Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization

Jaroslaw Szlichta[#], Parke Godfrey[*], Lukasz Golab[$], Mehdi Kargar[♮], Divesh Srivastava[◇]
[#]University of Ontario Institute of Technology, Canada
[*]York University, Canada
[$]University of Waterloo, Canada
[♮]University of Windsor, Canada
[◇]AT&T Labs-Research, USA
jaroslaw.szlichta@uoit.ca, godfrey@yorku.ca, lgolab@uwaterloo.ca, mkargar@uwindsor.ca, divesh@research.att.com

ABSTRACT

Integrity constraints (ICs) are useful for query optimization and for expressing and enforcing application semantics. However, formulating constraints manually requires domain expertise, is prone to human errors, and may be excessively time consuming, especially on large datasets. Hence, proposals for automatic discovery have been made for some classes of ICs, such as functional dependencies (FDs), and recently, order dependencies (ODs). ODs properly subsume FDs, as they can additionally express business rules involving order; e.g., an employee never has a higher salary while paying lower taxes than another employee.

We present a new OD discovery algorithm enabled by a novel polynomial mapping to a canonical form of ODs, and a sound and complete set of axioms (inference rules) for canonical ODs. Our algorithm has exponential worst-case time complexity, $O(2^{|\mathbf{R}|})$, in the number of attributes $|\mathbf{R}|$ and linear complexity in the number of tuples. We prove that it produces a complete and minimal set of ODs. Using real and synthetic datasets, we experimentally show orders-of-magnitude performance improvements over the prior state-of-the-art.

1. INTRODUCTION

1.1 Motivation

With the interest in data analytics at an all-time high, *data quality* and *query optimization* are being revisited to address the scale and complexity of modern data-intensive applications. Real data suffer from inconsistencies, duplicates and missing values [3, 4]. A recent Gartner Research Report study in 2012 revealed that, by 2017, one third of the largest global companies will experience data quality crises due to their inability to trust and govern their enterprise information. Deep analytics on large data warehouses, spanning thousands of lines of SQL code, are no longer restricted to well-tuned, canned batch reports. Instead, complex ad-hoc queries are increasingly required for business operations to make timely data-driven decisions. Without clean data and effective query optimization, organizations will not be able to take advantage of new data-driven opportunities.

Table 1: A table with employee salaries and tax information.

#	ID	yr	posit	bin	sal	perc	tax	grp	subg
t1	10	16	secr	1	5K	20%	1K	A	III
t2	11	16	mngr	2	8K	25%	2K	C	II
t3	12	16	direct	3	10K	30%	3K	D	I
t4	10	15	secr	1	4.5K	20%	0.9K	A	III
t5	11	15	mngr	2	6K	25%	1.5K	C	I
t6	12	15	direct	3	8K	25%	2K	C	II

Integrity constraints (ICs) are commonly used to characterize data quality and to optimize business queries. Prior work has focused on *functional dependencies* (FDs) and extensions thereof, such as *conditional* FDs [4]. However, FDs cannot capture relationships among *ordered* attributes, such as between timestamps and numbers, which are common in business data [19]. For example, consider Table 1, which shows employee tax records in which tax is calculated as a percentage (perc) of salary (sal). Both tax and percentage increase with salary.

We study order dependencies (ODs) [10, 17, 20], which naturally express such semantics. The OD salary *orders* group (grp), subgroup (subg) holds in Table 1: if we sort the table by salary, it is also sorted by group, subg; i.e., sorted by group, with ties broken by subg. (However, the OD salary *orders* subg, grp does not hold.) Similarly, the OD salary *orders* tax, percentage holds in Table 1. ODs subsume FDs as any FD can be *mapped* to an equivalent OD by prefixing the left-hand-side attributes onto the right-hand-side [17]. For example, if salary *functionally determines* tax, then salary *orders* salary, tax (and, in this case, vice versa).

The additional expressiveness of ODs makes them particularly suitable for improving data quality, where ODs can describe intended semantics and business rules; and their violations can point out possible data errors.

Furthermore, query optimizers can use ODs to eliminate costly operators such as joins and sorts and to identify *interesting orders*: ordered streams between query operators that exploit available indices, enable pipelining, and eliminate intermediate sorts and partitioning steps [20]. Sorting and interesting orders are integral parts of relational query optimizers, not only for SQL order-by and group-by, but for

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 7
Copyright 2017 VLDB Endowment 2150-8097/17/03.

```
select ...
from web_sales, item, date_dim
where ws_item_sk = i_item_sk and ...
      and ws_sold_date_sk = d_date_sk and
      d_date between
          cast('1999-02-22' as date) and
          (cast('1999-02-22' as date)
               + 30 days)...;
```

Figure 1: TPC-DS Q#23 with an expensive join.

instance also for sort-merge joins [19]. A practical application of order dependencies for improved design to reduce the indexing space was presented in [5].

Date and time are richly supported in the SQL standard and frequently appear in decision support queries over historical data. For example, 85 out of 99 queries in the TPC-DS benchmark[1] involve date operators and five involve time operators. If ODs were only applicable to date and time, they would already be very beneficial. As seen in Table 1, ODs are also useful in other domains such as sales, flight schedules, stock prices, salaries and taxes [17, 19, 20].

ODs also enable new optimizations for *online analytical processing* (OLAP) such as eliminating expensive joins [18], and improving the performance of queries with SQL functions and algebraic expressions (e.g., `date` *orders* `year(date)` and `date` *orders* `year(date)*100 + month(date)`) [11, 20].

Data warehouse queries reference fact tables which include foreign keys to dimension tables. Foreign keys are often surrogate keys (unique sequential integers). However, queries usually reference other natural dimension attributes, not surrogate keys; e.g., the "between" predicate on `d_date` in TPC-DS query #23 (Figure 1). This predicate requires a (potentially) expensive join between the fact table and the date dimension table to identify all surrogate key values falling within the 30-day window starting at 1999-02-22. This can be particularly expensive if the potentially very large fact table is partitioned by date across many compute nodes. Since the date range surrogate values cannot be determined from `d_date`, all partitions of the fact table must be scanned. However, if `d_date_sk` *orders* `d_date`, then it suffices to make two probes into the date dimension table: one to find the minimum relevant `d_date_sk` value corresponding to 1999-02-22; and one to find the maximum relevant `d_date_sk` value corresponding to 1999-02-22 plus 30 days [18]. This allows us to restate the "between" predicate in terms of these two surrogate key values rather than dates, and thus eliminate the join.

TPC-DS query #23 can be optimized as shown in Figure 2. The cardinality reduction due to the selection on the date table is usually greater than that due to the selections on other dimension tables. Thus, the first join done is between the fact table and the date dimension table. Eliminating this join brings significant benefits. The experimental evaluation in [18] showed that 13 out of 99 TPC-DS queries matched this rewrite, which led to an average runtime improvement of 48%. Similar rewrite rules can improve the performance of other queries, too, such as TPC-DS query #29, which has an "in" predicate, `d_year in (1998, 1998+1, 1998+2)`.

1.2 Problem Statement and Contributions

To use ODs for data cleaning and query optimization, we need to know which ODs hold on a given dataset. While

[1] http://tpc.org/tpcds/

```
select ... from web_sales, item,
     (select min(d_date_sk) as mindate
         from date_dim
         where d_date >= cast('1999-02-22' as date))
         as A,
     (select max(d_date_sk) as maxdate
         from date_dim
         where d_date <= cast('1999-02-22' as date)
                + 30 days)
         as Z
where ... and ws_sold_date_sk between
    A.mindate and Z.maxdate...;
```

Figure 2: Rewrite of TPC-DS Q#23 (eliminating join).

dependencies can be obtained manually through consultation with domain experts, this is known to be an expensive, time consuming, and error-prone process that requires expertise in the data dependency language [9]. The problem we study in this paper is *how to automatically discover ODs from data*. Automatically discovered ODs can then be manually validated by domain experts, which is an easier task than manual specification.

This problem has been studied, but is not well understood. Our aim is to provide a deeper understanding of OD discovery. An OD discovery algorithm was recently proposed by Langer and Naumann [10], which has a *factorial* worst-case time complexity in the number of attributes. (This is the only prior OD discovery work of which we are aware.) In contrast to FDs, ODs are naturally expressed with lists rather than sets of attributes. For instance, salary *orders* tax, percentage is different from salary *orders* percentage, tax whereas the two analogous FDs are equivalent. The first complete axiomatization for ODs is expressed in a list notation [17]; Langer and Naumann use this.

An insight we present is that ODs can be expressed with sets of attributes via a *polynomial mapping* to a set-based canonical form. The mapping allows us to design a *fast* and *effective* OD discovery algorithm that has "only" *exponential* worst-case complexity, $O(2^{|\mathbf{R}|})$, in the number of attributes $|\mathbf{R}|$ (and linear complexity in the number of tuples). This complexity is similar to previous FD and Inclusion Dependency discovery algorithms such as TANE [9].

We also develop *sound* and *complete* set-based axioms (inference rules) for ODs that enable pruning of the search space, which can alleviate the worst-case complexity in practice. To overcome the factorial complexity, the list-based algorithm in [10] intentionally omits ODs in which the same attributes are repeated in the left-hand-side and the right-hand-side such as year, salary *orders* year, bin. In contrast, our pruning rules do not affect completeness.

Finally, by introducing a set-based canonical form for ODs, we achieve greater compactness in our representation. Many ODs that are considered minimal by the algorithm in [10] are found to be redundant by our algorithm. This is quite important for efficiency of OD discovery. We do not need to rediscover the "same" ODs repeatedly–that is, ODs that can be inferred from ones we have already discovered–and we can more aggressively prune portions of the search space which would only have "repeats".

In Section 2, we formally define ODs. We then make the following contributions.

1. *Mapping.* We translate ODs into a novel set-based *canonical form* (Section 3) which leads to a new and efficient approach to OD discovery (Section 4). By mapping ODs to *equivalent* set-based *canonical* ODs, we illustrate that they can be discovered efficiently by

traversing a *set-containment lattice* with *exponential* worst-case complexity in the number of attributes (the same as for FDs [9]), and just *linear* complexity in the number of tuples (Section 4.7).

2. *Set-based Axiomatization.* We introduce axioms for set-based canonical ODs and prove these are *sound* and *complete* (Section 3.2). Our inference rules reveal insights into the nature of canonical ODs, which lead to optimizations of the OD discovery algorithm to avoid redundant computation (Section 4).
3. *Completeness and Interestingness.* We prove that our discovery algorithm produces a *complete* and *minimal* set of ODs (Section 4). We also propose an *interestingness* measure for ranking the discovered ODs, which makes manual verification easier and provides additional pruning.
4. *Experiments.* We provide an experimental study (Section 5) of the performance and effectiveness of our discovery techniques using real datasets. We report orders-of-magnitude performance improvements over previous work [10].

We discuss related work in Section 6. In Section 7, we conclude and consider future work.

2. PRELIMINARIES

In this section, we formally define ODs, present the established list-based axiomatization [17], and explain the two ways in which ODs can be violated.

2.1 Framework and Background

We use the following notational conventions.

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

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

Lists. $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ denote *lists* of attributes. $\mathbf{X}$ may represent the empty list, denoted as $[\,]$. $[\mathsf{A}, \mathsf{B}, \mathsf{C}]$ denotes an explicit list. $[\mathsf{A} \mid \mathbf{T}]$ denotes a list with *head* A and *tail* $\mathbf{T}$; i.e., the remaining list when the first element is removed. Let $\mathbf{XY}$ be shorthand for $\mathbf{X} \circ \mathbf{Y}$ ($\mathbf{X}$ concatenate $\mathbf{Y}$). Set $\mathcal{X}$ denotes the *set* of elements in *list* $\mathbf{X}$. Any place a set is expected but a list appears, the list is *cast* to a set; e.g., $t_\mathbf{X}$ denotes $t_\mathcal{X}$. Let $\mathbf{X}'$ denote some other permutation of elements of list $\mathbf{X}$.

Let an *order specification* be a list of attributes defining a lexicographic order, as in the SQL order-by clause. The order specification `order by group, subgroup` requires sorting by `group` in ascending order and, within each `group`, by `subgroup` in ascending order. This is a lexicographical ordering, a *nested* sort.

First, we define the operator '$\preceq_\mathbf{X}$', which defines a *weak total order* over any set of tuples, where $\mathbf{X}$ denotes an order specification. Unless otherwise specified, numbers are ordered numerically, strings are ordered lexicographically and dates are ordered chronologically (all *ascending*).

Definition 1. Let $\mathbf{X}$ be a list of attributes. For two tuples r and s, $\mathcal{X} \in \mathbf{R}$, $r \preceq_\mathbf{X} s$ if
- $\mathbf{X} = [\,]$; *or*
- $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}]$ and $r_\mathsf{A} < s_\mathsf{A}$; or
- $\mathbf{X} = [\mathsf{A} \mid \mathbf{T}]$, $r_\mathsf{A} = s_\mathsf{A}$, and $r \preceq_\mathbf{T} s$.

1. *Reflexivity*
$$\mathbf{XY} \mapsto \mathbf{X}$$
2. *Prefix*
$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{ZX} \mapsto \mathbf{ZY}}$$
3. *Transitivity*
$$\frac{\mathbf{X} \mapsto \mathbf{Y} \quad \mathbf{Y} \mapsto \mathbf{Z}}{\mathbf{X} \mapsto \mathbf{Z}}$$
4. *Normalization*
$$\mathbf{WXYXV} \leftrightarrow \mathbf{WXYV}.$$
5. *Suffix*
$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{X} \leftrightarrow \mathbf{YX}}$$
6. *Chain*
$$\frac{\mathbf{X} \sim \mathbf{Y}_1 \quad \forall_{i \in [1,n-1]} \mathbf{Y}_i \sim \mathbf{Y}_{i+1} \quad \mathbf{Y}_n \sim \mathbf{Z} \quad \forall_{i \in [1,n]} \mathbf{Y}_i\mathbf{X} \sim \mathbf{Y}_i\mathbf{Z}}{\mathbf{X} \sim \mathbf{Z}}$$

Figure 3: List-based axioms for ODs.

Let $r \prec_\mathbf{X} s$ if $r \preceq_\mathbf{X} s$ but $s \not\preceq_\mathbf{X} r$.

Next, we define order dependencies [10, 17, 20].

Definition 2. Let $\mathbf{X}$ and $\mathbf{Y}$ be order specifications, where $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R}$. $\mathbf{X} \mapsto \mathbf{Y}$ denotes an *order dependency* (OD), read as $\mathbf{X}$ *orders* $\mathbf{Y}$. We write $\mathbf{X} \leftrightarrow \mathbf{Y}$, read as $\mathbf{X}$ and $\mathbf{Y}$ are *order equivalent*, if $\mathbf{X}$ *orders* $\mathbf{Y}$ and $\mathbf{Y}$ *orders* $\mathbf{X}$. Table $\mathbf{r}$ over $\mathbf{R}$ *satisfies* $\mathbf{X} \mapsto \mathbf{Y}$ ($\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$) if, for all $r, s \in \mathbf{r}$, $r \preceq_\mathbf{X} s$ implies $r \preceq_\mathbf{Y} s$. $\mathbf{X} \mapsto \mathbf{Y}$ is said to *hold* for $\mathbf{R}$ ($\mathbf{R} \models \mathbf{X} \mapsto \mathbf{Y}$) if, for each admissible relational instance $\mathbf{r}$ of $\mathbf{R}$, table $\mathbf{r}$ satisfies $\mathbf{X} \mapsto \mathbf{Y}$. $\mathbf{X} \mapsto \mathbf{Y}$ is *trivial* if, for all $\mathbf{r}$, $\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$.

The OD $\mathbf{X} \mapsto \mathbf{Y}$ means that $\mathbf{Y}$'s values are monotonically non-decreasing with respect to $\mathbf{X}$'s values. Thus, if a list of tuples is ordered by $\mathbf{X}$, then it is also ordered by $\mathbf{Y}$, but not necessarily vice versa. This is to say, if one knows $\mathbf{X} \mapsto \mathbf{Y}$, then one knows that any ordering of tuples of any table that satisfies `order by` $\mathbf{X}$ must also satisfy `order by` $\mathbf{Y}$. We assume *ascending* (asc) order in the lexicographical ordering, which is the SQL default for any attributes for which directionality is not explicitly indicated. We do not consider *bidirectional* ODs in this paper, which allow a mix of ascending and descending (desc) orders [20]. We also only consider lexicographic order specifications, as per the SQL order-by semantics, and do not consider *pointwise* ODs [6, 7]. Working with lexicographical ODs is more useful for query optimization [10, 19, 20] than working with pointwise ODs, because the sequence of the attributes in an order specification as in order-by matters.

Example 1. *Recall Table 1, in which tax is calculated as a percentage of salary, and tax groups and subgroups are based on salary. Tax, percentage and group increase with salary. Furthermore, within the same group, subgroup increases with salary. Finally, within the same year, bin increases with salary. Thus, the following ODs hold:* [salary] $\mapsto$ [tax]*;* [salary] $\mapsto$ [percentage]*;* [salary] $\mapsto$ [group, subgroup]*; and* [year, salary] $\mapsto$ [year, bin]*. Let Table 1 have a clustered index on* year, salary*. Then, given the OD* [year, salary] $\mapsto$ [year, bin]*, a query with* `order by` year, bin *could be evaluated using the index on year and salary.*

2.2 List-based Axiomatization

Figure 3 shows a *sound* and *complete* list-based axiomatization for ODs [17]. The Chain axiom uses the notion of *order compatibility*, denoted as "$\sim$", defined below.

Definition 3. Two order specifications $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible*, denoted as $\mathbf{X} \sim \mathbf{Y}$, *iff* $\mathbf{XY} \leftrightarrow \mathbf{YX}$. The empty order specification (i.e., $[\,]$) is order compatible with *any* order specification.

EXAMPLE 2. [d_month] $\sim$ [d_week] *is valid: sorting by month and breaking ties by week is equivalent to sorting by week and breaking ties by month. However, the OD* [d_month] $\mapsto$ [d_week] *does not hold since any given month corresponds to several different weeks (in other words,* d_month *does not* functionally determine d_week*). Thus, order compatibility is a weaker notion than order dependency.*

2.3 Violations

ODs can be violated in two ways. We begin with the following theorem and then explain how the two conditions therein correspond to two possible sources of violations. Detailed proofs are omitted due to the space constraints; however, they can be found in the technical report [16].

THEOREM 1. *For every instance* $\mathbf{r}$ *of relation* $\mathbf{R}$*,* $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$ *and* $\mathbf{X} \sim \mathbf{Y}$*.*

Proof sketch: Suppose $\mathbf{X} \mapsto \mathbf{Y}$, therefore, by *Suffix* $\mathbf{X} \leftrightarrow \mathbf{YX}$ can be inferred. Hence, by *Prefix* and *Normalization* $\mathbf{X} \sim \mathbf{Y}$ holds. Next, assume that $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{X} \sim \mathbf{Y}$. By *Transitivity*, $\mathbf{X} \mapsto \mathbf{YX}$. Hence, by *Reflexivity* and *Transitivity*, $\mathbf{X} \mapsto \mathbf{Y}$. □

There is a strong relationship between ODs and FDs. Any OD implies an FD, modulo lists and sets, but not vice versa.

LEMMA 1. *For every instance* $\mathbf{r}$ *of relation* $\mathbf{R}$*, if an OD* $\mathbf{X} \mapsto \mathbf{Y}$ *holds, then the FD* $\mathcal{X} \rightarrow \mathcal{Y}$ *is true.*

Proof sketch: Let rows s, $t \in \mathbf{r}$. Assume that $s_\mathcal{X} = t_\mathcal{X}$. Hence, $s \preceq_\mathbf{X} t$ and $t \preceq_\mathbf{X} s$. By definition of an OD, $s \preceq_\mathbf{Y} t$ and $t \preceq_\mathbf{Y} s$. Therefore, $s_\mathcal{Y} = t_\mathcal{Y}$ holds. □

Also, there exists a correspondence between FDs and ODs.

THEOREM 2. *For relation* $\mathbf{R}$*, for every instance* $\mathbf{r}$*,* $\mathcal{X} \rightarrow \mathcal{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$*, for any list* $\mathbf{X}$ *over the attributes of* $\mathcal{X}$ *and any list* $\mathbf{Y}$ *over the attributes of* $\mathcal{Y}$*.*

Proof sketch: Assume an OD $\mathbf{X} \mapsto \mathbf{XY}$ does not hold. This means, there exist s, $t \in \mathbf{r}$, such that $s \preceq_\mathbf{X} t$ but $s \not\preceq_\mathbf{XY} t$. Therefore, $s_\mathcal{X} = t_\mathcal{X}$ and $s \prec_\mathbf{Y} t$. Also $s \prec_\mathbf{Y} t$ implies that $s_\mathcal{Y} \neq t_\mathcal{Y}$. Therefore, $\mathcal{X} \rightarrow \mathcal{Y}$ is not satisfied. However, if $\mathbf{X} \mapsto \mathbf{XY}$, then $\mathcal{X} \rightarrow \mathcal{XY}$. Hence, by Armstrong's axiom of Reflexivity and Transitivity, $\mathcal{X} \rightarrow \mathcal{Y}$. □

We are now ready to explain the two sources of OD violations: *splits* and *swaps* [17]. Langer et al. [10] utilize these concepts to validate ODs in their list-based OD discovery algorithm (Section 4.5). An OD $\mathbf{X} \mapsto \mathbf{Y}$ can be violated in two ways, as per Theorem 1. Split falsifies $\mathbf{X} \mapsto \mathbf{XY}$ ($\mathcal{X}$ does not functionally determine $\mathcal{Y}$) and swap falsifies $\mathbf{X} \sim \mathbf{Y}$.

Definition 4. A *split* with respect to an OD $\mathbf{X} \mapsto \mathbf{XY}$ is a pair of tuples s and t such that $s_\mathcal{X} = t_\mathcal{X}$ but $s_\mathcal{Y} \neq t_\mathcal{Y}$.

Definition 5. A *swap* with respect to $\mathbf{X} \sim \mathbf{Y}$ (i.e., with respect to $\mathbf{XY} \leftrightarrow \mathbf{YX}$) is a pair of tuples s and t such that $s \prec_\mathbf{X} t$, but $t \prec_\mathbf{Y} s$.

EXAMPLE 3. *In Table 1, there are three splits with respect to the OD* [position] $\mapsto$ [position, salary] *because* position *does not functionally determine* salary*. The violating tuple pairs are t1 and t4, t2 and t5, and t3 and t6. There is a swap with respect to* [salary] $\sim$ [subgroup]*, e.g., over pair of tuples t1 and t2.*

3. SET-BASED CANONICAL FORM

We now present our first set of contributions: a polynomial mapping from the list-based representation to a set-based canonical form for ODs, and a *sound* and *complete* axiomatization over this representation. These are the fundamental building blocks of our efficient OD discovery algorithm that will be discussed in Section 4.

3.1 Mapping to Canonical Form

Expressing ODs in a natural way relies on lists of attributes, as in the SQL order-by statement. One might well wonder whether lists are inherently necessary. Indeed, we provide a polynomial *mapping* of list-based ODs into *equivalent* set-based canonical ODs. The mapping allows us to develop an efficient OD discovery algorithm that traverses a much smaller set-containment lattice rather than the list-containment lattice used in prior work [10].

Two tuples s and t are *equivalent* with respect to a given set $\mathcal{X}$ if $s_\mathcal{X} = t_\mathcal{X}$. Any attribute set $\mathcal{X}$ partitions tuples into *equivalence classes* [9]. We denote the *equivalence class* of a tuple $t \in \mathbf{r}$ with respect to a given set $\mathcal{X}$ by $\mathcal{E}(t_\mathcal{X})$, i.e., $\mathcal{E}(t_\mathcal{X}) = \{s \in \mathbf{r} \mid s_\mathcal{X} = t_\mathcal{X}\}$. A *partition* of $\mathbf{r}$ over $\mathcal{X}$ is the set of equivalence classes, $\Pi_\mathcal{X} = \{\mathcal{E}(t_\mathcal{X}) \mid t \in \mathbf{r}\}$. For instance, in Table 1, $\mathcal{E}(t1_{\{\mathsf{year}\}}) = \mathcal{E}(t2_{\{\mathsf{year}\}}) = \mathcal{E}(t3_{\{\mathsf{year}\}}) = \{t1, t2, t3\}$ and $\Pi_{\mathsf{year}} = \{\{t1, t2, t3\}, \{t4, t5, t6\}\}$.

We now introduce a *canonical form* for ODs.

Definition 6. An attribute A is a *constant* within each equivalence class with respect to $\mathcal{X}$, denoted as $\mathcal{X}$: [] $\mapsto_{cst}$ A, if $\mathbf{X}' \mapsto \mathbf{X}'$A for any permutation $\mathbf{X}'$ of $\mathbf{X}$. Furthermore, two attributes A and B are order-compatible (i.e., no swaps) within each equivalence class with respect to $\mathcal{X}$, denoted as $\mathcal{X}$: A $\sim$ B, if $\mathbf{X}'$A $\sim \mathbf{X}'$B. ODs of the form of $\mathcal{X}$: [] $\mapsto_{cst}$ A and $\mathcal{X}$: A $\sim$ B are called *canonical* ODs, and the set $\mathcal{X}$ is called a *context*.

EXAMPLE 4. *In Table 1,* bin *is a constant in the context of* position (posit)*, written as* {position}: [] $\mapsto_{cst}$ bin*. This is because* $\mathcal{E}(t1_{\{\mathsf{position}\}}) \models$ [] $\mapsto_{cst}$ bin*,* $\mathcal{E}(t2_{\{\mathsf{position}\}}) \models$ [] $\mapsto_{cst}$ bin *and* $\mathcal{E}(t3_{\{\mathsf{position}\}}) \models$ [] $\mapsto_{cst}$ bin*.*

Also, there is no swap between bin *and* salary *in the context of* year*, i.e.,* {year}: bin $\sim$ salary*. This is because* $\mathcal{E}(t1_{\{\mathsf{year}\}}) \models$ bin $\sim$ salary *and* $\mathcal{E}(t4_{\{\mathsf{year}\}}) \models$ bin $\sim$ salary*. However, the canonical ODs* {year}: bin $\sim$ subgroup *and* {position}: [] $\mapsto_{cst}$ salary *do not hold as* $\mathcal{E}(t1_{\{\mathsf{year}\}}) \not\models$ bin $\sim$ subgroup *and* $\mathcal{E}(t1_{\{\mathsf{position}\}}) \not\models$ [] $\mapsto_{cst}$ salary*, respectively.*

Given a set of attributes $\mathcal{Y}$, for brevity, we state $\forall j$, Y_j to mean $\forall j \in [1..|\mathcal{Y}|]$, Y_j in the remainder of this section.

THEOREM 3.
An OD $\mathbf{X} \mapsto \mathbf{XY}$ *holds iff* $\forall j$*,* $\mathcal{X}$: [] $\mapsto_{cst}$ Y_j*.*

Proof sketch: Let $\mathbf{X} \mapsto \mathbf{XY}$ hold. Thus, by *Reflexivity* and *Transitivity* $\forall j$, $\mathbf{X} \mapsto \mathbf{XY}_j$. Hence, by *Prefix* and *Normalization* $\forall j$, $\mathbf{X}' \mapsto \mathbf{X}'\mathsf{Y}_j$, i.e, $\forall j$, $\mathcal{X}$: [] $\mapsto_{cst}$ Y_j. Next, assume $\forall j$, $\mathcal{X}$: [] $\mapsto_{cst}$ Y_j. Thus, $\forall j$, $\mathbf{X}' \mapsto \mathbf{X}'\mathsf{Y}_j$. Hence, it follows from *Union* [17] (if $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{X} \mapsto \mathbf{Z}$, then $\mathbf{X} \mapsto \mathbf{YZ}$) that $\mathbf{X} \mapsto \mathbf{XY}$. □

THEOREM 4.
$\mathbf{X} \sim \mathbf{Y}$ *iff* $\forall i, j$, $\{\mathsf{X}_1, .., \mathsf{X}_{i-1}, \mathsf{Y}_1, .., \mathsf{Y}_{j-1}\}$: $\mathsf{X}_i \sim \mathsf{Y}_j$*.*

Proof sketch: Let $\mathbf{X} \sim \mathbf{Y}$, hence, by *Downward Closure* [17] (if $\mathbf{XZ} \sim \mathbf{YV}$, then $\mathbf{X} \sim \mathbf{Y}$), $\forall i, j$, X_1, .., $\mathsf{X}_i \sim \mathsf{Y}_1$, .., Y_j

1. Reflexivity
$\mathcal{X}$: $[] \mapsto_{cst} \mathsf{A}$, $\forall \mathsf{A} \in \mathcal{X}$

2. Identity
$\mathcal{X}$: $\mathsf{A} \sim \mathsf{A}$

3. Commutativity
$$\frac{\mathcal{X}\text{: } \mathsf{A} \sim \mathsf{B}}{\mathcal{X}\text{: } \mathsf{B} \sim \mathsf{A}}$$

4. Strengthen
$$\frac{\mathcal{X}\text{: } [] \mapsto_{cst} \mathsf{A}}{\mathcal{X}\mathsf{A}\text{: } [] \mapsto_{cst} \mathsf{B}}$$
$\mathcal{X}$: $[] \mapsto_{cst} \mathsf{B}$

5. Propagate
$$\frac{\mathcal{X}\text{: } [] \mapsto_{cst} \mathsf{A}}{\mathcal{X}\text{: } \mathsf{A} \sim \mathsf{B}}$$

6. Augmentation-I
$$\frac{\mathcal{X}\text{: } [] \mapsto_{cst} \mathsf{A}}{\mathcal{Z}\mathcal{X}\text{: } [] \mapsto_{cst} \mathsf{A}}$$

7. Augmentation-II
$$\frac{\mathcal{X}\text{: } \mathsf{A} \sim \mathsf{B}}{\mathcal{Z}\mathcal{X}\text{: } \mathsf{A} \sim \mathsf{B}}$$

8. Chain
$$\frac{\begin{array}{l}\mathcal{X}\text{: } \mathsf{A} \sim \mathsf{B}_1 \\ \forall_{i \in [1,n-1]}, \mathcal{X}\text{: } \mathsf{B}_i \sim \mathsf{B}_{i+1} \\ \mathcal{X}\text{: } \mathsf{B}_n \sim \mathsf{C} \\ \forall_{i \in [1,n]}, \mathcal{X}\mathsf{B}_i : \mathsf{A} \sim \mathsf{C}\end{array}}{\mathcal{X}\text{: } \mathsf{A} \sim \mathsf{C}}$$

Figure 4: Set-based axiomatization for canonical ODs.

and by *Prefix*, *Normalization* and Theorem 1, $\forall i, j$, $\{\mathsf{X}_1, .., \mathsf{X}_{i-1}, \mathsf{Y}_1, .., \mathsf{Y}_{j-1}\}$: $\mathsf{X}_i \sim \mathsf{Y}_j$. Given $\forall i, j$, $\{\mathsf{X}_1, .., \mathsf{X}_{i-1}, \mathsf{Y}_1, .., \mathsf{Y}_{j-1}\}$: $\mathsf{X}_i \sim \mathsf{Y}_j$, by *Prefix*, *Normalization*, *Transitivity* and *Reflexivity*, it follows that $\mathbf{X} \sim \mathbf{Y}$. □

THEOREM 5. $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\forall j$, $\mathcal{X}$: $[] \mapsto_{cst} \mathsf{Y}_j$ *and* $\forall i, j$, $\{\mathsf{X}_1, .., \mathsf{X}_{i-1}, \mathsf{Y}_1, .., \mathsf{Y}_{j-1}\}$: $\mathsf{X}_i \sim \mathsf{Y}_j$.

Proof sketch: Since $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{X} \sim \mathbf{Y}$ (Theorem 1), therefore, by Theorems 3 and 4 $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\forall j$, $\mathcal{X}$: $[] \mapsto_{cst} \mathsf{Y}_j$ and $\forall i, j$, $\{\mathsf{X}_1, .., \mathsf{X}_{i-1}, \mathsf{Y}_1, .., \mathsf{Y}_{j-1}\}$: $\mathsf{X}_i \sim \mathsf{Y}_j$. □

The size of this mapping is the product $|\mathbf{X}| * |\mathbf{Y}|$.

EXAMPLE 5. *By Theorem 5, an OD* $[\mathsf{AB}] \mapsto [\mathsf{CD}]$ *can be mapped into the following equivalent set of canonical ODs:* $\{\mathsf{A}, \mathsf{B}\}$: $[] \mapsto_{cst} \mathsf{C}$, $\{\mathsf{A}, \mathsf{B}\}$: $[] \mapsto_{cst} \mathsf{D}$, $\{\}$: $\mathsf{A} \sim \mathsf{C}$, $\{\mathsf{A}\}$: $\mathsf{B} \sim \mathsf{C}$, $\{\mathsf{C}\}$: $\mathsf{A} \sim \mathsf{D}$, $\{\mathsf{A}, \mathsf{C}\}$: $\mathsf{B} \sim \mathsf{D}$.

3.2 Set-based Axiomatization

We present a *sound* and *complete set-based axiomatization* for ODs in Figure 4. For clarity, we denote names of list-based inference rules (Figure 3) with italic font and those of set-based inference rules (Figure 4) with regular font. Below, we show additional inference rules that can be inferred from the axioms in Figure 4, as they are used in Section 4.

LEMMA 2 (TRANSITIVITY).
$$\frac{\begin{array}{l}1.\ \forall j,\ \mathcal{X}\text{: } [] \mapsto_{cst} \mathsf{Y}_j \\ 2.\ \forall k,\ \mathcal{Y}\text{: } [] \mapsto_{cst} \mathsf{Z}_k\end{array}}{\forall k,\ \mathcal{X}\text{: } [] \mapsto_{cst} \mathsf{Z}_k}$$

LEMMA 3 (NORMALIZATION).
1. $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$, $\forall \mathsf{A} \in \mathcal{X}$

THEOREM 6. *The proposed set-based axiomatization for canonical ODs in Figure 4 is sound and complete.*

Proof sketch: Given the mapping presented in Theorem 5, to prove *soundness*, it is sufficient to show that all the set-based OD axioms (Figure 4) can be inferred from the list-based OD axioms (Figure 3). In order to prove *completeness* the remaining step is to show that all the list-based OD axioms follow from the set-based OD axioms. □

By Theorem 6, three set-based axioms, namely Reflexivity, Strengthen and Augmentation–I, are sound and complete for the FD fragment of the OD class, as these are the only canonical OD axioms that use the constant operator. In contrast, the sound and complete list-based axiomatization for ODs [17] is concealed within the first five axioms in Figure 3. The versatility and separability of the set-based axioms–between the FD fragment and the order-compatible fragment–allows us to design effective pruning rules for our OD discovery algorithm (Section 4).

4. OD DISCOVERY ALGORITHM

4.1 FASTOD Main Algorithm

Given the mapping of a list-based OD into equivalent set-based ODs (Section 3.1), we present an algorithm, named *FASTOD* (Algorithm 1), which *efficiently* discovers a complete and minimal set of set-based ODs over a given relation instance.

A canonical OD $\mathcal{X}$: $[] \mapsto_{cst} \mathsf{A}$ is *trivial* if $\mathsf{A} \in \mathcal{X}$ (Reflexivity). A canonical OD $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$ is *trivial* if $\mathsf{A} \in \mathcal{X}$ or $\mathsf{B} \in \mathcal{X}$ (Normalization, Lemma 3) or $\mathsf{A} = \mathsf{B}$ (Identity). A canonical OD $\mathcal{X}$: $[] \mapsto_{cst} \mathsf{A}$ is *minimal* if it is non-trivial and there is no context $\mathcal{Y} \subset \mathcal{X}$ such that $\mathcal{Y}$: $[] \mapsto_{cst} \mathsf{A}$ holds in $\mathbf{r}$ (Augmentation–I). A canonical OD $\mathcal{X}$: $\mathsf{A} \sim \mathsf{B}$ is *minimal* if it is non-trivial and there is no context $\mathcal{Y}$, where $\mathcal{Y} \subset \mathcal{X}$, such that $\mathcal{Y}$: $\mathsf{A} \sim \mathsf{B}$ holds in $\mathbf{r}$ (Aug–II), or $\mathcal{X}$: $[] \mapsto_{cst} \mathsf{A}$ or $\mathcal{X}$: $[] \mapsto_{cst} \mathsf{B}$ (Propagate) holds in $\mathbf{r}$. Our goal is to compute a complete, minimal set of ODs that hold in $\mathbf{r}$.

FASTOD traverses a *lattice* of all possible *sets* of attributes in a level-wise manner (Figure 5) since list-based ODs can be mapped into equivalent set-based ODs (Theorem 5). In contrast, the OD discovery algorithm from [10] traverses a lattice of all possible *lists* of attributes, which leads to factorial time complexity. In level $\mathcal{L}_l$, our algorithm generates candidate ODs with l attributes using *computeODs*($\mathcal{L}_l$). *FASTOD* starts the search from singleton sets of attributes and works its way to larger attribute sets through the set-containment lattice, level by level. When the algorithm is processing an attribute set $\mathcal{X}$, it verifies ODs of the form $\mathcal{X} \setminus \mathsf{A}$: $[] \mapsto_{cst} \mathsf{A}$ (let $\mathcal{X} \setminus \mathsf{A}$ be shorthand for $\mathcal{X} \setminus \{\mathsf{A}\}$), where $\mathsf{A} \in \mathcal{X}$ and $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$, where $\mathsf{A}, \mathsf{B} \in \mathcal{X}$ and $\mathsf{A} \neq \mathsf{B}$. This guarantees that only non-trivial ODs are considered.

Algorithm 1 FASTOD

Input: Relation $\mathbf{r}$ over schema $\mathbf{R}$
Output: Minimal set of ODs $\mathcal{M}$, such that $\mathbf{r} \models \mathcal{M}$

```
1: L_0 = {}
2: C_c^+({}) = R
3: C_s^+({}) = {}
4: l = 1
5: L_1 = {A | A ∈ R}
6: ∀_{A∈R} C_s^+(A) = {}
7: while L_l ≠ {} do
8:     computeODs(L_l)
9:     pruneLevels(L_l)
10:    L_{l+1} = calculateNextLevel(L_l)
11:    l = l + 1
12: end while
13: return M
```

The small-to-large search strategy of the discovery algorithm guarantees that only ODs that are minimal with respect to the context are added to the output set of ODs $\mathcal{M}$, and is used to prune the search space effectively. The OD

candidates generated in a given level are checked for *minimality* based on the previous levels and are added to a *valid* set of ODs $\mathcal{M}$ if applicable. The algorithm *pruneLevels*($\mathcal{L}_l$) prunes the search space by deleting sets from $\mathcal{L}_l$ with the knowledge gained during the checks in *computeODs*($\mathcal{L}_l$). The algorithm *calculateNextLevel*($\mathcal{L}_l$) forms the next level from the current level.

Our algorithm, for example, can detect the following ODs in the TPC-DS benchmark: {d_date_sk}: [] $\mapsto_{cst}$ [d_date]; {}: [d_date_sk] $\sim$ [d_date]; {d_date_sk}: [] $\mapsto_{cst}$ [d_year]; {}: [d_date_sk] $\sim$ [d_year]; {d_month}: [] $\mapsto_{cst}$ [d_quarter] and; {}: [d_month] $\sim$ [d_quarter]. These types of canonical ODs have been proven to be useful to eliminate joins and simplify group-by and order-by statements [18, 20].

Next, we explain, in turn, each of the algorithms that are called in the main loop of *FASTOD* (Lines 7–12).

4.2 Finding Minimal ODs

FASTOD traverses the lattice until all complete and minimal ODs are found. First, we deal with ODs of the form $\mathcal{X} \setminus A$: [] $\mapsto_{cst} A$, where $A \in \mathcal{X}$. To check if such an OD is minimal, we need to know if $\mathcal{Y} \setminus A$: [] $\mapsto_{cst} A$ is valid for $\mathcal{Y} \subset \mathcal{X}$. If $\mathcal{Y} \setminus A$: [] $\mapsto_{cst} A$, then by Augmentation–I $\mathcal{X} \setminus A$: [] $\mapsto_{cst} A$ holds. An OD $\mathcal{X}$: [] $\mapsto_{cst} A$ holds for any relational instance by Reflexivity, therefore, considering only $\mathcal{X} \setminus A$: [] $\mapsto_{cst} A$ guarantees that only non-trivial ODs are taken into account.

We maintain information about minimal ODs, in the form of $\mathcal{X} \setminus A$: [] $\mapsto_{cst} A$, in the *candidate* set $\mathcal{C}_c^+(\mathcal{X})$ [9].[2] If $A \in \mathcal{C}_c^+(\mathcal{X})$ for a given set $\mathcal{X}$, then A has not been found to depend on any proper subset of $\mathcal{X}$. Therefore, to find minimal ODs, it suffices to verify ODs $\mathcal{X} \setminus A$: [] $\mapsto_{cst} A$, where $A \in \mathcal{X}$ and $A \in \mathcal{C}_c^+(\mathcal{X} \setminus B)$ for all $B \in \mathcal{X}$.

Example 6. *Assume that* {B}: [] $\mapsto_{cst}$ A *and that we consider the set* $\mathcal{X} = \{A, B, C\}$. *As* {B}: [] $\mapsto_{cst}$ A *holds,* $A \notin \mathcal{C}_c^+(\mathcal{X} \setminus C)$. *Hence, the OD* {B, C}: [] $\mapsto_{cst}$ A *is not minimal.*

We now show how to prune the search space. Assume that $B \in \mathcal{X}$ and an OD $\mathcal{X} \setminus B$: [] $\mapsto_{cst} B$ holds. Then, by Lemma 4 below, inferring via the Strengthen axiom, the OD $\mathcal{X}$: [] $\mapsto_{cst} A$ cannot be minimal because B can be removed from the context $\mathcal{X}$ [9]. Observe that *FASTOD* does not have to know whether $\mathcal{X}$: [] $\mapsto_{cst} A$ holds.

Lemma 4. *Let* $B \in \mathcal{X}$ *and* $\mathcal{X} \setminus B$: [] $\mapsto_{cst} B$. *If* $\mathcal{X}$: [] $\mapsto_{cst} A$, *then* $\mathcal{X} \setminus B$: [] $\mapsto_{cst} A$.

Hence, we define the candidate set $\mathcal{C}_c^+(\mathcal{X})$, formally as follows. (Note that A may equal B in Definition 7).

Definition 7. $\mathcal{C}_c^+(\mathcal{X}) = \{A \in \mathbf{R} \mid$
$\forall_{B \in \mathcal{X}}\ \mathcal{X} \setminus \{A, B\}$: [] $\mapsto_{cst} B$ does not hold$\}$

Example 7. *Assume that* {B}: [] $\mapsto_{cst}$ C *and that* FASTOD *considers* {B, C}: [] $\mapsto_{cst}$ A. *Since* {B}: [] $\mapsto_{cst}$ C *holds, the OD* {B, C}: [] $\mapsto_{cst}$ A *is not minimal and* $A \notin \mathcal{C}_c^+(\mathcal{X})$, *where* $\mathcal{X} = \{A, B, C\}$.

[2]Since FDs are subsumed by ODs, FD discovery is part of *FASTOD*, and some of our techniques are similar to those from *TANE* [9]. However, *FASTOD* and *TANE* differ in many details, even for FD discovery; e.g., a pruning rule for removing nodes from the lattice (Section 4.5) and the key pruning rule (Section 4.6). Additionally, *FASTOD* includes new OD-specific pruning rules.

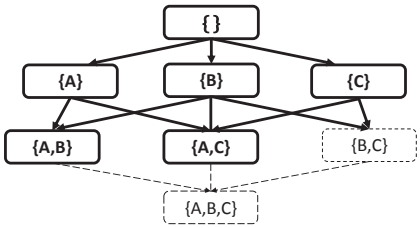

Figure 5: A pruned set lattice for attributes A, B, C. As {B, C} is discarded, only the non-dashed bold parts are accessed.

We now deal with ODs involving '$\sim$'. To verify if a potential OD of the form $\mathcal{X} \setminus \{A, B\}$: $A \sim B$, where $A, B \in \mathcal{X}$ and $A \neq B$, is *minimal*, we need to know if $\mathcal{Y} \setminus \{A, B\}$: $A \sim B$ holds for some proper subset $\mathcal{Y}$ of $\mathcal{X} \setminus \{A, B\}$, and that $\mathcal{X} \setminus \{A, B\}$: [] $\mapsto_{cst} A$ and $\mathcal{X} \setminus \{A, B\}$: [] $\mapsto_{cst} B$ do not hold. If $\mathcal{Y} \setminus \{A, B\}$: $A \sim B$, then by Augmentation–II, $\mathcal{X} \setminus \{A, B\}$: $A \sim B$ holds. Also, if $\mathcal{X} \setminus \{A, B\}$: [] $\mapsto_{cst} A$ or $\mathcal{X} \setminus \{A, B\}$: [] $\mapsto_{cst} B$, then by Propagate, $\mathcal{X} \setminus \{A, B\}$: $A \sim B$ holds. ODs $\mathcal{X} \setminus B$: $A \sim B$, $\mathcal{X} \setminus A$: $A \sim B$, and OD $\mathcal{X}$: $A \sim A$ are always true by Normalization and Identity, respectively. Hence, considering ODs in the form of $\mathcal{X} \setminus \{A, B\}$: $A \sim B$ guarantees that non-trivial ODs are taken into account.

We store information about minimal ODs in the form of $\mathcal{X} \setminus \{A, B\}$: $A \sim B$, in the *candidate* set $\mathcal{C}_s^+(\mathcal{X})$. If $\{A, B\} \in \mathcal{C}_s^+(\mathcal{X})$ for a given set $\mathcal{X}$, then $A \sim B$ has not been found to hold within the context of any subset of $\mathcal{X} \setminus \{A, B\}$; also, $\mathcal{X} \setminus \{A, B\}$: [] $\mapsto_{cst} A$ and $\mathcal{X} \setminus \{A, B\}$: [] $\mapsto_{cst} B$ do not hold. By Commutativity, if $\mathcal{X} \setminus \{A, B\}$: $A \sim B$, then $\mathcal{X} \setminus \{A, B\}$: $B \sim A$. Hence, only $\{A, B\}$ is stored in $\mathcal{C}_s^+(\mathcal{X})$ instead of both [A, B] and [B, A], since order compatibility is symmetric.

Example 8. *Let* {}: $A \sim B$, {A}: [] $\mapsto_{cst}$ C *and* $\mathcal{X} = \{A, B, C\}$. *Consider that* {C}: $A \sim B$ *and* {A}: $B \sim C$. *Hence,* {C}: $A \sim B$ *is not minimal as* {}: $A \sim B$; *therefore,* $\{A, B\} \notin \mathcal{C}_s^+(\mathcal{X})$. *Also,* {A}: $B \sim C$ *is not minimal since* {A}: [] $\mapsto_{cst}$ C; *therefore,* $\{B, C\} \notin \mathcal{C}_s^+(\mathcal{X})$.

Lemma 5. *Let* $C \in \mathcal{X}$ *and* $\mathcal{X} \setminus C$: [] $\mapsto_{cst} C$ *hold. If* $\mathcal{X}$: $A \sim B$, *then* $\mathcal{X} \setminus C$: $A \sim B$.

Proof sketch: Assume that, for some $C \in \mathcal{X}$, an OD $\mathcal{X} \setminus C$: [] $\mapsto_{cst} C$ holds. Therefore, by Propagate $\mathcal{X} \setminus C$: $A \sim C$ and $\mathcal{X} \setminus C$: $B \sim C$. Also, $\mathcal{X} \setminus C$: $A \sim B$ follows by Chain. □

Hence, $\mathcal{X}$: $A \sim B$ could not be minimal as C can be removed from the context $\mathcal{X}$. Note that *FASTOD* does not have to know whether $\mathcal{X}$: $A \sim B$ holds. As a result, we define the candidate set $\mathcal{C}_s^+(\mathcal{X})$ as follows. Note that C can be equal to A or B.

Definition 8. $\mathcal{C}_s^+(\mathcal{X}) = \{\{A, B\} \in \mathcal{X}^2 \mid A \neq B$ and $\forall_{C \in \mathcal{X}}\ \mathcal{X} \setminus \{A, B, C\}$: $A \sim B$ does not hold, and $\forall_{C \in \mathcal{X}}\ \mathcal{X} \setminus \{A, B, C\}$: [] $\mapsto_{cst} C$ does not hold$\}$

Example 9. *Assume that* {C}: [] $\mapsto_{cst}$ D *and that* FASTOD *considers* {C, D}: $A \sim B$. *Since* {C}: [] $\mapsto_{cst}$ D, *the OD* {C, D}: $A \sim B$ *is not minimal, and therefore,* $\{A, B\} \notin \mathcal{C}_s^+(\mathcal{X})$, *where* $\mathcal{X} = \{A, B, C, D\}$.

FASTOD's candidate sets do not increase in size during the execution of the algorithm (unlike *ORDER* [10]) because of the concise candidate representation. Also, we show in Section 5.3 that many "minimal" ODs in [10] are considered non-minimal (redundant) in our representation. Additional optimizations for our OD discovery algorithm are described in Sections 4.5 and 4.6.

4.3 Computing Levels

Algorithm 2 explains *calculateNextLevel*, which computes $\mathcal{L}_{l+1}$ from $\mathcal{L}_l$. It uses the subroutine *singleAttrDifferBlocks*($\mathcal{L}_l$) that partitions $\mathcal{L}_l$ into blocks (Line 2). Two sets belong to the same block if they have a common subset $\mathcal{Y}$ of length $l-1$ and differ in only one attribute, A and B, respectively. Therefore, the blocks are not difficult to calculate as sets $\mathcal{Y}$A and $\mathcal{Y}$B can be preserved as sorted sets of attributes. Unlike *FASTOD* and other usual use cases of Apriori [1] such as TANE, the OD discovery algorithm from [10] generates *all permutations* of attributes of size $l+1$ with the same *prefix* blocks of size $l-1$, which leads to its *factorial* worst-case time complexity.

Algorithm 2 calculateNextLevel($\mathcal{L}_l$)

1: $\mathcal{L}_l = []$
2: **for all** $\{\mathcal{Y}\mathsf{B}, \mathcal{Y}\mathsf{C}\} \in$ *singleAttrDiffBlocks*($\mathcal{L}_l$) **do**
3: $\quad\mathcal{X} = \mathcal{Y} \bigcup \{\mathsf{B}, \mathsf{C}\}$
4: $\quad$**if** $\forall_{\mathsf{A}\in\mathcal{X}}\ \mathcal{X} \setminus \mathsf{A} \in \mathcal{L}_l$ **then**
5: $\quad\quad$Add $\mathcal{X}$ to $\mathcal{L}_{l+1}$
6: $\quad$**end if**
7: **end for**
8: **return** $\mathcal{L}_{l+1}$

The level $\mathcal{L}_{l+1}$ contains only those sets of attributes of size $l+1$ which have all their subsets of size l in $\mathcal{L}_l$ (Line 4).

4.4 Computing Dependencies

Algorithm 3, *computeODs*($\mathcal{L}_l$), adds minimal ODs from level $\mathcal{L}_l$ to $\mathcal{M}$, in the form of $\mathcal{X} \setminus \mathsf{A}$: $[] \mapsto_{cst} \mathsf{A}$ and $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$, where $\mathsf{A}, \mathsf{B} \in \mathcal{X}$ and $\mathsf{A} \neq \mathsf{B}$. The following lemma shows that we can use the candidate set $\mathcal{C}_c^+(\mathcal{X})$ to test whether $\mathcal{X} \setminus \mathsf{A}$: $[] \mapsto_{cst} \mathsf{A}$ is minimal [9].

Lemma 6. *An OD* $\mathcal{X} \setminus \mathsf{A}$: $[] \mapsto_{cst} \mathsf{A}$, *where* $\mathsf{A} \in \mathcal{X}$, *is minimal iff* $\forall_{\mathsf{B}\in\mathcal{X}} \mathsf{A} \in \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{B})$.

Similarly, Lemma 7 states that we can use the candidate set $\mathcal{C}_s^+(\mathcal{X})$ to verify if $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$ is minimal.

Lemma 7. *The OD* $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$, *where* $\mathsf{A}, \mathsf{B} \in \mathcal{X}$ *and* $\mathsf{A} \neq \mathsf{B}$, *is minimal iff* $\forall_{\mathsf{C}\in\mathcal{X}\setminus\{\mathsf{A},\mathsf{B}\}}\ \{\mathsf{A}, \mathsf{B}\} \in \mathcal{C}_s^+(\mathcal{X} \setminus \mathsf{C})$, *and* $\mathsf{A} \in \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{B})$ *and* $\mathsf{B} \in \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{A})$.

Proof sketch: Assume first that $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$ is not minimal. Thus, there exists $\mathsf{C} \in \mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$ for which $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$: $\mathsf{A} \sim \mathsf{B}$ holds or there exists $\mathsf{D} \in \mathcal{X}$ such that $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{D}\}$: $[] \mapsto_{cst} \mathsf{D}$ holds. Then $\{\mathsf{A}, \mathsf{B}\} \notin \mathcal{C}_s^+(\mathcal{X} \setminus \mathsf{C})$, $\mathsf{A} \notin \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{B})$, or $\mathsf{B} \notin \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{A})$. Next, assume that there exists $\mathsf{C} \in \mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$ such that $\{\mathsf{A}, \mathsf{B}\} \notin \mathcal{C}_s^+(\mathcal{X} \setminus \mathsf{C})$, or $\mathsf{A} \notin \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{B})$ or $\mathsf{B} \notin \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{A})$. Therefore, $\exists \mathsf{D} \in \mathcal{X} \setminus \mathsf{C}$, such that $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$: $\mathsf{A} \sim \mathsf{B}$ or $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$: $[] \mapsto_{cst} \mathsf{C}$ or $\exists \mathsf{D} \in \mathcal{X} \setminus \mathsf{B}$ such that $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{D}\}$: $[] \mapsto_{cst} \mathsf{D}$ or $\exists \mathsf{E} \in \mathcal{X} \setminus \mathsf{A}$ such that $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$: $[] \mapsto_{cst} \mathsf{E}$. Hence, by Augmentation–II, Propagate, and Lemma 5, $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$ is not minimal. □

By Lemma 6, the steps in Lines 2, 10, 11 and 12 guarantee that the algorithm adds to $\mathcal{M}$ only the minimal ODs of the form $\mathcal{X} \setminus \mathsf{A}$: $[] \mapsto_{cst} \mathsf{A}$, where $\mathcal{X} \in \mathcal{L}_l$ and $\mathsf{A} \in \mathcal{X}$ [9].

Lemma 8. *Let* $\mathcal{C}_c^+(\mathcal{Y})$ *be correctly computed* $\forall \mathcal{Y} \in \mathcal{L}_{l-1}$. *computeODs*($\mathcal{L}_l$) *calculates correctly* $\mathcal{C}_c^+(\mathcal{X})$, $\forall \mathcal{X} \in \mathcal{L}_l$.

By Lemma 7, the steps in Lines 4, 6, 17–18 and 20–21 ensure that the algorithm adds to $\mathcal{M}$ only the minimal ODs of the form $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$, where $\mathcal{X} \in \mathcal{L}_l$, $\mathsf{A}, \mathsf{B} \in \mathcal{X}$ and $\mathsf{A} \neq \mathsf{B}$.

Algorithm 3 computeODs($\mathcal{L}_l$)

1: **for all** $\mathcal{X} \in \mathcal{L}_l$ **do**
2: $\quad\mathcal{C}_c^+(\mathcal{X}) = \bigcap_{\mathsf{A}\in\mathcal{X}} \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{A})$
3: $\quad$**if** $l = 2$ **then**
4: $\quad\quad\forall_{\mathsf{A},\mathsf{B}\in\mathbf{R}^2,\mathsf{A}\neq\mathsf{B}} \mathcal{C}_s^+(\{\mathsf{A}, \mathsf{B}\}) = \{\mathsf{A}, \mathsf{B}\}$
5: $\quad$**else if** $l > 2$ **then**
6: $\quad\quad\mathcal{C}_s^+(\mathcal{X}) = \{\{\mathsf{A}, \mathsf{B}\} \in \bigcup_{\mathsf{C}\in\mathcal{X}} \mathcal{C}_s^+(\mathcal{X} \setminus \mathsf{C}) \mid$
$\quad\quad\forall_{\mathsf{D}\in\mathcal{X}\setminus\{\mathsf{A},\mathsf{B}\}} \{\mathsf{A}, \mathsf{B}\} \in \mathcal{C}_s^+(\mathcal{X} \setminus \mathsf{D})\}$
7: $\quad$**end if**
8: **end for**
9: **for all** $\mathcal{X} \in \mathcal{L}_l$ **do**
10: $\quad$**for all** $\mathsf{A} \in \mathcal{X} \cap \mathcal{C}_c^+(\mathcal{X})$ **do**
11: $\quad\quad$**if** $\mathcal{X} \setminus \mathsf{A}$: $[] \mapsto_{cst} \mathsf{A}$ **then**
12: $\quad\quad\quad$Add $\mathcal{X} \setminus \mathsf{A}$: $[] \mapsto_{cst} \mathsf{A}$ to $\mathcal{M}$
13: $\quad\quad\quad$Remove A from $\mathcal{C}_c^+(\mathcal{X})$
14: $\quad\quad\quad$Remove all $\mathsf{B} \in \mathbf{R} \setminus \mathcal{X}$ from $\mathcal{C}_c^+(\mathcal{X})$
15: $\quad\quad$**end if**
16: $\quad$**end for**
17: $\quad$**for all** $\{\mathsf{A}, \mathsf{B}\} \in \mathcal{C}_s^+(\mathcal{X})$ **do**
18: $\quad\quad$**if** $\mathsf{A} \notin \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{B})$ or $\mathsf{B} \notin \mathcal{C}_c^+(\mathcal{X} \setminus \mathsf{A})$ **then**
19: $\quad\quad\quad$Remove $\{\mathsf{A}, \mathsf{B}\}$ from $\mathcal{C}_s^+(\mathcal{X})$
20: $\quad\quad$**else if** $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$ **then**
21: $\quad\quad\quad$Add $\mathcal{X} \setminus \{\mathsf{A}, \mathsf{B}\}$: $\mathsf{A} \sim \mathsf{B}$ to $\mathcal{M}$
22: $\quad\quad\quad$Remove $\{\mathsf{A}, \mathsf{B}\}$ from $\mathcal{C}_s^+(\mathcal{X})$
23: $\quad\quad$**end if**
24: $\quad$**end for**
25: **end for**

Lemma 9. *Let* $\mathcal{C}_c^+(\mathcal{Y})$ *and* $\mathcal{C}_s^+(\mathcal{Y})$ *be correct* $\forall \mathcal{Y} \in \mathcal{L}_{l-1}$. *computeODs*($\mathcal{L}_l$) *computes correctly* $\mathcal{C}_s^+(\mathcal{X})$, $\forall \mathcal{X} \in \mathcal{L}_l$.

Proof sketch: A pair of attributes $\{\mathsf{A}, \mathsf{B}\}$, such that $\{\mathsf{A}, \mathsf{B}\} \in \mathcal{X}^2$ and $\mathsf{A} \neq \mathsf{B}$, is in $\mathcal{C}_s^+(\mathcal{X})$ after the execution of *computeODs*($\mathcal{L}_l$) unless it is excluded from $\mathcal{C}_s^+(\mathcal{X})$ in Lines 6, 19 or 22. Thus, we can prove correctness by first showing that if $\{\mathsf{A}, \mathsf{B}\}$ is excluded from $\mathcal{C}_s^+(\mathcal{X})$ by *computeODs*($\mathcal{L}_l$), then $\{\mathsf{A}, \mathsf{B}\} \notin \mathcal{C}_s^+(\mathcal{X})$ by the definition of $\mathcal{C}_s^+(\mathcal{X})$. Next, by showing that if $\{\mathsf{A}, \mathsf{B}\} \notin \mathcal{C}_s^+(\mathcal{X})$ by the definition of $\mathcal{C}_s^+(\mathcal{X})$, then A is excluded from $\mathcal{C}_s^+(\mathcal{X})$ by *computeODs*($\mathcal{L}_l$). □

4.5 Pruning Levels and Completeness

Algorithm 4 shows *pruneLevels*($\mathcal{L}_l$), which implements an additional optimization. We delete node $\mathcal{X}$ from $\mathcal{L}_l$, where $l \geq 2$, if both $\mathcal{C}_c^+(\mathcal{X}) = \{\}$ and $\mathcal{C}_s^+(\mathcal{X}) = \{\}$.

Algorithm 4 pruneLevels($\mathcal{L}_l$)

1: **for all** $\mathcal{X} \in \mathcal{L}_l$ **do**
2: $\quad$**if** $l \geq 2$ **then**
3: $\quad\quad$**if** $\mathcal{C}_c^+(\mathcal{X}) = \{\}$ and $\mathcal{C}_s^+(\mathcal{X}) = \{\}$ **then**
4: $\quad\quad\quad$Delete $\mathcal{X}$ from $\mathcal{L}_l$
5: $\quad\quad$**end if**
6: $\quad$**end if**
7: **end for**

Lemma 10. *Deleting* $\mathcal{X}$ *from* $\mathcal{L}_l$ *for levels* $l \geq 2$, *if* $\mathcal{C}_c^+(\mathcal{X}) = \{\}$ *and* $\mathcal{C}_s^+(\mathcal{X}) = \{\}$ *has no effect on the output set of minimal ODs* $\mathcal{M}$.

Proof sketch: Follows from the fact that for all $\mathcal{Y}$ such that $\mathcal{Y} \supset \mathcal{X}$, $\mathcal{C}_c^+(\mathcal{Y}) = \{\}$ and $\mathcal{C}_s^+(\mathcal{Y}) = \{\}$. □

Example 10. *Let* A: $[] \mapsto_{cst} \mathsf{B}$, B: $[] \mapsto_{cst} \mathsf{A}$ *and* $\{\}$: $\mathsf{A} \sim \mathsf{B}$. *Since* $\mathcal{C}_c^+(\{\mathsf{A}, \mathsf{B}\}) = \{\}$ *and* $\mathcal{C}_s^+(\{\mathsf{A}, \mathsf{B}\})$ *as well as* $l = 2$, *by the pruning levels rule* (*Lemma 10*), *the node* $\{\mathsf{A}, \mathsf{B}\}$ *is deleted and the node* $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ *is not considered*

(*see Figure 5*). *This is justified as* {AB}: [] $\mapsto_{cst}$ C *is not minimal by Lemma 4,* {AC}: [] $\mapsto_{cst}$ B *is not minimal by Augmentation–I,* {BC}: [] $\mapsto_{cst}$ A *is not minimal by Augmentation–I,* {C}: A $\sim$ B *is not minimal by Augmentation–II,* {A}: B $\sim$ C *is not minimal by Propagate, and* {B}: A $\sim$ C *is not minimal by Propagate.*

THEOREM 7. *The FASTOD algorithm computes a complete, minimal set of ODs $\mathcal{M}$.*

Proof sketch: The algorithm *computeODs*($\mathcal{L}_l$) adds to $\mathcal{M}$ only the minimal ODs. (See the discussion in Section 4.3.) It can also be shown by induction that *computeODs*($\mathcal{L}_l$) calculates correctly the sets $\mathcal{C}_c^+(\mathcal{X})$ and $\mathcal{C}_s^+(\mathcal{X})$ for *all* $\mathcal{X} \in \mathcal{L}_l$, since Lemmas 8 and 9 hold. Deleting $\mathcal{X}$ from $\mathcal{L}_l$ for levels $l \geq 2$ if $\mathcal{C}_c^+(\mathcal{X}) = \{\}$ and $\mathcal{C}_s^+(\mathcal{X}) = \{\}$ has no effect on the output set of minimal ODs $\mathcal{M}$ of the algorithm by Lemma 10. Therefore, the *FASTOD* algorithm computes a sound and complete set of minimal ODs $\mathcal{M}$. □

While we guarantee that FASTOD finds a complete set of ODs (Sections 4.4–4.5), the ORDER algorithm in [10] intentionally omits some ODs for efficiency. When *ORDER* accesses the node [A, B, C] in the list-containment lattice, it generates the list-based ODs [B, C] $\mapsto$ [A] and [C] $\mapsto$ [A, B] and verifies if there are any splits and swaps (Definitions 4 and 5). However, it discards potential ODs with repeating attributes [10]. For instance, a valid OD [C] $\mapsto$ [C, A, B] (note, an FD) is missed, if [C] $\sim$ [A, B] does not hold. Similarly, a valid OD [C] $\sim$ [A, B] (an order compatible dependency) is missed, if [C] $\mapsto$ [C, A, B] does not hold. This causes the algorithm to discard ODs such as d_month $\sim$ d_week from Example 2 as d_month does not *functionally determine* d_week. Only if both [C] $\mapsto$ [C, A, B] and [C] $\sim$ [A, B] hold, then they are not missed, as just [C] $\mapsto$ [A, B] is checked by the *ORDER* algorithm. (By Theorem 1, [C] $\mapsto$ [A, B] *iff* [C] $\mapsto$ [C, A, B] and [C] $\sim$ [A, B].) Also, *ORDER* misses potential ODs with the same prefix; i.e., **XY** $\mapsto$ **XZ**, such as [year, salary] $\mapsto$ [year, bin] (Example 4). Furthermore, *ORDER* discards constants; i.e., ODs in the form of [] $\mapsto_{cst}$ **X**. For instance, if all data are from the year 2012, the year attribute is a constant.

4.6 Efficient OD Validation

Computing with Partitions. In order to verify whether the ODs $\mathcal{X}$: [] $\mapsto_{cst}$ A and $\mathcal{X}$: A $\sim$ B hold, we first compute the equivalence classes over the context $\mathcal{X}$, i.e., $\Pi_{\mathcal{X}}$. Next, within each equivalence class in $\Pi_{\mathcal{X}}$, we verify whether the ODs $\mathcal{X}$: [] $\mapsto_{cst}$ A and $\mathcal{X}$: A $\sim$ B hold.

The values of the columns are replaced with integers: 1, ..., n, in a way that the equivalence classes do not change and the ordering is preserved. That is, the same values are substituted by the same integers, and higher values are replaced by larger integers. Computation over integers is more time and space efficient. The value t_A is used as the identifier of the equivalence class $\mathcal{E}(t_A)$ of Π_A.

After computing the partitions Π_A, Π_B, ..., for all single attributes in **R** at level $l = 2$, we efficiently compute the partitions for subsequent levels in linear time by taking the product of refined partitions, i.e., $\Pi_{A \cup B} = \Pi_A \cdot \Pi_B$. Accordingly, for all other levels the partitions are not calculated from scratch for each set of attributes $\mathcal{X}$. In the general case, for all $|\mathcal{X}| \geq 2$, partitions $\Pi_{\mathcal{X}}$ are computed in linear time as products of partitions $\Pi_{\mathcal{Y}}$ and $\Pi_{\mathcal{Z}}$, i.e., $\Pi_{\mathcal{X}} = \Pi_{\mathcal{Y}} \cdot \Pi_{\mathcal{Z}}$, such that $\mathcal{Y}, \mathcal{Z} \subset \mathcal{X}$ and $|\mathcal{Y}| = |\mathcal{Z}| = |\mathcal{X} - 1|$.

τ_A Rank	Tuple #'s
1	{*t3, t5, t8*}
2	{*t1, t6*}
3	{*t4*}
4	{*t7*}
5	{*t2*}

(a) τ_A

$\Pi_{\mathcal{X}}$ ID	$\tau_A(\mathcal{E}(t_{\mathcal{X}}))$
$t1_{\mathcal{X}}$	{*t1*}
$t2_{\mathcal{X}}$	{*t2*}
$t3_{\mathcal{X}}$	{*t3, t5*}, {*t4*}
$t6_{\mathcal{X}}$	{*t6*}, {*t7*}
$t8_{\mathcal{X}}$	{*t8*}

(b) $\tau_A(\mathcal{E}(t_{\mathcal{X}}))$

Table 2: $\Pi_{\mathcal{X}} = \{\{t1\}, \{t2\}, \{t3, t4, t5\}, \{t6, t7\}, \{t8\}\}$, $\tau_A = \{\{t3, t5, t8\}, \{t1, t6\}, \{t4\}, \{t7\}, \{t2\}\}$

This is beneficial for a levelwise algorithm since only partitions from the previous level are needed.

To verify whether an OD $\mathcal{X}$: [] $\mapsto_{cst}$ A holds, for each equivalence class $\mathcal{E}(t_{\mathcal{X}}) \in \Pi_{\mathcal{X}}$, it is sufficient to check whether $|\Pi_A(\mathcal{E}(t_{\mathcal{X}}))| = 1$, which requires a single scan over the dataset (linear time operation in the number of tuples). However, there is an even simpler test to validate whether $\mathcal{X}$: [] $\mapsto_{cst}$ A holds: check if $|\Pi_{\mathcal{X}}| = |\Pi_{\mathcal{X} \cup A}|$ [9]. Verifying if an OD $\mathcal{X}$: A $\sim$ B holds is more involved but can also be computed efficiently. For all single attributes A $\in$ **R** at level $l = 2$, we calculate *sorted partitions* τ_A. A sorted partition τ_A is a list of equivalence classes according to the ordering imposed on the tuples by A. For instance, in Table 1, $\tau_{bin} = \{\{t1, t4\}, \{t2, t5\}, \{t3, t6\}\}$. Hence, when we verify for any subsequent level, whether $\mathcal{X}$: A $\sim$ B is valid, for each equivalence class $\mathcal{E}(t_{\mathcal{X}}) \in \Pi_{\mathcal{X}}$, we compute $\tau_A(\mathcal{E}(t_{\mathcal{X}}))$ by hashing tuples into sorted buckets with a single scan over τ_A (see the example in Table 2). This allows us to verify efficiently whether there is no swap over attributes A and B for each equivalence class in $\Pi_{\mathcal{X}}$ via a single scan. Hence, checking whether $\mathcal{X}$: A $\sim$ B can be done in linear time.

Key Pruning. When a key is found during the search of ODs, additional optimization methods can be applied.

LEMMA 11. *Let $\mathcal{X} \setminus A$ be a superkey and A $\in \mathcal{X}$. The OD $\mathcal{X} \setminus A$: [] $\mapsto_{cst}$ A is valid. The OD $\mathcal{X} \setminus A$: [] $\mapsto$ A is minimal iff $\mathcal{X} \setminus A$ is a key and $\forall B \in \mathcal{X}$, A $\in \mathcal{C}_c^+(\{\mathcal{X}\} \setminus B)$.*

Typically, an OD $\mathcal{X} \setminus A$: [] $\mapsto_{cst}$ A, A $\notin \mathcal{X}$, is verified in Line 11 of *computeODs*($\mathcal{L}_l$). However, if $\mathcal{X} \setminus A$ is a key, then $\mathcal{X} \setminus A$: [] $\mapsto_{cst}$ A always holds, and hence, we do not need to verify it. On the other hand, if $\mathcal{X} \setminus A$ is a superkey but not a key, then clearly the OD $\mathcal{X} \setminus A$: [] $\mapsto_{cst}$ A is not minimal. This is because there exists B $\in \mathcal{X}$, such that $\mathcal{X} \setminus B$ is a superkey, therefore, A $\notin \mathcal{C}_c^+(\{\mathcal{X}\} \setminus B)$).

Lemma 12 states that $\mathcal{X} \setminus \{A, B\}$: A $\sim$ B, A, B $\in \mathcal{X}$, also does not have to be verified in Line 20 of *computeODs*($\mathcal{L}_l$), if $\mathcal{X} \setminus \{A, B\}$ is a key since it is not minimal (Propagate).

LEMMA 12. *Let $\mathcal{X}$ be a key (or superkey) and A, B $\in$ **R** $\setminus$ $\mathcal{X}$. The OD $\mathcal{X}$: A $\sim$ B is valid but not minimal.*

Stripped Partitions. We replace partitions with a more compact version called *stripped partitions*. A stripped partition is a partition with equivalence classes of size one–call these singleton equivalence classes–discarded. A stripped representation of partition $\Pi_{\mathcal{X}}$ is denoted as $\Pi^*_{\mathcal{X}}$.

EXAMPLE 11. *In Table 1, $\Pi^*_{salary} = \{\{t2, t6\}\}$, whereas $\Pi_{salary} = \{\{t1\}, \{t2, t6\}, \{t3\}, \{t4\}, \{t5\}\}$.*

Removing singleton equivalence classes is correct as they cannot break any set-based ODs.

LEMMA 13. *Singleton equivalence classes over attribute set $\mathcal{X}$ cannot falsify any OD: $\mathcal{X}$: [] $\mapsto_{cst}$ A or $\mathcal{X}$: A $\sim$ B.*

If $\Pi^*_{\mathcal{X}} = \{\}$, then $\mathcal{X}$ is a superkey and the optimization with key pruning is triggered. The authors in [10] state that stripped partitions cannot be used with their list-based canonical form.

4.7 Complexity and Interestingness

Complexity Analysis. The previous state-of-the-art discovery algorithm for ODs, *ORDER*, has a *factorial* worst-case complexity in the number of attributes [10], as it traverses a lattice of attribute *permutations* with $\lfloor |\mathbf{R}|! * e \rfloor$ nodes. The authors in [10] claim that this is inevitable, due to the factorial search space of candidates, since ODs are defined over lists of attributes as opposed to FDs being defined over sets of attributes. We show that the worst-case complexity of our OD discovery algorithm is *exponential* in the number of attributes, as there exists a quadratic mapping of list-based ODs into the equivalent set-based ODs (Section 3.1), and there are $2^{|\mathbf{R}|}$ nodes in the set-containment lattice (Sections 4.1 and 4.3). This establishes a lower bound as ODs subsume FDs, and it has been already observed for FDs that the solution space is exponential, and a polynomial time algorithm cannot exist [9]. Additionally, the complexity is linear in the number of tuples (Section 4.6).

Interestingness. Given that ODs properly subsume FDs [20], the search space and the number of discovered ODs may be larger than for FDs even after pruning non-minimal ODs. To decrease the cognitive burden of human verification, we propose a new measure for *interestingness* of ODs based on their *coverage*, which can be used to rank them. Given an OD φ, $\mathcal{X}$: $[] \mapsto_{cst}$ A or $\mathcal{X}$: A $\sim$ B, we define interestingness, abbreviated *Inter*, as the number of tuple pairs covered by the OD (with respect to the context $\mathcal{X}$); i.e., the sum of squares of the sizes of the equivalence classes: $Inter(\varphi) = \sum_{\mathcal{E}(t_{\mathcal{X}}) \in \Pi_{\mathcal{X}}} |\mathcal{E}(t_{\mathcal{X}})|^2$.

Example 12. *In Table 1,* $Inter(\{\text{year}\}: \text{bin} \sim \text{salary}) = 3^2 + 3^2 = 18$ *and* $Inter(\{\text{year}, \text{group}\} : \text{bin} \sim \text{salary}) = 4 * 1^2 + 2^2 = 8$*; hence, the first OD is more interesting.*

Our interestingness metric prefers ODs with large equivalence classes. Observe that ODs with more attributes (with the same prefixes over the context) naturally have more unique equivalence classes. Thus, since stripped partitions (Lemma 13) do not violate any ODs, the discovered ODs are not necessary meaningful. This is also recognized as *overfitting*, and occurs when an OD is excessively complex. Whereas this approach is mainly data driven, it also takes into account indirectly the *succinctness* of rules; i.e., the size of ODs (schema size).

Approximate ODs. In practice, ODs may not hold exactly, perhaps due to errors in the data. We thus define approximate ODs that hold with some exceptions. As in prior work on approximate FD discovery, we compute the minimum number of tuples that must be removed from the given table for the OD to hold. We define the problem of discovering approximate ODs as follows: given a table $\mathbf{t}$ and a threshold ϵ, $0 \leq \epsilon \leq 1$, find all minimal ODs φ, such that $e(\varphi) \leq \epsilon$, where $e(\varphi) = min\{|\mathbf{r}| \mid \mathbf{r} \subseteq \mathbf{t}, \mathbf{t} \setminus \mathbf{r} \models \varphi\}/|\mathbf{t}|$.

The modification to the regular OD discovery algorithm needed to search for approximate ODs is the data verification step. The error $e(\mathcal{X}: [] \mapsto_{cst} \text{A})$ is computed from the partitions $\Pi_{\mathcal{X}}$ and $\Pi_{\mathcal{X}\cup\text{A}}$ [9]. Any equivalence class $\mathcal{E}(t_{\mathcal{X}})$ of $\Pi_{\mathcal{X}}$ is the union of one or more equivalence classes $\mathcal{E}_1(t_{\mathcal{X}\cup\text{A}})$, ..., $\mathcal{E}_n(t_{\mathcal{X}\cup\text{A}})$ of $\Pi_{\mathcal{X}\cup\text{A}}$. The minimum number of tuples to be removed over each equivalence class of $\Pi_{\mathcal{X}}$ is the size $|\mathcal{E}(t_{\mathcal{X}})|$ minus the size of the largest $\mathcal{E}_k(t_{\mathcal{X}\cup\text{A}})$ of $\Pi_{\mathcal{X}\cup\text{A}}$. A similar argument applies to computing the error $e(\mathcal{X}: \text{A} \sim \text{B})$. The minimum number of tuples to be removed over each equivalence class over of $\Pi_{\mathcal{X}}$ is the size $|\mathcal{E}(t_{\mathcal{X}})|$ minus the sum of the iteratively removed tuples (for an OD to hold) with the highest number of swaps with respect to attributes A and B over sorted partitions $\tau_{\text{A}}(\mathcal{E}(t_{\mathcal{X}}))$.

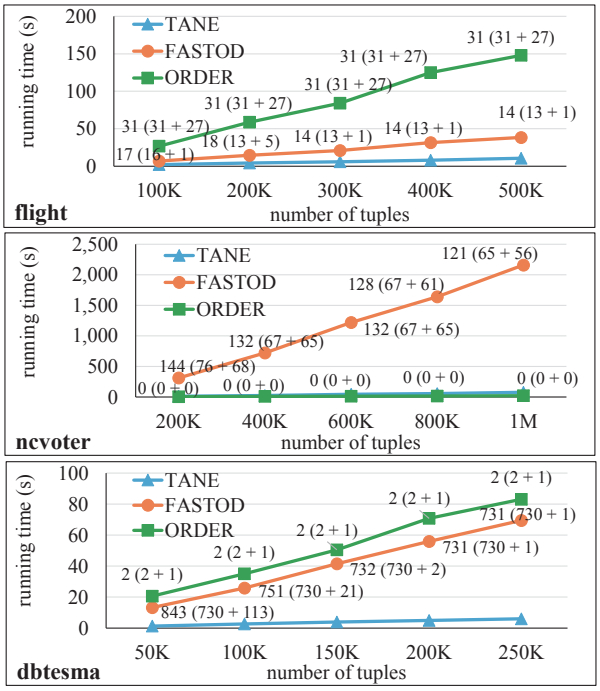

Figure 6: Scalability and effectiveness in $|\mathbf{r}|$.

5. EXPERIMENTS

In this section, we present an experimental evaluation of our techniques. Our experiments were run on a machine with an Intel Xeon CPU E5-2630 v3 2.4GHz with 64GB of memory. The algorithm was implemented in Java 8.

5.1 Data Characteristics

We use several real and synthetic datasets that have previously been used to evaluate FD and OD discovery algorithms [10]. They are published through the UCI Machine Learning Repository[3] and the Hasso-Plattner-Institute (HPI) repository[4]. The `flight` dataset (from the HPI repository) contains information about US domestic flights, with 500K tuples and 30 attributes (and 40 attributes over 1K tuples). The `ncvoter` dataset (from the ncsbe.gov repository) contains personal data of registered voters from North Carolina, with 1M tuples and 20 attributes. The `hepatitis` dataset provides information about the hepatitis disease of the liver, with 155 tuples and 20 attributes. `Dbtesma` is a synthetic dataset with 250K tuples and 30 attributes, available at the link provided for the HPI repository, produced by a data generator[5] for benchmarks and performance analysis.

[3] http://archive.ics.uci.edu/ml
[4] http://metanome.de
[5] http://sourceforge.net/projects/dbtesma

5.2 Scalability

Exp-1: Scalability in |r|. We measure the running time of *FASTOD* in milliseconds by varying the number of tuples, as reported in Figure 6. For now, ignore the other curves labeled ORDER and TANE, as well as the numbers next to the datapoints. We use random samples of 20, 40, 60, 80 and 100 percent of `flight`, `ncvoter` and `dbtesma`, all with 10 attributes for comparison purposes with *ORDER* [10], which does not terminate after 5 hours on larger schemas (details in Exp-3). Furthermore, on the entire available `flight` dataset schema with 30 attributes and 500K, both *TANE* and *FASTOD* run out of memory. In such cases, we recommend to set a threshold for the measure of interestingness to prune the rules and/or limit the order dependencies to be discovered only over the top levels of the lattice. The reported runtimes are averaged over ten executions. We do not use `hepatitis` in this experiment because it is too small for testing scalability in the number of tuples.

Figure 6 shows a linear runtime growth as computation is dominated by the verification of ODs which requires a scan of the dataset. Thus, our algorithm scales well for large datasets. Also, it appears to run faster on `flight` than on `ncvoter`. This is due to the varying effectiveness of our pruning strategies (details in Section 5.4).

Exp-2: Scalability in |R|. We measure the running time of *FASTOD* in milliseconds by varying the number of attributes by taking random projections of the tested datasets. We use the `flight` dataset with 1K tuples, the `ncvoter` dataset with 1K tuples, the `hepatitis` dataset with 155 tuples and the `dbtesma` dataset with 1K tuples. Again, we report the average runtimes over ten executions. Figure 7 shows that the running time increases exponentially with the number of attributes. (The Y axis of Figure 7 is in log scale.) This is not surprising because the number of minimal ODs over the set-containment lattice is exponential in the worst case (Section 4.7).

5.3 Comparative Study

Exp-3: Comparison with ORDER [10]. We now compare *FASTOD* with *ORDER* from [10]. We obtained a Java implementation of *ORDER* from www.metanome.de. As Figures 6 and 7 show, our algorithm can be orders of magnitude faster. For instance, on the `flight` dataset with 1K tuples and 20 attributes, *FASTOD* finishes the computation in less than 1 second, whereas *ORDER* did not terminate after 5 hours. (We aborted the experiments after 5 hours and denote it in figures as "* 5h".) The running time of *ORDER* is consistent with results reported in [10]. Similar observations can be made on the `dbtesma` dataset. This is expected as the worst-time complexity for *ORDER* is factorial in the number of attributes [10], whereas, for *FASTOD*, it is exponential in the number of attributes (Section 4.7).

The higher complexity of *ORDER* is remedied in some cases by its aggressive pruning strategies. For instance, the *swap pruning rule* states that if $\mathbf{X} \mapsto \mathbf{Y}$ is invalidated by a swap, then $\mathbf{XA} \mapsto \mathbf{YB}$ is not valid and is not considered when traversing the list-containment lattice. As a result, the *ORDER* algorithm misses ODs of the form $\mathbf{XA} \mapsto \mathbf{XAYB}$, e.g., $\mathcal{X}$A: [] $\mapsto_{cst}$ B is missed (recall Section 4.5). It also skips ODs in the form of $\mathbf{X} \mapsto \mathbf{XY}$. Similarly, if $\mathbf{XA} \mapsto \mathbf{YB}$ is invalidated by a split, then $\mathbf{XA} \sim \mathbf{YB}$ is not considered; e.g., $\mathcal{XY}$: A $\sim$ B is missed.

These observations explain why *ORDER* is faster than

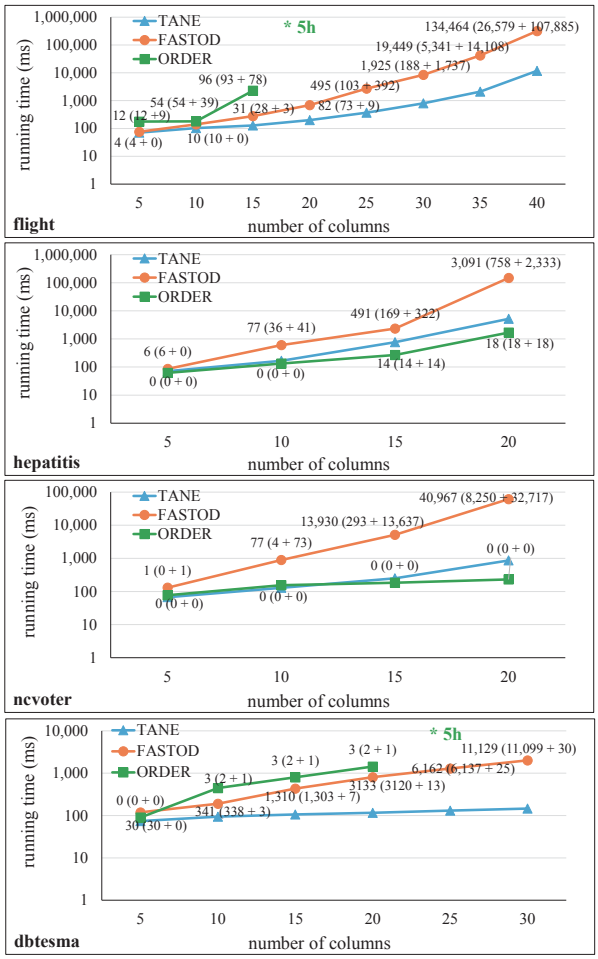

Figure 7: Scalability and effectiveness in |R|.

FASTOD (and even *TANE*) on the `hepatitis` dataset: here, *ORDER* discovers zero ODs, whereas *FASTOD*, which is sound and complete, discovers hundreds. Similar reasoning explains the runtimes over the `ncvoter` dataset.

While the swap pruning rule (and other pruning rules) in the *ORDER* algorithm can be deactivated to enable completeness, this has an extreme performance impact. For example, after disabling the swap pruning rule, the *ORDER* algorithm did not terminate within five hours in any of tested datasets. In fact, in [10] they have shown the same.

Also, our canonical representation is significantly more concise than the one in [10] even though *FASTOD* is complete. For brevity, we refer to set-based ODs in the form of $\mathcal{X} \setminus$ A: [] $\mapsto_{cst}$ A as FDs and $\mathcal{X} \setminus \{$A, B$\}$: A $\sim$ B as order compatible dependencies (OCDs). We report the number of set-based ODs (number of FDs and number of OCDs) in Figure 6 and 7 next to the runtime datapoints. For instance, for the `flight` dataset with 500K tuples and 10 attributes, the number of discovered set-based ODs by *FASTOD* is 14 (13 FDs and 1 OCD) and list-based ODs by *ORDER* is 31, which maps to 58 set-based ODs (31 FDs and 27 OCDs). Since all flight data are from the year 2012, the year attribute is a constant. Therefore, *FASTOD* discovers an OD {}: [] $\mapsto_{cst}$ [year]; however, *ORDER* produces ODs [quarter] $\mapsto$ [year], [dayOfMonth] $\mapsto$ [year], [dayOfWeek] $\mapsto$ [year] and so on, which follow from {}: [] $\mapsto_{cst}$ [year] by Aug–I and Propagate (but not vice versa!). For example,

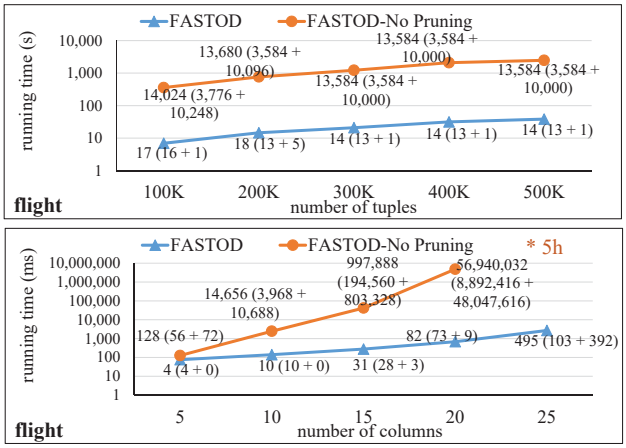

Figure 8: Impact of pruning.

{quarter}: [] $\mapsto_{cst}$ [year] can be inferred from Aug–I, and {}: [quarter] $\sim$ [year] can be deduced from Propagate.

Exp-4: Comparison with TANE [9]. Since ODs are closely related to FDs, as ODs properly subsume FDs, we also conduct a comparative study with *TANE* [9] (Java implementation). Here, we effectively measure the extra cost to capture the additional OD semantics: those of order. As expected, *TANE* is faster than *FASTOD* (see Figures 6 and 7). TANE uses a technique called *error rate* [9] to speed up the verification over FDs (detecting splits) that is not applicable to ODs in the form of $\mathcal{X} \setminus \{A, B\}$: A $\sim$ B (detecting swaps). However, both *TANE* and *FASTOD* scale linearly in the number of tuples and exponentially in the number of attributes. We argue that the extra cost of OD discovery is worthwhile, as ODs can convey many business rules that FDs cannot express. For instance, in the *flight* dataset with 1K tuples and 25 attributes, we found around 500 set-based minimal ODs, out of which around 100 are in the form of $\mathcal{X} \setminus$ A: [] $\mapsto_{cst}$ A (FDs) (the number of FDs detected by *TANE* and *FASTOD* is the same) and around 400 are in the form of $\mathcal{X} \setminus \{A, B\}$: A $\sim$ B (OCDs).

This experiment showed that the number of discovered ODs increases by an order of magnitude when we move from large to small datasets (in the number of tuples). In Exp-8, we will explore how to rank and prune ODs using our interestingness measure from Section 4.7.

5.4 Optimizations and Lattice Levels

Exp-5: Improving performance. We now quantify the runtime improvements due to the optimizations described in Sections 4.2 and 4.5. We perform this experiment over the `flight` dataset up to 500K tuples (and 10 attributes) as well as up to 25 attributes (and 1K tuples). Results are shown in Figure 8 for *FASTOD* and *FASTOD-No Pruning*; the Y axis is in log scale. Our optimizations provide a substantial performance improvement. For instance, the running time over the `flight` dataset with 1K tuples and 20 attributes drops from 80 minutes to less than 1 second. For the same dataset with 25 attributes, the running time drops from intractable (did not terminate after 5 hours) to under 3 seconds.

Exp-6: Pruning non-minimal ODs. We show that our optimizations prune a significant number of non-minimal ODs. Figure 8 reports the numbers of ODs with and without redundant ODs for the `flight` dataset. Our canonical representation can prune a large amount of inferred ODs. For instance, over the flight dataset with 1K tuples and 20 attributes, there are around 700 minimal ODs discovered (with pruning activated) and around 50 million non-minimal ODs (with pruning deactivated). Thus, our canonical representation is highly effective in avoiding redundancy.

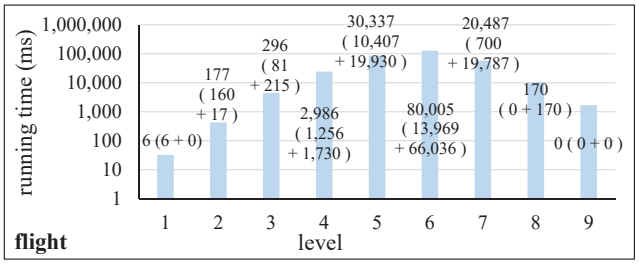

Figure 9: Experiment with lattice levels, $|\mathbf{r}| = 1$K, $|\mathbf{R}| = 40$.

Exp-7: Effectiveness over lattice levels. Here, we measure the running time and the number of ODs discovered at different levels of the lattice (Figure 9 with log scale on Y axis). Note that the level number (l) determines the size of the context for set-based ODs; i.e., the higher the level, the larger the context. We report results over the `flight` dataset with 1K tuples and 40 attributes. We believe that ODs with a smaller context are more interesting; for instance, they are more likely to be used in query optimization [19, 20]. Note that FASTOD discards redundant ODs if they can be inferred by ODs detected at lower levels of the lattice. Interestingly, most of the ODs are discovered efficiently over the first few levels of the lattice. Level 9 was the highest level for which FASTOD generated candidates. Since the set-lattice is diamond-shaped (Figure 5) and nodes are pruned over time, the time to process each level first increases, up to level 6, and decreases thereafter.

Exp-8: Evaluation of the *Inter* score. We conduct this experiment over the `flight` dataset with 100K tuples and 30 attributes (since the number of discovered ODs over 500K tuples and 10 attributes is small). We first report the precision of discovered ODs with $\mathcal{M}_{Top\text{-}K}$ being the Top-K ODs according to our *Inter* score (Section 4.7). We define *precision* as: (*# Meaningful ODs in* $\mathcal{M}_{Top\text{-}K}$) / $|\mathcal{M}_{Top\text{-}K}|$, where "*# Meaningful ODs in* $\mathcal{M}_{Top\text{-}K}$" is the number of meaningful (not overfitted) ODs in $\mathcal{M}_{Top\text{-}K}$ verified manually by experts (graduate students in computer science well versed in databases). The precision of our algorithm over the flight dataset is 88%. (The parameter K was set to TOP-100.) This experiment demonstrates that the proposed scoring function that combines coverage and (indirectly) succinctness can identify interesting ODs.

Next, we report the algorithm runtime with and without using a minimum *Inter* score as a pruning rule. The higher is the *Inter* threshold, the more aggressive is the pruning. We adjusted this threshold manually so TOP-100 ODs are selected. The running time over the `flight` dataset with the *Inter* score pruning strategy activated is an order of magnitude faster over 25K tuples and 30 attributes (39 sec versus 535 sec). For 100K tuples and 30 attributes, the algorithm with pruning finishes in 639 seconds and without pruning it runs out of memory. Thus, this optimization provides a substantial performance improvement.

6. RELATED WORK

Pointwise ODs were introduced in the context of database systems by Ginsburg and Hull [6]. In a pointwise ordering,

an instance of a database satisfies a pointwise order dependency $\mathcal{X} \rightsquigarrow \mathcal{Y}$ if, for all tuples s and t, for every attribute A in $\mathcal{X}$, $s[\mathsf{A}]$ *op* $t[\mathsf{A}]$ implies that for every attribute B in $\mathcal{Y}$ $s[\mathsf{B}]$ *op* $t[\mathsf{B}]$, where *op* $\epsilon\{<, >, \leq, \geq, =\}$. A sound and complete set of axioms was presented, and the inference problem was proven to be co-NP complete. Lexicographical ODs were studied in [12, 17, 20]. Ng [12] developed a theory of lexicographical ODs as well as a simpler version of pointwise ODs. Szlichta et al. [17] presented a sound and complete axiomatization for ODs. Bidirectional ODs, containing a mix of ascending and descending orders, were introduced by Szlichta et al. [20], where it was shown that the inference problem for both ODs and bidirectional ODs is co-NP-complete.

Golab et al. [7] introduced *sequential dependencies* (SDs), which specify that when tuples have consecutive antecedent values, their consequents must be within a specified range. The discovery problem was studied for approximate SDs [7].

Denial Constraints (DCs) [2], a universally quantified first-order logic formalism, are more expressive than FDs and ODs. DCs subsume pointwise ODs as they allow six operators for comparison of tuples $\{<, >, \leq, \geq, =, \neq\}$ (with $\neq$ as the addition). DCs also subsume lexicographical ODs as it is shown in [20] that pointwise ODs properly subsume lexicographical ODs. While the search space of FD and OD discovery is $2^{|\mathbf{R}|}$, the search space for DC discovery is much larger, at $O(2^{|\mathbf{R}|^2})$. Note that $2^{|\mathbf{R}|^2}$ grows faster than even factorial $|\mathbf{R}|!$, the runtime of *ORDER* in [10]. The authors in [2] design a set of (incomplete) axioms to develop pruning rules for DC discovery.

There has been much research on FD discovery [9, 13]. However, there has, to date, been little work on discovery of ODs. Langer and Naumann [10] propose the algorithm *ORDER* that uses list-containment lattice with factorial worst-case time complexity in the number of attributes, which establishes an upper bound. However, OD discovery is no harder than FD discovery as we show in Section 4.7.

Sorting is at the heart of many database operations; such as sort-merge join, index generation, duplicate elimination and order-by. The importance of sorted sets for query optimization and processing had been long recognized. The seminal query optimizer System R [14] paid particular attention to *interesting orders* by keeping track of all such ordered sets throughout the process of query optimization. FDs and ODs were shown to help generate interesting orders in [15] and in [20, 19], respectively. In [8], the authors explored the use of sorted sets for executing nested queries.

The significance of sorted sets has prompted researchers to look *beyond* the sets that have been explicitly generated. Malkemus et al. [11] and Szlichta et al. [19] showed how to use sorted sets created as generated columns (SQL functions and algebraic expressions) in predicates for query optimization. Relationships between sorted attributes discovered by reasoning over the physical schema have been also used to eliminate joins over data warehouse queries [18].

7. CONCLUSIONS

We presented an efficient algorithm for discovering ODs, which we showed to be substantially faster than the prior state-of-the-art. The technical innovation that made our algorithm possible is a novel mapping into a set-based canonical form and an axiomatization for set-based canonical ODs.

In future work, we plan to extend our OD discovery framework to bidirectional ODs [20] and conditional ODs that hold over portions of a relation. A conditional OD can be represented as a pair: embedded OD and range pattern tableau, that defines the rows over which the embedded dependency applies. Furthermore, while we have outlined a straightforward approach for discovery of approximate ODs in this paper, we plan to investigate more effective and efficient approaches in the future.

8. REFERENCES

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining, AAAI Press*, pages 307–328, 1996.

[2] X. Chu, I. Ilyas, and P. Papotti. Discovering Denial Constraints. *PVLDB*, 6(13):1498–1509, 2013.

[3] X. Chu, I. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[4] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

[5] J. Dong and R. Hull. Applying Approximate Order Dependency to Reduce Indexing Space. In *SIGMOD, 119-127*, 1982.

[6] S. Ginsburg and R. Hull. Order Dependency in the Relational Model. *TCS, 26(1): 149-195*, 1983.

[7] L. Golab, H. Karloff, F. Korn, and D. Srivastava. Sequential Dependencies. *PVLDB, 2(1): 574-585*, 2009.

[8] R. Guravannavar, H. Ramanujam, and S. Sudarshan. Optimizing Nested Queries with Parameter Sort Orders. In *VLDB, 481-492*, 2005.

[9] Y. Huhtala, J. Krkkinen, P. Porkka, and H. Toivonen. Efficient Discovery of Functional and Approximate Dependencies Using Partitions. In *ICDE*, pages 392–401, 1998.

[10] P. Langer and F. Naumann. Efficient order dependency detection. *VLDB J.*, 25(2):223–241, 2016.

[11] T. Malkemus, S. Padmanabhan, B. Bhattacharjee, and L. Cranston. Predicate Derivation and Monotonicity Detection in DB2 UDB. In *ICDE, 939-947*, 2005.

[12] W. Ng. An Extension of the Relational Data Model to Incorporate Ordered Domains. *TODS, 26(3) 344-383*, 2001.

[13] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J. Rudolph, M. Schnberg, J. Zwiener, and F. Naumann. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *PVLDB*, 8(10):1082–1093, 2015.

[14] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD, 23-34*, 1979.

[15] D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *SIGMOD, 57-67*, 1996.

[16] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *Technical report, 14 pages, http://arxiv.org/abs/1608.06169*, 2016.

[17] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of Order Dependencies. *PVLDB, 5(11): 1220-1231*, 2012.

[18] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, P. Pawluk, and C. Zuzarte. Queries on Dates: Fast Yet not Blind. In *EDBT, 497-502*, 2011.

[19] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, W. Qiu, and C. Zuzarte. Business-Intelligence Queries with Order Dependencies in DB2. In *EDBT, 750-761*, 2014.

[20] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. Expressiveness and Complexity of Order Dependencies. *PVLDB 6(14): 1858-1869*, 2013.