Distributed Order Dependency

Sebastian Schmidl, Juliane Waack[1]

Abstract: When dealing with large amounts of data, finding hidden relationships in that data can help with maintaining and improving data quality and optimizing query performance. Order dependencies (ODs) are among these kinds of hidden relationships. A few solutions for the automatic detection of ODs have been proposed. All of them are designed to run on a single system and have at least exponential runtime complexity. In this work we introduce an OD discovery algorithm, called DODO, that distributes the detection of ODs across several machines in a cluster, thereby improving both speed and scalability. DODO is built to be fault-tolerant and to work with a dynamically changing cluster size. We evaluate DODO in respect to its performance, scalability and robustness. We demonstrate that the single node setup of DODO is about twice as fast as the OCDDiscover algorithm by Consonni et al., which DODO is based on. Running DODO distributed across multiple nodes achieves a significant speed-up compared to the single node setup. This shows how distribution can be used to increase performance for OD discovery algorithms, which are limited by the power of the processors they are run on.

Keywords: Actor Model; Akka; Distributed Computing; Parallelization

1 Introduction

With the growing interest in data analytics and near real-time data processing, data quality and query optimization are gaining importance again. They can address the scale and complexity of current data-intensive applications. Without clean data and highly optimized queries, organizations will not be able to take full advantage of their existing and upcoming opportunities for data-driven applications. Data profiling attempts to understand and find hidden metadata, such as integrity constrains and relationships, in existing datasets. Discovered integrity constraints are used to characterize data quality and to optimize business query execution.

We take a look at order dependencies (ODs) [GH83; SGG12], which describe order relationships between lists of attributes in a dataset. ODs are closely related to functional dependencies that have been studied extensively in research [Li12]. Compared to functional dependencies, ODs capture the order of values in a column as well. An OD is noted with the $\mapsto$ symbol putting two lists of attributes over a relational dataset into relation: $X \mapsto Y$. The intuition is that when we order the relation based on the left-hand side list of attributes

[1] Hasso-Plattner-Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, {sebastian.schmidl, juliane.waack}@student.hpi.de

X and the OD $X \mapsto Y$ holds, then the relation is also ordered according to Y. The ordering based on a list of attributes is lexicographical and produces the same result as the `ORDER BY`-clause in SQL.

The automatic discovery of ODs is hard, because ODs are naturally expressed with lists rather than sets of attributes, which would be the case for functional dependencies. This results in a large candidate space that is factorial in the number of attributes in the dataset. A large search space means a bad worst-case time complexity for the corresponding discovery algorithm and also increases the potential memory consumption of the approach. In the process of finding a fast and scalable discovery algorithm several definitions for a minimal set of ODs have been proposed. Langer; Naumann [LN16] use the original list-based form introduced in [SGG12], Szlichta et al. [Sz17] use a set-based canonical form, and Consonni et al. [Co19] use the notion of order compatibility to aid their discovery approach. Nevertheless all current algorithms for OD discovery have at least an exponential runtime complexity.

The discovery of a minimal set of ODs for relative small datasets consisting of only a thousand records with up to 100 columns can still take hours to complete. The OD discovery for those small datasets is CPU-bound. This means that more computational power can decrease the time needed for the algorithms to complete. Vertical scaling is limited by the available hardware, which is expensive. We therefore propose a scalable, fault-tolerant, and distributed OD discovery algorithm, called DODO, that scales horizontally across multiple nodes and allows dynamic cluster sizes. Our algorithm uses the minimality definition by Consonni et al. [Co19] and outperforms their discovery algorithm OCDDiscover by a factor of two on a single node. Our experiments with running DODO across multiple nodes shows a good scalability and a significant reduction of computation times. To our knowledge, DODO is the first implementation of a distributed OD discovery algorithm.

This report is structured as follows: In Sect. 2 we review related work. Our novel distributed OD discovery approach DODO is described in Sect. 3, followed by our experimental evaluation of DODO in Sect. 4. We end our paper with a conclusion and potential future work in Sect. 5.

2 Related work

In comparison to the related field of functional dependencies, literature on ODs is comparatively rare. The idea of considering the ordering of attributes in a relation as a kind of dependency was first introduced in 1982 by Ginsburg; Hull [GH83]. Their work formalizes the so-called *point-wise ordering* with a complete set of inference rules and shows that the problem of inference in their formalism is co-NP-complete [GH83]. Ginsburg; Hull's definition of *point-wise ordering* specifies that a set of attributes in a relation orders another set of attributes.

In 2012, Szlichta et al. [SGG12] introduced a definition for ODs, which considers lists of attributes instead of sets of attributes. This leads to a lexicographical ordering of tuples as generated by the `ORDER BY` operator in SQL. The definition of ODs by Szlichta et al. was used throughout the following research in this area [Co19; LN16; Sz17] and is also the basis of this work.

The problem of efficient discovery of ODs in existing datasets using Szlichta et al.'s definition was first approached by Langer; Naumann [LN16]. Their algorithm is called ORDER and first computes all potential OD candidates and then traverses the candidate lattice in a bottom-up manner employing pruning rules. ORDER has a factorial worst-case complexity over the number of attributes in the relation. Unfortunately, the aggressive pruning rules of ORDER affect the completeness of their results [Sz17].

Szlichta et al. present another approach to OD discovery called FASTOD. It is complete and faster than ORDER. FASTOD uses a polynomial mapping of list-based ODs to a set-based canonical form, which reduces the algorithm's worst-case complexity to $O(2^{|n|})$, in the number of attributes n.

Most recently, Consonni et al. [Co19] presented a third approach on OD discovery using the concept of order compatibility dependencies (OCDs), called OCDDiscover. It has a factorial worst-case runtime, but Consonni et al. showed that it can outperform FASTOD on some datasets. OCDDiscover is capable of running multi-threaded, which makes it easier to adapt the algorithm to a distributed system. This was the state of published research when we started on our project, which is why we based our own algorithm on OCDDiscover. Since then, Godfrey et al. [Go19] have published an Errata Note demonstrating an error in Consonni et al.'s proof of minimality, which renders the results of OCDDiscover incomplete. We still base our approach on OCDDiscover, since the goal of our work is the distribution of an already existing algorithm with a focus on reliability and scalability.

3 Distributed discovery of ODs

We now present our approach to OD discovery, called DODO. First, we explain how OCDDiscover [Co19] traverses the candidate space to find minimal OCDs and ODs in Sect. 3.1. This is the algorithmic foundation of our approach, which we slightly adopt to be able to distribute the discovery across several nodes in a cluster (see Sect. 3.2). Sect. 3.3 then describes DODO's architecture and Sect. 3.5 the communication protocols used to make a DODO cluster fault-tolerant and elastic.

3.1 The OCDDiscover algorithm

The OCDDiscover algorithm creates the search space to find ODs over OCDs [Co19]. This reduces the number of candidates that the algorithm must check, because OCDs are symmetrical. Consonni et al. prove that if a OD holds, than a functional dependency and a OCD hold as well. Based on this theorem, they then use a breadth-first search strategy to identify OCD relations in the dataset to discover minimal dependencies before longer ones.

The OCDDiscover algorithm consists of two phases. In the first phase, OCDDiscover performs an initial pruning step that removes constant columns and reduces order equivalent columns of the original column set. The constant columns and order equivalent columns are part of the output. They would generate a huge amount of ODs and those ODs can easily reconstructed from the result later on.

In the second phase, the actual OD discovery takes place. OCDDiscover generates OCD candidates from the remaining columns in levels, one after the other. For each level the algorithm checks if the OCD candidates hold. If a candidate holds, the candidate is further checked for being order dependent, the results are emitted, and child OCD candidates are generated. This step includes further pruning rules [Co19]. If the OCD candidate does not hold, no new candidates starting from it are generated. All newly generated candidates are then put into a list for the next level.

Consonni et al. proved in their paper, that they would be able to find all minimal ODs. However, as Godfrey et al. showed in their Errata Note, Consonni et al. made a mistake in one of their proofs and this method of building the search space does not create minimal ODs with repeated prefixes [Go19].

3.2 The DODO algorithm

We use Consonni et al.'s work as the basis of our discovery algorithm. Conceptually, DODO works the same as OCDDiscover. This means that we also perform a breadth-first search in a OCD candidate tree and our algorithm also consists of an initial pruning step followed by the actual search step. However, our implementation of the algorithm differs.

Instead of generating our candidates level-wise, we use a task-based approach with a single task queue. We initiate the task queue with the initial OCD candidates comparable to the first level generation by Consonni et al. Each task's input is the OCD candidate. Its outputs are the results from checking the candidate and a set of new tasks that can be empty. The results are emitted as output and the new tasks (child candidates) are added to the task queue. This is possible, because OCD candidate checking and the generation of new candidates is independent from the other candidates [Co19]. This setup allows us to parallelize the processing, because the tasks in the task queue can be processed independently and concurrently. To distribute our algorithm, we just spread the tasks, ideally evenly, across our cluster.

3.3 Implementation of the DODO algorithm

We implement the DODO algorithm using Akka [Li18], Akka Clustering [Li19] and the Scala programming language [Éc19]. Akka is a toolkit for building distributed message-driven applications using the actor programming model that was first introduced in Orleans [Be14]. It provides tools for concurrency, distribution, and elasticity.

The DODO algorithm is designed as a peer-to-peer cluster. All nodes in the cluster are equal and employ the same internal architecture. The nodes are arranged on a ring, so that every node has two neighbors. Based on this cluster layout, we define communication protocols (see Sect. 3.5) to balance the workload, to recover from node failures, and to allow dynamic cluster sizes. Each node runs its own actor system with the complete set of DODO actors (Sect. 3.4 describes them in more detail) and works on the checking of OCDs and the generation of new OCD candidates. Fig. 1 shows the cluster architecture and the internal architecture of the nodes' actor systems exemplarily by node two.

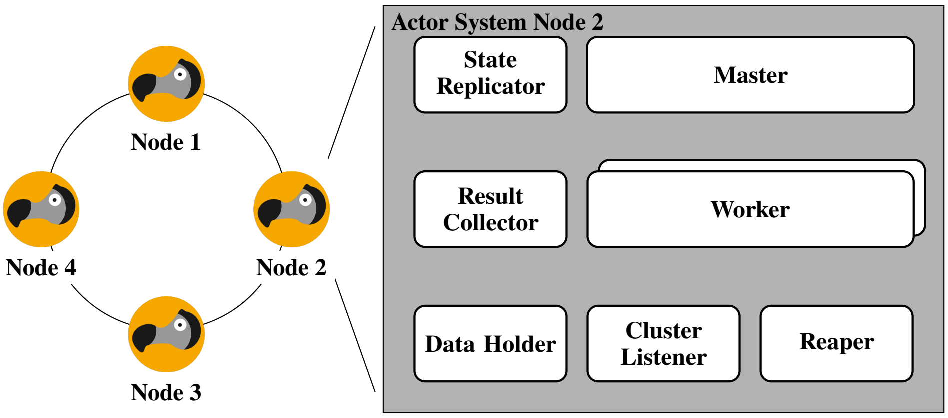


Fig. 1: DODO architecture

The execution of the DODO algorithm in the cluster is initiated by a seed node. The seed node plays a special role in our cluster and it is the only exception to the peer-to-peer approach. It is started before all other nodes and has the following unique tasks:

- The seed node is the initial contact point for all nodes that want to join the cluster.
- It always loads the dataset from disk into memory.
- It performs the initial pruning and generates the first OCD candidates.

The seed node is the first node in the cluster and may be the only one in the cluster for some time. If the seed node fails during this first phase, the whole cluster shuts down and must be restarted. We do not implement resilience for this first phase. This seemed to be a reasonable constraint, though, because the first phase only takes a few seconds.

After the seed node started up, it immediately starts with the discovery phase. All other nodes that already joined the cluster wait for the seed node to finish data loading and initial pruning. The discovery phase can not start until the seed node removed all constant columns and reduced the order equivalent columns. Therefore, the seed node broadcasts the reduced column set to all joined nodes once it finished pruning them. Our algorithm requires all nodes to have access to the whole dataset, because we do not partition the candidates and every node can check every candidate. This means that nodes load the dataset from their neighbors using data streaming, where the seed node is the data origin.

Every node needs the dataset, the reduced column set, and some OCD candidates to start the discovery. Later joining nodes, therefore, ask their direct neighbors in the ring for the dataset and the reduced column set. The distribution of OCD candidates is implemented using a work stealing protocol, which we describe in Sect. 3.5.1.

3.4 DODO node architecture

Every node in the cluster has the same internal architecture, because the nodes form a peer-to-peer cluster, where all nodes are equal. Every node can become the seed node that performs the initialization steps at the beginning of the algorithm. The node architecture is shown in Fig. 1 on the right as an example for node 2. Each node employs its own actor system running actors of seven different actor types.

Master For each node we use a master-worker pattern to utilize the multi-processing and multi-threading capabilities of the node. The Master actor is the node-local coordinator responsible for holding the processing state and communicating with the other nodes. It uses a data holder actor and a state replicator actor to outsource some of the management logic. Both those actor types are explained later on. The Master actor's state consists of a OCD candidate queue and a pending work queue. We use a pull-based approach for the distribution of tasks to Worker actors on a node. An idle Worker asks the Master for the next work packet (a batch of OCD candidates) and the Master sends the packet to the Worker actor and moves the candidates to the pending work queue. After receiving the results from the Worker the Master removes the candidates from the pending work queue. This means that the Master does not have to monitor the state of every single Worker. The Master also implements the work stealing protocol (see Sect. 3.5.1) to balance the work across the nodes in the cluster and it initiates the node shutdown when their is no more work left and all ODs have been found (see Sect. 3.5.2).

Worker A node can have an arbitrary number of Worker actors. We propose to use $n - 1$ Workers, where n is the number of available threads on the node. This ensures that one thread is still available even if the Workers are blocking the threads, which should not occur. The Workers perform the actual work of our algorithm. They check OCD and OD candidates and derive new ones from the found OCDs. The new candidates are pruned and then sent back to the Master. This work consumes most of the processing resources and is therefore spread across multiple Worker actors on each node. The Workers send all found OCDs and ODs to the local Result Collector actor, which records them in a persistent file.

Result Collector The Result Collector actor receives the ODs and OCDs from the local Workers. It collects and counts these results and writes them to a file, so they can be recovered even if the node fails unexpectedly.

Data Holder The Data Holder actor is responsible for loading the dataset and for storing the dataset in heap. If we provide a file path to a dataset to the node during startup, the Data Holder will load and parse the dataset from disk. It also performs type inference to ensure the correct ordering relation for dates, numbers and text. If no file path is available, the Data Holder asks the node's neighbors on the ring (see Cluster Listener actor) to stream the dataset to it. Once the dataset is loaded in memory, the Data Holder makes the reference available to the Master and the Worker actors. From this point on it is able to stream the dataset to other nodes as well.

State Replicator The State Replicator actor is a utility actor for the Master actor. It holds the state of its two neighboring nodes and shares the state of its own local Master actor with them. This enables us to recover work from unexpectedly failed nodes and is explained in more detail in Sect. 3.5.3.

Cluster Listener Each node in the DODO cluster has a position on a conceptual node-ring. The position is determined based on the node's network address consisting of hostname and port information. Leaving and joining nodes change the arrangement of the nodes on the ring. The Cluster Listener actor is responsible for creating a representation of the ring structure and for listening on cluster events introducing or removing nodes. It determines the position of the local node on the ring and shares the neighboring node's address information with the other local actors. Whenever the neighbors of the local node change, the updated neighbors are sent to the State Replicator to change the target for state updates. In contrast, the Data Holder only receives information about the neighboring nodes, when it specifically asks for them, because it only needs the information when requesting the dataset from another node.

Reaper The Reaper actor watches all other actors and cleanly shuts down the local actor system once all of them are terminated.

3.5 Communication protocols

This section introduces the three core communication protocols used by DODO. They allow DODO to dynamically scale to a different number of nodes in a cluster and to be fault-tolerant in the case of message loss and node failures. DODO is able to rebalance work across the cluster. This allows us to dynamically add more nodes to an already running cluster. This is implemented by the work stealing protocol introduced in the next section (Sect. 3.5.1). Sect. 3.5.2 then describes the downing protocol, which ensures that all nodes only shut down if there is no more work available in the whole cluster. In Sect. 3.5.3, we present our state replication protocol that synchronizes the nodes' states to their cluster neighbors to prevent loss of candidates.

3.5.1 Work stealing protocol

When a node is out of work, either because it just joined the cluster or because all the candidates it checked got pruned and no new candidates were generated, it tries to take over some of the work of the other nodes in the cluster. We call this process work stealing. It ensures that no node is idle while the others are still checking candidates and it balances the workload over the cluster. The workload of a node is defined by the number of candidates waiting to be processed in its candidate queue. Ideally every node would have processed nearly the same number of candidates at the time the algorithm ends.

The work stealing protocol consists of three phases: preparation, stealing, and acknowledgment. It is initiated when a node's candidate queue is empty, even if some of its workers are still processing candidates and might return new ones soon. Starting work stealing this early reduces idle times. The node with the empty candidate queue (the thief node) starts the preparation phase by asking all other nodes for their current workload. Because some nodes might be very busy, it only waits for a specific duration to receive the results. Only the nodes from which the thief received answers are considered as potential victims. The thief node calculates the average workload of the potential victim nodes and itself. This is the ideal workload for every node. To improve the work distribution in the cluster, the thief only requests candidates from nodes that have a workload above average and only so many candidates that its own workload does not exceed the average. It then proceeds to the stealing phase.

In the stealing phase the thief sends out the requests to steal work from the other nodes. Those requests are directed to the selected victims and contain the amount of requested candidates. The victim node sends back the requested number of candidates to the thief, but at most half of the candidates in its candidate queue. This ensures that the victim node is not running out of work directly after sending out its candidates to another node, which would lead to the candidates being send back and forth between two or more nodes. The number of candidates left in the victim's queue might be less than initially computed, because it could have processed a lot of them in the meantime. To prevent data loss, the victim is not directly deleting the stolen candidates from its queue, but it is moving them into another queue, awaiting the acknowledgment of the thief.

In the acknowledgment phase, the thief node concurrently receives the candidates from the victims and adds them to its own pending queue. It sends the acknowledgments to the victims, so that they can delete the stolen candidates. To ensure that the sent candidates arrive at the thief node and will get processed, the victim uses Akka's actor monitoring to get notified if the thief node dies. In the case that the thief fails before it could acknowledge the receipt of the candidates, the victim will recover the to-be-stolen candidates and add them back to its own candidate queue.

3.5.2 Downing protocol

When all ODs have been found and there are no more candidates to check, the system should shut itself down cleanly. DODO is distributed across different nodes that act relatively autonomous. This means they have to communicate with each other to figure out if all of the nodes are out of work. The downing protocol defines the communication required to achieve a complete and clean shutdown of the cluster ensuring that all OCD candidates have been processed.

A node starts the downing protocol, when it has no more candidates in its candidate queue, all local Worker actors have finished processing, and an attempt to get more work using work stealing was unsuccessful. We have to make sure that no more Workers are processing candidates, because each candidate that is being processed could generate new candidates for the candidate queue. If the three mentioned criteria a fulfilled, a DODO node changes into a downing state, where it is idle and waiting for shutdown. It must now ensure that all other cluster nodes also have no more work left. If another node still holds unprocessed candidates or is still in the process of creating new candidates, these candidates could be redistributed in the cluster using work stealing. This means that the idle node can contribute its resources instead of shutting itself down. Therefore, the node asks all cluster members if they are ready for shutdown. It waits for a response from every other node in the cluster, also being aware of nodes leaving the cluster and subsequently not awaiting their response anymore. If any of the other nodes responds with a message indicating that they are still holding or processing candidates, the node leaves the downing state and starts the work stealing protocol again. If all nodes have either left the cluster or responded that they have no more work, the node shuts itself down. This process is the same for all other nodes in the cluster and eventually all nodes will shut down.

3.5.3 State replication protocol

The state replication protocol ensures that a DODO cluster can tolerate complete node failures without losing work. Every node regularly replicates its state to its left and right neighbor in the cluster ring. Only if three neighboring nodes fail simultaneously would DODO lose the unprocessed candidates of one node. In the case of this rare failure scenario, the whole DODO cluster would need to be restarted to ensure complete results.

The basis of the state replication protocol is the node ring, because the node's position on the ring determines the state replication targets. The ring structure is computed by the Cluster Listener actors based on the nodes' network addresses. It is implemented as double linked list, where the last node also holds a reference to the first node and around. Every node replicates its state to its predecessor and successor in this list. The State Replication actors are responsible for sending their nodes' state to its neighbors in a regular interval. They are also the receiving endpoint and storage for the incoming state messages from the other nodes. The state replication interval can be configured by the user of the DODO algorithm.

If it is time to replicate the state the State Replicator asks the Master for its current state of the candidate queue and the pending work queue. It then takes the neighbor's addresses, received from the local Cluster Listener actor, and sends out the state and an increasing version number to them. The receiving nodes compare the incoming version number with their local state's version. If the new state version is higher, they save the new state and forget the old one. If it is not, they discard the incoming message, because their existing state replica is more recent. We do not use acknowledgments for state transmissions, because the regular replication interval ensures that the replication is tried again in the case of message loss and we want to reduce the stress on the network.

There is a special case, when a state replication is triggered outside of the regular interval. When a new node joins the cluster, the Cluster Listener updates and reorders its list of all the nodes in the cluster. It then notifies the State Replicator actor about the changed neighbors. If the node's neighbors have changed, the State Replicator sends its own state to its new neighbor right away and marks the state of the node that is no longer its neighbor for deletion. It will be overwritten once the State Replicator received the state of its new neighbor. This ensures that the new node arrangement is quickly reflected in the state replication status and keeps the time, where a node only has one other state replica small.

In the case of a node failure or when a node leaves the cluster, the cluster must recover the work the leaving node did not finish. Both neighbors of the leaving node are alerted about the event by their local Cluster Listener. The Cluster Listener automatically rearranges the nodes in the ring and informs the State Replicator about the left node and its new neighbor. This new neighbor was the leaving nodes other neighbor. This means that the new neighbor also holds a replica of the left node's state. To determine which of the two nodes has the most up to date state of the leaving node, they exchange the version numbers of their state replicas. The node with the higher version number adds the leaving nodes state to its own candidate queue to ensure that all candidates get processed. The node with the lower version number deletes its local copy of the leaving node's state. The two nodes then share their updated state with each other to reach a three-fold state replication for all nodes again.

There might have elapsed some time between the last state replication and the node failure. This means that in the meantime the failing node generated candidates that are not reflected in the state replicas. The new candidates are no problem, because they can be recomputed from the existing candidates in the state replica. The results were made persistent on the local disk of the failed node and can be recovered. However, as some candidates are processed by the failed node and the node that recovered the state, DODO will produce duplicate results in the presence of node failures.

4 Evaluation

In this section, we present an initial evaluation of the performance and fault-tolerance of our algorithm. In Sect. 4.1, we evaluate the correctness of the results produced by our algorithm when run in different settings and under faults, Sect. 4.2 then compares the performance with our baseline OCDDiscover by Consonni et al. Sect. 4.3 explores the scalability of DODO across multiple nodes. The code for OCDDiscover was provided to us by the authors.

To evaluate the performance of the different OD discovery approaches, we measure the overall runtime the algorithm needed to compute all ODs and OCDs. The runtime of the algorithm includes startup and shutdown times and the algorithm must terminate to produce a valid result. As a consequence, when running DODO in a distributed setting, this measurement will include node discovery and shutdown synchronization times.

All experiments were performed on homogeneous nodes with 20 cores and 64 GB of main memory. DODO is implemented in Scala and the nodes run it with Java 1.8 with the Java JVM heap space limited to 40 GB. We adopted the *flight_1k* dataset available from the Information Systems Group at Hasso Plattner Institute[3] to reduce the computation time to an order of minutes. This was achieved by removing sparse columns. We ended up with a dataset consisting of 1000 rows and 36 columns: *flight_r1k_c36*.

4.1 Correctness and fault-tolerance

Because we based our approach on OCDDiscover by Consonni et al., we don't expect our algorithm to output a complete minimal set of ODs. Our goal is to output the same OCDs and ODs as OCDDiscover.

We find that DODO computes the same OCDs and ODs as OCDDiscover, independent of whether DODO is run on a single node or distributed across multiple nodes. We did not compare the actual results for our medium-size test dataset *flight_r1k_c36*, because it contains too many ODs. But Tab. 1 shows, that DODO finds the same number of ODs in the dataset as OCDDiscover. Column four in Tab. 1 indicates the sum of found OCDs and ODs. `iw,jn, kfailure(s)` indicate an experiment with DODO using `i` workers per node, the system consisting of `j` nodes overall and `k` nodes of them were killed during the experiment.

If DODO experiences node failures, it redistributes the OCD candidates to the remaining nodes using the state replication protocol (see Sect. 3.5.3). This leads to duplicate checks and along with that to the same OD being found multiple times, as is the case in the experiment `8w, 8n, 1failure`. DODO finds more ODs in this experiment than there are in the dataset. This is due to the fact that we do not yet perform deduplication of the found ODs.

[3] `https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html`

Experiment	CCs	OECs	OCDs + ODs	runtime
OCDDiscover, 8w	7	5	49 514	7 m 09 s
8w, 1n, 0failure	7	5	49 514	3 m 42 s
8w, 8n, 0failure	7	5	49 514	49 s
8w, 8n, 1failure	7	5	50 055	54 s
8w, 8n, 2failures	7	5	48 374	55 s
8w, 8n, 3failures	7	5	48 352	52 s

Tab. 1: Result comparison between OCDDiscover, DODO, and DODO in the presence of node failures.

The experiments with two and more failing nodes do not find all ODs in the dataset. Unfortunately, we performed the experiments manually and the first node failure was introduced before the node was able to replicate its state to the other nodes in the cluster. This caused the system to loose all OCD candidates of this node. But DODO was able to recover from the following node failures, because those nodes were already able to replicate their state.

In Tab. 1, we already notice that DODO in a single nodes setup with eight workers is nearly 2x as fast as OCDDiscover with eight threads. Running DODO on multiple nodes decreases the time even more as one would expect. If there are no failures, DODO outputs the correct results independent of the number of nodes it is deployed on.

4.2 Comparing with OCDDiscover

In Fig. 2, we compare the performance of DODO with OCDDiscover on a single node using our test dataset. OCDDiscover is only capable of running on a single-node setup, but can scale by increasing the number of used threads. We measure the runtime of the algorithms to find all ODs with a different degree of parallelism. OCDDiscover scales by increasing the number of threads and DODO scales with the number of workers in a single node setup.

We find that DODO with a single worker already outperforms the single-threaded version of OCDDiscover by over a factor of two. OCDDiscover switches to a completely different implementation when it is run multi-threaded. This version performs slightly better compared to DODO. Here, DODO can only achieve a performance increase by a factor of around 1.8 compared to OCDDiscover. Apart from that, DODO scales better than OCDDiscover. DODO achieves a performance increase of a factor of 2.8 when run with 18 workers compared to OCDDiscover with 18 threads. This shows that our implementation using the actor model with no explicit synchronization barriers is superior to the explicit multi-threading implementation by Consonni et al.

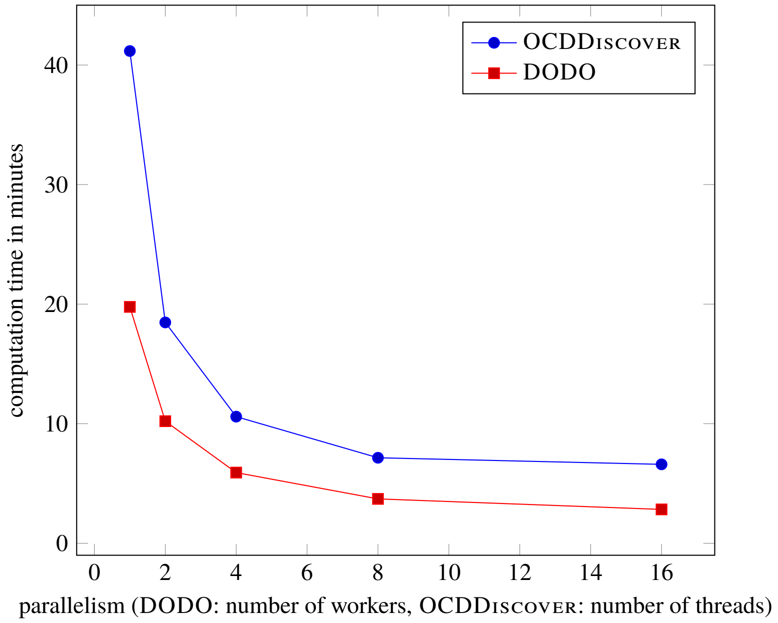
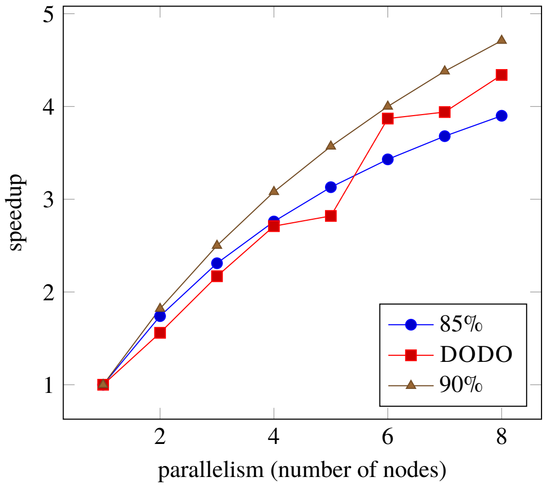


Fig. 2: Single node runtime comparison of our approach and the baseline algorithm OCDDiscover with the test dataset *flight_r1k_c36*.

4.3 Scalability

In Sect. 4.2, we already showed that DODO outperforms our baseline OCDDiscover in a single node setup. In this section, we analyze the scalability of DODO when run on multiple nodes in a cluster setup.

Fig. 3 shows the results of our cluster scalability experiments using our test dataset *flight_r1k_c36*, where we measured the runtime of DODO with different cluster sizes. Fig. 3a plots the speed of running DODO on multiple nodes and Fig. 3b shows the actual runtime of DODO.



(a) Computation time speedup of DODO compared to theoretical 85% and 90% parallel code.

# nodes	computation time
1	3 m 37 s
2	2 m 19 s
3	1 m 40 s
4	1 m 20 s
5	1 m 17 s
6	56 s
7	55 s
8	50 s

(b) Computation time for finding all ODs in the dataset of different cluster sizes.

Fig. 3: Scaling our algorithm over the number of nodes and keeping the number of workers fixed at eight workers per node using the test dataset *flight_r1k_c36*.

We find that using DODO in a cluster setup greatly reduces overall computation time despite the additional effort spend on node discovery, state replication and inter-node synchronization. Finding all ODs in our test dataset takes about three and a half minutes on a single node. This time is already reduced to under a minute when using six or more nodes in the DODO cluster. As we can see in Fig. 3a, DODO can achieve a speedup of 4.3 for eight nodes compared to the single node setup. This indicates that between 85% and 90% of DODO's code is executable in parallel. The outliers in the diagram can be explained by our downing protocol implementation, which waits for all nodes to come to a mutual decision that all candidates were processed before shutting down nodes. This protocol can delay the shutdown procedure when the timeouts do overlap just slightly or one node still works on the last OCDs candidates.

5 Conclusion

In this work, we presented DODO, a scalable, fault-tolerant, distributed OD discovery algorithm implemented on top of the Akka toolkit. DODO can be deployed on a single node or in a cluster. It makes use of work stealing to distribute load and to deal with dynamic cluster sizes. A state replication protocol ensures fault-tolerance in the case of message loss and node failures. We based DODO's OD discovery approach on OCDDiscover introduced by Consonni et al. [Co19]. Our approach outperforms OCDDiscover by about a factor of two on a single node setup and we can even scale out across several nodes. Our experiments show that distributing an OD discovery algorithm across nodes greatly reduces computation time of the algorithm. DODO in cluster mode with eight nodes achieves a four times speedup compared to a single node DODO setup. This proves that employing distributed algorithms to aid OD discovery reduces computation times and increases their robustness. This opens the way for using OD discovery to find hidden dependencies in big data.

DODO currently uses Consonni et al.'s definition of minimality. As Godfrey et al. [Go19] have pointed out, this does not find all minimal OCDs in the dataset. A valuable next step would therefore be the change of the discovery algorithm to correctly prune the search space according to [Go19]. Additionally, we currently do not remove duplicates from the results and we do not automatically merge the result sets from different nodes. Duplicates are produced when DODO recovers from node failures and some OCD candidates must be re-checked on another node to prevent data loss.

DODO is implemented using the actor programming model. It provides a dynamic computation aspect. This can be used to run different OD discovery approaches in a single system at the same time and to exchange information between them. This could decrease the computation time further, because the exchanged information can be used to prune the search spaces for the different approaches dynamically. Future work has to evaluate if the benefits of this approach outweigh their disadvantages, such as increased communication overhead and running two approaches simultaneously. Our approach does not make use of this aspect of the actor model so far.

Finally, we did not yet perform micro-benchmarks with DODO. It would be interesting to further evaluate (i) the memory boundaries of DODO, (ii) the different settings for batch-sizes and timeout durations, and (iii) the impact of the work stealing and state replication protocols on the performance of our algorithm.

References

[Be14] Bernstein, P. A.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J.: Orleans: Distributed virtual actors for programmability and scalability. MSR-TR-2014–41/, 2014.

[Co19] Consonni, C.; Sottovia, P.; Montresor, A.; Velegrakis, Y.: Discoverying order dependencies through order compatibility. In: International Conference on Extending Database Technology. 2019.

[Éc19] École Polytechnique Fédérale Lausanne (EPFL): The Scala Programming Language, 2019, URL: `https://www.scala-lang.org/`, visited on: 09/10/2019.

[GH83] Ginsburg, S.; Hull, R.: Order dependency in the relational model. Theoretical computer science 26/1-2, pp. 149–195, 1983.

[Go19] Godfrey, P.; Golab, L.; Kargar, M.; Srivastava, D.; Szlichta, J.: Errata Note: Discovering Order Dependencies through Order Compatibility, 2019, arXiv: `1v01020.5091 [cs.DB]`.

[Li12] Liu, J.; Li, J.; Liu, C.; Chen, Y.: Discover Dependencies from Data—A Review. IEEE Transactions on Knowledge and Data Engineering 24/2, pp. 251–264, 2012.

[Li18] Lightbend, Inc.: Akka, 2018, URL: `https://akka.io/`, visited on: 08/15/2018.

[Li19] Lightbend, Inc.: Akka Documentation – Clustering, 2019, URL: `https://doc.akka.io/docs/akka/current/index-cluster.html`, visited on: 09/10/2019.

[LN16] Langer, P.; Naumann, F.: Efficient order dependency detection. The VLDB Journal—The International Journal on Very Large Data Bases 25/2, pp. 223–241, 2016.

[SGG12] Szlichta, J.; Godfrey, P.; Gryz, J.: Fundamentals of order dependencies. Proceedings of the VLDB Endowment 5/11, pp. 1220–1231, 2012.

[Sz17] Szlichta, J.; Godfrey, P.; Golab, L.; Kargar, M.; Srivastava, D.: Effective and complete discovery of order dependencies via set-based axiomatization. Proceedings of the VLDB Endowment 10/7, pp. 721–732, 2017.