CONTENTS

Efficient Discovery of Approximate Order Dependencies

Reza Karegar
University of Waterloo, CA
mkaregar@uwaterloo.ca

Parke Godfrey
York University, CA
godfrey@yorku.ca

Lukasz Golab
University of Waterloo, CA
lgolab@uwaterloo.ca

Mehdi Kargar
Ryerson University, CA
kargar@ryerson.ca

Divesh Srivastava
AT&T Chief Data Office, US
divesh@att.com

Jaroslaw Szlichta
Ontario Tech Univ, CA
jarek@ontariotechu.ca

ABSTRACT

Order dependencies (ODs) capture relationships between ordered domains of attributes. Approximate ODs (AODs) capture such relationships even when there exist exceptions in the data. During automated discovery of dependencies, *validation* is the process of verifying whether a dependency holds. We present an algorithm for validating AODs with significantly improved runtime performance over existing methods, and prove that it is *minimal* and has *optimal* runtime. By replacing the validation step in a recent algorithm for AOD discovery with ours, we achieve orders-of-magnitude improvements in performance.

1 INTRODUCTION

1.1 Motivation

Functional dependencies (FDs) specify that the values of given attributes *functionally determine* the value of a target attribute. *Order dependencies* extend FDs to state that, additionally, the *order* of tuples with respect to the values from the domains of given attributes determines the *order* of the values from the domain of the target attribute. Table 1 shows a dataset with employee salaries. In this table, the OD that sal *orders* taxGrp holds. If one sorts the table by sal, it is sorted by taxGrp as well.

An OD implies the corresponding FD; e.g., that sal orders taxGrp implies that sal *functionally determines* taxGrp. *Order compatibility* (OC) captures the *co-ordering* aspect of an OD *without* the corresponding FD. Two lists of attributes are *order compatible* if there exists an arrangement for the tuples in the table in which the tuples are sorted according to both. Any OD can thus be equivalently represented by a pair of an OC and an FD [13]. In Table 1, that taxGrp is *order compatible* with sal holds. Note that taxGrp does *not* order sal, as an FD does not hold.

There has been recent work to automate the *discovery* of ODs from data [1, 4, 6, 10, 11]. In practice, however, constraints rarely hold *perfectly* in the data. Real data are dirty, containing wrong and inconsistent values that may violate semantically valid dependencies. This motivates the need for discovering *approximate* ODs (AODs), ODs that hold in the data but with *exceptions*. Discovered ODs deemed semantically valid can be used for data cleaning, to detect erroneous tuples, where measures are then taken to repair the errors [8]. AODs are useful even when the data are not dirty, as there can be exceptions to general rules. AODs help avoid overfitting by discovering more general dependencies.

In Table 1, tax is a fixed percentage of salary in each tax group; i.e., one, three, or eight percent. However, perc includes a concatenated zero in some rows due to data entry errors (e.g., 10% instead of 1% in t_1). Because of this, the OC that salary is order compatible with tax does *not* hold, even though this OC is intended. Similarly, the FD that pos, exp functionally determines sal does *not* hold, due to the exception of tuples t_6 and t_7, two employees with the same position and years of experience but having different salaries. With approximate ODs, we can still discover such concise and meaningful rules in these instances.

Table 1: Employee salaries

#	pos	exp	sal	taxGrp	perc	tax	bonus
t_1	sec	1	20K	A	10%	2K	1K
t_2	sec	3	25K	A	10%	2.5K	1K
t_3	dev	1	30K	A	1%	0.3K	3K
t_4	sec	5	40K	B	30%	12K	2K
t_5	dev	3	50K	B	3%	1.5K	4K
t_6	dev	5	55K	B	30%	16.5K	4K
t_7	dev	5	60K	B	3%	1.8K	4K
t_8	dev	-1	90K	C	8%	7.2K	7K
t_9	dir	8	200K	C	8%	16K	10K

Approximate ODs were introduced in [10]. Their definition of AODs, as is ours herein, is based on the concept of "tuple removal." Given a table and an OD, a *removal set* is a set of tuples which, if removed from the table, results in the OD holding. A *minimal* removal set is one with the smallest cardinality. An *approximation factor* can be defined with respect to a table and an OD, as the ratio of the size of a minimal removal set over the size of the table. For instance, for Table 1 and the OC that pos, exp is order compatible with pos, sal, the minimal removal set and the approximation factor are $\{t_8\}$ and $1/9 \approx 0.11$, respectively.

Given a table $\mathbf{r}$ and an approximation threshold $0 \leq \epsilon \leq 1$, the *discovery problem* for AODs is to find the complete set of minimal *valid* AODs in $\mathbf{r}$ w.r.t. ϵ. Exact ODs are a special case of AODs with an approximation factor of zero. Given a table $\mathbf{r}$, an OD φ, and a threshold ϵ, the problem of *validating* the candidate OD as an AOD involves verifying whether the approximation factor of φ, denoted by $e(\varphi)$, is less than or equal to ϵ.

1.2 Contributions

The extension for AOD discovery in [10, 11], however, is impractical due to its performance. While the approximate FD component can be validated in linear time [3, 10], to validate the approximate OC (AOC) component in the search, they iteratively remove the tuple—or one of the tuples, in the case of a tie—that causes the largest number of violations. This has two weaknesses: the runtime is quadratic in the number of tuples, and it is not guaranteed to find a minimal removal set.

That it is quadratic makes it prohibitively expensive to run on larger datasets. (The validation step for a candidate exact OD has a linear runtime in the number of tuples.) So while the OD discovery algorithm in [10, 11] is shown to scale to datasets with millions of tuples, it is infeasible to run their adapted AOC discovery algorithm over even moderately sized datasets. During

benchmarking, we found in some discovery runs that more than 99% of the running time is spent on validating AOC candidates.

That it deliberately does not guarantee finding a minimal removal set means that the algorithm may overestimate the approximation factor of an AOC candidate. Thus, *true* AOCs with respect to the approximation threshold can be eliminated (while the exact OD discovery algorithm is complete).

In this paper, we resolve this major bottleneck in AOD discovery via an algorithm with optimal runtime and guaranteed minimal removal set for validating AOC candidates. This brings performance of AOD discovery on par with that of OD discovery, while making the AOD discovery complete.

The paper is structured as follows, with the following key contributions. In Sec. 2, we provide background and discuss related work. In Sec. 3.1, we illustrate the established OD and AOD discovery framework—which we then adapt herein—and, in Sec. 3.2, the *iterative validation algorithm* [10, 11] it employs. In Sec. 3.3, we contribute a minimal and optimal validation algorithm based on longest increasing subsequences that decreases the runtime from quadratic to log-linear. In Sec. 4, we present our experimental results, with the following contributions. We demonstrate that AOD discovery using our validation algorithm scales to datasets with millions of tuples and tens of attributes (Exp-1 and Exp-2). We compare our adapted AOC discovery against the previous approach and demonstrate that ours is orders of magnitude faster (Exp-3). As discovering AODs enables the application of pruning rules earlier than for discovering ODs, AOD discovery can be just as efficient, if not more so. Our AOD discovery algorithm gains up to 76% improvement in runtime compared against the (exact) OD discovery algorithm (Exp-5). Given our AOD discovery algorithm is complete, we discover more AODs, and *semantically more general* AODs (thus, of higher quality). We show that we find more AODs, both due to our better scalability and the minimality of our removal sets (Exp-4 and Exp-6). Finally, in Section 5, we conclude with suggestions for future work.

2 PRELIMINARIES AND RELATED WORK

2.1 Definitions and Notation

$\mathbf{R}$ denotes a relational schema, $\mathbf{r}$ represents a table instance, and s and t denote tuples. A and B denote individual attributes and $\mathcal{X}$ and $\mathcal{Y}$ *sets* of attributes. *Lists* of attributes are presented using $\mathbf{X}$ and $\mathbf{Y}$; [] denotes the empty list and $[A \mid \mathbf{T}]$ denotes a list with *head* attribute A and *tail* list $\mathbf{T}$. Tuples t_A and $t_{\mathcal{X}}$ denote the projections of tuple t on A and $\mathcal{X}$, respectively. Wherever a set is expected but a list appears, the list is cast to a set; e.g., $t_{\mathbf{X}}$ is equivalent to $t_{\mathcal{X}}$. $\mathbf{X}'$ represents an arbitrary permutation of the values of a list $\mathbf{X}$ or set $\mathcal{X}$.

Definition 2.1. (nested order) Let $\mathbf{X}$ be a list of attributes where $\mathcal{X} \in \mathbf{R}$. Given two tuples, s and t, $s \preceq_{\mathbf{X}} t$ *iff*

- $\mathbf{X} = [\,]$; *or*
- $\mathbf{X} = [A \mid \mathbf{T}]$ and $s_A < t_A$; or
- $\mathbf{X} = [A \mid \mathbf{T}]$, $s_A = t_A$, and $s \preceq_{\mathbf{T}} t$.

Let $s \prec_{\mathbf{X}} t$ *iff* $s \preceq_{\mathbf{X}} t$ but $t \not\preceq_{\mathbf{X}} s$.

Next, we define order dependencies [1, 4, 6, 10, 11, 13].

Definition 2.2. (order dependency) Let $\mathbf{X}$ and $\mathbf{Y}$ be lists of attributes where $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R}$. $\mathbf{X} \mapsto \mathbf{Y}$ denotes an *order dependency*, read as $\mathbf{X}$ *orders* $\mathbf{Y}$. Table $\mathbf{r}$ satisfies $\mathbf{X} \mapsto \mathbf{Y}$ ($\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$) iff, for all $s, t \in \mathbf{r}$, $s \preceq_{\mathbf{X}} t$ implies $s \preceq_{\mathbf{Y}} t$. $\mathbf{X}$ and $\mathbf{Y}$ are *order equivalent* (denoted as $\mathbf{X} \leftrightarrow \mathbf{Y}$), iff $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$.

Definition 2.3. (order compatibility) Let $\mathbf{X}$ and $\mathbf{Y}$ be lists of attributes where $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R}$. $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible*, denoted as $\mathbf{X} \sim \mathbf{Y}$, iff $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

The order dependency $\mathbf{X} \mapsto \mathbf{Y}$ means that $\mathbf{Y}$'s values are monotonically non-decreasing with respect to $\mathbf{X}$'s values. Therefore, if one orders the tuples by $\mathbf{X}$, they are also ordered by $\mathbf{Y}$. The order compatibility (OC) $\mathbf{X} \sim \mathbf{Y}$ means that there exists a total order of the tuples in which they are ordered according to both $\mathbf{X}$ and $\mathbf{Y}$.

Example 2.4. In Table 1, the OD sal $\mapsto$ taxGrp holds. The OC taxGrp $\sim$ sal holds, even though the OD taxGrp $\mapsto$ sal does not.

ODs have a strong correspondence with OCs and FDs. An OD $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \sim \mathbf{Y}$ (OC) and $\mathcal{X} \rightarrow \mathcal{Y}$ (FD) hold. This gives two sources of violations for ODs: *swap*s and *split*s [13].

Definition 2.5. (swap) A *swap* with respect to OC $\mathbf{X} \sim \mathbf{Y}$ is a pair of tuples s and t such that $s \prec_{\mathbf{X}} t$ but $t \prec_{\mathbf{Y}} s$.

Definition 2.6. (split) A *split* with respect to FD $\mathcal{X} \rightarrow \mathcal{Y}$ is a pair of tuples s and t such that $s_{\mathcal{X}} = t_{\mathcal{X}}$ but $s_{\mathcal{Y}} \neq t_{\mathcal{Y}}$

Example 2.7. In Table 1, given the OD pos, exp $\mapsto$ pos, sal, tuples t_7 and t_8 constitute a swap (the OC pos, exp $\sim$ pos, sal), and tuples t_6 and t_7 constitute a split (the FD pos, exp $\rightarrow$ pos, sal).

Definition 2.8. tuples s and t are *equivalent* w.r.t. set of attributes $\mathcal{X}$ *iff* $s_{\mathcal{X}} = t_{\mathcal{X}}$. An attribute set $\mathcal{X}$ partitions tuples into *equivalence classes* [3]. The *equivalence class* of tuple $t \in \mathbf{r}$ w.r.t. $\mathcal{X}$ is denoted by $\mathcal{E}(t_{\mathcal{X}})$; i.e., $\mathcal{E}(t_{\mathcal{X}}) = \{s \in \mathbf{r} \mid s_{\mathcal{X}} = t_{\mathcal{X}}\}$. Given a set of attributes $\mathcal{X}$, a *partition* of the table with respect to $\mathcal{X}$ is the set of all equivalence classes; i.e., $\Pi_{\mathcal{X}} = \{\mathcal{E}(t_{\mathcal{X}}) \mid t \in \mathbf{r}\}$.

Example 2.9. In Table 1, $\mathcal{E}(t_{1\{\text{pos}\}}) = \mathcal{E}(t_{2\{\text{pos}\}}) = \mathcal{E}(t_{4\{\text{pos}\}}) = \{t_1, t_2, t_4\}$, and $\Pi_{\text{pos}} = \{\{t_1, t_2, t_4\}, \{t_3, t_5, t_6, t_7, t_8\}, \{t_9\}\}$.

2.2 A Canonical Mapping

A natural representation of ODs relies on lists of attributes, as in the ORDER BY statement in SQL, where the order of attributes in the list matters; e.g., the OD pos, sal $\mapsto$ pos, exp is different than the OD pos, sal $\mapsto$ exp, pos. This is unlike FDs, where the order of attributes does not matter, as with the GROUP BY statement in SQL. Working within this list-based representation, however, has led to discovery frameworks with factorial worst-case runtimes in the number of attributes [6]. Fortunately, lists are *not* inherently necessary to express ODs. In [10, 11], the authors rely on a polynomial mapping of list-based ODs into a logically *equivalent* collection of set-based *canonical* ODs to devise a discovery framework with exponential worst-case runtime in the number of attributes and linear in the number of tuples.

Definition 2.10. (canonical order compatibility) Given a set of attributes $\mathcal{X}$, $\mathbf{X}'A \sim \mathbf{X}'B$ is the OC that states that attributes A and B are *order compatible* within each equivalence class of $\mathcal{X}$. We write this as $\mathcal{X}$: A $\sim$ B in the canonical notation, factoring out the common prefix, and refer to this as a *canonical* OC.

Definition 2.11. (order functional dependency) Given a set of attributes $\mathcal{X}$, the FD that states that an attribute A is *constant* within each equivalence class of $\mathcal{X}$ is equivalent to the list-based OD $\mathbf{X}' \mapsto \mathbf{X}'A$. We write this as $\mathcal{X}$: [] $\mapsto$ A in the canonical notation, and refer to this as an *order functional dependency* (OFD).

Given a canonical OC of $\mathcal{X}$: A $\sim$ B or an OFD of $\mathcal{X}$: [] $\mapsto$ A, the set $\mathcal{X}$ is referred to as the *context* of the respective canonical OC or OFD. Intuitively, the context is the common prefix on the left- and right-side of the corresponding list-based OC or OD.

Canonical OCs and OFDs constitute the canonical ODs; i.e., OD ≡ OC + OFD. The OD of $\mathbf{X}'A \mapsto \mathbf{X}'B$ is logically equivalent to the canonical OC of $\mathcal{X}$: A ∼ B and OFD of $\mathcal{X}$A: [] ↦ B. This is $\mathcal{X}$: A ↦ B written in the canonical form.

Example 2.12. In Table 1, sal and bonus are order compatible w.r.t. the context pos; i.e., {pos}: sal ∼ bonus. In the same table, bonus is constant w.r.t. the context pos, sal; i.e., {pos, sal}: [] ↦ bonus. Therefore, sal orders bonus w.r.t. the context pos; i.e., {pos}: sal ↦ bonus.

This mapping generalizes: an OD $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \mapsto \mathbf{XY}$ *and* $\mathbf{X} \sim \mathbf{Y}$. These can be encoded into an equivalent set of canonical OFDs and OCs as follows. In the context of $\mathcal{X}$, all attributes in $\mathcal{Y}$ must be constants. In the context of all prefixes of **X** and of **Y**, the trailing attributes must be order compatible:

$$\mathbf{R} \models \mathbf{X} \mapsto \mathbf{XY} \textit{ iff } \forall A \in \mathbf{Y}.\ \mathbf{R} \models \mathcal{X}\text{: } [] \mapsto A \text{ and}$$

$$\mathbf{R} \models \mathbf{X} \sim \mathbf{Y} \textit{ iff } \forall i, j.\ \mathbf{R} \models [X_1, \ldots, X_{i-1}][Y_1, \ldots, Y_{j-1}]\text{: } X_i \sim Y_j.$$

Thus, list-based ODs can be polynomially *mapped* to a set of *equivalent* canonical ODs; i.e., canonical OCs and OFDs [10, 11]. In this work, we refer to canonical OCs simply as OCs.

Example 2.13. The OD [A, B] ↦ [C, D] is equivalent to the following canonical ODs: {A, B}: [] ↦ C, {A, B}: [] ↦ D, {}: A ∼ C, {A}: B ∼ C, {C}: A ∼ D, and {A, C}: B ∼ D.

While various algorithms have been proposed for discovering ODs, most are not *complete*. The algorithm described in [6] relies on the list-based definition and employs aggressive pruning rules to compensate for its factorial time complexity, but which make it deliberately incomplete. The authors in [4] claim completeness but their algorithm misses ODs in which the same attributes are repeated on the left- and right-hand side. A similar completeness claim has been made in [1], which was shown to be incorrect in [12]. The set-based OD discovery algorithm proposed in [10] does offer a sound and complete discovery of ODs. Thus, we build our algorithm atop the framework introduced in [10].

2.3 Definition of Approximate ODs

We refer to canonical AOCs and approximate OFDs (AOFDs) collectively as AODs. We define AODs based on the fewest tuples that must be removed from a table for an OD to hold. This definition was used for AODs in [10]; their AOC validation step (for the only currently existing AOD discovery algorithm) has a quadratic runtime. For AOFDs, validation takes linear time [3].

Definition 2.14. Given a table **r** and an OD φ, a set of tuples **s** is a *removal set* w.r.t. φ *iff* $\mathbf{r} \setminus \mathbf{s} \models \varphi$. Let $|\mathbf{r}|$ denote the *cardinality* of **r**, the number of tuples in **r**. A removal set **s** is a *minimal* removal set *iff* it has the smallest cardinality over all removal sets; i.e., $|\mathbf{s}| = \min(\{|\mathbf{s}| \mid \mathbf{s} \subseteq \mathbf{r},\ \mathbf{r} \setminus \mathbf{s} \models \varphi\})$. Given **s**, the *approximation factor* $e(\varphi)$ is defined as $|\mathbf{s}|/|\mathbf{r}|$.

Example 2.15. Consider Table 1 and the OC of sal ∼ tax. Here, $\mathbf{s} = \{t_1, t_2, t_4, t_6\}$ and $e(\text{sal} \sim \text{tax}) = 4/9 \approx 0.44$, as $\mathbf{r} \setminus \mathbf{s} = \{t_3, t_5, t_7, t_8, t_9\}$ does not contain any swaps with respect to sal ∼ tax and no smaller set $\mathbf{s}'$ exists such that $\mathbf{r} \setminus \mathbf{s}' \models \text{sal} \sim \text{tax}$.

Given a table **r** and an approximation threshold ϵ, $0 \leq \epsilon \leq 1$, the problem of discovering AODs involves finding all minimal (non-redundant that follow from others) ODs φ such that $e(\varphi) \leq \epsilon$. In this work, we focus on the problem of validating AODs; i.e., verifying whether the approximation factor of a given AOD is less than or equal to a provided threshold. We present an optimal algorithm for doing so and incorporate it into an existing OD discovery framework.

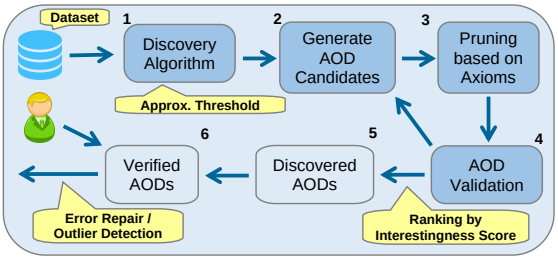

Figure 1: System framework.

As discussed in Sec. 2.2, OCs and OFDs constitute canonical ODs; i.e., OD ≡ OC + OFD. There already exists an efficient linear-time algorithm for validating AOFDs, as described in [3]. In this work, we present an optimal validation algorithm for AOCs. Note that when discovering *approximate* OCs and OFDs given an approximation threshold ϵ, AOD ≡ AOC + AOFD does not necessarily hold. If AOC $\mathcal{X}$: A ∼ B and AOFD $\mathcal{X}$A: [] ↦ B hold with approximation factors $e_1, e_2 \leq \epsilon$, respectively, it is not guaranteed for the corresponding AOD of $\mathcal{X}$: A ↦ B to also hold with respect to ϵ. As to be discussed in Sec. 3.3, however, our validation algorithm can easily be extended to validate list-based approximate ODs as well.

3 DISCOVERING APPROXIMATE OD'S

In Sec. 3.1, we describe our framework to discover set-based canonical AODs. In Sec. 3.2, we describe the iterative validation algorithm proposed in [10, 11], analyze its runtime, and provide an example of it failing to find a minimal removal set and thus overestimating the number of tuples that must be removed. In Sec. 3.3, we present our efficient validation algorithm, based on the longest increasing subsequence (LIS) problem, analyze its runtime, and prove its minimality and optimality.

3.1 Discovery Framework

The algorithm starts the search from singleton sets of attributes and proceeds to traverse the set-based attribute lattice in a level-wise manner [10, 11]. At each level, and when processing the attribute set $\mathcal{X}$, the algorithm verifies AOCs of the form $\mathcal{X} \setminus \{A, B\}$: A ∼ B for which A, B ∈ $\mathcal{X}$ and A ≠ B, and AOFDs of the form $\mathcal{X} \setminus \{A\}$: [] ↦ A for which A ∈ $\mathcal{X}$.

Figure 1 illustrates the framework. Candidate AODs are generated based on the attribute sets at the current level of the lattice. Using the dependencies found in previous levels of the lattice, these candidates are then pruned by axioms to avoid redundant dependencies that follow from already discovered ones [10]. Our algorithm validates whether each candidate dependency holds approximately, given the approximation threshold as input. Valid AODs are then scored and ranked, using the measure of interestingness introduced in [10]. These discovered AODs can then be manually verified by domain experts, to be then used for tasks such as error repair or outlier detection, which is an easier task than manual specification.

3.2 The Iterative Validation Algorithm

We first discuss the algorithm described in [10, 11] to validate an AOC given a threshold ϵ. To validate an AOC, the authors compute a removal set **s** by iteratively removing a tuple with the largest number of swaps, which does not guarantee to produce the minimal removal set. This is repeated until either the OC

Algorithm 1 Approx-OC-iterative

Input: Table $\mathbf{r}$, OC $\mathcal{X}$: A ∼ B, and approximation threshold ϵ.
Output: Approximation factor e and removal set $\mathbf{s}$, or "INVALID"

```
1:  s = {}
2:  for all E ∈ Π_X do
3:      t = order E by [A ASC, B ASC]
4:      t_swapCnt = countInversions(t_B)
5:      order t by swapCnt ASC
6:      while t is not empty do
7:          t = t.dropLast()
8:          if t_swapCnt == 0 then break
9:          for all s ∈ t do
10:             if s_A,B and t_A,B are swapped then s_swapCnt −= 1
11:         end for
12:         order t by swapCnt ASC
13:         add t to s
14:         if |s| > ϵ|r| then return "INVALID"
15:     end while
16: end for
17: return |s|/|r|, s
```

holds or the number of removed tuples crosses the threshold $\epsilon|\mathbf{r}|$, in which case the AOC candidate is considered invalid. Note that after removing each tuple, the number of swaps for the remaining tuples must be updated.

Algorithm 1 validates a candidate using the iterative approach. The steps in Lines 3 to 15 are repeated on tuples within each equivalence class with respect to the context. Line 4 uses a variant of merge sort to count the number of *inversions* in the projection of sorted tuples over B, which is equivalent to the number of swaps for each tuple. Line 7 removes a tuple with the most swaps and Lines 9 to 11 update the number of swaps for the remaining tuples. Line 14 exits if the approximation threshold is crossed.

Example 3.1. Consider Table 1 and the OC sal ∼ tax. Tuple t_7 has swaps with tuples t_1, t_2, t_4, and t_6, which is more than any tuple in the table, and is thus removed. In following steps, tuples t_5, t_3, t_6, and t_4 are removed. Therefore, $\mathbf{s} = \{t_3, t_4, t_5, t_6, t_7\}$ is reported as a removal set for this AOC, and the approximation factor is computed as $5/9 \approx 0.56$. This is larger than the actual approximation factor for this AOC; i.e., 0.44.

Let m denote the number of tuples in an equivalence class. Lines 3 to 5 have runtime $O(m \log m)$. Lines 7 to 14 inside the loop take $O(m)$ time. Note that since the value of swapCnt for each tuple is bounded by m, sorting the tuples in Line 12 (as well as Line 5) can be done in $O(m)$ time using counting sort. In the worst case, this loop is repeated ϵn times, where ϵ and n denote the approximation threshold and the number of tuples in the table, respectively. Therefore, in the worst case, where $m = n$, the runtime of this algorithm is $O(n \log n + \epsilon n^2)$.

3.3 Our Optimal Validation Algorithm

We now present Algorithm 2 based on the *longest increasing subsequence* (LIS) problem to validate an AOC candidate. Lines 3 to 5 are repeated for the tuples in each equivalence class with respect to the context. Line 3 orders the tuples by [A, B] in ascending order. Next, Line 4 finds a longest non-decreasing subsequence (LNDS) of the projection of tuples over B. (As OCs are symmetric, we can also sort by [B, A] and find a LNDS of projections over A.) Line 5 adds the tuples that are *not* in the LNDS to the removal set. Finally, Line 7 checks whether the OC holds approximately with respect to the threshold, and returns the appropriate output.

Algorithm 2 Approx-OC-optimal

Input: Table $\mathbf{r}$, OC $\mathcal{X}$: A ∼ B, and approximation threshold ϵ.
Output: Approximation factor e and removal set $\mathbf{s}$, or "INVALID"

```
1: s = {}
2: for all E ∈ Π_X do
3:     t = order E by [A ASC, B ASC]
4:     L = computeLNDS(t_B)
5:     s = s ∪ (t_B \ L)
6: end for
7: if |s| ≤ ϵ|r| then return |s|/|r|, s else return "INVALID"
```

Example 3.2. Consider Table 1 and the OD sal ∼ tax. After ordering the tuples according to sal and breaking ties by tax, the projection of the tuples over tax is the list $[2K, 2.5K, 0.3K, 12K, 1.5K, 16.5K, 1.8K, 7.2K, 16K]$. The LNDS of this list is $[0.3K, 1.5K, 1.8K, 7.2K, 16K]$ and thus, the removal set is $\mathbf{s} = \{t_1, t_2, t_4, t_6\}$. Thus, the approximation factor is $4/9 \approx 0.44$.

Again, let m denote the number of tuples in an equivalence class. Sorting the tuples in each equivalence class takes $O(m \log m)$ time (Line 3). To compute a LNDS of a list with length m, a dynamic programming algorithm from [2] with small modifications and with runtime $O(m \log m)$ is employed (Line 4). In Line 5, since L is a subsequence of $\mathbf{t}_B$, $\mathbf{t}_B \setminus L$ can be computed in $O(m)$ time by traversing both lists once. Therefore, the worst case runtime of this algorithm, which occurs when $m = n$, is $O(n \log n)$.

We now prove minimality and optimality of our algorithm. [1]

Theorem 3.3. *The set* $\mathbf{s}$ *generated using Algorithm 2 is a minimal removal set with respect to the given* AOC.

Theorem 3.4. *Algorithm 2 has the optimal runtime for validating an* AOC *candidate.*

Our validation algorithm easily extends to AODs of the form $\mathcal{X}$: A ↦ B. We again use Algorithm 2, but in Line 3, tuples are ordered according to the ascending order over A, but ties are broken according to the *descending* order over B. Intuitively, this forces the solution to the LNDS problem in Algorithm 2 to remove all *splits* in the table (removal of *swaps* is already ensured similar to Algorithm 2 for AOCs).[2]

4 EXPERIMENTS

We implemented our approximate OC validation algorithm on top of a Java implementation of the set-based OD discovery framework from [10]. We implemented our new LIS-based algorithm as well as the iterative algorithm using the same technologies to ensure that the improvements in runtime are not due to implementation differences. Unless mentioned otherwise, we set the approximation threshold to 10% and use ten attributes. We run our experiments on a machine with Xeon CPU 2.4GHz with 64GB RAM, and use datasets from the Bureau of Transportation Statistics and the North Carolina State Board of Elections:

(1) **flight** contains information such as date, origin, destination, and airline about flights in the United States and has 1M tuples and 35 attributes (https://www.bts.gov).
(2) **ncvoter** contains information such as registration number, age, and address about voters in North Carolina and has 5M tuples and 30 attributes (https://www.ncsbe.gov).

[1] Due to space limits, proofs of theorems can be found in the technical report [5].
[2] This idea can be extended to list-based AODs of the form $\mathbf{X} \mapsto \mathbf{Y}$, by ordering tuples in ascending order of $\mathbf{X}$ and breaking ties using the descending order over $\mathbf{Y}$.

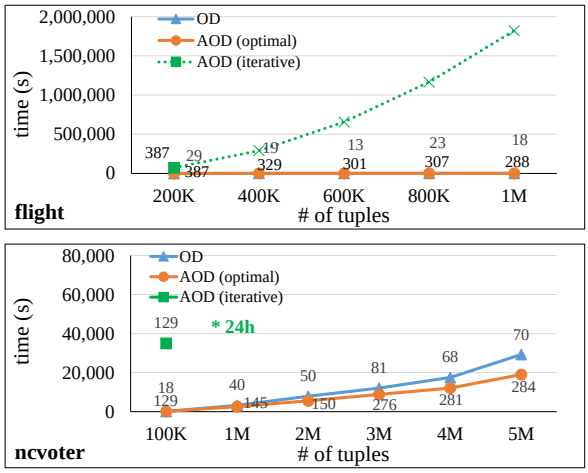

Figure 2: Scalability in |r|.

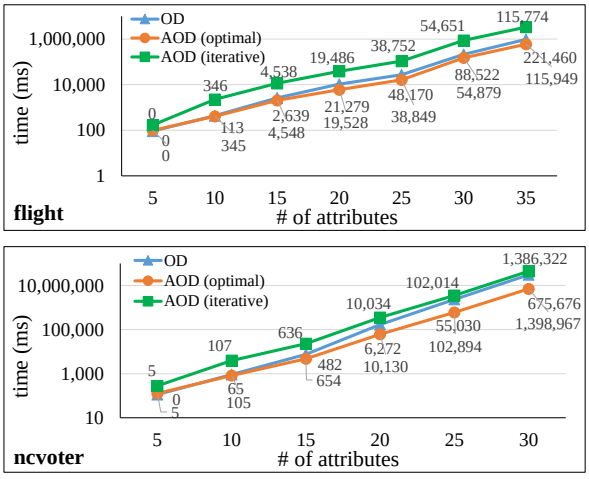

Figure 3: Scalability in |R|.

4.1 Scalability

Exp-1: Scalability in |r|. We measure the runtime (in seconds) of the AOD discovery framework that uses our validation algorithm by varying the number of tuples in our datasets, as reported in Figure 2. For now, ignore the curves labeled "OD" and "AOD (iterative)", as well as the numbers next to the datapoints. The AOD discovery framework implemented using our optimal algorithm scales up to millions of tuples.

Exp-2: Scalability in |R|. Next, we measure the runtime of the discovery framework in milliseconds, by varying the number of attributes in our datasets, as illustrated in Figure 3. We use 1K tuples of our datasets (to allow experiments with a large number of attributes in reasonable time) and vary the number of attributes in multiples of five. In this experiment, the runtime has an exponential growth (the Y-axis in Figure 3 is in log scale). This is expected since the number of ODs increases exponentially with the number of attributes.

4.2 Comparison with the Iterative Algorithm

Exp-3: Runtime comparison with the iterative algorithm. As discussed in Section 3, our AOC validation algorithm has time complexity $O(n \log n)$, while the iterative algorithm proposed in [10, 11] has time complexity $O(n \log n + \epsilon n^2)$. Figures 2, 3, and 4 illustrate the running times of the AOD discovery framework when using these two validation algorithms.

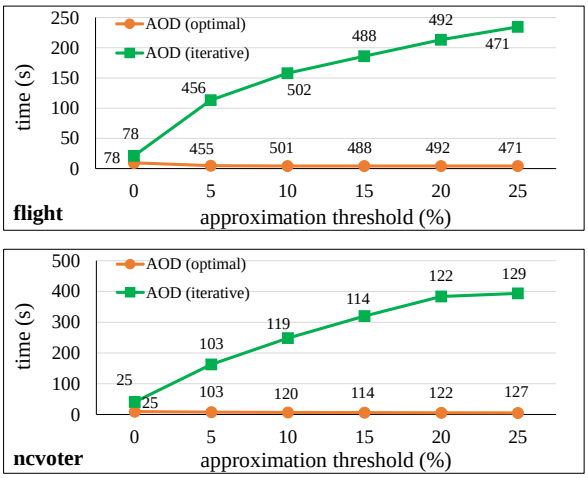

Figure 4: The effect of the approximation threshold.

As shown in Figure 2, while when using our algorithm, the framework can discover AOCs in datasets with up to millions of tuples, when using the iterative algorithm, it does not terminate within 24 hours on 400K and 1M tuples of the flight and ncvoter datasets, respectively (the running times for the flight dataset have been projected with dashed lines for better comparison). In cases where the framework equipped with the iterative algorithm terminates within the time limit, it is orders of magnitude slower. In Figure 3, while the differences are not as pronounced (as the number of tuples is too small), using our validation algorithm still makes the framework almost an order of magnitude faster.

We next experiment with the approximation threshold, by using 10K tuples from our datasets and setting the approximation threshold to 0, 5, 10, 15, 20, and 25 percent. As Figure 4 illustrates, while a larger approximation threshold does not increase the runtime of our algorithm, (the runtime decreases in some cases due to better pruning opportunities), it increases the runtime of the iterative approach at an almost linear rate. This aligns with the time complexity of these algorithms, as analyzed in Section 3.

As mentioned in Section 1, validating AOCs becomes the bottleneck of the AOD discovery framework when using the iterative algorithm. This is verified in our experiments, as up to 99.6% of the total runtime is spent on validation. Using our LIS-based validation algorithm, we reduce the time spent on validating AOCs by up to 99.8%, which results in the orders-of-magnitude improvement in runtime discussed before.

Exp-4: Removal sets and validating AOCs using the iterative algorithm. While our validation algorithm guarantees finding a minimal removal set for a given OC (as is proved in Section 3.3), the iterative algorithm may *overestimate* the size of a minimal removal set. This results in removal sets which are on average around 1% larger than the true minimal removal set.

Overestimating the approximation factor may result in missing valid AOCs if the true approximation factor is close to the input threshold. In Fig. 2, 3, and 4, the numbers inside the plots indicate the number of OCs or AOCs found by an algorithm. We have not listed the number of AOFDs since this work focuses on discovering AOCs. (Wherever the plots for our algorithm and the algorithm for exact ODs overlap, the numbers on the bottom correspond to our approach.) The iterative approach misses up to 2% of the valid AOCs found using our optimal approach.

Missing these AOCs could have potentially severe consequences. For instance, in the flight dataset, the AOC of arrivalDelay ~ lateAircraftDelay holds with an approximation factor of 9.5%.

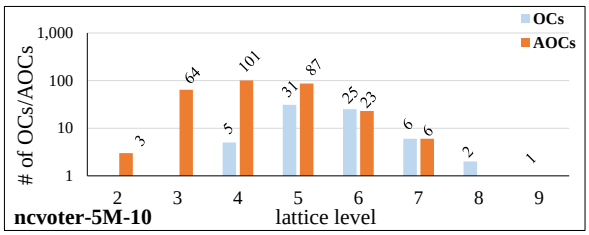

Figure 5: Number of discovered OCs/AOCs in each level.

This AOC points out that generally, delays in arrival are due to the aircraft and not other causes; e.g., security or weather delays. However, the iterative algorithm overestimates the approximation factor as 10.5%. This results in the framework missing this valid AOC when using an approximation threshold of 10%. Note that missing some AOCs results in different pruning opportunities, and, as a result, the set of discovered AOCs, which explains why the iterative algorithm discovers more AOCs in some cases.

Furthermore, as has been discussed for Exp-3, the running time of the iterative algorithm on larger datasets is prohibitively long. On such datasets, using the iterative algorithm results in missing *all* valid AOCs. For instance, in the ncvoter dataset with 5M tuples and with the approximation threshold set to 20%, the AOC of municipalityAbbrv ∼ municipalityDesc is discovered, which points to exceptions in creating abbreviations for municipalities; e.g., "Raleigh" is abbreviated as "RAL", while "Charlotte" is abbreviated as "CLT". However, this AOC does not hold in our 100K sample of tuples when using this threshold. Therefore, this dependency would have been missed by using the iterative validation algorithm, as it exceeds the time limit on the full dataset.

4.3 Comparison with Exact OD Discovery

Exp-5: Lattice level of AOCs and runtime improvements. AOCs tend to reside in lower levels of the lattice (with smaller contexts). In our scalability experiments in the number of tuples (Exp-1), the AOCs are on average 1.2 levels lower on the lattice. Similarly, in experiments in the number of attributes (Exp-2), the AOCs are on average 0.5 levels lower on the lattice. Figure 5 shows the number of OCs or AOCs found at each level of the lattice, when using 5M tuples and 10 attributes of the ncvoter dataset. On this dataset, the average lattice level of the discovered dependencies drops from 5.6 to 4.3 when using our approximate algorithm. As discussed in [10, 11], dependencies found in lower levels of the lattice are likely to be more interesting.

Furthermore, as discussed in Section 3.1, our discovery framework first validates candidates on lower levels of the lattice, and then applies pruning rules to generate the candidates on higher levels of the lattice (step 3 in Figure 1). Therefore, by finding AOCs in lower levels, the algorithm can use pruning rules more effectively earlier in the discovery process, resulting in pruning some candidates on higher levels of the lattice and validating fewer candidates in total. The effects of such pruning opportunities are not noticed when using the iterative validation algorithm, due to its prohibitively long running time. However, we optimally reduce the runtime of the validation step, resulting in runtime improvements for the discovery framework.

Figures 2 and 3 show the running times of the algorithms for discovering exact and approximate ODs. Even though validation of AOCs has a worse runtime compared to exact OCs, i.e., $O(n \log n)$, as opposed to $O(n)$, due to the extra pruning opportunities described above, the total runtime of the discovery framework for AODs can even be lower than the discovery framework for exact ODs; i.e., up to 34% and 76% faster in experiments in the number of tuples and attributes, respectively. The pronounced effect in the experiments in the number of attributes is due to having a smaller number of tuples.

Exp-6: Discovered AOCs compared to OCs. The exact algorithm fails to discover meaningful OCs in presence of anomalies, or even if a single value is erroneous. However, valid AOCs may hold in such instances. Other than the AOCs discussed in Exp-4, in the flight dataset, we discovered the AOC city ∼ airportName with a 27% approximation factor, which indicates that the names of airports usually begin with the name of the corresponding cities. Furthermore, the AOC streetAddress ∼ mailAddress holds in the ncvoter dataset with an approximation factor of 18%. These AOCs can point to anomalies and data quality issues, e.g., wrong address formats, misaligned mailing and residence addresses, and non-standard / erroneous airport names.

As shown in Figures 2 and 3, by discovering AOCs, we can find more dependencies in the data. Even if there are fewer AOCs than OCs (e.g., the flight dataset in Exp-2), the discovered dependencies are on lower levels of the lattice, as shown in Exp-5, which makes them more interesting [10, 11]. If the number of discovered dependencies is too large, the interestingness measure proposed in [11] can be used to rank the AOCs. In fact, the example AOCs that we have identified in Exp-4 and in this experiment, were all ranked as the most interesting AOCs.

5 CONCLUSIONS

We proposed a new validation algorithm for approximate ODs and proved its minimality and runtime optimality. We then implemented our approach in an existing canonical OD discovery framework and demonstrated significant gains compared to existing frameworks for discovering exact and approximate ODs. In future work, we will study new approaches for discovering approximate ODs, such as hybrid sampling, as done in [7] for FDs. We will also extend our approximate OD discovery framework to distributed settings, similar to the work in [9].

REFERENCES

[1] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis. 2019. Discovering order dependencies through order compatibility. *EDBT* (2019), 409–420.
[2] M. Fredman. 1975. On computing the length of longest increasing subsequences. *Discrete Mathematics* 11, 1 (1975), 29 – 35.
[3] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Computer J.* 42 (1999), 100–111.
[4] Yifeng Jin, L. Zhu, and Zijing Tan. 2020. Efficient Bidirectional Order Dependency Discovery. *ICDE* (2020), 61–72.
[5] Reza Karegar, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. 2021. Efficient Discovery of Approximate Order Dependencies. *Technical report, 7 pages, http://arxiv.org/abs/2101.02174* (2021).
[6] P. Langer and F. Naumann. 2016. Efficient Order Dependency Detection. *The VLDB Journal* 25, 2 (2016), 223–241.
[7] T. Papenbrock and F. Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. *SIGMOD* (2016), 821–833.
[8] Y. Qiu, Tan, K. Z., Yang, X. Yang, and N. Guo. 2018. Repairing data violations with order dependencies. *DASFAA* (2018), 283–300.
[9] H. Saxena, L. Golab, and I. Ilyas. 2019. Distributed Implementations of Dependency Discovery Algorithms. *PVLDB* 12, 11 (2019), 1624–1636.
[10] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2017. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB* 10, 7 (2017), 721–732.
[11] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2018. Effective and Complete Discovery of Bidirectional Order Dependencies via Set-Based Axioms. *The VLDB Journal* 27, 4 (2018), 573–591.
[12] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2020. Erratum for discovering order dependencies through order compatibility. *EDBT* (2020), 659–663.
[13] J. Szlichta, P. Godfrey, and J. Gryz. 2012. Fundamentals of Order Dependencies. *PVLDB* 5, 11 (2012), 1220–1231.

CONTENTS

Efficient Discovery of Approximate Order Dependencies

Reza Karegar
University of Waterloo, CA
mkaregar@uwaterloo.ca

Parke Godfrey
York University, CA
godfrey@yorku.ca

Lukasz Golab
University of Waterloo, CA
lgolab@uwaterloo.ca

Mehdi Kargar
Ryerson University, CA
kargar@ryerson.ca

Divesh Srivastava
AT&T Chief Data Office, US
divesh@att.com

Jaroslaw Szlichta
Ontario Tech Univ, CA
jarek@ontariotechu.ca

ABSTRACT

Order dependencies (ODs) capture relationships between ordered domains of attributes. Approximate ODs (AODs) capture such relationships even when there exist exceptions in the data. During automated discovery of dependencies, *validation* is the process of verifying whether a dependency holds. We present an algorithm for validating AODs with significantly improved runtime performance over existing methods, and prove that it is *minimal* and has *optimal* runtime. By replacing the validation step in a recent algorithm for AOD discovery with ours, we achieve orders-of-magnitude improvements in performance.

1 INTRODUCTION

1.1 Motivation

Functional dependencies (FDs) specify that the values of given attributes *functionally determine* the value of a target attribute. *Order dependencies* extend FDs to state that, additionally, the *order* of tuples with respect to the values from the domains of given attributes determines the *order* of the values from the domain of the target attribute. Table 1 shows a dataset with employee salaries. In this table, the OD that sal *orders* taxGrp holds. If one sorts the table by sal, it is sorted by taxGrp as well.

An OD implies the corresponding FD; e.g., that sal orders taxGrp implies that sal *functionally determines* taxGrp. *Order compatibility* (OC) captures the *co-ordering* aspect of an OD *without* the corresponding FD. Two lists of attributes are *order compatible* if there exists an arrangement for the tuples in the table in which the tuples are sorted according to both. Any OD can thus be equivalently represented by a pair of an OC and an FD [13]. In Table 1, that taxGrp is *order compatible* with sal holds. Note that taxGrp does *not* order sal, as an FD does not hold.

There has been recent work to automate the *discovery* of ODs from data [1, 4, 6, 10, 11]. In practice, however, constraints rarely hold *perfectly* in the data. Real data are dirty, containing wrong and inconsistent values that may violate semantically valid dependencies. This motivates the need for discovering *approximate* ODs (AODs), ODs that hold in the data but with *exceptions*. Discovered ODs deemed semantically valid can be used for data cleaning, to detect erroneous tuples, where measures are then taken to repair the errors [8]. AODs are useful even when the data are not dirty, as there can be exceptions to general rules. AODs help avoid overfitting by discovering more general dependencies.

In Table 1, tax is a fixed percentage of salary in each tax group; i.e., one, three, or eight percent. However, perc includes a concatenated zero in some rows due to data entry errors (e.g., 10% instead of 1% in t_1). Because of this, the OC that salary is order compatible with tax does *not* hold, even though this OC is intended. Similarly, the FD that pos, exp functionally determines sal does *not* hold, due to the exception of tuples t_6 and t_7, two employees with the same position and years of experience but having different salaries. With approximate ODs, we can still discover such concise and meaningful rules in these instances.

Table 1: Employee salaries

#	pos	exp	sal	taxGrp	perc	tax	bonus
t_1	sec	1	20K	A	10%	2K	1K
t_2	sec	3	25K	A	10%	2.5K	1K
t_3	dev	1	30K	A	1%	0.3K	3K
t_4	sec	5	40K	B	30%	12K	2K
t_5	dev	3	50K	B	3%	1.5K	4K
t_6	dev	5	55K	B	30%	16.5K	4K
t_7	dev	5	60K	B	3%	1.8K	4K
t_8	dev	-1	90K	C	8%	7.2K	7K
t_9	dir	8	200K	C	8%	16K	10K

Approximate ODs were introduced in [10]. Their definition of AODs, as is ours herein, is based on the concept of "tuple removal." Given a table and an OD, a *removal set* is a set of tuples which, if removed from the table, results in the OD holding. A *minimal* removal set is one with the smallest cardinality. An *approximation factor* can be defined with respect to a table and an OD, as the ratio of the size of a minimal removal set over the size of the table. For instance, for Table 1 and the OC that pos, exp is order compatible with pos, sal, the minimal removal set and the approximation factor are $\{t_8\}$ and $1/9 \approx 0.11$, respectively.

Given a table $\mathbf{r}$ and an approximation threshold $0 \leq \epsilon \leq 1$, the *discovery problem* for AODs is to find the complete set of minimal *valid* AODs in $\mathbf{r}$ w.r.t. ϵ. Exact ODs are a special case of AODs with an approximation factor of zero. Given a table $\mathbf{r}$, an OD φ, and a threshold ϵ, the problem of *validating* the candidate OD as an AOD involves verifying whether the approximation factor of φ, denoted by $e(\varphi)$, is less than or equal to ϵ.

1.2 Contributions

The extension for AOD discovery in [10, 11], however, is impractical due to its performance. While the approximate FD component can be validated in linear time [3, 10], to validate the approximate OC (AOC) component in the search, they iteratively remove the tuple—or one of the tuples, in the case of a tie—that causes the largest number of violations. This has two weaknesses: the runtime is quadratic in the number of tuples, and it is not guaranteed to find a minimal removal set.

That it is quadratic makes it prohibitively expensive to run on larger datasets. (The validation step for a candidate exact OD has a linear runtime in the number of tuples.) So while the OD discovery algorithm in [10, 11] is shown to scale to datasets with millions of tuples, it is infeasible to run their adapted AOC discovery algorithm over even moderately sized datasets. During

benchmarking, we found in some discovery runs that more than 99% of the running time is spent on validating AOC candidates.

That it deliberately does not guarantee finding a minimal removal set means that the algorithm may overestimate the approximation factor of an AOC candidate. Thus, *true* AOCs with respect to the approximation threshold can be eliminated (while the exact OD discovery algorithm is complete).

In this paper, we resolve this major bottleneck in AOD discovery via an algorithm with optimal runtime and guaranteed minimal removal set for validating AOC candidates. This brings performance of AOD discovery on par with that of OD discovery, while making the AOD discovery complete.

The paper is structured as follows, with the following key contributions. In Sec. 2, we provide background and discuss related work. In Sec. 3.1, we illustrate the established OD and AOD discovery framework—which we then adapt herein—and, in Sec. 3.2, the *iterative validation algorithm* [10, 11] it employs. In Sec. 3.3, we contribute a minimal and optimal validation algorithm based on longest increasing subsequences that decreases the runtime from quadratic to log-linear. In Sec. 4, we present our experimental results, with the following contributions. We demonstrate that AOD discovery using our validation algorithm scales to datasets with millions of tuples and tens of attributes (Exp-1 and Exp-2). We compare our adapted AOC discovery against the previous approach and demonstrate that ours is orders of magnitude faster (Exp-3). As discovering AODs enables the application of pruning rules earlier than for discovering ODs, AOD discovery can be just as efficient, if not more so. Our AOD discovery algorithm gains up to 76% improvement in runtime compared against the (exact) OD discovery algorithm (Exp-5). Given our AOD discovery algorithm is complete, we discover more AODs, and *semantically more general* AODs (thus, of higher quality). We show that we find more AODs, both due to our better scalability and the minimality of our removal sets (Exp-4 and Exp-6). Finally, in Section 5, we conclude with suggestions for future work.

2 PRELIMINARIES AND RELATED WORK

2.1 Definitions and Notation

$\mathbf{R}$ denotes a relational schema, $\mathbf{r}$ represents a table instance, and s and t denote tuples. A and B denote individual attributes and $\mathcal{X}$ and $\mathcal{Y}$ *sets* of attributes. *Lists* of attributes are presented using $\mathbf{X}$ and $\mathbf{Y}$; [] denotes the empty list and $[A \mid \mathbf{T}]$ denotes a list with *head* attribute A and *tail* list $\mathbf{T}$. Tuples t_A and $t_{\mathcal{X}}$ denote the projections of tuple t on A and $\mathcal{X}$, respectively. Wherever a set is expected but a list appears, the list is cast to a set; e.g., $t_{\mathbf{X}}$ is equivalent to $t_{\mathcal{X}}$. $\mathbf{X}'$ represents an arbitrary permutation of the values of a list $\mathbf{X}$ or set $\mathcal{X}$.

Definition 2.1. (nested order) Let $\mathbf{X}$ be a list of attributes where $\mathcal{X} \in \mathbf{R}$. Given two tuples, s and t, $s \preceq_{\mathbf{X}} t$ *iff*

- $\mathbf{X} = [\,]$; *or*
- $\mathbf{X} = [A \mid \mathbf{T}]$ and $s_A < t_A$; or
- $\mathbf{X} = [A \mid \mathbf{T}]$, $s_A = t_A$, and $s \preceq_{\mathbf{T}} t$.

Let $s \prec_{\mathbf{X}} t$ *iff* $s \preceq_{\mathbf{X}} t$ but $t \not\preceq_{\mathbf{X}} s$.

Next, we define order dependencies [1, 4, 6, 10, 11, 13].

Definition 2.2. (order dependency) Let $\mathbf{X}$ and $\mathbf{Y}$ be lists of attributes where $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R}$. $\mathbf{X} \mapsto \mathbf{Y}$ denotes an *order dependency*, read as $\mathbf{X}$ *orders* $\mathbf{Y}$. Table $\mathbf{r}$ satisfies $\mathbf{X} \mapsto \mathbf{Y}$ ($\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$) iff, for all $s, t \in \mathbf{r}$, $s \preceq_{\mathbf{X}} t$ implies $s \preceq_{\mathbf{Y}} t$. $\mathbf{X}$ and $\mathbf{Y}$ are *order equivalent* (denoted as $\mathbf{X} \leftrightarrow \mathbf{Y}$), iff $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$.

Definition 2.3. (order compatibility) Let $\mathbf{X}$ and $\mathbf{Y}$ be lists of attributes where $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R}$. $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible*, denoted as $\mathbf{X} \sim \mathbf{Y}$, iff $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

The order dependency $\mathbf{X} \mapsto \mathbf{Y}$ means that $\mathbf{Y}$'s values are monotonically non-decreasing with respect to $\mathbf{X}$'s values. Therefore, if one orders the tuples by $\mathbf{X}$, they are also ordered by $\mathbf{Y}$. The order compatibility (OC) $\mathbf{X} \sim \mathbf{Y}$ means that there exists a total order of the tuples in which they are ordered according to both $\mathbf{X}$ and $\mathbf{Y}$.

Example 2.4. In Table 1, the OD sal $\mapsto$ taxGrp holds. The OC taxGrp $\sim$ sal holds, even though the OD taxGrp $\mapsto$ sal does not.

ODs have a strong correspondence with OCs and FDs. An OD $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \sim \mathbf{Y}$ (OC) and $\mathcal{X} \rightarrow \mathcal{Y}$ (FD) hold. This gives two sources of violations for ODs: *swap*s and *split*s [13].

Definition 2.5. (swap) A *swap* with respect to OC $\mathbf{X} \sim \mathbf{Y}$ is a pair of tuples s and t such that $s \prec_{\mathbf{X}} t$ but $t \prec_{\mathbf{Y}} s$.

Definition 2.6. (split) A *split* with respect to FD $\mathcal{X} \rightarrow \mathcal{Y}$ is a pair of tuples s and t such that $s_{\mathcal{X}} = t_{\mathcal{X}}$ but $s_{\mathcal{Y}} \neq t_{\mathcal{Y}}$

Example 2.7. In Table 1, given the OD pos, exp $\mapsto$ pos, sal, tuples t_7 and t_8 constitute a swap (the OC pos, exp $\sim$ pos, sal), and tuples t_6 and t_7 constitute a split (the FD pos, exp $\rightarrow$ pos, sal).

Definition 2.8. tuples s and t are *equivalent* w.r.t. set of attributes $\mathcal{X}$ *iff* $s_{\mathcal{X}} = t_{\mathcal{X}}$. An attribute set $\mathcal{X}$ partitions tuples into *equivalence classes* [3]. The *equivalence class* of tuple $t \in \mathbf{r}$ w.r.t. $\mathcal{X}$ is denoted by $\mathcal{E}(t_{\mathcal{X}})$; i.e., $\mathcal{E}(t_{\mathcal{X}}) = \{s \in \mathbf{r} \mid s_{\mathcal{X}} = t_{\mathcal{X}}\}$. Given a set of attributes $\mathcal{X}$, a *partition* of the table with respect to $\mathcal{X}$ is the set of all equivalence classes; i.e., $\Pi_{\mathcal{X}} = \{\mathcal{E}(t_{\mathcal{X}}) \mid t \in \mathbf{r}\}$.

Example 2.9. In Table 1, $\mathcal{E}(t_{1\{\text{pos}\}}) = \mathcal{E}(t_{2\{\text{pos}\}}) = \mathcal{E}(t_{4\{\text{pos}\}}) = \{t_1, t_2, t_4\}$, and $\Pi_{\text{pos}} = \{\{t_1, t_2, t_4\}, \{t_3, t_5, t_6, t_7, t_8\}, \{t_9\}\}$.

2.2 A Canonical Mapping

A natural representation of ODs relies on lists of attributes, as in the ORDER BY statement in SQL, where the order of attributes in the list matters; e.g., the OD pos, sal $\mapsto$ pos, exp is different than the OD pos, sal $\mapsto$ exp, pos. This is unlike FDs, where the order of attributes does not matter, as with the GROUP BY statement in SQL. Working within this list-based representation, however, has led to discovery frameworks with factorial worst-case runtimes in the number of attributes [6]. Fortunately, lists are *not* inherently necessary to express ODs. In [10, 11], the authors rely on a polynomial mapping of list-based ODs into a logically *equivalent* collection of set-based *canonical* ODs to devise a discovery framework with exponential worst-case runtime in the number of attributes and linear in the number of tuples.

Definition 2.10. (canonical order compatibility) Given a set of attributes $\mathcal{X}$, $\mathbf{X}'A \sim \mathbf{X}'B$ is the OC that states that attributes A and B are *order compatible* within each equivalence class of $\mathcal{X}$. We write this as $\mathcal{X}$: A $\sim$ B in the canonical notation, factoring out the common prefix, and refer to this as a *canonical* OC.

Definition 2.11. (order functional dependency) Given a set of attributes $\mathcal{X}$, the FD that states that an attribute A is *constant* within each equivalence class of $\mathcal{X}$ is equivalent to the list-based OD $\mathbf{X}' \mapsto \mathbf{X}'A$. We write this as $\mathcal{X}$: [] $\mapsto$ A in the canonical notation, and refer to this as an *order functional dependency* (OFD).

Given a canonical OC of $\mathcal{X}$: A $\sim$ B or an OFD of $\mathcal{X}$: [] $\mapsto$ A, the set $\mathcal{X}$ is referred to as the *context* of the respective canonical OC or OFD. Intuitively, the context is the common prefix on the left- and right-side of the corresponding list-based OC or OD.

Canonical OCs and OFDs constitute the canonical ODs; i.e., OD ≡ OC + OFD. The OD of $\mathbf{X}'A \mapsto \mathbf{X}'B$ is logically equivalent to the canonical OC of $\mathcal{X}$: A ∼ B and OFD of $\mathcal{X}$A: [] ↦ B. This is $\mathcal{X}$: A ↦ B written in the canonical form.

Example 2.12. In Table 1, sal and bonus are order compatible w.r.t. the context pos; i.e., {pos}: sal ∼ bonus. In the same table, bonus is constant w.r.t. the context pos, sal; i.e., {pos, sal}: [] ↦ bonus. Therefore, sal orders bonus w.r.t. the context pos; i.e., {pos}: sal ↦ bonus.

This mapping generalizes: an OD $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \mapsto \mathbf{XY}$ *and* $\mathbf{X} \sim \mathbf{Y}$. These can be encoded into an equivalent set of canonical OFDs and OCs as follows. In the context of $\mathcal{X}$, all attributes in $\mathcal{Y}$ must be constants. In the context of all prefixes of **X** and of **Y**, the trailing attributes must be order compatible:

$$\mathbf{R} \models \mathbf{X} \mapsto \mathbf{XY} \textit{ iff } \forall A \in \mathbf{Y}.\ \mathbf{R} \models \mathcal{X}\text{: } [] \mapsto A \text{ and}$$

$$\mathbf{R} \models \mathbf{X} \sim \mathbf{Y} \textit{ iff } \forall i, j.\ \mathbf{R} \models [X_1, \ldots, X_{i-1}][Y_1, \ldots, Y_{j-1}]\text{: } X_i \sim Y_j.$$

Thus, list-based ODs can be polynomially *mapped* to a set of *equivalent* canonical ODs; i.e., canonical OCs and OFDs [10, 11]. In this work, we refer to canonical OCs simply as OCs.

Example 2.13. The OD [A, B] ↦ [C, D] is equivalent to the following canonical ODs: {A, B}: [] ↦ C, {A, B}: [] ↦ D, {}: A ∼ C, {A}: B ∼ C, {C}: A ∼ D, and {A, C}: B ∼ D.

While various algorithms have been proposed for discovering ODs, most are not *complete*. The algorithm described in [6] relies on the list-based definition and employs aggressive pruning rules to compensate for its factorial time complexity, but which make it deliberately incomplete. The authors in [4] claim completeness but their algorithm misses ODs in which the same attributes are repeated on the left- and right-hand side. A similar completeness claim has been made in [1], which was shown to be incorrect in [12]. The set-based OD discovery algorithm proposed in [10] does offer a sound and complete discovery of ODs. Thus, we build our algorithm atop the framework introduced in [10].

2.3 Definition of Approximate ODs

We refer to canonical AOCs and approximate OFDs (AOFDs) collectively as AODs. We define AODs based on the fewest tuples that must be removed from a table for an OD to hold. This definition was used for AODs in [10]; their AOC validation step (for the only currently existing AOD discovery algorithm) has a quadratic runtime. For AOFDs, validation takes linear time [3].

Definition 2.14. Given a table **r** and an OD φ, a set of tuples **s** is a *removal set* w.r.t. φ *iff* $\mathbf{r} \setminus \mathbf{s} \models \varphi$. Let $|\mathbf{r}|$ denote the *cardinality* of **r**, the number of tuples in **r**. A removal set **s** is a *minimal* removal set *iff* it has the smallest cardinality over all removal sets; i.e., $|\mathbf{s}| = \min(\{|\mathbf{s}| \mid \mathbf{s} \subseteq \mathbf{r},\ \mathbf{r} \setminus \mathbf{s} \models \varphi\})$. Given **s**, the *approximation factor* $e(\varphi)$ is defined as $|\mathbf{s}|/|\mathbf{r}|$.

Example 2.15. Consider Table 1 and the OC of sal ∼ tax. Here, $\mathbf{s} = \{t_1, t_2, t_4, t_6\}$ and $e(\text{sal} \sim \text{tax}) = 4/9 \approx 0.44$, as $\mathbf{r} \setminus \mathbf{s} = \{t_3, t_5, t_7, t_8, t_9\}$ does not contain any swaps with respect to sal ∼ tax and no smaller set $\mathbf{s}'$ exists such that $\mathbf{r} \setminus \mathbf{s}' \models \text{sal} \sim \text{tax}$.

Given a table **r** and an approximation threshold ϵ, $0 \leq \epsilon \leq 1$, the problem of discovering AODs involves finding all minimal (non-redundant that follow from others) ODs φ such that $e(\varphi) \leq \epsilon$. In this work, we focus on the problem of validating AODs; i.e., verifying whether the approximation factor of a given AOD is less than or equal to a provided threshold. We present an optimal algorithm for doing so and incorporate it into an existing OD discovery framework.

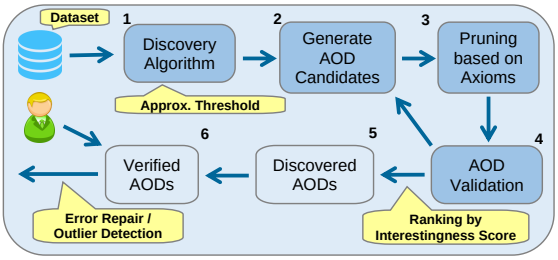

Figure 1: System framework.

As discussed in Sec. 2.2, OCs and OFDs constitute canonical ODs; i.e., OD ≡ OC + OFD. There already exists an efficient linear-time algorithm for validating AOFDs, as described in [3]. In this work, we present an optimal validation algorithm for AOCs. Note that when discovering *approximate* OCs and OFDs given an approximation threshold ϵ, AOD ≡ AOC + AOFD does not necessarily hold. If AOC $\mathcal{X}$: A ∼ B and AOFD $\mathcal{X}$A: [] ↦ B hold with approximation factors $e_1, e_2 \leq \epsilon$, respectively, it is not guaranteed for the corresponding AOD of $\mathcal{X}$: A ↦ B to also hold with respect to ϵ. As to be discussed in Sec. 3.3, however, our validation algorithm can easily be extended to validate list-based approximate ODs as well.

3 DISCOVERING APPROXIMATE OD'S

In Sec. 3.1, we describe our framework to discover set-based canonical AODs. In Sec. 3.2, we describe the iterative validation algorithm proposed in [10, 11], analyze its runtime, and provide an example of it failing to find a minimal removal set and thus overestimating the number of tuples that must be removed. In Sec. 3.3, we present our efficient validation algorithm, based on the longest increasing subsequence (LIS) problem, analyze its runtime, and prove its minimality and optimality.

3.1 Discovery Framework

The algorithm starts the search from singleton sets of attributes and proceeds to traverse the set-based attribute lattice in a level-wise manner [10, 11]. At each level, and when processing the attribute set $\mathcal{X}$, the algorithm verifies AOCs of the form $\mathcal{X} \setminus \{A, B\}$: A ∼ B for which A, B ∈ $\mathcal{X}$ and A ≠ B, and AOFDs of the form $\mathcal{X} \setminus \{A\}$: [] ↦ A for which A ∈ $\mathcal{X}$.

Figure 1 illustrates the framework. Candidate AODs are generated based on the attribute sets at the current level of the lattice. Using the dependencies found in previous levels of the lattice, these candidates are then pruned by axioms to avoid redundant dependencies that follow from already discovered ones [10]. Our algorithm validates whether each candidate dependency holds approximately, given the approximation threshold as input. Valid AODs are then scored and ranked, using the measure of interestingness introduced in [10]. These discovered AODs can then be manually verified by domain experts, to be then used for tasks such as error repair or outlier detection, which is an easier task than manual specification.

3.2 The Iterative Validation Algorithm

We first discuss the algorithm described in [10, 11] to validate an AOC given a threshold ϵ. To validate an AOC, the authors compute a removal set **s** by iteratively removing a tuple with the largest number of swaps, which does not guarantee to produce the minimal removal set. This is repeated until either the OC

Algorithm 1 Approx-OC-iterative

Input: Table $\mathbf{r}$, OC $\mathcal{X}$: A ∼ B, and approximation threshold ϵ.
Output: Approximation factor e and removal set $\mathbf{s}$, or "INVALID"

```
1:  s = {}
2:  for all E ∈ Π_X do
3:      t = order E by [A ASC, B ASC]
4:      t_swapCnt = countInversions(t_B)
5:      order t by swapCnt ASC
6:      while t is not empty do
7:          t = t.dropLast()
8:          if t_swapCnt == 0 then break
9:          for all s ∈ t do
10:             if s_A,B and t_A,B are swapped then s_swapCnt −= 1
11:         end for
12:         order t by swapCnt ASC
13:         add t to s
14:         if |s| > ϵ|r| then return "INVALID"
15:     end while
16: end for
17: return |s|/|r|, s
```

holds or the number of removed tuples crosses the threshold $\epsilon|\mathbf{r}|$, in which case the AOC candidate is considered invalid. Note that after removing each tuple, the number of swaps for the remaining tuples must be updated.

Algorithm 1 validates a candidate using the iterative approach. The steps in Lines 3 to 15 are repeated on tuples within each equivalence class with respect to the context. Line 4 uses a variant of merge sort to count the number of *inversions* in the projection of sorted tuples over B, which is equivalent to the number of swaps for each tuple. Line 7 removes a tuple with the most swaps and Lines 9 to 11 update the number of swaps for the remaining tuples. Line 14 exits if the approximation threshold is crossed.

Example 3.1. Consider Table 1 and the OC sal ∼ tax. Tuple t_7 has swaps with tuples t_1, t_2, t_4, and t_6, which is more than any tuple in the table, and is thus removed. In following steps, tuples t_5, t_3, t_6, and t_4 are removed. Therefore, $\mathbf{s} = \{t_3, t_4, t_5, t_6, t_7\}$ is reported as a removal set for this AOC, and the approximation factor is computed as $5/9 \approx 0.56$. This is larger than the actual approximation factor for this AOC; i.e., 0.44.

Let m denote the number of tuples in an equivalence class. Lines 3 to 5 have runtime $O(m \log m)$. Lines 7 to 14 inside the loop take $O(m)$ time. Note that since the value of swapCnt for each tuple is bounded by m, sorting the tuples in Line 12 (as well as Line 5) can be done in $O(m)$ time using counting sort. In the worst case, this loop is repeated ϵn times, where ϵ and n denote the approximation threshold and the number of tuples in the table, respectively. Therefore, in the worst case, where $m = n$, the runtime of this algorithm is $O(n \log n + \epsilon n^2)$.

3.3 Our Optimal Validation Algorithm

We now present Algorithm 2 based on the *longest increasing subsequence* (LIS) problem to validate an AOC candidate. Lines 3 to 5 are repeated for the tuples in each equivalence class with respect to the context. Line 3 orders the tuples by [A, B] in ascending order. Next, Line 4 finds a longest non-decreasing subsequence (LNDS) of the projection of tuples over B. (As OCs are symmetric, we can also sort by [B, A] and find a LNDS of projections over A.) Line 5 adds the tuples that are *not* in the LNDS to the removal set. Finally, Line 7 checks whether the OC holds approximately with respect to the threshold, and returns the appropriate output.

Algorithm 2 Approx-OC-optimal

Input: Table $\mathbf{r}$, OC $\mathcal{X}$: A ∼ B, and approximation threshold ϵ.
Output: Approximation factor e and removal set $\mathbf{s}$, or "INVALID"

```
1: s = {}
2: for all E ∈ Π_X do
3:     t = order E by [A ASC, B ASC]
4:     L = computeLNDS(t_B)
5:     s = s ∪ (t_B \ L)
6: end for
7: if |s| ≤ ϵ|r| then return |s|/|r|, s else return "INVALID"
```

Example 3.2. Consider Table 1 and the OD sal ∼ tax. After ordering the tuples according to sal and breaking ties by tax, the projection of the tuples over tax is the list $[2K, 2.5K, 0.3K, 12K, 1.5K, 16.5K, 1.8K, 7.2K, 16K]$. The LNDS of this list is $[0.3K, 1.5K, 1.8K, 7.2K, 16K]$ and thus, the removal set is $\mathbf{s} = \{t_1, t_2, t_4, t_6\}$. Thus, the approximation factor is $4/9 \approx 0.44$.

Again, let m denote the number of tuples in an equivalence class. Sorting the tuples in each equivalence class takes $O(m \log m)$ time (Line 3). To compute a LNDS of a list with length m, a dynamic programming algorithm from [2] with small modifications and with runtime $O(m \log m)$ is employed (Line 4). In Line 5, since L is a subsequence of $\mathbf{t}_B$, $\mathbf{t}_B \setminus L$ can be computed in $O(m)$ time by traversing both lists once. Therefore, the worst case runtime of this algorithm, which occurs when $m = n$, is $O(n \log n)$.

We now prove minimality and optimality of our algorithm. [1]

Theorem 3.3. *The set* $\mathbf{s}$ *generated using Algorithm 2 is a minimal removal set with respect to the given* AOC.

Theorem 3.4. *Algorithm 2 has the optimal runtime for validating an* AOC *candidate.*

Our validation algorithm easily extends to AODs of the form $\mathcal{X}$: A ↦ B. We again use Algorithm 2, but in Line 3, tuples are ordered according to the ascending order over A, but ties are broken according to the *descending* order over B. Intuitively, this forces the solution to the LNDS problem in Algorithm 2 to remove all *splits* in the table (removal of *swaps* is already ensured similar to Algorithm 2 for AOCs).[2]

4 EXPERIMENTS

We implemented our approximate OC validation algorithm on top of a Java implementation of the set-based OD discovery framework from [10]. We implemented our new LIS-based algorithm as well as the iterative algorithm using the same technologies to ensure that the improvements in runtime are not due to implementation differences. Unless mentioned otherwise, we set the approximation threshold to 10% and use ten attributes. We run our experiments on a machine with Xeon CPU 2.4GHz with 64GB RAM, and use datasets from the Bureau of Transportation Statistics and the North Carolina State Board of Elections:

(1) **flight** contains information such as date, origin, destination, and airline about flights in the United States and has 1M tuples and 35 attributes (https://www.bts.gov).
(2) **ncvoter** contains information such as registration number, age, and address about voters in North Carolina and has 5M tuples and 30 attributes (https://www.ncsbe.gov).

[1] Due to space limits, proofs of theorems can be found in the technical report [5].
[2] This idea can be extended to list-based AODs of the form $\mathbf{X} \mapsto \mathbf{Y}$, by ordering tuples in ascending order of $\mathbf{X}$ and breaking ties using the descending order over $\mathbf{Y}$.

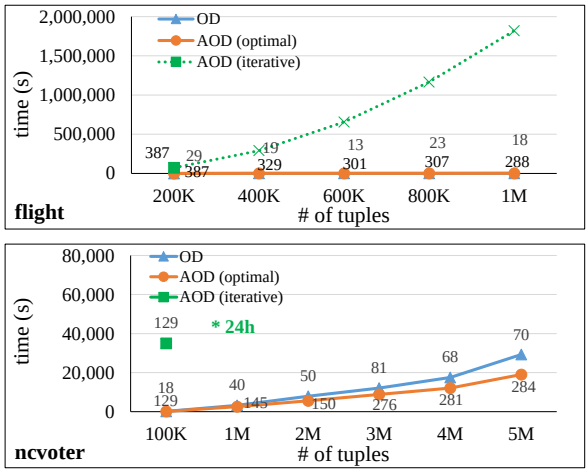

Figure 2: Scalability in |r|.

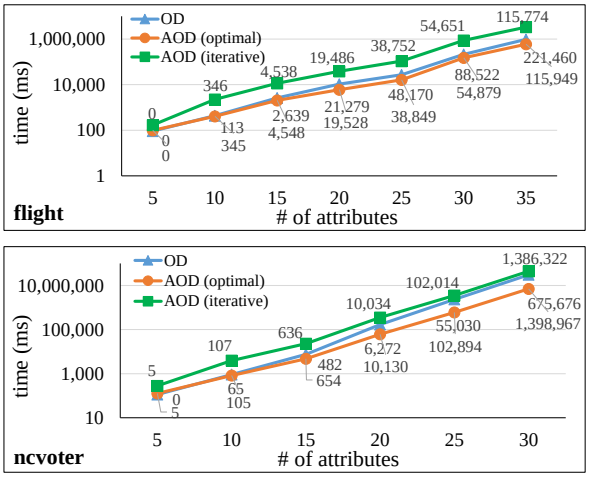

Figure 3: Scalability in |R|.

4.1 Scalability

Exp-1: Scalability in |r|. We measure the runtime (in seconds) of the AOD discovery framework that uses our validation algorithm by varying the number of tuples in our datasets, as reported in Figure 2. For now, ignore the curves labeled "OD" and "AOD (iterative)", as well as the numbers next to the datapoints. The AOD discovery framework implemented using our optimal algorithm scales up to millions of tuples.

Exp-2: Scalability in |R|. Next, we measure the runtime of the discovery framework in milliseconds, by varying the number of attributes in our datasets, as illustrated in Figure 3. We use 1K tuples of our datasets (to allow experiments with a large number of attributes in reasonable time) and vary the number of attributes in multiples of five. In this experiment, the runtime has an exponential growth (the Y-axis in Figure 3 is in log scale). This is expected since the number of ODs increases exponentially with the number of attributes.

4.2 Comparison with the Iterative Algorithm

Exp-3: Runtime comparison with the iterative algorithm. As discussed in Section 3, our AOC validation algorithm has time complexity $O(n \log n)$, while the iterative algorithm proposed in [10, 11] has time complexity $O(n \log n + \epsilon n^2)$. Figures 2, 3, and 4 illustrate the running times of the AOD discovery framework when using these two validation algorithms.

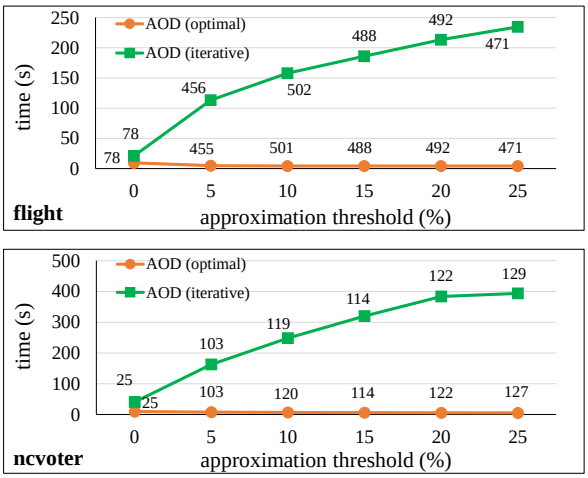

Figure 4: The effect of the approximation threshold.

As shown in Figure 2, while when using our algorithm, the framework can discover AOCs in datasets with up to millions of tuples, when using the iterative algorithm, it does not terminate within 24 hours on 400K and 1M tuples of the flight and ncvoter datasets, respectively (the running times for the flight dataset have been projected with dashed lines for better comparison). In cases where the framework equipped with the iterative algorithm terminates within the time limit, it is orders of magnitude slower. In Figure 3, while the differences are not as pronounced (as the number of tuples is too small), using our validation algorithm still makes the framework almost an order of magnitude faster.

We next experiment with the approximation threshold, by using 10K tuples from our datasets and setting the approximation threshold to 0, 5, 10, 15, 20, and 25 percent. As Figure 4 illustrates, while a larger approximation threshold does not increase the runtime of our algorithm, (the runtime decreases in some cases due to better pruning opportunities), it increases the runtime of the iterative approach at an almost linear rate. This aligns with the time complexity of these algorithms, as analyzed in Section 3.

As mentioned in Section 1, validating AOCs becomes the bottleneck of the AOD discovery framework when using the iterative algorithm. This is verified in our experiments, as up to 99.6% of the total runtime is spent on validation. Using our LIS-based validation algorithm, we reduce the time spent on validating AOCs by up to 99.8%, which results in the orders-of-magnitude improvement in runtime discussed before.

Exp-4: Removal sets and validating AOCs using the iterative algorithm. While our validation algorithm guarantees finding a minimal removal set for a given OC (as is proved in Section 3.3), the iterative algorithm may *overestimate* the size of a minimal removal set. This results in removal sets which are on average around 1% larger than the true minimal removal set.

Overestimating the approximation factor may result in missing valid AOCs if the true approximation factor is close to the input threshold. In Fig. 2, 3, and 4, the numbers inside the plots indicate the number of OCs or AOCs found by an algorithm. We have not listed the number of AOFDs since this work focuses on discovering AOCs. (Wherever the plots for our algorithm and the algorithm for exact ODs overlap, the numbers on the bottom correspond to our approach.) The iterative approach misses up to 2% of the valid AOCs found using our optimal approach.

Missing these AOCs could have potentially severe consequences. For instance, in the flight dataset, the AOC of arrivalDelay ~ lateAircraftDelay holds with an approximation factor of 9.5%.

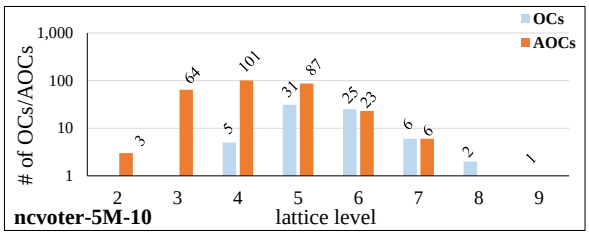

Figure 5: Number of discovered OCs/AOCs in each level.

This AOC points out that generally, delays in arrival are due to the aircraft and not other causes; e.g., security or weather delays. However, the iterative algorithm overestimates the approximation factor as 10.5%. This results in the framework missing this valid AOC when using an approximation threshold of 10%. Note that missing some AOCs results in different pruning opportunities, and, as a result, the set of discovered AOCs, which explains why the iterative algorithm discovers more AOCs in some cases.

Furthermore, as has been discussed for Exp-3, the running time of the iterative algorithm on larger datasets is prohibitively long. On such datasets, using the iterative algorithm results in missing *all* valid AOCs. For instance, in the ncvoter dataset with 5M tuples and with the approximation threshold set to 20%, the AOC of municipalityAbbrv ∼ municipalityDesc is discovered, which points to exceptions in creating abbreviations for municipalities; e.g., "Raleigh" is abbreviated as "RAL", while "Charlotte" is abbreviated as "CLT". However, this AOC does not hold in our 100K sample of tuples when using this threshold. Therefore, this dependency would have been missed by using the iterative validation algorithm, as it exceeds the time limit on the full dataset.

4.3 Comparison with Exact OD Discovery

Exp-5: Lattice level of AOCs and runtime improvements. AOCs tend to reside in lower levels of the lattice (with smaller contexts). In our scalability experiments in the number of tuples (Exp-1), the AOCs are on average 1.2 levels lower on the lattice. Similarly, in experiments in the number of attributes (Exp-2), the AOCs are on average 0.5 levels lower on the lattice. Figure 5 shows the number of OCs or AOCs found at each level of the lattice, when using 5M tuples and 10 attributes of the ncvoter dataset. On this dataset, the average lattice level of the discovered dependencies drops from 5.6 to 4.3 when using our approximate algorithm. As discussed in [10, 11], dependencies found in lower levels of the lattice are likely to be more interesting.

Furthermore, as discussed in Section 3.1, our discovery framework first validates candidates on lower levels of the lattice, and then applies pruning rules to generate the candidates on higher levels of the lattice (step 3 in Figure 1). Therefore, by finding AOCs in lower levels, the algorithm can use pruning rules more effectively earlier in the discovery process, resulting in pruning some candidates on higher levels of the lattice and validating fewer candidates in total. The effects of such pruning opportunities are not noticed when using the iterative validation algorithm, due to its prohibitively long running time. However, we optimally reduce the runtime of the validation step, resulting in runtime improvements for the discovery framework.

Figures 2 and 3 show the running times of the algorithms for discovering exact and approximate ODs. Even though validation of AOCs has a worse runtime compared to exact OCs, i.e., $O(n \log n)$, as opposed to $O(n)$, due to the extra pruning opportunities described above, the total runtime of the discovery framework for AODs can even be lower than the discovery framework for exact ODs; i.e., up to 34% and 76% faster in experiments in the number of tuples and attributes, respectively. The pronounced effect in the experiments in the number of attributes is due to having a smaller number of tuples.

Exp-6: Discovered AOCs compared to OCs. The exact algorithm fails to discover meaningful OCs in presence of anomalies, or even if a single value is erroneous. However, valid AOCs may hold in such instances. Other than the AOCs discussed in Exp-4, in the flight dataset, we discovered the AOC city ∼ airportName with a 27% approximation factor, which indicates that the names of airports usually begin with the name of the corresponding cities. Furthermore, the AOC streetAddress ∼ mailAddress holds in the ncvoter dataset with an approximation factor of 18%. These AOCs can point to anomalies and data quality issues, e.g., wrong address formats, misaligned mailing and residence addresses, and non-standard / erroneous airport names.

As shown in Figures 2 and 3, by discovering AOCs, we can find more dependencies in the data. Even if there are fewer AOCs than OCs (e.g., the flight dataset in Exp-2), the discovered dependencies are on lower levels of the lattice, as shown in Exp-5, which makes them more interesting [10, 11]. If the number of discovered dependencies is too large, the interestingness measure proposed in [11] can be used to rank the AOCs. In fact, the example AOCs that we have identified in Exp-4 and in this experiment, were all ranked as the most interesting AOCs.

5 CONCLUSIONS

We proposed a new validation algorithm for approximate ODs and proved its minimality and runtime optimality. We then implemented our approach in an existing canonical OD discovery framework and demonstrated significant gains compared to existing frameworks for discovering exact and approximate ODs. In future work, we will study new approaches for discovering approximate ODs, such as hybrid sampling, as done in [7] for FDs. We will also extend our approximate OD discovery framework to distributed settings, similar to the work in [9].

REFERENCES

[1] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis. 2019. Discovering order dependencies through order compatibility. *EDBT* (2019), 409–420.
[2] M. Fredman. 1975. On computing the length of longest increasing subsequences. *Discrete Mathematics* 11, 1 (1975), 29 – 35.
[3] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Computer J.* 42 (1999), 100–111.
[4] Yifeng Jin, L. Zhu, and Zijing Tan. 2020. Efficient Bidirectional Order Dependency Discovery. *ICDE* (2020), 61–72.
[5] Reza Karegar, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. 2021. Efficient Discovery of Approximate Order Dependencies. *Technical report, 7 pages, http://arxiv.org/abs/2101.02174* (2021).
[6] P. Langer and F. Naumann. 2016. Efficient Order Dependency Detection. *The VLDB Journal* 25, 2 (2016), 223–241.
[7] T. Papenbrock and F. Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. *SIGMOD* (2016), 821–833.
[8] Y. Qiu, Tan, K. Z., Yang, X. Yang, and N. Guo. 2018. Repairing data violations with order dependencies. *DASFAA* (2018), 283–300.
[9] H. Saxena, L. Golab, and I. Ilyas. 2019. Distributed Implementations of Dependency Discovery Algorithms. *PVLDB* 12, 11 (2019), 1624–1636.
[10] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2017. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB* 10, 7 (2017), 721–732.
[11] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2018. Effective and Complete Discovery of Bidirectional Order Dependencies via Set-Based Axioms. *The VLDB Journal* 27, 4 (2018), 573–591.
[12] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2020. Erratum for discovering order dependencies through order compatibility. *EDBT* (2020), 659–663.
[13] J. Szlichta, P. Godfrey, and J. Gryz. 2012. Fundamentals of Order Dependencies. *PVLDB* 5, 11 (2012), 1220–1231.