Discovering Order Dependencies through Order Compatibility

Cristian Consonni
University of Trento
cristian.consonni@unitn.it

Paolo Sottovia
University of Trento
ps@disi.unitn.it

Alberto Montresor
University of Trento
alberto.montresor@unitn.it

Yannis Velegrakis
Utrecht University and University of Trento
velgias@disi.unitn.eu

ABSTRACT

A relevant task in the exploration and understanding of large datasets is the discovery of hidden relationships in the data. In particular, functional dependencies have received considerable attention in the past. However, there are other kinds of relationships that are significant both for understanding the data and for performing query optimization. Order dependencies belong to this category. An *order dependency* states that if a table is ordered on a list of attributes, then it is also ordered on another list of attributes. The discovery of order dependencies has been only recently studied. In this paper, we propose a novel approach for discovering order dependencies in a given dataset. Our approach leverages the observation that discovering order dependencies can be guided by the discovery of a more specific form of dependencies called *order compatibility dependencies*. We show that our algorithm outperforms existing approaches on real datasets. Furthermore, our algorithm can be parallelized leading to further improvements when it is executed on multiple threads. We present several experiments that illustrate the effectiveness and efficiency of our proposal and discuss our findings.

1 INTRODUCTION

In the big data era, the volume and complexity of available datasets has grown so much that data engineers are having a hard time interpreting the information contained in them. In such a reality, the ability to discover hidden dependencies in some automatic way is fundamental. Dependencies across different parts of the data play a significant role in query optimization, since redundant information may be ignored making the query evaluation faster. Furthermore, parts of the data may be replaced with others that are easier to manipulate, without affecting the final result. Data profiling may help with data quality since it highlights constraints that may exist in the data but are not fully satisfied and have not been enforced when designing the database.

Dependency discovery is not a new challenge. Functional and inclusion dependencies are the most common type of dependencies and have been studied extensively [11]. A *functional dependency* states that if two different data elements sharing a common structure have the same part A, then some other part B should also have the same value. An *inclusion dependency* states that the values of the data elements in some part A must be a subset of the values in a subpart B of some other portion of the dataset.

An example of functional dependency can be seen in Table 1, that shows a relational table with data regarding yearly incomes, savings and taxes. Assume that the tax system is a progressive one that categorizes the different incomes into brackets, each of them characterized by a tax percentage. Thus, there is a functional dependency from the income amount to the tax brackets, i.e., `income` → `bracket`. Since for every income range the percentage is fixed, there are two other functional dependencies from the income to the tax amount and vice-versa, i.e., `income` → `tax` and `tax` → `income`. Using the transitive property of functional dependencies a new one can be inferred, i.e., `tax` → `bracket`.

A closer look at Table 1 can illustrate another, stronger, form of dependency: as the income is increasing, bracket and tax amount are increasing as well. In other words, if we were to order the table based on the income column, each one of the bracket and the tax amount columns will also end up being ordered. This form of dependency is known as an *order dependency* and is typically noted with the ↦ symbol, i.e., `income` ↦ `tax`, which is read as: `income` *orders* `tax`.

The knowledge encoded by order dependencies can be applied to various tasks during the entire data life-cycle [**?**]: in the design phase, order dependencies can be exploited to assist schema design [16] or for selecting indexes [**?**]; if data are extracted from unstructured sources, order dependencies can aid knowledge discovery, to find hidden properties of the data; in the context of data profiling [10], data integration and cleansing [3], order dependencies can be used to describe a dataset; for data quality [5], order dependencies can be used as requirements or constraints [**?**].

The most important application of order dependencies is their use in the optimization of queries; in particular, they can be used to rewrite the `ORDER BY` clauses in SQL queries in ways similar to that of functional dependencies for the `GROUP BY` statements [14, 17]. Consider the following query:

```
SELECT income, bracket, tax
FROM TaxInfo
ORDER BY income, bracket, tax
```

TaxInfo

name	income	savings	bracket	tax
T. Green	35,000	3,000	1	5,250
J. Smith	40,000	4,000	1	6,000
J. Doe	40,000	3,800	1	6,000
S. Black	55,000	6,500	2	8,500
W. White	60,000	6,500	2	9,500
M. Darrel	80,000	10,000	3	14,000

Table 1: A relational table with financial information.

Given that the order dependencies income ↦ tax and income ↦ bracket hold, the query optimizer can infer that sorting by income makes the ordering on the other two columns redundant, so the ORDER BY clause can be simplified to ORDER BY income.

The concept of order dependency in the context of database systems first appeared under the name of *point-wise order* [6–8]. A point-wise ordering specifies that a *set* of columns orders another *set* of columns. In the example of Table 1, the point-wise order dependency income,tax ⇝ bracket holds because if both of the tuples (income, tax) and (tax, income) are lexicographically ordered, then the column bracket is ordered in the same way. A new definition for order dependency was later introduced [16] to represent an order-preserving mapping between *lists* of attributes. In contrast to point-wise ordering, the new definition was distinguishing tuples with attributes in different order, thus having lists of attributes instead of sets.

There are cases where two lists of attributes order each other when taken together. This property is known as *order compatibility* and is denoted with the symbol ∼. In Table 1, e.g., it holds that

$$(\texttt{income}, \texttt{savings}) \mapsto (\texttt{savings}, \texttt{income}) \quad \text{and}$$
$$(\texttt{savings}, \texttt{income}) \mapsto (\texttt{income}, \texttt{savings})$$

and thus income ∼ savings. Another way to see an order compatibility dependency between two columns is that their values are both monotonically non-decreasing when they are considered pairwise.

Dependencies are typically derived from design specifications, from the context of queries or from other known dependencies using inference rules. Discovering dependencies by analyzing the data is a process known as *dependency discovery* [11]. It conceptually requires to check for all potential dependencies if they hold in the database instance under examination, which may be time consuming. Thus, there is interest in developing strategies that limits the number of combinations to be checked. The task becomes even more challenging in the case of order dependencies, where the order of attributes matters, leading to a search space much larger than that of functional dependencies.

In this work we study ways for efficiently discovering order dependencies. We follow a bottom-up approach in which we start by checking short lists of columns and progressively check longer and longer lists. In this process, once an order dependency between two lists of attributes is found not to hold, we prune the search space by ignoring larger lists that include them. In this way, many of the combinations that would have normally been checked are avoided.

We advocate that this whole process can be significantly improved by framing the discovery of order dependencies in the context of order compatibility dependencies. This is based on a recently introduced theorem [16] that established that an order dependency holds if and only if a functional and an order compatibility dependency hold between the two attribute lists of the order dependency. We illustrate in details how the order compatibility dependencies can be exploited to find order dependencies and propose a new algorithm for finding them.

Recently, two algorithms to automatically detect order dependencies in relational data have been proposed: ORDER, proposed by Langer and Naumann [10], and FASTOD proposed by Szlichta et al. [**?**]. ORDER explores a lattice of order dependency candidates, in a level-wise fashion reminiscent of the TANE algorithm [9]. After building a dependency candidate, ORDER checks its validity against the data and then it applies pruning rules to reduce the search space over the lattice. ORDER has been shown to be incomplete [**?**], i.e. it does not find the complete set of order dependencies. In particular, this approach is unable to discover dependencies with repeated attributes, for example, the order dependency (income, savings) ↦ savings of Table 1 cannot be discovered. Dependencies of this form, however, may not be inferred from other dependencies and are useful in the case of queries that involve ordering with multi-column indexes. In the example of Table 1, an index over (income, savings) can be used to simplify the clause ORDER BY savings. FASTOD [**?**] is based on a different axiomatization of order dependencies that allows mapping dependencies between lists of attributes to dependencies between sets of attributes written in a canonical form. In this way, several order dependencies are mapped to the same set-based canonical form. FASTOD explores the space of order dependencies of this set-based canonical form, still retaining the ability to find a complete set of dependencies. While we have reproduced the results presented in the original work, we have found that an implementation error of the original work produces wrong results over simple datasets, this vitiates the validity of their results and the comparison with our approach.

The approach we present in this paper is able to provide a complete set of dependencies that is based on the idea that the whole process of order dependency discovery could be performed through the search of *order compatibility dependencies*. While our approach has a higher worst-case complexity than FASTOD, it outperforms all the state-of-the-art approaches [10**?**] when tested over real datasets.

In particular, our contributions are the following:

- we introduce a definition of minimality for a set of order compatibility dependencies that we show being complete in the sense that it can recover all valid order compatibility dependencies that hold over a given instance of relational data;
- we propose a novel algorithm for finding order dependencies that is complete and can perform the detection of order dependencies in parallel.
- we perform an exhaustive experimental evaluation that shows the performance of our algorithm in comparison with existing works, including a study of its scalability over big datasets and multiple threads.
- we discuss possible solutions for the discovery of the most important order dependencies in the case of dataset that could not be managed (too many columns) in a reasonable amount of time.

The paper is structured as follows: in Section 2 we review the relevant definitions and theorems that formalize the connection between order dependencies and order compatibility dependencies. In Section 3 we prove that order dependency discovery can be guided by order compatibility dependencies without losing completeness. Our novel algorithm is presented in Section 4, while Section 5 contains the discussion of our experimental evaluation. Finally, a thorough review of the related work can be found in Section 6 and we present our conclusions in Section 7.

2 BACKGROUND

To provide the background of our problem, we first review the definition of order dependency and its axiomatization, then we describe the formal relation between order dependencies (ODs),

functional dependencies (FDs) and order compatibility dependencies (OCDs).

2.1 Notational Conventions and Definitions

We adopt the notational conventions summarized in Table 2, consistently with the literature [16]. Let $\boldsymbol{R}$ be a relation over the set of attributes $\mathcal{U}$ and $\boldsymbol{r}$ be a table instance of $\boldsymbol{R}$, i.e. a set of tuples under $\boldsymbol{R}$'s schema. A tuple p can be projected over a single attribute A, over a set of attributes $\mathcal{X}$ and over a list of attributes $\mathbf{X}$ by subscripting the tuple as follows: p_A, $p_{\mathcal{X}}$, $p_{\mathbf{X}}$.

We assume that a total ordering is defined over each of the attributes, denoted $\leq_A$; in the following, however, we will drop the attribute specification and use $\leq$, as the attribute will be always clear from the context. Order dependencies are defined based on the operator $\preccurlyeq$, which is equivalent to a lexicographical ordering over a list of attributes, and is defined by:

Definition 2.1 (operator $\preccurlyeq$). Given a list of attributes $\mathbf{X} := [A|\mathbf{T}]$ and two tuples $p, q \in \boldsymbol{r}$, the operator $\preccurlyeq$ (and its associated operator $\prec$) are defined as follows:

$$\begin{aligned} p_{\mathbf{X}} \preccurlyeq q_{\mathbf{X}} \quad &\Leftrightarrow \quad \big(p_A < q_A\big) \vee \\ &\quad \big((p_A = q_A) \wedge (\mathbf{T} = [\cdot] \vee p_{\mathbf{T}} \preccurlyeq q_{\mathbf{T}})\big) \qquad (1) \\ p_{\mathbf{X}} \prec q_{\mathbf{X}} \quad &\Leftrightarrow \quad p_{\mathbf{X}} \preccurlyeq q_{\mathbf{X}} \wedge p_{\mathbf{X}} \neq q_{\mathbf{X}} \end{aligned}$$

The $\preccurlyeq$ operator reproduces the ordering clause `ORDER BY ASC` in SQL [16].

Based on the comparison operator of Definition 2.1, we can introduce the concept of *order dependency* [16].

Definition 2.2 (Order dependency (OD)). Given a relation $\boldsymbol{R}$ and two lists $\mathbf{X}$ and $\mathbf{Y}$, $\mathbf{X} \mapsto \mathbf{Y}$ is an *order dependency* if, for any instance $\boldsymbol{r}$ of $\boldsymbol{R}$ and for every pair of tuples $p, q \in \boldsymbol{r}$, the following implication holds:

$$p_{\mathbf{X}} \preccurlyeq q_{\mathbf{X}} \Rightarrow p_{\mathbf{Y}} \preccurlyeq q_{\mathbf{Y}} \qquad (2)$$

If both $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$ hold, we say that $\mathbf{X}$ and $\mathbf{Y}$ are *order equivalent* and we write $\mathbf{X} \leftrightarrow \mathbf{Y}$. If $\mathbf{XY} \leftrightarrow \mathbf{YX}$ we say that $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible* and we write $\mathbf{X} \sim \mathbf{Y}$. We will discuss the latter relation in Section 2.2.

Order dependencies satisfy the set of axioms $\mathcal{J}_{\text{OD}}$, introduced by Szlichta et al. [16], which are reported in Table 3. These axioms are analogous to the Armstrong axioms for functional dependencies [1].

Relations:
- $\boldsymbol{R}$, written as a capital letter in bold italics, is a *relation* over a set of attributes $\mathcal{U}$;
- $\boldsymbol{r}$, written as a lowercase letter in bold italics, is a *table instance* over $\boldsymbol{R}$, i.e. a *set of tuples*;
- Single *attributes* are represented with capital letters: A, B, and C;
- *Tuples* are represented with lowercase letters: p, q, s, and t.

Lists:
- Bold capital letters are *lists* of attributes: $\mathbf{X}$,$\mathbf{Y}$, and $\mathbf{Z}$. they can represent the empty list $[\cdot]$;
- A list is denoted with square brackets $[A,B,C]$. A list $[A|\mathbf{T}]$ is composed by a *head* A and a *tail* $\mathbf{T}$;
- $\mathbf{XY}$ is a shorthand for $\mathbf{X} \circ \mathbf{Y}$, $\mathbf{X}A$ and $A\mathbf{X}$ are shorthands for $\mathbf{X} \circ [A]$ and $[A] \circ \mathbf{X}$ respectively, AB denotes $[A,B]$.

Table 2: Notational conventions

AX1: *Reflexivity*

$$\mathbf{XY} \mapsto \mathbf{X}$$

AX2: *Prefix*

$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{ZX} \mapsto \mathbf{ZY}}$$

AX3: *Normalization*

$$\mathbf{WXYXV} \leftrightarrow \mathbf{WXYV}$$

AX4: *Transitivity*

$$\frac{\mathbf{X} \mapsto \mathbf{Y} \quad \mathbf{Y} \mapsto \mathbf{Z}}{\mathbf{X} \mapsto \mathbf{Z}}$$

AX5: *Suffix*

$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{X} \leftrightarrow \mathbf{YX}}$$

AX6: *Chain*

$$\frac{\begin{array}{c}\mathbf{X} \sim \mathbf{Y}_1 \\ \forall_{i \in [1,n-1]} \mathbf{Y}_i \sim \mathbf{Y}_{i+1} \\ \mathbf{Y}_n \sim \mathbf{Z} \\ \forall_{i \in [1,n]} \mathbf{Y}_i\mathbf{X} \sim \mathbf{Y}_i\mathbf{Z}\end{array}}{\mathbf{X} \sim \mathbf{Z}}$$

Table 3: The set of axioms $\mathcal{J}_{\text{OD}}$ for order dependencies [16]

2.2 Decomposing Order Dependencies

We show how an order dependency can be decomposed in a pair composed of one functional dependency and one order compatibility dependency.

Functional dependencies. Functional dependencies encode the fact that, in a relation, one attribute determines completely another attribute.

Definition 2.3 (Functional dependency (FD)). Given a relation $\boldsymbol{R}$ and two sets of attributes $\mathcal{X}$ and $\mathcal{Y}$, $\mathcal{X} \rightarrow \mathcal{Y}$ is a *functional dependency* if, for any instance $\boldsymbol{r}$ of $\boldsymbol{R}$ and for every pair of tuples $p, q \in \boldsymbol{r}$, the following implication holds:

$$p_{\mathcal{X}} = q_{\mathcal{X}} \Rightarrow p_{\mathcal{Y}} = q_{\mathcal{Y}} \qquad (3)$$

Order Compatibility Dependencies. Order compatibility dependencies encode the fact that in a relation two lists of attributes show the same monotonicity. If we order either combination of the lists in non-decreasing order, they end up both ordered such their values are both monotonically non-decreasing.

Definition 2.4 (order compatibility dep. (OCD)). Given a relation $\boldsymbol{R}$ and two lists of attributes $\mathbf{X}$ and $\mathbf{Y}$ in $\boldsymbol{R}$, $\mathbf{X} \sim \mathbf{Y}$ is a *order compatibility dependency* if, for any instance $\boldsymbol{r}$ of $\boldsymbol{R}$ and for every pair of tuples $p, q \in \boldsymbol{r}$, the following implications hold:

$$p_{\mathbf{XY}} \preccurlyeq q_{\mathbf{XY}} \Leftrightarrow p_{\mathbf{YX}} \preccurlyeq q_{\mathbf{YX}} \qquad (4)$$

Order dependencies are a stricter relation between two attributes with respect to functional and order compatibility dependencies; furthermore, when an order dependency between two lists of attributes $\mathbf{X}$ and $\mathbf{Y}$ holds, an order compatibility dependency between $\mathbf{X}$ and $\mathbf{Y}$ holds as well. In this sense, order dependencies combine the fact that an attribute functionally determines and have the same monotonicity of another when ordered together.

We present previous results [16] that formally highlight the nature of the relationship between these dependencies, showing that when an order dependency does not hold there are only two possible scenarios called *split* and *swap*. When $\mathbf{X}$ does not order $\mathbf{Y}$, i.e. when the order dependency between $\mathbf{X}$ and $\mathbf{Y}$ does not hold, we write $\mathbf{X} \nmapsto \mathbf{Y}$.

	A	B
t_1	1	4
t_2	2	5
t_3	3	6
t_4	3	7
t_5	4	1

Table 4: A relational table containing both a split and a swap between the two attributes A and B.

Split. A split indicates the case where there exists a pair of tuples that have the same values when projected over the attributes **X**, but have different values over the attributes **Y**. Formally:

Definition 2.5 (Split). Two tuples s,$t \in \boldsymbol{r}$ form a split over two lists of attributes **X**, **Y** iff $s_{\mathbf{X}} = t_{\mathbf{X}}$ but $s_{\mathbf{Y}} \neq t_{\mathbf{Y}}$, or equivalently:

$$\exists s, t \in \boldsymbol{r} : s_{\mathbf{X}} = t_{\mathbf{X}} \wedge s_{\mathbf{Y}} > t_{\mathbf{Y}}$$

When an OD between two attribute lists is valid, then a FD is valid as well (Theorem 15, [16])

Theorem 2.6 (ODs subsume FDs). *For every instance $\boldsymbol{r}$ of relation $\boldsymbol{R}$, if the OD $\mathbf{X} \mapsto \mathbf{Y}$ holds, then the FD $\mathcal{X} \rightarrow \mathcal{Y}$ holds.*

Whereas if there is a split between two lists of attributes **X** and **Y**, there is no guarantee that ordering data will result in ordered tuples over **Y** and **XY**; in other words, both the ODs $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{X} \mapsto \mathbf{XY}$ do not hold. Furthermore, a split falsifies the functional dependency $\mathcal{X} \rightarrow \mathcal{Y}$ as well.

The relationship between order and functional dependencies is formalized as follows (Theorem 13, [16]):

Theorem 2.7 (FD and OD correspondence). *For every instance $\boldsymbol{r}$ of a relation $\boldsymbol{R}$, the functional dependency $\mathcal{X} \rightarrow \mathcal{Y}$ holds iff $\mathbf{X} \mapsto \mathbf{XY}$ holds for all lists **X** that order the attributes of $\mathcal{X}$ and all lists **Y** that order the attributes of $\mathcal{Y}$.*

In Table 4, tuples t_3 and t_4 form a split for the attributes A and B, thus $A \nmapsto B$ and $A \nmapsto AB$. The functional dependency $A \rightarrow B$ is not valid, as well.

Swap. A *swap* indicates the case where there exists a pair of tuples whose values projected over two lists of attributes **X** and **Y** are swapped, i.e. they are sorted differently when they are ordered with respect to **X** or **Y**. Formally:

Definition 2.8 (Swap). Two tuples s,$t \in \boldsymbol{R}$ form a swap over two list of attributes **X** and **Y**, iff $s_{\mathbf{X}} \prec t_{\mathbf{X}}$ but $t_{\mathbf{Y}} \prec s_{\mathbf{Y}}$, or equivalently:

$$\exists s, t \in \boldsymbol{r} : s_{\mathbf{X}} \prec t_{\mathbf{X}} \wedge s_{\mathbf{Y}} > t_{\mathbf{Y}}$$

Swaps between **X** and **Y** falsify the ODs of the form $\mathbf{X} \mapsto \mathbf{Y}$, $\mathbf{Y} \mapsto \mathbf{X}$, and $\mathbf{XY} \leftrightarrow \mathbf{YX}$. For example, tuples t_1 and t_5 in Table 4 form a swap for attributes A and B, thus $A \nmapsto B$, $AB \nmapsto B$, and $AB \nmapsto BA$.

Splits and swaps establish a correspondence between order dependencies, functional dependencies and order compatibility dependencies, as in the following theorem (Theorem 15, [16]):

Theorem 2.9 (OD = FD + OCD). $\mathbf{X} \mapsto \mathbf{Y}$ *holds iff* $\mathcal{X} \rightarrow \mathcal{Y}$ *(*$\mathbf{X} \mapsto \mathbf{XY}$*) and* $\mathbf{X} \sim \mathbf{Y}$ *(*$\mathbf{XY} \leftrightarrow \mathbf{YX}$*) hold.*

In summary, when an order dependency between two lists of attributes **X** and **Y** holds:

- a functional dependency $\mathcal{X} \rightarrow \mathcal{Y}$ holds, which implies the absence of split conditions;
- an order compatibility dependency between **X** and **Y** holds, which implies the absence of swap conditions.

We exploit this relation to guide our discovery algorithm as established in Section 4.2.

3 ORDER DEPENDENCY THROUGH ORDER COMPATIBILITY

This section introduces the concepts that lay the foundations to our approach.

3.1 Minimality of Discovered Dependencies

Similarly to what has been done for functional dependencies, we introduce the notion of minimality of a set of order compatibility dependencies. In principle, minimality is the property for which a set of dependencies equipped with inference rules can recover all the dependencies that are valid in a given instance of relational data. Axioms AX1-AX6 presented in Table 3 provide inference rules for order dependencies.

The concept of minimality for order dependencies, as for other types of dependencies, has several uses. First of all, it allows reasoning about the validity of pruning rules, e.g., to show that they do not lead to loss of information about valid dependencies in the relation.

Minimality serves also the purpose of compressing information to a manageable size: in fact, if we take a relation with n attributes, in the worst case where each of which is order equivalent to every other, the minimal set of ODs would contain $n-1$ dependencies ($A \leftrightarrow B, A \leftrightarrow C$, etc.), while the set of all valid dependencies would contain $\mathcal{O}((n!)^2)$ elements: all the possible combinations of attributes on the left-hand and right-hand sides, a prohibitively large number.

Finally, real applications may not need the whole list of dependencies: for example, in knowledge discovery, redundant dependencies do not add value to the properties discovered and too many dependencies can cause the most important ones to be missed; in query optimization, the only useful dependencies are those that can be applied to the queries to be performed.

Any pruning rule applied by a dependency discovery algorithm needs to respect minimality, in the sense that it should allow the recovery of the full set of valid dependencies. A complete algorithm must find at least all *minimal* order dependencies over an instance $\boldsymbol{r}$ of a relation $\boldsymbol{R}$.

To introduce the concept of minimality for ODs, we start by presenting the concepts of closure and equivalence of sets of order dependencies.

Definition 3.1 (Closure). The *closure* of the set of ODs $\mathcal{M}$, denoted $\mathcal{M}^+$, is the set of ODs that are logically implied from $\mathcal{M}$ by the axioms $\mathcal{J}_{\text{OD}} = \{\text{AX1} - \text{AX6}\}$ defined in Table 3.

$$\mathcal{M}^+ = \{\mathbf{X} \mapsto \mathbf{Y} \mid \mathcal{M} \vdash_{\mathcal{J}_{\text{OD}}} \mathbf{X} \mapsto \mathbf{Y}\}$$

Definition 3.2 (Equivalence of sets of ODs). Two sets $\mathcal{M}_1$ and $\mathcal{M}_2$ of order dependencies are *equivalent* if and only if they have the same closure $\mathcal{M}_1^+ = \mathcal{M}_2^+$.

The closure of a set of minimal dependencies is the set of all the dependencies that are valid over $\boldsymbol{r}$. We build this definition first showing which lists of attributes are in minimal form and then when an OCD is minimal. Finally, we prove that our definition of minimality is complete.

Order compatibility dependencies employ attribute lists instead of attribute sets, thus we introduce the concepts of *minimal* attribute list. An attribute list is minimal if it has no embedded order dependency, i.e., the list of attributes is the shortest possible.

Definition 3.3 (minimal attribute list). An attribute list $\mathbf{X}$ is minimal if there is no other list $\mathbf{X}'$ such that:

- $\mathbf{X}'$ is smaller than $\mathbf{X}$, and
- $\mathbf{X}$ and $\mathbf{X}'$ are order equivalent.

For example, the attribute list *ABA* is never minimal, in fact, by the Normalization axiom (AX3) we know that $ABA \leftrightarrow AB$ and *AB* is a shorter list than *ABA*. Instead, *AB* is a minimal attribute list, unless $A \leftrightarrow B$.

An OCD is minimal if both sides are given as minimal attribute lists and there are no repeated attributes:

Definition 3.4 (minimal OCD). An OCD $\mathbf{X} \sim \mathbf{Y}$ is minimal if:

- $\mathbf{X}$ and $\mathbf{Y}$ are minimal attribute lists;
- $\mathcal{X} \cap \mathcal{Y} = \varnothing$.

In the following theorem, we show that OCDs with repeated attributes can be derived from OCDs without repeated attributes:

Theorem 3.5 (Completeness of minimal OCD). *Order compatibility dependencies with repeated attributes can be derived from OCD without repeated attributes.*

Proof. The proof of this theorem is split in three cases:

(1) OCDs of the form $\mathbf{XY} \sim \mathbf{XZ}$ can be derived from $\mathbf{Y} \sim \mathbf{Z}$;
(2) OCDs of the form $\mathbf{XY} \sim \mathbf{MY}$ can be derived from $\mathbf{XY} \sim \mathbf{M}$ and $\mathbf{X} \sim \mathbf{MY}$;
(3) OCDs of the form $\mathbf{XY} \sim \mathbf{MYN}$ can be derived from $\mathbf{X} \sim \mathbf{M}$, $\mathbf{XY} \sim \mathbf{M}$, $\mathbf{X} \sim \mathbf{MY}$ and $\mathbf{XY} \sim \mathbf{MN}$;

which are covered respectively by Theorems 3.10, 3.11 and 3.12 which are presented separately for clarity in Section 3.3. □

The following result extends the Downward Closure theorem (Theorem 12, [16]):

Theorem 3.6. *Downward closure for OCD*

$$\frac{\mathbf{XY} \sim \mathbf{ZV}}{\mathbf{X} \sim \mathbf{Z}}$$

Theorems 3.5 and 3.6 provide the justification for the structure of the search tree used in our approach, as explained in Section 4.2. In particular, we derive the following pruning rule:

Theorem 3.7 (Pruning rule for OCD).

$$\frac{\mathbf{X} \not\sim \mathbf{Z}}{\mathbf{XY} \not\sim \mathbf{ZV}}$$

Proof. This theorem is the contronominal preposition of Theorem 3.6. □

Theorem 3.5 justifies the reduction of the search space that we highlight in the following section.

3.2 Dimension of the Search Space

The number of valid ODs over a relation $\boldsymbol{R}$ is vast: in fact, if $\boldsymbol{R}$ has n attributes, then all permutations of length k of n elements must be considered both for the left-hand side and the right-hand side of the OD. We address a limitation of the previous work by Langer and Naumann [10] and we show that some order dependencies with repeated attributes cannot be derived from other dependencies without repeated attributes.

For example, in Table 5 (a) we have that $AB \mapsto B$, instead in Table 5 (b) we have $AB \mapsto B$ and $A \sim B$. For both tables the dependencies $A \mapsto B$ and $B \mapsto A$ do not hold, thus the validity of $AB \mapsto B$ and $A \sim B$, in this case, cannot be inferred from shorter ODs.

	(a)		(b)	
	A	**B**	**A**	**B**
t_1	1	4	1	4
t_2	2	5	2	5
t_3	3	6	3	6
t_4	3	7	3	7
t_5	4	1	4	7

Table 5: Two relations where the ODs $A \mapsto B$ and $B \mapsto A$ do not hold; furthermore in (a) $AB \mapsto B$ while in (b) $AB \mapsto B$ and $A \sim B$.

Finding all valid order dependencies thus requires, in principle, the need for checking all combinations $\mathbf{X} \mapsto \mathbf{Y}$ where both $\mathbf{X}$ and $\mathbf{Y}$ can be permutations of length k of the n attributes in $\boldsymbol{R}$ with $1 \leq k \leq n$. If we denote with $S(n)$ the number of k-permutations of n elements we have:

$$S(n) = \lfloor e \cdot n! \rfloor - 1$$

Excluding trivial ODs of the form $\mathbf{X} \mapsto \mathbf{X}$, the number of candidates that needs to be checked would be:

$$C(n) = \big(S(n) - 1\big) \cdot \big(S(n) - 1\big) - \big(S(n) - 1\big) \propto \mathcal{O}((n!)^2) \quad (5)$$

With $n = 10$, there are more than $97 \cdot 10^{12}$ candidates ODs.

In contrast to general order dependencies, OCDs candidates with repeated attributes, i.e., $\mathbf{X} \to \mathbf{Y}$ or $\mathbf{X} \sim \mathbf{Y}$ where $\mathcal{X} \cap \mathcal{Y} \neq \varnothing$, are redundant in the sense that their validity can be inferred from the validity of other dependencies of the same type without repeated attributes and with shorter attribute lists.

Theorem 3.8. $\mathbf{X} \sim \mathbf{Y}$ *iff* $\mathbf{XY} \mapsto \mathbf{Y}$

Proof. We prove the implication in each direction:

⇒ By definition $\mathbf{X} \sim \mathbf{Y}$ implies that both the order dependencies $\mathbf{XY} \mapsto \mathbf{YX}$ and $\mathbf{YX} \mapsto \mathbf{XY}$ are valid. By Reflexivity (AX1) $\mathbf{YX} \mapsto \mathbf{Y}$ and thus by Transitivity (AX4) the order dependency $\mathbf{XY} \mapsto \mathbf{Y}$ is valid.
⇐ Conversely, if $\mathbf{XY} \mapsto \mathbf{Y}$, by Suffix (AX5) $\mathbf{XY} \mapsto \mathbf{YXY}$ and Normalization (AX3) $\mathbf{XY} \mapsto \mathbf{YX}$. □

□

This means that ODs of the form $\mathbf{XY} \mapsto \mathbf{Y}$ and OCDs of the form $\mathbf{X} \sim \mathbf{Y}$ are equivalent. We are thus enabled to solve the problem by considering only OCDs without repeated attributes, and thus the dimension of the search space is reduced to $\mathcal{O}(n!)$.

As shown in Theorem 2.9, if $\mathbf{X} \mapsto \mathbf{Y}$ then $\mathbf{X} \sim \mathbf{Y}$ is valid; we can thus derive the following theorem, which will provide the foundation for the pruning rules detailed in Section 4.2.1.

Theorem 3.9.

$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{XZ} \sim \mathbf{Y}} \quad (6)$$

Proof. By the Augmentation theorem [16], $\mathbf{X} \mapsto \mathbf{Y}$ implies $\mathbf{XZ} \mapsto \mathbf{Y}$. By Theorem 2.9 of Section 2.2, $\mathbf{XZ} \mapsto \mathbf{Y}$ implies $\mathbf{XZ} \sim \mathbf{Y}$. □

3.3 Completeness of Minimal OCD

We divide the proof of the completeness of our definition of minimality for OCDs in three parts: first, in the following theorem we prove that attribute lists with repeated attributes at the beginning are redundant:

Theorem 3.10 (Completeness of minimal OCD - 1).

$$\frac{\mathbf{Y} \sim \mathbf{Z}}{\mathbf{XY} \sim \mathbf{XZ}}$$

Proof. By the Shift theorem [16] and the fact that $\mathbf{X} \leftrightarrow \mathbf{X}$ by Reflexivity (AX1):

$$\frac{\mathbf{YZ} \mapsto \mathbf{ZY} \quad \mathbf{X} \leftrightarrow \mathbf{X}}{\mathbf{XYZ} \mapsto \mathbf{XZY}}$$

by Normalization (AX3) and Replace [16] $\mathbf{XYXZ} \mapsto \mathbf{XZXY}$. Analogously by the Shift theorem [16] starting from $\mathbf{ZY} \mapsto \mathbf{YZ}$ we obtain $\mathbf{XZXY} \mapsto \mathbf{XYXZ}$. Thus $\mathbf{XYXZ} \leftrightarrow \mathbf{XZXY}$, i.e., $\mathbf{XY} \sim \mathbf{XZ}$ □

The following theorem proves that attribute lists with repeated attributes at the end are also redundant:

Theorem 3.11 (Completeness of minimal OCD - 2).

$$\frac{\mathbf{X} \sim \mathbf{Y} \quad \mathbf{XZ} \sim \mathbf{Y} \quad \mathbf{X} \sim \mathbf{YZ}}{\mathbf{XZ} \sim \mathbf{YZ}}$$

Proof. (1) using $\mathbf{XY} \leftrightarrow \mathbf{YX}$ and $\mathbf{XZY} \leftrightarrow \mathbf{YXZ}$, by Normalization (AX3) $\mathbf{XZY} \leftrightarrow \mathbf{XZYZ}$ and by Replace [16] $\mathbf{YXZ} \leftrightarrow \mathbf{XZYZ}$;

(2) using $\mathbf{XY} \leftrightarrow \mathbf{YX}$ and $\mathbf{XYZ} \leftrightarrow \mathbf{YZX}$, by Normalization (AX3) $\mathbf{YZX} \leftrightarrow \mathbf{YZXZ}$, by Replace [16] $\mathbf{YXZ} \leftrightarrow \mathbf{YZX}$ and by Transitivity (AX4) $\mathbf{YXZ} \leftrightarrow \mathbf{YZXZ}$;

By Transitivity (AX4) $\mathbf{YXZ} \leftrightarrow \mathbf{XZYZ}$ and $\mathbf{YXZ} \leftrightarrow \mathbf{YZXZ}$ imply $\mathbf{XZYZ} \leftrightarrow \mathbf{YZXZ}$, i.e., $\mathbf{XZ} \sim \mathbf{YZ}$. □

Finally, the following theorem proves that attribute lists with repeated attributes in the middle are redundant:

Theorem 3.12 (Completeness of minimal OCD - 3).

$$\frac{\mathbf{X} \sim \boldsymbol{M} \quad \mathbf{XY} \sim \boldsymbol{M} \quad \mathbf{X} \sim \boldsymbol{MY} \quad \mathbf{XY} \sim \boldsymbol{MN}}{\mathbf{XY} \sim \boldsymbol{MYN}}$$

Proof.

(1) from $\mathbf{XY} \sim \mathbf{MN}$, by Normalization (AX3) $\mathbf{XYMYN} \leftrightarrow \mathbf{MNXY}$;

(2) from $\mathbf{XY} \sim \mathbf{M}$ and $\mathbf{X} \sim \mathbf{MY}$, using $\mathbf{X} \sim \mathbf{M}$ and Replace [16] we get $\mathbf{MYX} \leftrightarrow \mathbf{XYM}$ and $\mathbf{MXY} \leftrightarrow \mathbf{MYX} \leftrightarrow \mathbf{XYM}$;

(3) from (2), by the Shift theorem [16] with $\mathbf{MY} \leftrightarrow \mathbf{MY}$ and $\mathbf{MNXY} \leftrightarrow \mathbf{XYMMYN}$ we get $\mathbf{MYMNXY} \leftrightarrow \mathbf{MYXYMMYN}$;

(4) by Normalization (AX3) $\mathbf{MYMNXY} \leftrightarrow \mathbf{MYNXY}$;

(5) from $\mathbf{MYXYMMYN}$, using $\mathbf{MYX} \leftrightarrow \mathbf{XYM}$ and Normalization (AX3) we get $\mathbf{XYMYMYN}$ and finally $\mathbf{XYMYN}$;

From points (3), (4) and (5) we finally get $\mathbf{MYNXY} \leftrightarrow \mathbf{XYMYN}$, i.e., $\mathbf{XY} \sim \mathbf{MYN}$. □

4 THE OCDDISCOVER ALGORITHM

We present now the details of our algorithm, called OCDDISCOVER, by first examining its search strategy to cover all the possible combinations and then presenting an implementation in pseudocode.

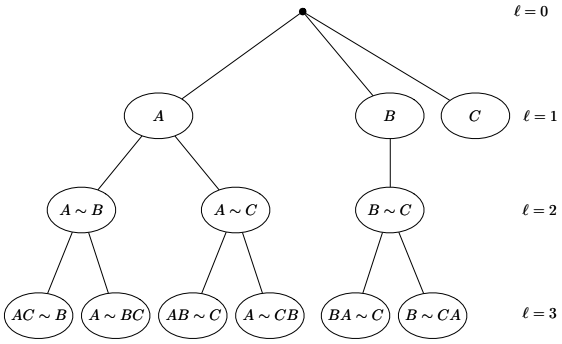

Figure 1: Permutation tree for a table with $n = 3$ attributes.

4.1 Column Reduction

Given that the search space grows with the number of columns, we start our discovery algorithm focusing on the columns showing special properties and we perform two operations: (a) the removal of constant columns; (b) the reduction of order-equivalent columns. The dependencies provided by these operations are an integral part of the results provided by our algorithm.

Removal of constant columns. Constant columns generate a huge amount of ODs; in fact, over an instance $\boldsymbol{r}$ a constant column C is ordered by any other attribute list $\mathbf{X}$.[1] Thus, we remove all constant columns and we collect the corresponding dependencies.

Reduction of order-equivalent columns. Order-equivalent columns as $A \leftrightarrow B$ describe a relation in which both the directions of the order dependency hold. By the Replace theorem (Theorem 6, [16]), we can replace any order dependency where A appears with another dependency with any instance of A replaced with B, that is:

$$\mathbf{X}A\mathbf{Y} \mapsto \mathbf{M}A\mathbf{N} \Leftrightarrow \mathbf{X}B\mathbf{Y} \mapsto \mathbf{M}B\mathbf{N}$$

We check any combination of order-equivalent dependencies, i.e. for all $A, B \in \mathcal{U}$ we verify the validity of $A \mapsto B$ and $B \mapsto A$, and we build the equivalence classes of columns using the Tarjan algorithm [19].

We choose a representative from each of these equivalence classes; we then remove all other columns. We store this information to later recover the redundant dependencies.

4.2 Search Tree

We use a breadth-first search strategy for identifying OCD relations in $\boldsymbol{r}$; in this way, shorter minimal dependencies are discovered before longer ones. At the first level, we consider the set of all pairs of single attributes. Given that OCDs are commutative, we build this set by enumerating all the attributes with $A_1, A_2, \ldots, A_n$ and taking all the pairs (A_i, A_j) such that $\{(A_i, A_j) \mid A_i, A_j \in \mathcal{U}, i < j\}$.

Figure 1 shows the tree $\mathcal{T}$ of generated candidates for a relation $\boldsymbol{r}$ with attributes $\mathcal{U} = \{A,B,C\}$ where all possible candidates are generated.

Each OCD candidate $\mathbf{X} \sim \mathbf{Y}$ is checked for order compatibility; we are then confronted with two possibilities:

[1] If C is constant column, the following property holds for any tuple p, q in any instance $\boldsymbol{r}$ of $\boldsymbol{R}$: $p_{\mathbf{X}} \leq q_{\mathbf{X}} \Rightarrow p_C = q_C$, where the second part of the implication is always true by definition of constant column.

- if the candidate is not order compatible, we do not generate any other candidate starting from it, as stated by Theorem 3.7 in Section 3.1;
- if the candidate is order compatible, we generate new OCD candidates in the following way: for each attribute not already present in the OCD, for each $A \in \mathcal{U} \setminus \{\mathcal{X} \cup \mathcal{Y}\}$, we add it to the right of each attribute list, i.e. $\mathbf{X}A \sim \mathbf{Y}$ and $\mathbf{X} \sim \mathbf{Y}A$, then we apply further pruning rules as explained in Section 4.2.1.

4.2.1 Pruning Rules. When we find a new OCD $\mathbf{X} \sim \mathbf{Y}$, we further check the validity of the OD $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$.

From Theorem 3.9 we derive the following pruning rules:

- if $\mathbf{X} \mapsto \mathbf{Y}$ we do not generate the candidates of the form $\mathbf{XZ} \overset{?}{\sim} \mathbf{Y}$, i.e. the left-hand children candidates of $\mathbf{X} \sim \mathbf{Y}$ are pruned;
- if $\mathbf{Y} \mapsto \mathbf{X}$, we do not generate the candidates of the form $\mathbf{X} \overset{?}{\sim} \mathbf{YZ}$, i.e. the right-hand children candidates of $\mathbf{X} \sim \mathbf{Y}$ are pruned;

If both dependencies are valid we prune all the subtree from the given OCD candidate.

With reference to Figure 1, if the order compatible dependency $A \sim B$ is valid:

- if $A \mapsto B$, the children candidate $AC \overset{?}{\sim} B$ is pruned;
- if $B \mapsto A$, the children candidate $A \overset{?}{\sim} BC$ is pruned.

4.2.2 Parallelizability. OCDDISCOVER explores the tree of candidates breadth-first. Each branch of the tree can be visited independently since each OCD candidate is independent from the others; furthermore, a candidate is generated for each level if and only if its father in the tree was a valid order dependency. We exploit this structure to parallelize the execution of OCDDISCOVER by assigning candidates from different branches to different queues; each queue is then processed by a different thread. With reference to Algorithm 1, if we have K cores available, we can have K independent subtrees $\mathcal{T}_\ell^1, \mathcal{T}_\ell^2, \ldots, \mathcal{T}_\ell^K$ (lines $7-12$) each one containing the OCD candidates belonging to a different branch of the tree. The number of queues used by the algorithm can be chosen as a run-time parameter provided by the user.

4.3 Order Checking

One of the most important steps in our approach is the check of order compatibility candidates.

Single check. Given an OCD candidate in the form $\mathbf{X} \overset{?}{\sim} \mathbf{Y}$, we need to verify if it holds. The general definition of order compatibility states that $\mathbf{X} \sim \mathbf{Y} \equiv \mathbf{XY} \leftrightarrow \mathbf{YX}$, i.e. $\mathbf{X}$ and $\mathbf{Y}$ are order compatible if $\mathbf{XY} \mapsto \mathbf{YX}$ and $\mathbf{YX} \mapsto \mathbf{XY}$; however, with the following theorem, we can reduce the problem of checking the validity of an OCD to a single check.

THEOREM 4.1. $\mathbf{XY} \mapsto \mathbf{YX}$ *is valid iff* $\mathbf{X} \sim \mathbf{Y}$.

PROOF.

$\Rightarrow$ we have to prove that $\mathbf{XY} \mapsto \mathbf{YX} \Rightarrow \mathbf{YX} \mapsto \mathbf{XY}$. If, by contradiction, $\mathbf{YX} \nmapsto \mathbf{XY}$, then:

$$\exists\, p,q \mid p_{\mathbf{YX}} \leq q_{\mathbf{YX}} \Rightarrow p_{\mathbf{XY}} > q_{\mathbf{XY}} \tag{7}$$

thus since $p_{\mathbf{XY}} > q_{\mathbf{XY}}$ we can distinguish two cases:

– if $p_{\mathbf{X}} > q_{\mathbf{X}}$, we can conclude that:

$$q_{\mathbf{XY}} < p_{\mathbf{XY}} \Rightarrow q_{\mathbf{YX}} > p_{\mathbf{YX}}$$

thus $\mathbf{XY} \nmapsto \mathbf{YX}$;

– if $p_{\mathbf{X}} = q_{\mathbf{X}} \wedge p_{\mathbf{Y}} > q_{\mathbf{Y}}$, we always obtain a contradiction with the condition expressed in Eq. 7;

thus both $\mathbf{XY} \mapsto \mathbf{YX}$ and $\mathbf{YX} \mapsto \mathbf{XY}$ are valid and $\mathbf{X} \sim \mathbf{Y}$;

$\Leftarrow$ by definition if $\mathbf{X} \sim \mathbf{Y}$ then both $\mathbf{YX} \mapsto \mathbf{XY}$ and $\mathbf{XY} \mapsto \mathbf{YX}$ are valid;

□

Checking with Indexes. To compute if an order dependency candidate holds, we sort the relation by the left-hand side attributes of the candidate. We build an index that contains only the position of each tuple in the order in which it appears. Then, we iterate over the tuples following this index, and we check if the attributes on the right-hand side violate the ordering, specifically if we detected a pair of tuples forming a swap. We break the detection loop as soon as we find a violation and return true otherwise. In the worst case, the number of comparisons to be made is $O(m + m \log m)$ where m is the total number of tuples in the relation.

`NULL` Values. In real-world datasets, and in many of the test cases that will be analyzed in Section 5, data contains `NULL` values, which destroy the total ordering assumption because they can not be compared with the other values. We use the standard SQL semantics given by `set ansi nulls ON`, i.e. `NULL` equals `NULL`, and `NULLS FIRST` for sorting.

4.4 Description of the Algorithm

Algorithm 1 is the main algorithm that implements our approach. The input is given by the instance of relational data $\boldsymbol{r}$ and its set of attributes $\mathcal{U}$.

Algorithm 1 OCDDISCOVER

Input: $\boldsymbol{r}$: a relational instance
Input: $\mathcal{U}$: the set of attributes associated with $\boldsymbol{r}$

```
1:  function OCDDISCOVER(r, U)
2:      ℓ ← 2
3:      U' ← COLUMNSREDUCTION(U)
4:      T_1 ← {(A, B) | A, B ∈ U', B ≻ A}
5:      while T_ℓ ≠ ∅ do                                  ▷ Main loop
6:          T_{ℓ+1} ← ∅
7:          for each (X, Y) ∈ T_ℓ do
8:              if CHECKCANDIDATE(XY, YX, r) then
9:                  emit X ~ Y
10:                 T_{ℓ+1} ← T_{ℓ+1} ∪ GENERATENEXTLEVEL(X, Y, U')
11:             end if
12:         end for
13:         ℓ ← ℓ + 1
14:     end while
15: end function
```

In Line 3, function COLUMNREDUCTION() (not shown here) is called to apply the operations described in Section 4.1: the removal of constant attributes and the reduction of order-equivalent columns. This function prints a list of order-equivalence relations and a list of constant attributes and returns a reduced set of attributes $\mathcal{U}'$ where (a) the constant columns are removed; and (b) for each class of order-equivalent attributes, one representative is chosen.

The initial tree of OCD candidates is built in Line 4; by the commutativity of OCDs, only half of the combinations are added.

Figure 1 shows that the second level of the tree ($\ell = 2$) contains only the initial candidates $A \overset{?}{\sim} B$, $A \overset{?}{\sim} C$ and $B \overset{?}{\sim} C$.

Then, the algorithm continues with the main loop where each OCD candidate $\mathbf{X} \overset{?}{\sim} \mathbf{Y}$ is tested against the data in $\boldsymbol{r}$ in the form of an OD candidate $\mathbf{XY} \overset{?}{\mapsto} \mathbf{YX}$ using the function checkCandidate(), which is described in Algorithm 2.

The function checkCandidate() iterates over the index built on the left-hand side of the candidate with generateIndex() and checks that the values over the attributes in the right-hand side are in the same order. The loop is terminated early if a violation is detected.

Algorithm 2 checkCandidate

Input: $\mathbf{X}, \mathbf{Y}$: an OD candidate
Input: $\boldsymbol{r}$: the instance of relational data
Output: **true** if $\mathbf{X} \mapsto \mathbf{Y}$, **false** otherwise.

```
1:  function checkCandidate(X, Y, r)
2:    l_r ← len(r)
3:    index ← generateIndex(X,Y,r)
4:    for i ← 1 to l_r − 1 do
5:      for each A ∈ Y do
6:        if r[index[i], A] > r[index[i + 1], A] then
7:          return false
8:        else if r[index[i], A] < r[index[i + 1], A] then
9:          return true
10:       end if
11:     end for
12:   end for
13:   return true
14: end function
```

If the candidate is a valid OCD, we emit it as a result and generate the new candidates through generateNextLevel(), which is described in Algorithm 3.

The function generateNextLevel() builds a set containing all the OCD candidates of the form $\mathbf{X}A \sim \mathbf{Y}$ and $\mathbf{X} \sim \mathbf{Y}A$, where A is an attribute that does not already belong to the lists $\mathbf{X}$ and $\mathbf{Y}$, this corresponds to creating a new level of the tree presented in Section 4.2 using the pruning rules of Section 4.2.1. The function further checks if the ODs $\mathbf{X} \overset{?}{\mapsto} \mathbf{Y}$ or $\mathbf{Y} \overset{?}{\mapsto} \mathbf{X}$ hold. If so, it applies the pruning rules and returns the remaining candidates. We emit the valid ODs found in Lines 9 and 16 of Algorithm 3.

The new candidates are added to the queue of candidates to check for the next level in line 10 of Algorithm 1. The loop terminates when there are no candidates left.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the results of our approach, with particular emphasis on its scalability in the number of rows and columns, and we compare it with previous work on order dependency detection. We analyze the results of ocddiscover over 6 real-world datasets and 5 synthetic datasets.

Our algorithm is implemented in Java 1.7 and is designed to work on the Metanome data profiling framework [12] as a multi-threaded program.

All experiments were run on a `i686` Intel Xeon E52440 2.40 GHz machine with 12 cores in hyper-threading and 128 GB RAM, over a Linux kernel `v4.15.0`. The execution environment is a 64-bit Oracle JDK version `1.8.0_171`, with the JVM heap space limited to 110 GB.

Algorithm 3 generateNextLevel

Input: $\mathbf{X}, \mathbf{Y}$: an OCD candidate
Input: $\mathcal{U}'$: the set of reduced attributes of relation $\boldsymbol{R}$
Output: C: the candidate OCD generated from $\mathbf{X} \sim \mathbf{Y}$

```
1:  function generateNextLevel(X, Y, U')
2:    C ← ∅
3:    A+ ← U' − set(X) − set(Y)
4:    if ¬checkCandidate(X, Y, r) then          ▷ X ↦ Y
5:      for each A ∈ A+ do
6:        C.add((XA, Y))
7:      end for
8:    else                                      ▷ X ↦ Y
9:      emit X ↦ Y
10:   end if
11:   if ¬checkCandidate(Y, X, r) then          ▷ Y ↦ X
12:     for each A ∈ A+ do
13:       C.add((X, YA))
14:     end for
15:   else                                      ▷ Y ↦ X
16:     emit Y ↦ X
17:   end if
18:   return C
19: end function
```

5.1 Datasets

We use the datasets provided by the Information Systems Group of Hasso-Plattner-Institut.[2] These datasets are the same used by the previous work on order dependency discovery by Langer and Naumann [10]. We have also created three simple additional synthetic datasets, called `YES`, `NO`, and `NUMBERS` created to highlight the differences of our approach with previous works. In particular, `YES` and `NO` reproduce, respectively, the examples in Tables 5 (a) and 5 (b), while `NUMBERS` is shown in Table 7.

Table 6 presents the datasets and their properties; for each dataset, the table reports: the dataset name, the number of rows $|\boldsymbol{r}|$, the number of attributes $|\mathcal{U}|$, the number of functional dependencies discovered by the fastfds algorithm [**?**] $|\mathcal{F}_d|$, the number of ODs discovered $|\mathcal{O}_d|$ by order. For fastod we provide: (a) the number of FDs discovered $|\mathcal{F}_d|$, (b) the number of ODs discovered $|\mathcal{O}_d|$. For ocddiscover we provide: (a) the number of OCDs discovered $|\mathcal{O}_c|$, which are missed by order [10], since they are order dependencies with repeated attributes; (b) the number of ODs discovered $|\mathcal{O}_d|$; and (c) the total number of dependency candidates checked during the execution of the algorithm.

Execution time is averaged across 5 independent runs and we set a time threshold at 5 hours. When the time limit is reached, for order, and fastod we are unable to present the number of dependencies discovered so far, while for ocddiscover we report the number of dependencies discovered and the number of checks made until the limit.

5.2 Comparison with Previous Work

We discuss the results of the extensive comparison with the previous state-of-the-art algorithms for detecting order dependencies. The code used for the comparison has been provided by the respective authors.[3]

[2] https://hpi.de/naumann/projects/repeatability/data-profiling/fd-algorithms.html
[3] the source code for order is available at: https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html#c168192 while the source of fastod has been provided to us by the authors through direct communication.

Dataset properties			FASTFDS [?]	ORDER [10]		FASTOD [?]			OCDDISCOVER			
Dataset	$\|r\|$	$\|\mathcal{U}\|$	$\|\mathcal{F}_d\|$	$\|\mathcal{O}_d\|$	**time (ms)**	$\|\mathcal{O}_d\|$	$\|\mathcal{F}_d\|$	**time (ms)**	$\|\mathcal{O}_d\|$	$\|\mathcal{O}_c\|$	**time (ms)**	**checks**
DBTESMA	250,000	30	89,571	−*	5 h*	400	89,571	4,641,485	138	0	**337,289**	4,118
DBTESMA_1K	1000	30	11,099	−*	5 h*	30	11,099	5,799	138	0	**1,835**	4,118
FLIGHT_1K	1,000	109	−*	−†	−†	−*	−*	5 h*	3,216,069*	29,404,555*	5 h*	7,473,951
HEPATITIS	155	20	8,250	0	**182**	32,717	8,250	211,903	0	5	361	556
HORSE	300	29	128,727	31	46,907	−*	−*	5 h*	31	7	**618**	1,185
LETTER	20,000	17	61	0	**1,215**	−*	−*	5 h*	0	0	1,720	272
LINEITEM	6,001,215	16	−*	1	**982,075**	−*	−*	5 h*	1	0	1,039,517	255
NCVOTER_1K	1,000	19	758	18	**796**	2,333	758	90,000	18	1	872	338
NO	5	2	1	0	323	0	1	24	0	0	**4**	2
YES	5	2	0	0	329	1	0	28	1	1	**3**	2
NUMBERS	7	4	4	0	331	6	4	325	0	0	**28**	12

Table 6: Datasets and execution statistics for the OCDDISCOVER, ORDER [10], and FASTOD [?] algorithms. "*" indicates that the execution has reached the time limit of 5 hours, while "†" that it has exceeded the memory limit of 110GB. When the time limit is reached, for OCDDISCOVER we present partial results.

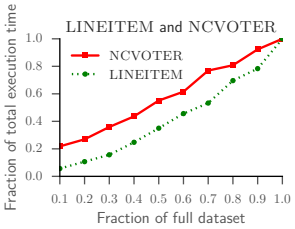

Figure 2: Normalized execution times for row scalability.

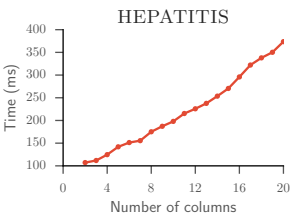

Figure 3: Execution times for column scalability for HEPATITIS.

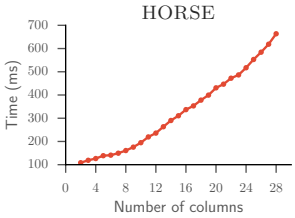

Figure 4: Execution times for column scalability for HORSE.

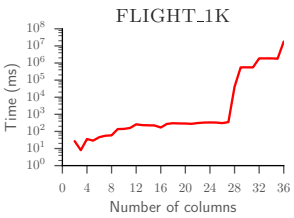

Figure 5: Execution times for column scalability for FLIGHT_1K.

A	B	C	D
1	3	1	1
2	2	3	2
2	3	2	2
2	5	2	2
3	1	2	3
4	4	4	2
4	5	3	2

Table 7: NUMBERS dataset.

To compare our algorithm with ORDER and FASTOD we need to transform OCDs back to ODs. In fact, in a relation $\boldsymbol{R}$ over attributes A, B and C where $A \leftrightarrow B$, $A \sim C$ and $B \sim C$, the set of OCDs $\mathcal{O} = \{A \sim B, A \sim C, B \sim C\}$ is minimal following Definition 3.4. However, COLUMNSREDUCTION() would discover the order-equivalence $A \leftrightarrow B$ and choose one attribute as a representative, e. g. A. Thus OCDDISCOVER would return as valid OCDs only $\mathcal{O} = \{A \sim C\}$. From this information, we infer the remaining dependency $B \sim C$ using the axioms $\mathcal{J}_{\text{OD}}$. We perform this expansion and compare the results produced by OCDDISCOVER, ORDER and FASTOD. The function performing the expansion is not shown in Algorithm 1, but the times reported in Table 6 include it. This step did not impact the running time of OCDDISCOVER.

5.2.1 Comparison with Langer and Naumann [10]. For ORDER, dependencies are considered to be *completely non-trivial*, if their left- and right-hand side attribute lists are disjoint. However, we argue that limiting the discovery of order dependencies to candidates where the left-hand side and the right-hand side are completely disjoint gives incomplete results. Our algorithm, instead, is complete.

Following Theorem 3.8 in Section 3.2, order dependencies of the form $\mathbf{XY} \mapsto \mathbf{Y}$ can be inferred from order compatibility dependencies of the form $\mathbf{X} \sim \mathbf{Y}$. Note that these dependencies have repeated attributes between its left- and right-hand sides.

We show the difference between our approach and previous work with the YES and NO datasets. As reported in Table 6, the ORDER algorithm does not find any order dependency in either of the YES and NO datasets. OCDDISCOVER, instead, finds correctly the order compatibility dependency $A \sim B$, i.e., the order equivalence $AB \leftrightarrow BA$, in YES.

Our approach detects all the dependencies found by ORDER, and additional dependencies on HEPATITIS, NCVOTER_1K, and HORSE. For FLIGHT_1K and NCVOTER_ALLC we found several dependencies but we were not able to compare the results with ORDER because the latter does not report the discovered dependencies when the time limit is reached.

Provided that the order compatibility dependencies found by OCDDISCOVER are translated to the corresponding OD in the form $\mathbf{XY} \mapsto \mathbf{Y}$, OCDDISCOVER effectively discovers a minimal set of ODs even following the definition of minimality provided by Langer and Naumann [10].

When a candidate dependency is found to be false, pruning rules are applied. For this reason, notwithstanding the factorial dimension of the search space, datasets with several columns, such as datasets HEPATITIS and HORSE, are successfully and completely tested. When pruning cannot be applied, the generation of candidates grows – e.g., more than 7 million candidates are generated in FLIGHT_1K. For this dataset, OCDDISCOVER detects more than 32 million ODs. In this particular case, the number of checks is smaller than the number of discovered dependencies because we also count the dependencies inferred from constant and order-equivalent columns reported by the COLUMNSREDUCTION() function.

Furthermore, Table 6 shows that using order compatibility dependencies does not hinder the performance of the detection. In

all dataset tested, the performance of our algorithm with respect to the execution time is comparable to ORDER; in some cases, we obtain significant speedups up to a factor of 75, e.g. in HORSE.

5.2.2 Comparison with Szlichta et al. [?] As shown in Table 6, OCDDISCOVER and FASTOD compute a different number of order dependencies. We claim that this difference, which also affects also the results published in [**?**], is due to an implementation error in the code. Table 7 presents an instance of a relational table where the implementation of FASTOD we received finds several order dependencies that are not actually present in the data, e.g. $[B] \mapsto [AC]$. Other datasets were also affected by this issue, but, unfortunately, we were not able to isolate and resolve the root cause of this incorrect behavior. In addition, FASTOD considers all columns as if they contain data of type `String`, thus using a lexicographical ordering, while ORDER and OCDDISCOVER perform type inference over the datasets provided, and use the natural ordering for real and integer numbers. We have also implemented for OCDDISCOVER the possibility of forcing lexicographical ordering, i.e., treat all data as if they were of type `String`, but we do not report these results since this change does not affect the execution time of our approach.

Furthermore, the scalability experiments reported by Szlichta et al. [**?**] used trimmed-down versions of the datasets. For the row scalability experiments, the datasets were reduced to 1,000 rows, while for the column scalability experiments columns were chosen at random. For this reason, we report in Table 6 both the DBTESMA and DBTESMA_1K datasets, which are respectively the full dataset with 250,000 rows, and a trimmed-down version with the first 1,000 rows. For column scalability, we tested OCDDISCOVER and FASTOD against HEPATITIS, FLIGHT_1K, and NCVOTER_1K, which were used in full, so we were able to compare the performance of the two algorithms.

As shown in Table 6, OCDDISCOVER gains significant speed-ups. This highlights the fact that, while the worst-case complexity of OCDDISCOVER is $(O)(|\boldsymbol{r}|!)$, which is greater than that of FASTOD, $O(2^{|\boldsymbol{r}|})$, the execution time depends on the actual number of dependencies contained in the dataset.

5.3 Performance

In the following, we analyze the scalability of our approach with respect to the number of rows, the number of columns, and the number of parallel threads used.

5.3.1 Scalability in the number of rows. We performed our analysis on the synthetic dataset LINEITEM with 6,001,215 rows and 16 columns. We also test the algorithm on the NCVOTER dataset. This dataset has 938,084 rows and 94 columns, but since our algorithm did not terminate on the full datasets, we consider only a subset of 20 randomly chosen columns. Figure 2 shows the results for the scalability experiment on LINEITEM and NCVOTER. Ten samples of the original dataset have been created, ranging from 10% to 100% of the rows with a step of 10%. Five repetitions have been performed for each of the samples and the average is reported. Variance is very small and thus not shown.

The experiments show that the algorithm scales almost linearly with the number of rows, and it is able to find a complete set of OCDs over datasets with millions of rows.

The execution time would be expected to grow log-linearly with respect to the number of rows, due to the indexing phase; but an increasing number of rows may correspond to a smaller number of dependencies; thus, the pruning phase could reduce the number of checks to be computed.

Previous work, instead, has shown the ability to scale linearly on the number of rows performing the check of dependency candidates with sorted partitions computed from the data. This method could have been re-implemented in our approach as well, but it would have been out of the scope of this paper.

5.3.2 Scalability in the number of columns. Scalability over columns is the key challenge in detecting dependencies in relational data since in many cases the dimension of the search scales with the number of columns.

We choose the HORSE and HEPATITIS datasets, that are well-suited to evaluate the influence of an increasing number of columns, given that their execution completes. We also consider FLIGHT_1K that has a very high number of columns and does not terminate.

The evaluation approach is as follows: we start with two random columns from each dataset, and we incrementally add more randomly-chosen columns, until the total number of columns in the dataset is reached.

To avoid skewing the results, we generate 50 samples of each dataset with the process described above and we run our algorithm over these samples. We average the execution time of OCDDISCOVER of each sample with c columns over all the 50 samples.

Figures 3 and 4 show the results of the column scalability experiment on the HEPATITIS and HORSE datasets.

Figure 5 shows how the algorithm behaves when the number of dependencies discovered in the data grows on a single run. Times on the y-axis are in logarithmic scale. OCDDISCOVER is very susceptible to columns that are quasi-constant, i.e., attributes with very few distinct values, but not constant. In this case, OCDDISCOVER cannot eliminate these columns during the column-reduction phase. As argued in Section 4.1, constant columns are ordered by any other attribute; quasi-constant columns are associated with a large number of valid OCDs and consequently the size of the tree to be explored grows enormously.

In fact, the slowdown corresponding to the sample with 28 columns in Figure 5 is caused by the addition of a column with 3 distinct values. This column appears on the right-hand side of more than 94% of the dependencies found in that sample.

5.3.3 Scalability over parallel threads. As described in Section 4.2.2, OCDDISCOVER can be run over multiple threads.

Figure 6 shows the results of the multithreading scalability experiments on the LETTER, LINEITEM, and DBTESMA datasets. On the y-axis times are normalized to the runtime over a single thread, which is the maximum for each case. Table 8 reports the executions times.

	Time (s) vs number of threads			
Dataset	**1**	**2**	**4**	**8**
LETTER	5.7	4.6	3.6	3.4
LINEITEM	2,848.8	1,770.0	1,243.5	1,040.0
DBTESMA	2,228.9	1,240.0	686.0	414.0

Table 8: Execution time of OCDDISCOVER versus number of threads.

As it is shown in Figure 6, using multiple parallel threads shortens the execution time of our dependency discovery algorithm.

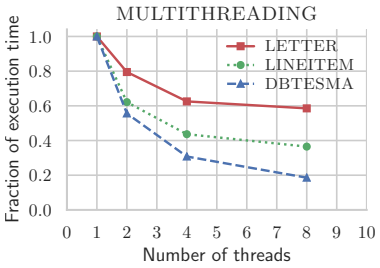

Figure 6: Execution times for LETTER (red straight line), LINEITEM (green dotted line), DBTESMA (blue dashed line), when executed over multiple threads, normalized over the runtime over a single thread.

The amount of this improvement varies based on the characteristics of each dataset and in particular, depends on the of OCD candidates checked.

If we compare LETTER and LINEITEM, we see that while the number of checks performed on each dataset is comparable (272 for LETTER versus 255 for LINEITEM), the number of their rows differ by several orders of magnitude ($\sim 20k$ lines for LETTER, versus $\sim 6M$ for LINEITEM). This implies that checking the validity of an OCD candidate for LINEITEM takes longer than checking a candidate for LETTER. Thus the relative gain when splitting the work over multiple threads is greater for LINEITEM.

Instead, comparing LINEITEM and DBTESMA we can see that for the latter dataset the number of checks performed is much higher, thus the workload of candidate checking can be spread over multiple threads, leading to greater relative improvements.

5.4 Quasi-constant Columns

The quasi-constant column scenario is challenging for our algorithm. We further develop the idea of measuring how varied the values in a column are by measuring its *entropy*.

Definition 5.1 (Entropy). Given an attribute $A \in \mathcal{U}$ of an instance $\boldsymbol{r}$ of a relation $\boldsymbol{R}$, the entropy of A is defined as:

$$H(A) = -\sum_{[a]} p_{[a]} \log\left(p_{[a]}\right) \tag{8}$$

where $[a]$ are the equivalence classes of distinct values in A and $p_{[a]}$ is the probability of extracting an instance of class a, computed as the relative frequency of instances class $[a]$ over the total number of tuples in $\boldsymbol{r}$:

$$p_{[a]} = \frac{|\{t \in \boldsymbol{r} \mid t_A \in [a]\}|}{|\boldsymbol{r}|}$$

For constant columns there is only one equivalence class and $p_{[a]} = 1$, thus $H(A) = 0$. If all values are distinct, for each $[a]$, $|[a]| = 1$ and $p_{[a]} = 1/|\boldsymbol{r}|$:

$$H(A) = -\sum_{[a]} \frac{1}{|\boldsymbol{r}|} \log\left(1/|\boldsymbol{r}|\right) = \log\left(|\boldsymbol{r}|\right)$$

We test the idea that progressively less diverse columns cause the slowdown of OCDDISCOVER by taking the FLIGHT_1K dataset and running it over multiple samples build with the following criteria: we calculate the entropy of each column in FLIGHT_1K and then we build samples of increasing size in the number of columns by adding progressively the columns with decreasing entropy, i.e., we start with the columns with the greatest number of distinct values and we progressively add columns with less distinct values. Eventually, the constant columns are added.

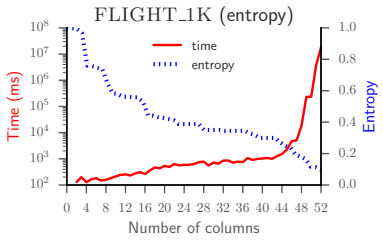

Figure 7: Execution times (red straight line) for the FLIGHT_1K when adding columns of decreasing entropy (blue dotted line). On the y-axis, times (left-hand side) are in logarithmic scale, entropy (right-hand side) is normalized with respect to the maximum value over the dataset.

The result of the execution on OCDDISCOVER over this set of samples is reported in Figure 7. With 50 columns the OCDDISCOVER completes in 4 minutes, adding the 51st column the execution time grows by an order of magnitude to over 1 hour. With the addition of the 52nd column, the algorithm reaches the time limit of 5 hours. The 50th, 51st, and 52nd columns have respectively 4, 2, and 2 distinct values respectively.

With respect to applications, this insight could be exploited to develop algorithms that return results for the most diverse columns, which can be the most interesting with respect to other properties of the data such as unique column combinations (UCC). Detection of unique column combinations is usually performed to find primary keys candidates that may be also interesting candidates from the point of view of ordering and query optimization.

In summary, the column scalability experiments show that OCDDISCOVER can find a complete set of ODs over datasets with tens of columns. Furthermore, OCDDISCOVER can be easily adapted to perform the detection over a set of interesting columns, where the interestingness of an attribute can be determined providing a function measuring the properties chosen by the user.

6 RELATED WORK

Functional Dependencies: Theory, Discovery and Applications. Literature on functional dependencies is vast [1, 9, 11, 13]. Applications of functional dependencies span several fields from query optimization [4], to data cleaning [3], to data quality management [5]. Given this variety of applications, several algorithms for the discovery of functional dependencies have been developed. The first algorithm to be proposed was TANE [9], which has served as inspiration for many subsequent efforts [11, 13]. Research on better functional dependency discovery algorithms is still ongoing [12]. Functional dependencies have been extended in several ways: from conditional functional dependencies [2], to approximate (or partial) functional dependencies [11].

Order Dependencies Theory and Applications. In the 1980's, Ginsburg and Hull were the first to consider the idea of analyzing orderings between the attributes of a relation as a kind of dependency [6–8]. They introduced the concept of point-wise ordering [8], that is a relation where a set of attributes orders another set of attributes. Recently, Szlichta et al. introduced the concept of order dependency [16], which is the one used throughout this paper. Order dependencies are defined over lists of attributes, and can be formalized in a similar way to functional dependencies [16, 18]. In this paper we focused on dependencies where the

attributes are all ordered in the same direction, also called "unidirectional" order dependencies; however order dependencies can be generalized to "polarized" or "bidirectional" ODs where a different direction of the ordering can be specified for each attribute on either side of the dependency [15]. One of the main theoretical problems concerning dependencies in relational data is the problem of inference. For order dependencies this problem is shown to be co-NP-complete [**?**].

Applications of Order Dependencies. Notwithstanding their recent theoretical formulation, order dependencies have been already used in several applications, such as query optimization.

Sorting is a fundamental database operation. Since the seminal works [7], research has focused on developing optimization strategies for dealing with queries with an `ORDER BY` clause [14]. Order dependencies can be used for this purpose, as it has been shown with an implementation of a query optimizer in IBM DB2. Optimizing queries with order dependencies yields significant speedups in execution times over the well-known TPC-DS benchmark and on queries taken from real-world scenarios [17].

Discovery Algorithms for Order Dependencies. These applications are driving the need for discovering order dependencies in existing datasets. The first work proposing an algorithm to solve this problem is the one by Langer and Naumann [10]. Their approach, called ORDER, follows the path of TANE, computing the potential order dependency candidates from the permutations of attributes and traversing the lattice in a level-wise, bottom-up manner. Pruning rules are applied to reduce the number of candidates to check, with the caveat of eliminating only redundant relations that can be later inferred from the dependencies discovered together with the axioms. However, as shown in this paper, this approach does not consider all the possible order dependency candidates, discarding repeated attributes. While this gives a significant advantage in execution time, reducing the worst-case time complexity of the algorithm to $O(|\boldsymbol{r}|!)$, its major drawback is the possibility of losing completeness. More recently Szlichta et al. [**?**] proposed an algorithm called FASTOD that is complete and faster than ORDER. This algorithm exploits a novel polynomial mapping that transforms ODs with lists of attributes into canonical forms of ODs that are established between sets of attributes. FASTOD has exponential worst-case time complexity, $O(2^{|\boldsymbol{r}|})$, in the number of attributes. Recently, this work was extended in order to discover bidirectional order dependencies [**?**].

7 CONCLUSIONS

In this work, we presented a novel method for discovering order dependencies (ODs). We based our approach on the fact that an order dependency is valid if and only if both a functional dependency (FD) and an order compatibility dependency (OCD) are valid. Thus, we designed a novel and efficient algorithm – called OCDDISCOVER – where the search for ODs is guided by checking the validity of OCDs. Our approach outperforms existing two state-of-the-art algorithms, ORDER [10] and FASTOD [**?**] with respect to ORDER, we are complete, meaning that we detect order dependencies that are ignored. We have shown that these dependencies cannot be inferred by other detected dependencies. While the worst-case complexity of OCDDISCOVER is greater than FASTOD, the execution time on real datasets depends on the actual number of dependencies found, thus our algorithm outperforms FASTOD. Furthermore, we presented an extensive set of experiments that illustrate that our approach can be executed in parallel over multiple threads. We have also suggested that considering the entropy of attributes can lead to further developments in discovery the most interesting order dependencies. As a future work, we would like to consider dynamic inputs, where additional rows and columns may be added at runtime.

REFERENCES

[1] W. W. Armstrong. Dependency Structures of Data Base Relationships. In *IFIP congress*, volume 74, pages 580–583. Geneva, Switzerland, 1974.

[2] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 746–755. IEEE, 2007.

[3] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 458–469. IEEE, 2013.

[4] H. Darwen and C. Date. The role of functional dependencies in query decomposition. *Relational Database Writings*, 1991, 1989.

[5] W. Fan and F. Geerts. Foundations of data quality management. *Synthesis Lectures on Data Management*, 4(5):1–217, 2012.

[6] S. Ginsburg and R. Hull. Ordered attribute domains in the relational model. In *XP2 Workshop on Relational Database Theory*, 1981.

[7] S. Ginsburg and R. Hull. Order dependency in the relational model. *Theoretical computer science*, 26(1):149–195, 1983.

[8] S. Ginsburg and R. Hull. Sort sets in the relational model. *Journal of the ACM (JACM)*, 33(3):465–488, 1986.

[9] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

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

[11] J. Liu, J. Li, C. Liu, Y. Chen, et al. Discover dependencies from data – A review. *IEEE Transactions on Knowledge and Data Engineering*, 24(2):251–264, 2012.

[12] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with metanome. *Proceedings of the VLDB Endowment*, 8(12):1860–1863, 2015.

[13] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[14] D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. *ACM SIGMOD Record*, 25(2):57–67, 1996.

[15] J. Szlichta, P. Godfrey, and J. Gryz. Chasing polarized order dependencies. In *AMW*, pages 168–179, 2012.

[16] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of order dependencies. *Proceedings of the VLDB Endowment*, 5(11):1220–1231, 2012.

[17] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, W. Qiu, and C. Zuzarte. Business-intelligence queries with order dependencies in db2. In *EDBT*, pages 750–761, 2014.

[18] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. Axiomatic system for order dependencies. In *AMW*, 2013.

[19] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.