns

To evaluate DISTOD's ability to scale with the number of attributes in a dataset, we perform an experiment on our widest dataset plista-sub.csv with 1 001 rows and 63 columns and scale its number of attributes from five to 60 by increments of five. We measure the runtime in seconds and vary the number of attributes by taking random projections of the plista-sub.csv dataset. Figure 13 shows that the runtime of DISTOD grows exponentially with the number of attributes of the dataset. This is expected, because the number of minimal bODs in the set containment lattice is exponential in the worst case, as we can also see on the result size measurements in Figure 13. For the plista-sub.csv dataset with 40 columns or more, the candidate lattice outgrows DISTOD's memory limit and DISTOD is not able to free up memory without guaranteeing complete and minimal results. It terminates early without finding all minimal bODs.

As the result size grows exponentially with the number of attributes in a dataset, DISTOD's runtime and memory consumption grows exponentially as well. To overcome this limitation, we introduced two semantic pruning strategies in Section 9. Interestingness pruning and size limitation 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. The plista-sub.csv dataset with 45 columns can be mined in 64 seconds (60 interesting bODs) when using interestingness pruning. For the whole dataset with 61 columns, DISTOD terminates within 4.5 minutes and finds 98 interesting bODs. Limiting the size of bODs to a maximum of six involved attributes achieves similar results: For the plista-sub.csv dataset with 45 columns, DISTOD

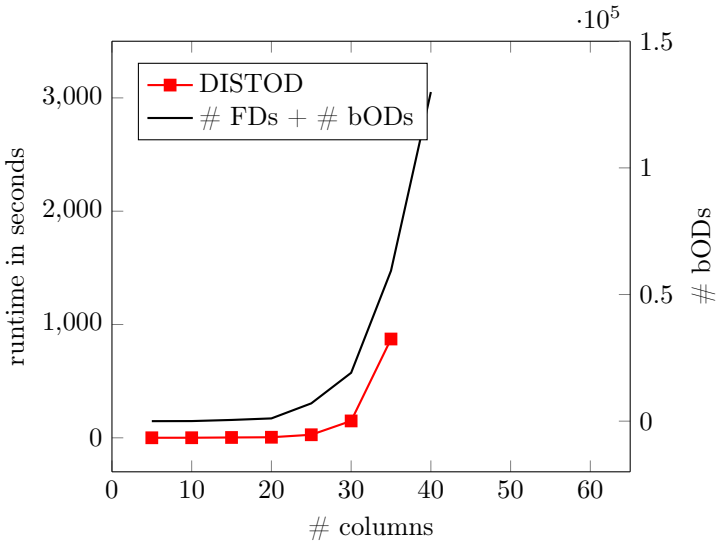


Figure 13: Runtime of DISTOD in seconds when scaling the number of columns from 5 to 60 of the plista-sub.csv dataset with 1 001 rows.

with size limitation turned on finds 532 bODs in 89 seconds. The whole plista-sub.csv dataset with 61 columns can be mined in under five minutes (809 bODs) when we use the size limitation feature.

10.7 In-depth experiments

In this section, we look at some in-depth experiments. Section 10.7.1 contrasts the memory consumption of DISTOD with its two alternatives FASTOD-BID and DIST-FASTOD-BID and Section 10.7.2 shows the impact of DISTOD's partition caching by the `PartitionMgr` actor on the runtime.

10.7.1 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. We 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 8 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.

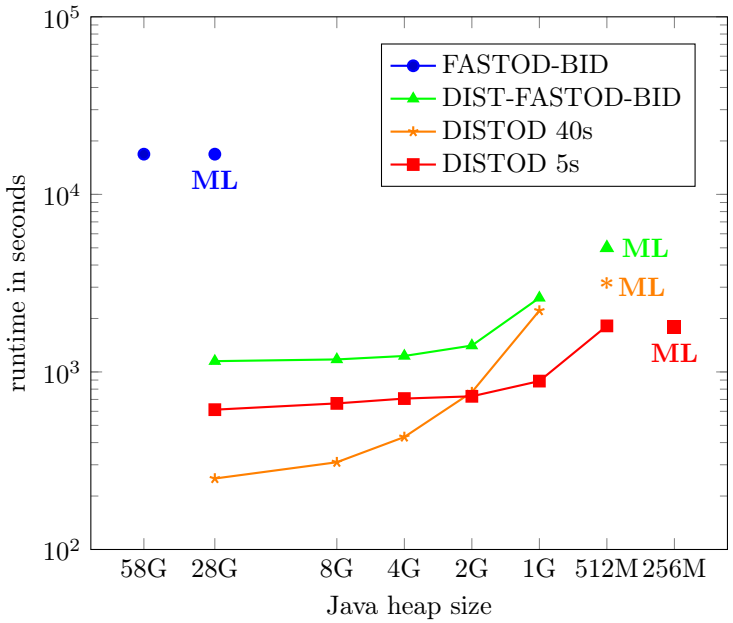


Figure 14: Runtime of DISTOD, DIST-FASTOD-BID, and FASTOD-BID on the letter-sub.csv dataset in seconds when varying the available memory.

However, we set the partition cleanup interval to a more aggressive value of 5 seconds (cf. Section 7.4). We expect that this allows us to free up more memory quicker, but influences DISTOD's runtime negatively. For DIST-FASTOD-BID, the available memory for the Spark driver process as well as all executors is limited to our selected memory value.

Figure 14 shows the runtimes of DISTOD, DIST-FASTOD-BID, and FASTOD-BID on the letter-sub.csv dataset when we reduce the available memory from 8 GB to 256 MB. Figure 14 also shows the runtime of the algorithms for the maximum memory configuration of 28 GB. Since FASTOD-BID already hit the memory limit (denoted with ML) with 28 GB memory, we included its runtime with 58 GB from Table 3 (Section 10.2).

FASTOD-BID uses a lot of memory, because it's level-wise algorithm stores all partitions of the current and 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. Only the partitions from the previous level are freed up when the transition from one level to the next is completed. FASTOD-BID takes more than 4.5 hours to process the letter-sub.csv dataset with 58 GB of memory. Note that FASTOD-BID can only utilize the memory of a single node. On a single node and 28 GB of memory, DISTOD terminates on the letter-sub.csv dataset after 38 minutes; this is 7× faster than FASTOD-BID while using less memory.

DIST-FASTOD-BID can process the letter-sub.csv dataset when the available memory for every executor and the Spark driver process is above or equal to 1 GB. When we limit the memory to 512 MB the Spark executors fail and the Spark job does not terminate. DIST-FASTOD-BID is not able to find bODs in the dataset letter-sub.csv in this case.

Figure 14 shows this experiment marked with ML. The runtime of DIST-FASTOD-BID increases when we reduce the available memory.

DISTOD is able to process the letter-sub.csv dataset with 512 MB or more memory. Even with only 512 MB of available memory and the partition cleanup interval set to 5 seconds, 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. They free up memory by removing all stored stripped partitions from the cache, which allows DISTOD to continue processing with less memory but increased processing time. We can see this in Figure 14 based on the overproportional runtime increase from 888 seconds (1024 MB) to 1817 seconds (512 MB). However, for the experiment with only 256 MB of memory available, the candidate states outgrow the memory limit and freeing up stripped partitions does not help anymore and the leader node of the DISTOD cluster hits the memory limit (ML). When available memory is low enough, the candidate states become the memory limiting factor.

Figure 14 also shows the runtimes of DISTOD when we keep the partition cleanup interval at 40 seconds. For the experiment with a limit of 28 GB of memory, DISTOD with a partition cleanup interval of 5 seconds takes 613 seconds to process the letter-sub.csv dataset; DISTOD with a partition cleanup interval of 40 seconds takes only 251 seconds (cf. Table 3 in Section 10.2). This shows that a small partition cleanup interval negatively impacts DISTOD's runtime when the available memory is adequately sized. It allows DISTOD to efficiently run with lower memory bounds though. When the partition cleanup interval is small, DISTOD removes stripped partitions from the cache more frequently. However, the number of stripped partitions that get removed from the cache too early, because they would be needed again later on, correlates inversely with the size of the partition cleanup interval. DISTOD has to recompute removed partitions, which increases DISTOD's runtime to process the dataset. For the measurements with a high memory limit, such as the measurement with 28 GB or 8 GB memory, using a larger partition cleanup interval improves the performance of DISTOD. However, if DISTOD's available memory is near its minimum memory requirement, a small partition cleanup interval also means that DISTOD can effectively free up memory to prevent partition evictions. If the partition cleanup can not keep up with the memory allocation by the generation of new stripped partitions, the memory usage exceeds the partition eviction threshold and DISTOD will remove all stripped partitions from the cache over and over again. The resulting runtime increase is higher as the runtime increase by the more frequent partition cleanups, because *all required stripped partitions* have to be recomputed from the initial set of stripped partitions every time a partition eviction is triggered. In the case of a 40 seconds partition cleanup interval, the higher memory consumption through the less frequent partition cleanups leads to DISTOD hitting the memory limit sooner than with a 5 seconds partition cleanup interval; already at 512 MB. For environments with limited memory, a small partition cleanup interval allows DISTOD to process a dataset more efficiently and, thus, faster.

Dataset	Caching off	Caching on	Diff
iris-sub.csv	375	414	+ 10 %
chess-sub.csv	2 873	2 393	− 17 %
abalone-sub.csv	1 951	1 396	− 28 %
bridges-sub.csv	2 979	2 871	− 04 %
adult-sub.csv	127 889	58 135	− 55 %
letter-sub.csv	828 082	249 417	− 70 %
hepatitis-sub.csv	108 094	102 466	− 05 %
flight-sub.csv	70 355	70 572	− 01 %
fd-reduced-sub.csv	200 402	101 846	− 49 %
flight-long.csv	390 670	207 679	− 47 %

Table 4: Runtimes of DISTOD in milliseconds when partition caching is turned off or on for some selected datasets. Column *Diff* reports the runtime increase or decrease when partition caching is turned on w. r. t. DISTOD's runtime when partition caching is turned off.

10.7.2 Partition caching

We study the impact of DISTOD's partition caching mechanism (cf. Section 7.1) on the runtime for various datasets. We measure the runtime of DISTOD with partition caching enabled and disabled in milliseconds and report the results in Table 4. The experiments were performed on all twelve nodes of our testing cluster and the partition caching setting was applied to all nodes in the same way.

DISTOD with partition caching enabled is on average 26 % faster then when partition caching is disabled. For the datasets adult-sub.csv and letter-sub.csv, partition caching decreases DISTOD's runtime even by 55 % and 70 % respectively. The higher runtime on the iris-sub.csv dataset with partition caching is due to the overall small runtime on this dataset and the runtime measurement fluctuation because we performed only a single measurement for each run in this experiment. If partition caching is disabled, DISTOD computes all stripped partitions from the initial partitions using the direct partition product (Section 7.3.2) 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 (Section 7.3.1).

11 Conclusion

In this thesis, 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 (nodes) that form a compute cluster. DISTOD discovers set-based bODs and makes use of the smaller set-containment lattice instead of a list-containment lattice for list-based bODs. This allows DISTOD to have an exponential worst-case runtime complexity in the number of attributes and linear complexity in the number of tuples. DISTOD discovers a complete set of minimal bODs w. r. t. the minimality definition by Szlichta et al. [25]. Our algorithm outperforms the single-threaded bOD discovery algorithm FASTOD-BID [25] by orders of magnitude and the distributed DIST-FASTOD-BID [21], which employs Spark to distribute the bOD discovery process, by factors of five to eight; for wider datasets even by an order of magnitude. With DISTOD's elasticity property and the semantic pruning strategies, DISTOD is able to discover bODs in datasets of practically relevant size, e. g. the plista-sub.csv dataset with 61 columns and 1k rows can be mined in under five minutes. For our concluding statements, we summarize our findings and contributions in Section 11.1 and discuss possible future work in Section 11.2.

11.1 Summary

Reactive search strategy: We introduced a novel, reactive search strategy to distribute and parallelize the discovery of bODs. Our search strategy traverses a lattice of all possible sets of attributes breadth-first. We break up the levels of the candidate lattice into individual tasks that can be processed independently. In this task-based approach, we can handle the two canonical forms for set-based bODs independently from each other, which allows for a fine-grained job distribution of the candidate validation jobs. Our search strategy supports elasticity by incorporating new nodes into the discovery process and by re-distributing jobs from leaving nodes to the other nodes of the cluster. We did not propose an overarching fault tolerance strategy yet (node failures simply require algorithm restarts), but valid bODs are progressively spilled to disk so that intermediate results do not get lost in the case of node failures. This provides best possible results in the case of failures, because we discover small bODs before large ones and small dependencies are considered most relevant [18].

Centralized, but parallel candidate generation: Our reactive search strategy is implemented using a centralized candidate generation algorithm. The candidate generation algorithm is executed only on the leader node, because a distributed implementation would introduce a high communication overhead to ensure consistency of the intermediate results and the search state. To increase its performance, the candidate generation component on the leader node is, however, parallelized.

The candidate generation component ensures correctness and completeness of the discovered bODs. BOD candidates are packed into validation jobs and distributed to the other

nodes in the cluster. Starting with an initial set of candidates, the candidate generation algorithm reactively generates new candidates when bOD validation results arrive at the leader.

Improved validation algorithm for order compatible bODs: We improve the efficiency of the order compatible bOD candidate validation algorithm from [25] by using inverted sorted partitions. An inverted sorted partition is a new index data structure that maps the tuple identifiers to their position in the dataset when we sort the dataset by a specific attribute. The time to create the inverted sorted partitions is negligible, because the inverted sorted partitions are computed once and reused for all order compatible bOD candidate validations. Our revised validation algorithm requires two scans over the stripped context partition for each candidate validation. FASTOD-BID's order compatible bOD validation algorithm [25] requires one scan over a sorted partition and one scan over the stripped context partition. Stripped partitions get smaller for larger attribute sets and sorted partitions always include all tuple identifiers of the input dataset. In the worst case, the stripped partition also includes all tuple identifiers and the runtime of our algorithm is the same as the runtime of FASTOD-BID's algorithm. In practice, however, our algorithm has to iterate over less tuple identifiers.

Dynamic generation of stripped partitions: In order to scale out the generation of stripped partitions, we generate stripped partitions on every node locally. Since a node cannot predict which stripped partition is required next, the stripped partitions are generated on demand. If a node receives a validation job, it generates only the stripped partitions required to execute the job. A particular stripped partition may be required for multiple jobs. To prevent redundant work, each node caches its stripped partitions. The partition cache also speeds up the generation of further stripped partitions, because we use a recursive partition generation scheme. Stripped partitions for larger attribute sets are smaller, which reduces the time needed to generate a succeeding stripped partition. If the chain of partition generation jobs for the recursive partition generation scheme gets too large, we switch to a direct partition product that uses the initial partitions to compute the required stripped partition in one step. The threshold to switch between recursive generation and direct partition product can be set manually.

Efficient data management: We efficiently manage the main memory consumption and the data communication costs of our approach. To reduce data communication costs, we replicate the initial partitions to all nodes once and generate all other stripped partitions locally on each node. The initial partitions consist of the stripped partition for the empty candidate set and the inverted sorted partitions for all attributes in the dataset. Based on these initial partitions, each node is able to generate all other stripped partitions. Besides the initial partitions, we send only small messages, such as the validation jobs, their results, or cluster state changes, over the network.

Our algorithm effectively deals with limited memory. The stripped partitions in the partition caches take up most of the main memory. We reduce the overall memory consumption of our algorithm by periodically removing unused stripped partitions from the partition caches. However, every node stores the set of initial partitions, because all other stripped

partitions can be recomputed based on the initial partitions. Every node in the cluster also monitors its main memory usage and if the usage grows too high, i. e., it exceeds a configurable threshold, the node frees up memory by removing all temporary stripped partitions from its local partitions cache (partition eviction). The partition eviction decision is made by each node individually, because their memory usage can differ. These techniques allow DISTOD to process datasets with less memory than its two competitors, FASTOD-BID and DIST-FASTOD-BID.

Semantic pruning strategies: The number of valid bODs for a dataset can grow very large (cf. Table 3 on Page 65). To restrict the result sizes and to overcome the exponential worst-case time complexity in the number of attributes for very large datasets, we adapt two semantic pruning strategies to our distributed setting: Interestingness pruning [25] and size limitation [18]. To facilitate interestingness pruning, we scale out the calculation of the interestingness score alongside the candidate validations and to facilitate size limitation, we prune large bODs from the candidate lattice in our centralized candidate generation component. Both strategies reduce the result size at the cost of losing completeness and, thus, they are implemented as optional features that can be enabled on demand.

11.2 Future work

We found some aspects of DISTOD and bOD discovery in general that did not fit into the scope of this thesis but may inspire future work:

Sophisticated fault tolerance: Distributed algorithms are exposed to a variety of additional failures compared to single-node algorithms. We did not yet consider node or network failures in the implementation of our distributed algorithm DISTOD, because the focus of this research project was on improving the performance of bOD discovery. For DISTOD to be fault-tolerant, it needs to be able to recover from follower node failures and potentially also leader node failures. To make a distributed leader-follower system fault tolerant, several standard protocols and techniques exists, such as work package tracking, state snapshotting, state replication, and redundant actor roles; we leave their straightforward implementation to future work.

State compression: Our evaluation has shown that DISTOD can process datasets with less memory than the existing approaches FASTOD-BID and DIST-FASTOD-BID. Still, the memory limiting factor in DISTOD is its central candidate state data structure on the leader node. It stores the intermediate results and the pruning information of the candidate lattice, which grows exponentially with the number of attributes in the dataset. To reduce DISTOD's memory footprint even further, future work should investigate in strategies that reduce the memory consumption and growth of the candidates states.

Transformation of set-based bODs to list-based bODs: Our algorithm and its two competitors output set-based canonical bODs. Many use cases for bODs, however, operate on list-based bODs. To make use of already existing methods and strategies, a

transformation algorithm is needed that transforms entire sets of discovered minimal set-based bODs into sets of minimal list-based bODs. In this way, bOD discovery benefits from the reduced search space of set-based bODs while providing the bODs in list-based form to existing query [28] or database design [6] optimization techniques. A careful evaluation of the transformation algorithm is necessary to show that discovering set-based bODs and transforming them back to list-based bODs is indeed faster than performing the discovery directly on list-based bODs. With solely set-based bODs, bOD use cases need to query the discovered result sets for specific list-based bODs (cf. Section 3.3) that can easily be checked using the bOD axioms [25].

Hybrid distributed bOD discovery: Jin, Zhu, and Tan have recently proposed a hybrid bOD discovery approach inspired by [4] and [17]. This approach iteratively discovers bODs on a sample of the dataset and then validates them on the complete dataset [12]. We think that bringing this idea into our distributed algorithm DISTOD could enable the discovery of bODs on even larger datasets.

References

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. „Integrating Compression and Execution in Column-Oriented Database Systems“. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2006, pp. 671–682.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. „Profiling Relational Data: A Survey“. In: *The VLDB Journal* 24.4 (2015), pp. 557–581.

[3] Ziawasch Abedjan et al. *Data Profiling*. Morgan & Claypool Publishers, 2018. ISBN: 978-1-68173-447-7.

[4] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. „Efficient Denial Constraint Discovery with Hydra“. In: *Proceedings of the VLDB Endowment* 11.3 (2017), pp. 311–323.

[5] Cristian Consonni et al. „Discovering Order Dependencies through Order Compatibility“. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 2019, pp. 409–420.

[6] Jirun Dong and Richard Hull. „Applying Approximate Order Dependency to Reduce Indexing Space“. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 1982, pp. 119–127.

[7] École Polytechnique Fédérale Lausanne (EPFL). *The Scala Programming Language*. Version 2.13.1. 2019.

[8] Seymour Ginsburg and Richard Hull. „Order Dependency in the Relational Model“. In: *Theoretical Computer Science* 26.1 (1983), pp. 149–195.

[9] Ykä Huhtala et al. „Efficient Discovery of Functional and Approximate Dependencies Using Partitions“. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 1998, pp. 392–401.

[10] Ykä Huhtala et al. „Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies“. In: *The Computer Journal* 42.2 (1999), pp. 100–111.

[11] Ihab F. Ilyas and Xu Chu. „Trends in Cleaning Relational Data: Consistency and Deduplication“. In: *Foundations and Trends in Databases* 5.4 (2015), pp. 281–393.

[12] YIfeng Jin, Lin Zhu, and Zijing Tan. „Efficient Bidirectional Order Dependency Discovery“. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 61–73.

[13] Philipp Langer and Felix Naumann. „Efficient Order Dependency Detection“. In: *The VLDB Journal* 25.2 (2016), pp. 223–241.

[14] Lightbend Inc. *Akka: Build Powerful Reactive, Concurrent, and Distributed Applications More Easily*. Version 2.6.3. 2020.

[15] Jixue Liu et al. „Discover Dependencies from Data — A Review“. In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 24.2 (2012), pp. 251–264.

[16] Frank McSherry, Michael Isard, and Derek G. Murray. „Scalability! But at What Cost?“ In: *Proceedings of the USENIX Conference on Hot Topics in Operating Systems (HotOS)*. USENIX Association, 2015, pp. 14–14.

[17] Thorsten Papenbrock and Felix Naumann. „A Hybrid Approach for Efficient Unique Column Combination Discovery“. In: *Proceedings of the Conference Datenbanksysteme in Business, Technologie Und Web (BTW)*. Gesellschaft für Informatik, 2017, pp. 195–204.

[18] Thorsten Papenbrock and Felix Naumann. „A Hybrid Approach to Functional Dependency Discovery“. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2016, pp. 821–833.

[19] Thorsten Papenbrock et al. „Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms“. In: *Proceedings of the VLDB Endowment* 8.10 (2015), p. 12.

[20] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. „Distributed Discovery of Functional Dependencies“. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1590–1593.

[21] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. „Distributed Implementations of Dependency Discovery Algorithms“. In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1624–1636.

[22] P. Griffiths Selinger et al. „Access Path Selection in a Relational Database Management System“. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 1979, pp. 23–34.

[23] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. „Fundamentals of Order Dependencies“. In: *Proceedings of the VLDB Endowment* 5.11 (2012), pp. 1220–1231.

[24] Jaroslaw Szlichta et al. „Business-Intelligence Queries with Order Dependencies in DB2“. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 2014, pp. 750–761.

[25] Jaroslaw Szlichta et al. „Effective and Complete Discovery of Bidirectional Order Dependencies via Set-Based Axioms“. In: *The VLDB Journal* 27.4 (2018), pp. 573–591.

[26] Jaroslaw Szlichta et al. „Effective and Complete Discovery of Order Dependencies via Set-Based Axiomatization“. In: *Proceedings of the VLDB Endowment* 10.7 (2017), pp. 721–732.

[27] Jaroslaw Szlichta et al. „Erratum for Discovering Order Dependencies through Order Compatibility“. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 2020.

[28] Jaroslaw Szlichta et al. „Expressiveness and Complexity of Order Dependencies“. In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1858–1869.

[29] Jaroslaw Szlichta et al. „Queries on Dates: Fast yet Not Blind“. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. Association for Computing Machinery, 2011, pp. 497–502.

[30] The Apache Software Foundation. *Apache Flink - Stateful Computations over Data Streams*. 2019. URL: `https://flink.apache.org/` (visited on 08/03/2020).

[31] The Apache Software Foundation. *Apache Spark - Unified Analytics Engine for Big Data*. 2018. URL: `https://spark.apache.org/` (visited on 08/03/2020).

[32] Shouzhong Tu and Minlie Huang. „Scalable Functional Dependencies Discovery from Big Data“. In: *Proceedings of the International Conference on Multimedia Big Data (BigMM)*. IEEE, 2016, pp. 426–431.

[33] Feiyue Ye et al. „A Framework for Mining Functional Dependencies from Large Distributed Databases“. In: *Proceedings of the International Conference on Artificial Intelligence and Computational Intelligence (AICI)*. IEEE, 2010, pp. 109–113.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die digital eingereichte Version ist mit der vorliegenden Arbeit identisch.

Datum	Unterschrift (Sebastian Schmidl, B.Sc.)