The VLDB Journal (2016) 25:223–241
DOI 10.1007/s00778-015-0412-3

REGULAR PAPER

Efficient order dependency detection

Philipp Langer[1] · **Felix Naumann**[1]

Received: 9 July 2015 / Revised: 4 November 2015 / Accepted: 11 November 2015 / Published online: 9 December 2015

Abstract Order dependencies (ODs) describe a relationship of order between lists of attributes in a relational table. ODs can help to understand the semantics of datasets and the applications producing them. They have applications in the field of query optimization by suggesting query rewrites. Also, the existence of an OD in a table can provide hints on which integrity constraints are valid for the domain of the data at hand. This work is the first to describe the discovery problem for order dependencies in a principled manner by characterizing the search space, developing and proving pruning rules, and presenting the algorithm ORDER, which finds *all* order dependencies in a given table. ORDER traverses the lattice of permutations of attributes in a level-wise bottom-up manner. In a comprehensive evaluation, we show that it is efficient even for various large datasets.

Keywords Data profiling · Functional dependencies · Metadata

1 Order dependency discovery

Sorting, and thereby, order is an integral part of databases, not only from the viewpoint of the user, who explicitly initiates a sort with the ORDER BY statement in SQL, but also as an internal aspect of database management systems, e.g., for implementing indexes or sort-merge joins. Order dependencies, which describe a relationship of order between attributes in a relational schema, were first introduced by Ginsburg and Hull [6]. Order dependencies have again gained traction in research in the last few years, and several interesting use cases have been proposed in the context of query optimization and integrity constraints.

Among the various use cases of data profiling are query optimization, data preparation for data cleansing, and establishing a comprehension of the structure of a dataset to aid data integration, scientific data management, and data analytics [13]. Integrity constraints, which are brought to light by data profiling, can be used to enhance and ensure data quality.

Below, we show several order dependencies in real-world datasets, which are suitable for definition as constraints to ensure data quality. We also demonstrate how order dependencies can be leveraged for query rewrites in the optimizer of a database management system for faster query processing.

Consider the order dependencies shown in Table 1, which (among many others) were found in a 10,000 table sample of the *WikiTables* project [15]. The *WikiTables* project provides a collection of more than 1.3 million tables extracted from the English version of the online encyclopedia Wikipedia. Table 1 shows tables from Wikipedia articles alongside the order dependencies we found in them.

The first article, "Demographics of Azerbaijan" contains a table of vital statistics for Azerbaijan for the years 1960–2011 in the column `Year`, and the average population of Azerbaijan in the column `Average population`. The order dependency "`Year` *orders* `Average population`" expresses the fact that the population of Azerbaijan never declined until 2011. In fact, the population of Azerbaijan steadily increased, which we can deduce from the order dependency `Average population` *orders* `Year`, which holds as well (not listed). "`Rank` *orders* `Time`" is another

✉ Felix Naumann
felix.naumann@hpi.de

Philipp Langer
philipp.langer@student.hpi.de

1 Hasso Plattner Institute, Potsdam, Germany

Table 1 Order dependencies found in a sample of 10,000 tables of the *WikiTables* project

Wikipedia article (table)	Order dependency
Demographics of Azerbaijan (Vital statistics)	`Year` *orders* `Average population`
1994 Parliamentary Election (Kurunegala)	**% of votes** *orders* `# of seats`
State highways in New Jersey (State Highways)	`Length (mi)` *orders* `Length (km)`
Men's 200m (Swimming at Olympics 2012)	`Rank` *orders* `Time`
2008 African Futsal Championship (Group 2)	`Date` and `Time` *order* `Match no.`

Table 2 Employee table with name, rank, and salary

emp_name	rank	salary (k)
Smith	1	40
Johnson	1	40
Williams	1	45
Brown	2	60
Davis	2	60
Miller	3	70
Wilson	4	100

order dependency that we encountered frequently in the *WikiTables* dataset, in tables on various sports events. In this case, the sports event is the men's 200-m swimming finale at the 2012 Summer Olympics in London.

Szlichta et al. [21] describe how order dependencies can be used for query optimization. For example, consider the SQL statement "SELECT `emp_name` FROM `employees` ORDER BY `rank, salary`," which queries the names of all employees shown in Table 2, sorted by `rank` and – within equal ranks – by `salary`. A database management system may initiate a sort operation on `rank` and – within equal ranks – on `salary`. If, however, the order dependency "`salary` *orders* `rank`" is known to the optimizer, it can rewrite the above ORDER BY statement to "ORDER BY `salary`," because the order dependency "`salary` *orders* `rank`" implies that the ordering of `rank` is subsumed by the ordering of `salary`. By reducing the ORDER BY statement, it is, for instance, more likely that the entire sorting operation can be satisfied with the help of an index. Likewise, a query "…ORDER BY `rank`" can be rewritten to "ORDER BY `salary`," which is beneficial if there is an index on `salary`, but not on `rank`. Further, Dong and Hull show how order dependencies can be used to reduce indexing space [5].

These use cases require knowledge about the order dependencies in a dataset. In the case of query optimization, Szlichta et al. [21] show how from order dependencies declared by a database administrator, new order dependencies can be inferred. They also show that order dependencies may originate from the use of algebraic expressions or SQL functions within a query: For instance, the order dependency "`date` *orders* (`date` + 30 days)" always holds. However, this leaves room for optimization on attributes that are not generated from other attributes, such as in the example provided in Table 2. Their methods for query optimization [21] benefit from our order dependency detection algorithms, because (i) order dependencies need not be entered manually by the database administrator, which is a tedious and error-prone task and (ii) the number of order dependencies that are automatically detected is complete, providing more opportunities for query rewrites. As for all dependency detection algorithms, order dependencies can be detected on instances only. In practice, this means that a database administrator or domain expert should verify the order dependencies discovered by our algorithm, before they can be defined as integrity constraints.

In general, the detection of dependencies, such as functional dependencies, inclusion dependencies, and unique column combinations, in datasets is a fundamental task of *data profiling* [1]. Dependency detection problems usually have two subproblems: the validation problem and the search problem. Whereas the validation problem checks the validity of a *single* dependency (e.g., "is the functional dependency $A \rightarrow B$ valid?"), the discovery problem defines the task of finding the set of all dependencies (over combination of columns) that hold in a given dataset. The resulting exponential number of dependency candidates poses a challenging problem and is the main reason for an often intolerably long runtime of dependency detection algorithms. However, intelligent generation of candidates and early pruning can alleviate the worst case in practice. Order dependencies are no exception to the rule of high complexity of dependency detection algorithms: Because order dependencies – as opposed to functional dependencies – are defined over *lists* of attributes rather than *sets*, the order of the attributes on the left- and right-hand side matters, resulting in a *factorial* number of order dependency candidates.

Finding an algorithm for the validation of a single dependency that is polynomial in the number of rows is typically straightforward. For example, for functional dependencies, inclusion dependencies, and unique column combinations, such an algorithm follows directly from their respective definitions. Naturally, large datasets with many rows and the large number of dependency candidates (and thus, the number of dependency validations) make it desirable to develop intelligent and more efficient validation methods.

In summary, this article is the first to tackle the problem of automatically discovering order dependencies and makes the following contributions:

- After introducing some notation (Sect. 2), we propose the novel notion and definition of *minimality* for order dependencies (Sect. 3).
- We define a data structure and an algorithm to efficiently validate an order dependency (Sect. 4).
- We describe pruning rules (Sect. 5) and the algorithm ORDER, which uses these rules to detect all ODs in a table (Sect. 6).
- We evaluate the performance of our algorithm on various real-world datasets (Sect. 7); all datasets and code are available online (www.metanome.de).

2 Preliminaries

Here, we introduce the notation used throughout this paper and establish an intuition for order dependencies by including several examples. We investigate the influence of the comparison operator ($\leq, <, \ldots$) in order dependencies in Sect. 2.4. Section 2.5 formulates relationships between order dependencies and other constraints. We describe the search space for the OD detection problem as a lattice of permutations of attributes in Sect. 2.6.

2.1 Notation

We denote the schema of a relation with $\mathcal{R}$. A relational instance over $\mathcal{R}$ is denoted by r. A unary order dependency over a schema $\mathcal{R}$ is a statement of the form $A \rightarrow_\theta B$, where $A, B \in \mathcal{R}$, and $\theta \in \{\leq, <, =, >, \geq\}$. The order dependency $A \rightarrow_\theta B$ is *valid* in a relational instance r over $\mathcal{R}$ iff for any two tuples s, t in r, $s[A]\,\theta\,t[A] \Rightarrow s[B]\,\theta\,t[B]$ as defined in [20]. Referring to Table 2, the order dependency `salary` $\rightarrow_\leq$ `rank` holds, but not `rank` $\rightarrow_\leq$ `salary`, because the order of employees Johnson and Williams is not fixed wrt. their rank. Slightly unintuitively, `rank` $\rightarrow_>$ `salary` also holds, because the ranks of Johnson and Williams are not in a $>$ order.

With this definition, functional dependencies (FDs) are a special case of order dependencies with θ as "$=$". For the remainder of this paper, θ is restricted to "$<$" and "$\leq$". The discovery algorithms presented can trivially be adjusted to discover order dependencies under "$>$" and "$\geq$", instead. The order dependency discovery algorithm ORDER presented in Sect. 6 finds order dependencies under $\rightarrow_<$; we show the equivalence of $\rightarrow_<$ and $\rightarrow_\geq$ later. The algorithm can also be used for FD discovery if a validity check for functional dependencies is used, because the lattice of attribute lists that ORDER traverses contains all FD candidates as well. Specialized and more efficient FD discovery algorithms are reviewed in [17].

As opposed to functional dependencies, order dependencies are sensitive to the order of the attributes on the left- and right-hand side. Therefore, order dependencies are defined over lists (rather than sets) of attributes. Such a list of attributes is denoted in bold capital letters from the end of the alphabet, such as $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$, following [20]. The term [ ] denotes the empty list. The size of an attribute list $\mathbf{X}$, denoted as $|\mathbf{X}|$, is the number of attributes in $\mathbf{X}$ and $|[\ ]| = 0$. We write $s[\mathbf{X}] \leq t[\mathbf{Y}]$ (for $|\mathbf{X}| = |\mathbf{Y}|$) to express that the tuple constructed by projection of tuple s on $\mathbf{X}$ is lexicographically smaller than or equal to the tuple constructed by projection of tuple t on $\mathbf{Y}$. We may also represent an attribute list explicitly from singleton attributes A, B, and C as ABC. Consequently, an OD may have the form $ABC \rightarrow_\theta D$ (implicit concatenation of A, B, and C to a list of attributes), or $\mathbf{X} \rightarrow_\theta \mathbf{Y}$ for two attribute lists $\mathbf{X}$ and $\mathbf{Y}$. An order dependency is *completely non-trivial*, if its left- and right-hand side attribute lists are disjoint.

2.2 OD variants

Bidirectional order dependencies allow differing sort orders (ascending, descending) to be assigned to the left- and right- hand side of an OD [22]. Hence, bidirectional order dependencies generalize order dependencies, but are not the focus of this paper. A *sequential dependency* (SD) $X \rightarrow_g Y$ expresses that in a table sorted on X, the difference between two consecutive values in Y lies in the interval g [7]. Sequential dependencies with $g = [0, \infty]$ correspond to order dependencies. Thus, SDs are more general than order dependencies. Golab et al. also describe partial SDs (SDs that nearly hold) and conditional SDs (CSDs) in [7]. They present a framework to discover tableaux for CSDs, assuming the embedded SD is given. Because discovery of (embedded) SD candidates is not considered, and we neither consider the discovery of partial ODs nor of conditional ODs in this work, their work is orthogonal to ours.

Order dependencies are sensitive to the kind of ordering we impose on the tuples in a table. Consider Table 3 as an example. Let the ordering imposed on the `salary` attribute be the natural ordering of integers. We could interpret the ordering of `employee_type` as "Developer" $<$ "Manager" $<$ "Chief Executive Officer." This means that we compare the employees by their ranks. The OD `employee_type` $\rightarrow_<$ `salary` is valid under this ordering, stating higher ranks earn higher salaries. If, however, we simply compare the employees using a lexicographical string comparison on `employee_type`,

Table 3 Employee table to demonstrate different orderings

`employee_type`	`salary`
Developer	50,000
Manager	70,000
Chief executive officer	200,000

we have "Developer" > "Chief Executive Officer," and the OD `employee_type` $\rightarrow_<$ `salary` is no longer valid.

We assume a natural order over three domains: Numerical order of numbers (e.g., integers, floating point decimals), lexicographical order on strings, and chronological order of dates. If the domain of an attribute is not known, it can be determined in a linear scan over the input table before running the OD detection algorithm.

Order dependencies may span multiple attributes in $\mathcal{R}$. The concept of such *n-ary* order dependencies gives rise to the question of how tuples and not just values should be ordered. There are at least two possibilities, namely lexicographical and pointwise ordering [14]. The *lexicographical ordering* of tuples states that $(s_1, \ldots, s_n) < (t_1, \ldots, t_n)$ if there exists some $i \leq n$ such that $s_i < t_i$ and for all $j < i, s_j = t_j$ [14]. We assume this ordering semantics, as it is also used in [20] and corresponds to the SQL semantics.

In contrast, a *pointwise* ordering of tuples states that $(s_1, \ldots, s_n) < (t_1, \ldots, t_n)$ if for all $1 \leq i \leq n, s_i < t_i$. As opposed to lexicographical ordering, which is total, pointwise ordering of tuples is only a partial order. Thus for the pointwise ordering semantics, the discovery problem for ODs is fundamentally different from that for lexicographical ODs. Our algorithm ORDER (Sect. 6) discovers lexicographical ODs, enabling the major use-case of query optimization, which is explained in Sect. 1. Extensions to our algorithm to cover some of the other variants are also discussed in Sect. 6.

2.3 Intuition and examples

The order dependency $A \rightarrow_\theta B$ may be read as "A orders B." An intuitive way to think of order dependencies is that after sorting a table by some attribute A, it may be sorted by some other attribute B simultaneously, which implies $A \rightarrow_\theta B$.

The concepts of a *split* and a *swap* have been introduced in [20] as the only two ways an order dependency under the operator "$\leq$" can be falsified. We additionally introduce a *merge* as one of the ways an order dependency under the operator "$<$" can be falsified. Using these three concepts, we give alternative definitions of validity for functional dependencies and order dependencies, which are helpful for our algorithm description.

Table 4 Tuples t_1 and t_2 form a *split* among (AB, C) and a *merge* among (C, AB)

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

Table 5 Tuples t_2 and t_3 form a *swap* among (AB, CD) and (CD, AB)

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

Definition 1 *(Split)* Tuples s and t in a relational instance r form a *split* among the pair of attribute lists $(\mathbf{X}, \mathbf{Y})$, if $s[\mathbf{X}] = t[\mathbf{X}]$, but $s[\mathbf{Y}] \neq t[\mathbf{Y}]$.

Definition 2 *(Merge)* Tuples s and t in a relational instance r form a *merge* among the pair of attribute lists $(\mathbf{X}, \mathbf{Y})$, if $s[\mathbf{X}] \neq t[\mathbf{X}]$, but $s[\mathbf{Y}] = t[\mathbf{Y}]$.

Refer to Table 4 for an example of a *split* among (AB, C) and a *merge* among (C, AB). Not only in this example, but in general, a *split* among $(\mathbf{X}, \mathbf{Y})$ implies a *merge* among $(\mathbf{Y}, \mathbf{X})$ and vice versa. This symmetry is also apparent from the definition of a *split* and *merge* given above. Introducing both as separate concepts facilitates the distinction between the two types of order dependencies: *Splits* invalidate only order dependencies under "$\leq$", and *merges* invalidate only order dependencies under "$<$".

Definition 3 *(Swap)* Tuples s and t in a relational instance r form a *swap* among the pair of attribute lists $(\mathbf{X}, \mathbf{Y})$, if $s[\mathbf{X}] < t[\mathbf{X}]$, but $s[\mathbf{Y}] > t[\mathbf{Y}]$.

Table 5 shows a *swap* among (AB, CD) and (CD, AB). Not only in this example, but in general, a *swap* among $(\mathbf{X}, \mathbf{Y})$ implies a *swap* among $(\mathbf{Y}, \mathbf{X})$.

We can define the validity of functional dependencies and order dependencies by means of *splits*, *swaps*, and *merges*, as Lemmata 1, 2, and 3 show.

Lemma 1 *A functional dependency $\mathcal{X} \rightarrow A$ is valid, iff there is no split among (B, A) for any $B \in \mathcal{X}$.*

Proof Recall the original definition of validity for functional dependencies: A functional dependency $\mathcal{X} \rightarrow A$ $(\mathcal{X} \subseteq \mathcal{R}, A \in \mathcal{R})$ is valid, if for all tuples s, t in a relational instance r, $s[B] = t[B]$ for all $B \in \mathcal{X} \implies s[A] = t[A]$.

For brevity, we write $s[\mathcal{X}] = t[\mathcal{X}]$ to mean $s[B] = t[B]$ for all $B \in \mathcal{X}$.
(1) "$\Leftarrow$": Let there be no *split* among $\mathcal{X}$ and A, i.e., for all $s, t \in r$, $s[\mathcal{X}] = t[\mathcal{X}] \implies s[A] = t[A]$. Therefore, by the definition of validity for FDs, $\mathcal{X} \rightarrow A$ is valid.
(2) "$\Rightarrow$": Assume there is a *split* among (B, A) for some $B \in \mathcal{X}$. Then, $s[\mathcal{X}] = t[\mathcal{X}]$, but $s[A] \neq t[A]$ for some tuples s, t. Tuples s and t invalidate the FD $\mathcal{X} \rightarrow A$. □

Lemma 2 *An order dependency* $\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$ *is valid, iff there is neither a split nor a swap among* $(\mathbf{X}, \mathbf{Y})$.

Proof Recall the original definition of validity for order dependencies under "$\leq$": An order dependency $\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$ is valid, if for all tuples s, t in a relational instance r, $s[\mathbf{X}] \leq t[\mathbf{X}] \implies s[\mathbf{Y}] \leq t[\mathbf{Y}]$.
(1) "$\Leftarrow$": Assume $\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$ is not valid. Then, $\exists u, v$, s.t. $u[\mathbf{X}] \leq v[\mathbf{X}]$, and $u[\mathbf{Y}] > v[\mathbf{Y}]$. Case 1: $u[\mathbf{X}] = v[\mathbf{X}]$. Because $u[\mathbf{Y}] > v[\mathbf{Y}]$, u and v form a *split* among $(\mathbf{X}, \mathbf{Y})$. Case 2: $u[\mathbf{X}] < v[\mathbf{X}]$. Because $u[\mathbf{Y}] > v[\mathbf{Y}]$, u and v form a *swap* among $(\mathbf{X}, \mathbf{Y})$.
(2) "$\Rightarrow$": Case 1: Assume there is a *split* among $(\mathbf{X}, \mathbf{Y})$. Then, $s[\mathbf{X}] = t[\mathbf{X}]$, but $s[\mathbf{Y}] \neq t[\mathbf{Y}]$ for some tuples s and t. s and t invalidate the OD $\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$. Case 2: Assume there is a *swap* among $(\mathbf{X}, \mathbf{Y})$. Then, $s[\mathbf{X}] < t[\mathbf{X}]$, but $s[\mathbf{Y}] > t[\mathbf{Y}]$ for some tuples s and t. s and t invalidate the OD $\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$. □

Lemma 3 *An order dependency* $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ *is valid, iff there is neither a merge nor a swap among* $(\mathbf{X}, \mathbf{Y})$.

Proof Recall the original definition of validity for order dependencies under "$<$": An order dependency $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ is valid, if for all tuples s, t in a relational instance r, $s[\mathbf{X}] < t[\mathbf{X}] \implies s[\mathbf{Y}] < t[\mathbf{Y}]$.
(1) "$\Leftarrow$": Assume $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ is not valid. Then, $\exists u, v$, s.t. $u[\mathbf{X}] < v[\mathbf{X}]$, and $u[\mathbf{Y}] \geq v[\mathbf{Y}]$. Case 1: $u[\mathbf{Y}] > v[\mathbf{Y}]$. Because $u[\mathbf{X}] < v[\mathbf{X}]$ and $u[\mathbf{Y}] > v[\mathbf{Y}]$, there is a *swap* among $(\mathbf{X}, \mathbf{Y})$. Case 2: $u[\mathbf{Y}] = v[\mathbf{Y}]$: Because $u[\mathbf{X}] < v[\mathbf{X}]$ and $u[\mathbf{Y}] = v[\mathbf{Y}]$, there is a *merge* among $(\mathbf{X}, \mathbf{Y})$.
(2) "$\Rightarrow$": Let $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ be valid. Case 1: Assume there is a *merge* among $(\mathbf{X}, \mathbf{Y})$. Then, $s[\mathbf{X}] \neq t[\mathbf{X}]$, but $s[\mathbf{Y}] = t[\mathbf{Y}]$ for some tuples s, t. Since either $s[\mathbf{X}] < t[\mathbf{X}]$ or $t[\mathbf{X}] < s[\mathbf{X}]$, s and t invalidate the OD $\mathbf{X} \rightarrow_{<} \mathbf{Y}$. Case 2: Assume there is a *swap* among $(\mathbf{X}, \mathbf{Y})$. Then, $s[\mathbf{X}] < t[\mathbf{X}]$, but $s[\mathbf{Y}] > t[\mathbf{Y}]$ for some tuples s, t. s and t invalidate the OD $\mathbf{X} \rightarrow_{<} \mathbf{Y}$. □

We say that the OD $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ is "invalidated" by a *merge*, if there is a *merge* among $(\mathbf{X}, \mathbf{Y})$. In analogy, the OD $\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$ is invalidated by a *split*, if there is a *split* among $(\mathbf{X}, \mathbf{Y})$. The OD $\mathbf{X} \rightarrow_{\theta} \mathbf{Y}$ with $\theta \in \{\leq, <\}$ is invalidated by a *swap* if there is a *swap* among $(\mathbf{X}, \mathbf{Y})$.

In Table 6, we provide simple exemplary relations that illustrate the relationship between order dependencies and *splits*, *swaps*, and *merges*. For instance, Table 6(a) depicts

Table 6 Tables with valid and invalid ODs under the operators "$<$" and "$\leq$"

A	B
1	2
1	**2**
3	**2**
4	5

(a) $A \not\rightarrow_{<} B$ (**merge**)

A	B
1	**2**
1	**3**
3	4
4	5

(b) $A \not\rightarrow_{\leq} B$ (**split**)

A	B
1	2
1	2
3	**5**
4	**4**

(c) $A \not\rightarrow_{\leq} B$ (**swap**)

the valid order dependency $A \rightarrow_{\leq} B$, because there are neither *splits* nor *swaps*. However, tuples $(1, 2)$ and $(3, 2)$ form a *merge* among (A, B) (a *split* among (B, A)), which is why $A \not\rightarrow_{<} B$ $(B \not\rightarrow_{\leq} A)$. Table 6(b) contains a *split* among (A, B) in the tuples $(1, 2)$ and $(1, 3)$. That is why $A \not\rightarrow B$ and $A \not\rightarrow_{\leq} B$. However, $A \rightarrow_{<} B$, because there are no *swaps* or *merges* among (A, B). As there is a *swap* among attributes (A, B) with tuples $(3, 5)$ and $(4, 4)$ in Table 6(c) we have $A \not\rightarrow_{\leq} B$ and $A \not\rightarrow_{<} B$. Because there is no *split*, $A \rightarrow B$ is valid.

2.4 Influence of the comparison operator

Until now, we have investigated the properties of order dependencies under the operators "$<$" and "$\leq$" separately. We now prove a useful, unifying relationship between them:

Theorem 1 $X \rightarrow_{<} Y$ *is valid* $\iff$ $Y \rightarrow_{\leq} X$ *is valid*

Proof (1) "$\Rightarrow$": Assume $\mathbf{Y} \rightarrow_{\leq} \mathbf{X}$ is not valid, i.e., $\exists t, u$, s.t. $t[\mathbf{Y}] \leq u[\mathbf{Y}]$ and $t[\mathbf{X}] > u[\mathbf{X}]$.
Case 1: $t[\mathbf{Y}] < u[\mathbf{Y}]$, i.e., $u[\mathbf{X}] < t[\mathbf{X}]$ and $u[\mathbf{Y}] > t[\mathbf{Y}]$. Then tuples u and t violate the OD $\mathbf{X} \rightarrow_{<} \mathbf{Y}$.
Case 2: $t[\mathbf{Y}] = u[\mathbf{Y}]$, i.e., $u[\mathbf{X}] < t[\mathbf{X}]$ and $u[\mathbf{Y}] = t[\mathbf{Y}]$. Then tuples u and t violate the OD $\mathbf{X} \rightarrow_{<} \mathbf{Y}$.
(2) "$\Leftarrow$":
Assume $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ is not valid, i.e., $\exists t, u$, s.t. $t[\mathbf{X}] < u[\mathbf{X}]$ and $t[\mathbf{Y}] \geq u[\mathbf{Y}]$.
Case 1: $t[\mathbf{Y}] > u[\mathbf{Y}]$, i.e., $t[\mathbf{X}] < u[\mathbf{X}]$ and $t[\mathbf{Y}] > u[\mathbf{Y}]$.
Case 2: $t[\mathbf{Y}] = u[\mathbf{Y}]$, i.e., $t[\mathbf{X}] < u[\mathbf{X}]$ and $t[\mathbf{Y}] = u[\mathbf{Y}]$.
In both cases, tuples u and t violate $\mathbf{Y} \rightarrow_{\leq} \mathbf{X}$. □

Theorem 1 implies that to find *all* n-ary order dependencies under the operator "$\leq$", it suffices to find *all* n-ary order dependencies under the operator "$<$" (and vice versa). We leverage this insight in the design of the order dependency discovery algorithm ORDER, which is described in Sect. 6.

2.5 ODs and other constraints

As functional dependencies are defined over sets of attributes, we introduce the notation $\mathcal{X}$ to denote a *set* of attributes as opposed to **X** for an (ordered) *list* of attributes. Szlichta et al. show that order dependencies under "$\leq$" subsume functional dependencies in [22], i.e., every order dependency $\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$ is also a functional dependency for the sets $\mathcal{X}, \mathcal{Y}$:

Lemma 4 *$\mathbf{X} \rightarrow_{\leq} \mathbf{Y}$ is valid $\Rightarrow \mathcal{X} \rightarrow \mathcal{Y}$ is valid [22].*

And thus, with Theorem 1 we have

Lemma 5

$$\begin{aligned} \mathbf{X} \rightarrow_{<} \mathbf{Y} \textit{ is valid} &\Rightarrow \mathbf{Y} \rightarrow_{\leq} \mathbf{X} \textit{ is valid} \\ &\Rightarrow \mathcal{Y} \rightarrow \mathcal{X} \textit{ is valid} \end{aligned}$$

There is also a relationship between ODs and unique column combinations (*uniques*), which are sets of attributes in $\mathcal{R}$, in which no two rows have identical values (i.e., key candidates). A valid OD under "$<$" with a unique left-hand side has several interesting implications, as the following lemma shows:

Lemma 6 *If $\mathbf{V} \rightarrow_{<} \mathbf{W}$ is valid and $\mathbf{V}$ is unique, the following statements are true for any attribute lists $\mathbf{X}$ and $\mathbf{Y}$ over a relation $\mathcal{R}$.*

$$\mathbf{W} \textit{ is unique.} \quad (1)$$
$$\mathbf{W} \rightarrow_{<} \mathbf{V} \textit{ is valid.} \quad (2)$$
$$\mathbf{VX} \rightarrow_{<} \mathbf{WY} \textit{ is valid.} \quad (3)$$
$$\mathbf{WY} \rightarrow_{<} \mathbf{VX} \textit{ is valid.} \quad (4)$$
$$\mathbf{VX} \rightarrow_{\leq} \mathbf{WY} \textit{ is valid.} \quad (5)$$
$$\mathbf{WY} \rightarrow_{\leq} \mathbf{VX} \textit{ is valid.} \quad (6)$$
$$\mathcal{V} \rightarrow \mathcal{W} \textit{ is valid.} \quad (7)$$
$$\mathcal{W} \rightarrow \mathcal{V} \textit{ is valid.} \quad (8)$$

Proof Because **V** is unique, and $\mathbf{V} \rightarrow_{<} \mathbf{W}$ is valid, **W** needs to be unique as well (Statement 1): If **W** contained duplicate rows, there would be a *merge* among (**V**, **W**) and thus $\mathbf{V} \nrightarrow_{<} \mathbf{W}$.

With both **V** and **W** a unique, there can be no *merge* among (**W**, **V**). There can also be no *swap* among (**W**, **V**), because then there would be a *swap* among (**V**, **W**) and $\mathbf{V} \nrightarrow_{<} \mathbf{W}$. Hence, $\mathbf{W} \rightarrow_{<} \mathbf{V}$ is valid (Statement 2).

$\mathbf{VX} \rightarrow_{<} \mathbf{WY}$ and $\mathbf{WY} \rightarrow_{<} \mathbf{VX}$ are valid as well (Statements 3 and 4): In general, because of lexicographical ordering of tuples, the tuples projected on an attribute list **S** are always in the same order as the tuples projected on **ST** for any **T** if **S** is unique. Thus, tuples projected on **VX** and **WY** have the same order than tuples projected on **V** and **W**, respectively.

With Theorem 1, Statements 5 and 6 follow directly from Statements 3 and 4. Statements 7 and 8 then follow from $\mathbf{W} \rightarrow_{<} \mathbf{V}$ and $\mathbf{V} \rightarrow_{<} \mathbf{W}$ and Lemma 5. □

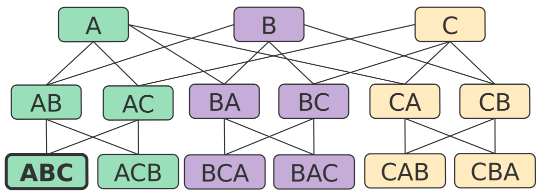

Fig. 1 A candidate lattice created from three attributes. The *highlighted node* ABC generates the ODs $A \rightarrow_{<} BC$ and $AB \rightarrow_{<} C$

In Sect. 5, we show how Statements 2–6 can be used to prune the search space.

2.6 Complexity

To find all order dependencies in a given table, we can represent the search space as a lattice of attribute permutations. In a relation $\mathcal{R}$ with $|\mathcal{R}| = n$, this lattice contains $P(n,k) := \binom{n}{k} \cdot k!$ nodes in level k as shown in Fig. 1. Each such node in level k contains k attributes and represents $k-1$ ODs: For example, the node $ABCD$ in level 4 represents the three OD candidates $A \rightarrow_{<} BCD$, $AB \rightarrow_{<} CD$, and $ABC \rightarrow_{<} D$, which are classified as either valid or invalid by the algorithm ORDER presented in Sect. 6. The permutation lattice is generated level by level from the bottom up as detailed in Sect. 5.

The sum over all levels of the lattice is $\sum_{k=0}^{n} P(n,k)$, which is the series A000522 in [18]. Halbeisen et al. prove $\sum_{k=0}^{n} P(n,k) = \lfloor n! \cdot e \rfloor$ in [8]. Thus, the permutation lattice containing *all* order dependencies, i.e., the search space, is factorial of size $\lfloor n! \cdot e \rfloor$.

The worst-case complexity of many existing discovery algorithms that traverse a candidate lattice depends on the number of candidates in that lattice. For example, the functional dependency discovery algorithm TANE traverses a set containment lattice with $2^{|\mathcal{R}|}$ nodes, and has an exponential worst-case complexity in the number of attributes [10]. Moreover, in the worst case, the *solution space* of minimal functional dependencies is also exponential, so a polynomial time algorithm cannot exist. Similar observations have been made for the discovery of inclusion dependencies [12] and unique column combinations [9].

3 Minimal order dependencies

Minimality is an important property of dependencies, such as functional dependencies, inclusion dependencies, and of other constraints, such as unique column combinations. Apart from the obvious benefit of conciseness of representation, the concept of minimality is useful for pruning the search space during dependency discovery. For instance, the

Table 7 Minimality of ODs. For instance, $CA \rightarrow_{\leq} B$ is not minimal, because already $C \rightarrow_{\leq} B$ is valid

A	B	C	D
2	1	1	1
1	2	2	1

algorithm TANE introduces a candidate set C^+ to effectively exclude non-minimal FDs from the search for FDs [10].

In the case of order dependencies, there is another non-obvious advantage of *minimality*. Szlichta et al. present the algorithm *ReduceOrder** [19] to rewrite SQL *order specifications*, i.e., a list of attributes in an ORDER BY. ODs with fewer attributes have a higher likelihood of matching an order specification and are thus more useful.

The minimality definition for order dependencies is more complex than that for functional dependencies. An FD $\mathcal{X} \rightarrow A$ is minimal if there is no $\mathcal{Y} \subset \mathcal{X}$, s.t. $\mathcal{Y} \rightarrow A$ is valid. An equivalent definition for ODs is not meaningful, because simply removing attributes from the left- or right-hand side of an OD does not respect the sensitivity of ODs for the order of the attributes. For instance, consider Table 7. Obviously, the OD $B \rightarrow_{\leq} C$ is valid. However, unlike for FDs, examining $AB \rightarrow_{\leq} C$ is not futile, because $AB \nrightarrow_{\leq} C$. In contrast, $BA \rightarrow_{\leq} C$ is valid (but not minimal).

We consolidate the ideas from *ReduceOrder** (in Definition 4) and two prefix-based inference rules to a novel definition of minimality for order dependencies under "$<$" in Definition 5.

Definition 4 (*Order-minimality*) An attribute list $\mathbf{X}$ is *minimal*, iff for any disjoint, contiguous sub-lists $\mathbf{V}$ and $\mathbf{W}$ in $\mathbf{X}$ where

1. $\mathbf{W}$ directly precedes $\mathbf{V}$ in $\mathbf{X}$, *or*
2. $\mathbf{W}$ follows (not necessarily directly) after $\mathbf{V}$ in $\mathbf{X}$,

$\mathbf{V} \rightarrow_{\leq} \mathbf{W}$ is not valid.

Definition 4 describes that the ordering of an attribute list $\mathbf{X}$ may be subsumed within that of an attribute list $\mathbf{Y}$ with $|\mathbf{Y}| < |\mathbf{X}|$. That is the case if there is an "embedded OD" in $\mathbf{X}$. The same argument is made for the algorithm *ReduceOrder** and proven to be correct as inference rules *Left Eliminate* (Case 1 of Definition 4) and *Eliminate* (Case 2 of Definition 4) in [20].

Definition 5 *(Minimality of order dependencies)* The order dependency $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ is minimal, iff

1. $\nexists\, \mathbf{V} \in \text{PREFIXES}(\mathbf{X})$, s.t. $\mathbf{V} \nrightarrow_{<} \mathbf{Y}$, *and*
2. $\nexists\, \mathbf{W} \in \text{PREFIXES}(\mathbf{Y})$, s.t. $\mathbf{X} \rightarrow_{<} \mathbf{W}$ is valid, *and*
3. $\mathbf{X}$ is minimal, *and*
4. $\mathbf{Y}$ is minimal.

where PREFIXES($\mathbf{X}$) returns the set of all prefixes of $\mathbf{X}$, e.g., $\text{PREFIXES}(ABCD) = \{A, AB, ABC\}$.

Because of Theorem 1, minimality for ODs under "$\leq$" can be defined analogously by switching the left- for the right-hand side in Rules 1 and 2.

4 Efficient validity check

The definition of ODs suggests a naïve algorithm for checking the validity of an order dependency $A \rightarrow_{\theta} B$: For every pair of tuples s and t, test whether $s[A]\,\theta\, t[A] \Rightarrow s[B]\,\theta\, t[B]$. However, this procedure demands a quadratic number of tuple comparisons. Alternatively, one could sort the relation by the left-hand side attributes and verify sortation for the right-hand side, still incurring $n \log n + n$ comparisons for n tuples. We provide an efficient data structure, sorted partitions, in Sect. 4.1, which is then used in Sect. 4.2 to check the validity of an order dependency in linear time. New sorted partitions can be created from those of smaller attribute lists to avoid accessing the underlying data again and again. This procedure is described as the *product* of sorted partitions in Sect. 4.3.

4.1 Sorted partitions

A *sorted partition* $\tau_{\mathbf{X}}$ is a list of sets, where the sets (or, equivalence classes) carry tuples with equal values in the attribute list $\mathbf{X}$, and the sets are sorted into a list according to the ordering imposed on the tuples in $\mathbf{X}$. In other words, a sorted partition $\tau_{\mathbf{X}}$ of an attribute list $\mathbf{X}$ partitions the tuples of a table into equivalence classes with equal values in $\mathbf{X}$, which are comparable to one another.

$\tau_{\mathbf{X}}^{k}$ represents the kth smallest tuples of $\mathbf{X}$. $|\tau_{\mathbf{X}}|$ denotes the number of equivalence classes in $\tau_{\mathbf{X}}$. For this purpose, it is not necessary to store entire tuples of data values. Instead, it suffices to store only the tuple identifiers (row indices). Using tuple identifiers results in faster comparisons, because they are represented as integers and make comparisons independent of the size of the tuples. The idea to represent tuples by their identifiers is used in other dependency detection algorithms as well. For instance, TANE introduces *partitions* [10], which resemble the *sorted partitions* defined here, except that they are defined as a set of sets, rather than a list of sets, because for detection of functional dependencies, one need not consider the order of the tuples. DUCC, an algorithm for discovering all unique column combinations in a dataset, uses position list indices (PLIs) to employ row-based pruning [9]. PLIs are also used in TANE, where they are called *stripped partitions* [10].

Table 8 shows a table containing the weight of shipped goods along with their cost of shipping. The correspond-

Table 8 Sorted partitions and data from which they are created

Tid	Weight	Shipping cost
(a) Table with shipped goods		
0	5	14
1	10	22
2	3	10
3	10	25
4	5	14
5	20	40

(b) Corresponding sorted partitions. Tuples are denoted by their identifiers

$\tau_{weight} = (\{2\}, \{0, 4\}, \{1, 3\}, \{5\})$

$\tau_{cost} = (\{2\}, \{0, 4\}, \{1\}, \{3\}, \{5\})$

ing sorted partitions τ_{weight} and τ_{cost} are given as well. The sorted partition τ_{weight} can be constructed by sorting the table by *weight* (in ascending order), building the equivalence classes from equal values in the sorted stream of values in *weight*, and incrementally adding these equivalence classes to the list τ_{weight}. In this example, 3 is the smallest value in *weight*, and it occurs only once. We add the set $\{2\}$ to τ_{weight}, because the value 3 is found at position 2 in the table. There are two occurrences of the second-smallest value 5 in *weight*, at positions 0 and 4. Thus, we add the set $\{0, 4\}$ to τ_{weight}, etc.

The *stripped partitions* of TANE remove any equivalence classes of size 1. Unfortunately, this is not possible here: Consider $\tau_A = (\{0, 1\}, \{2\}, \{3\})$ and $\tau_B = (\{0, 1\}, \{3\}, \{2\})$. Stripping these sorted partitions of their equivalence classes of size 1 results in a valid OD $A \rightarrow_< B$. However, $A \nrightarrow_< B$, because tuples 2 and 3 form a *swap* among (A, B).

So far, we have considered sorted partitions only on single attributes. However, sorted partitions for attribute lists $|\mathbf{X}| > 1$ are needed to check n-ary order dependencies. As defined in Sect. 2, we consider ODs on lexicographically ordered tuples. Because tuples are represented by their identifiers regardless of their size, sorted partitions for attribute lists have the same compact representation as those for single attributes. For example, the sorted partition for the attribute list (*weight*, *cost*) in Table 8 is $\tau_{weight,\ cost} = (\{2\}, \{0, 4\}, \{1\}, \{3\}, \{5\})$. We obtain this sorted partition in the same way as the sorted partition for the single attributes except that we consider the order of *tuples*: The tuple (3, 10) is the smallest tuple in the attribute list (*weight*, *cost*). It occurs only once at position 2 in the table. Hence, we add the set $\{2\}$ to $\tau_{weight,\ cost}$. The tuple (5, 14) occurs twice in the table, at position 0 and 4. Thus, we add the set $\{0, 4\}$ to $\tau_{weight,\ cost}$. The ordering of the next smallest tuples (10, 22) and (10, 25) is created using lexicographical ordering. Since the first components of the tuples are equal (both have the value 10), the ordering of the respective second components establishes the ordering of the entire tuples, i.e., because $(10, 22) < (10, 25)$, the set $\{1\}$ is added to $\tau_{weight,\ cost}$ immediately before $\{3\}$.

4.2 Efficient OD validation

The sorted partitions $\tau_{\mathbf{X}}$ and $\tau_{\mathbf{Y}}$ can be used for checking the validity of $\mathbf{X} \rightarrow_\theta \mathbf{Y}$ efficiently. As discussed earlier, order dependencies under "<" may be invalidated either by a *merge* or by a *swap*. In the remainder of this section, we first show how sorted partitions and the concept of a *merge* are related. We then show in Sect. 6.2 that *swaps* let us prune more candidates from the lattice of attribute lists than *merges* alone. Thus, we later on present a linear-time algorithm that determines from two sorted partitions $\tau_{\mathbf{X}}$ and $\tau_{\mathbf{Y}}$ if there is at least one *swap* among $(\mathbf{X}, \mathbf{Y})$.

Lemma 7 provides a useful formalization of the connection between sorted partitions and the concept of a *merge*.

Lemma 7 *There is a merge among (X, Y)* $\iff$ *the relation* $e \subseteq \tau_{\mathbf{X}} \times \tau_{\mathbf{Y}}$ *is not injective, with* $(x, y) \in e$ *if* $\exists t_y \in y$, *s.t.* $t_y \in x$, *where* t_y *is a tuple identifier in* y *and* $x \in \tau_{\mathbf{X}}$, $y \in \tau_{\mathbf{Y}}$.

Proof If there is a *merge* among $(\mathbf{X}, \mathbf{Y})$, there are two distinct tuple identifiers in different equivalence classes in $\tau_{\mathbf{X}}$, but in the same equivalence class in $\tau_{\mathbf{Y}}$. Mapping these equivalence classes yields a relation that is not injective. If the relation e is not injective, it maps two distinct tuple identifiers from different equivalence classes in $\tau_{\mathbf{X}}$ to one equivalence class in $\tau_{\mathbf{Y}}$. These tuple identifiers represent different values in $\mathbf{X}$, but equal values in $\mathbf{Y}$, and hence a *merge* among $\mathbf{X}$ and $\mathbf{Y}$. □

By construction, sorted partitions contain equivalence classes of tuples that are sorted ascendingly. We can conclude that the OD $\mathbf{X} \rightarrow_< \mathbf{Y}$ is invalidated by a *swap* if $\tau_{\mathbf{X}}$ and $\tau_{\mathbf{Y}}$ do not contain the tuple identifiers in the same order. For example, let $\tau^1_{\mathbf{X}} = \{5, 1, 4\}$ be the smallest tuples in $\mathbf{X}$. If the tuples identified by 5, 1, and 4 are not among the smallest tuples in $\mathbf{Y}$, there is a *swap* among $(\mathbf{X}, \mathbf{Y})$, and $\mathbf{X} \nrightarrow_< \mathbf{Y}$. We use this insight in Algorithm 1 (CHECKFORSWAP), which checks whether there is at least one *swap*, and if not, a *merge* among $(\mathbf{X}, \mathbf{Y})$, or if $\mathbf{X} \rightarrow_< \mathbf{Y}$ is valid.

CHECKFORSWAP detects if there is at least one *swap* between two sorted partitions $\tau_{\mathbf{X}}$ and $\tau_{\mathbf{Y}}$ by trying to match tuple identifiers from equivalence classes of $\tau_{\mathbf{X}}$ and $\tau_{\mathbf{Y}}$. This is achieved with a linear search (line 3) over the equivalence classes in $\tau_{\mathbf{X}}$ and $\tau_{\mathbf{Y}}$. CHECKFORSWAP distinguishes two cases—if the number of remaining, i.e., not-yet-mapped tuple identifiers in $e_{\mathbf{X}}$ compared to those remaining in $e_{\mathbf{Y}}$ is (i) smaller (line 6) or (ii) greater or equal (line 15). In case (i), CHECKFORSWAP checks whether $e_{\mathbf{X}}$ is a subset of $e_{\mathbf{Y}}$. If this is not the case (line 8), there is at least one tuple in

Algorithm 1 CHECKFORSWAP$_<$

```
Input: $\tau_{\mathbf{X}}, \tau_{\mathbf{Y}}$
Output: "swap" if there is at least one swap among $(\mathbf{X}, \mathbf{Y})$,
        "merge" if there is no swap, but a merge,
        "valid" if $\mathbf{X} \rightarrow_< \mathbf{Y}$ is valid
1: $next_{e_{\mathbf{X}}} \leftarrow next_{e_{\mathbf{Y}}} \leftarrow true$; $i \leftarrow 1$; $j \leftarrow 1$
2: $merge \leftarrow$ false
3: while $i < \tau_{\mathbf{X}}, j < \tau_{\mathbf{Y}}$
4:     if $(next_{e_{\mathbf{X}}})$ $e_{\mathbf{X}} \leftarrow \tau_{\mathbf{X}}^i$
5:     if $(next_{e_{\mathbf{Y}}})$ $e_{\mathbf{Y}} \leftarrow \tau_{\mathbf{Y}}^j$
6:     if $|e_{\mathbf{X}}| < |e_{\mathbf{Y}}|$
7:         if not $e_{\mathbf{X}} \subseteq e_{\mathbf{Y}}$
8:             return "swap"
9:         else
10:            $merge \leftarrow true$
11:            $i \leftarrow i + 1$
12:            $next_{e_{\mathbf{X}}} \leftarrow true$
13:            $e_{\mathbf{Y}} \leftarrow e_{\mathbf{Y}} - e_{\mathbf{X}}$
14:            $next_{e_{\mathbf{Y}}} \leftarrow false$
15:    else
16:        if not $e_{\mathbf{Y}} \subseteq e_{\mathbf{X}}$
17:            return "swap"
18:        else
19:            $j \leftarrow j + 1$
20:            $next_{e_{\mathbf{Y}}} \leftarrow true$
21:            $e_{\mathbf{X}} \leftarrow e_{\mathbf{X}} - e_{\mathbf{Y}}$
22:            if $|e_{\mathbf{X}}| = 0$
23:                $i \leftarrow i + 1$
24:                $next_{e_{\mathbf{X}}} \leftarrow true$
25:            else
26:                $next_{e_{\mathbf{X}}} \leftarrow false$
27: if $(merge)$ return "merge"
28: else return "valid"
```

$e_{\mathbf{X}}$ that appears not in $e_{\mathbf{Y}}$, but in a later equivalence class in $\tau_{\mathbf{Y}}$. Hence, there is a *swap* among $(\mathbf{X}, \mathbf{Y})$, and the algorithm terminates. If, however, $e_{\mathbf{X}} \subset e_{\mathbf{Y}}$ (line 9), we increment the counter i to retrieve the next equivalence class from $\tau_{\mathbf{X}}$ in the next iteration. Also, we update $e_{\mathbf{Y}}$ to remove all tuples that are also present in $e_{\mathbf{X}}$ (line 13), s.t. they are not mapped again in the next iteration.

Note that in case (i) and $e_{\mathbf{X}} \subseteq e_{\mathbf{Y}}$, we have also found a *merge*: Since the number of tuples in $e_{\mathbf{X}}$ is smaller than those in $e_{\mathbf{Y}}$, there must be at least one tuple in $e_{\mathbf{Y}}$ that is mapped to another equivalence class in $\tau_{\mathbf{X}}$. By Lemma 7, there is hence a *merge* among $(\mathbf{X}, \mathbf{Y})$. CHECKFORSWAP saves this information in the variable *merge* (line 10).

In case (ii), CHECKFORSWAP performs the check for a *swap* in line 16. The counter j is updated accordingly if no *swap* was found. An additional check whether $e_{\mathbf{X}}$ is empty (line 22) then determines whether the next equivalence class from $\tau_{\mathbf{X}}$ should be retrieved in the next iteration (this happens if $e_{\mathbf{Y}} = e_{\mathbf{X}}$). Finally, if no *swap* was found, the algorithm either returns that there is a *merge* among $(\mathbf{X}, \mathbf{Y})$ (as determined in line 10). If there is also no *merge*, the OD $\mathbf{X} \rightarrow_< \mathbf{Y}$ is valid (line 28).

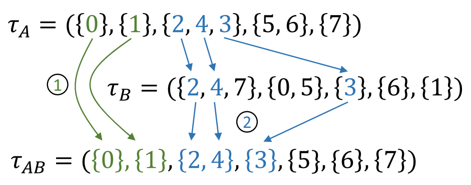
$\tau_A = (\{0\}, \{1\}, \{2, 4, 3\}, \{5, 6\}, \{7\})$
①
$\tau_B = (\{2, 4, 7\}, \{0, 5\}, \{3\}, \{6\}, \{1\})$
②
$\tau_{AB} = (\{0\}, \{1\}, \{2, 4\}, \{3\}, \{5\}, \{6\}, \{7\})$

Fig. 2 Sorted partition product of τ_A and τ_B yields τ_{AB}. The product can be implemented efficiently using a *hash-join-like* procedure as shown in Fig. 3

4.3 Product of sorted partitions

A stream of tuples, which is known to be sorted by A may be "nearly" sorted by AB as well, where the number of tuples that are not (guaranteed to be) sorted, depends on the number of non-distinct values in A. During a run of the algorithm ORDER presented in Sect. 6, this situation is encountered frequently, because the nodes in the permutation lattice of level l are all prefixes of some node in level $l + 1$.

As the order of the tuples in an attribute list is represented by a sorted partition, we present an algorithm for efficiently calculating what we call the *product* of sorted partitions, which—assuming constant-time lookup in hashing-based data structures—runs in linear time. Using this algorithm, the OD discovery algorithm ORDER touches the actual data only once for the initial creation of the sorted partitions. In the further run of the algorithm, sorted partitions are created from sorted partitions of attribute lists of smaller size, thereby saving many comparisons that would originate from sorting the data again.

Before presenting the actual algorithm to calculate the product of two sorted partitions, we briefly explain the idea in Fig. 2. Let τ_A and τ_B be two sorted partitions for single attributes A and B, respectively. Note that the following arguments are the same for sorted partitions of attribute lists, because tuples are represented by their identifiers regardless of their size. We distinguish two cases: (i) equivalence classes (represented as sets) of size 1 —these are appended to τ_{AB} as is highlighted in green, and (ii) equivalence classes of size greater than 1—these are split into new equivalence classes according to the order of the tuple identifiers in τ_B and then appended to τ_{AB} (highlighted in blue).

Because of the lexicographical ordering imposed on the tuples, the values in A dominate the sorting of tuples of AB: Sequences of different values in a tuple stream sorted by A appear in the same order in the same position in the tuple stream sorted by AB. For sorted partitions, this means that equivalence classes of size 1 in τ_A are also present in τ_{AB} in the same order as in τ_A. In Fig. 2, this is the case for equivalence classes $\{0\}$, $\{1\}$, and $\{7\}$. In the extreme case, if A is a unique column, $\tau_A = \tau_{A\mathbf{X}}$ for any attribute list $\mathbf{X}$ over $\mathcal{R}$.

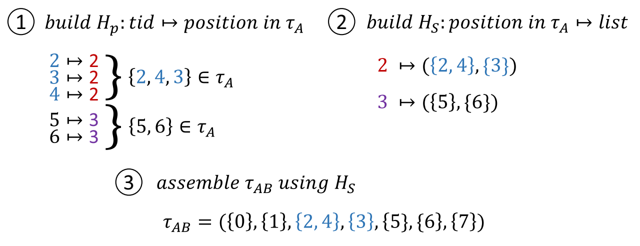
① build H_p: tid ↦ position in τ_A
② build H_S: position in τ_A ↦ list
2 ↦ 2
3 ↦ 2
4 ↦ 2
{2, 4, 3} ∈ τ_A
5 ↦ 3
6 ↦ 3
{5, 6} ∈ τ_A
2 ↦ ({2, 4}, {3})
3 ↦ ({5}, {6})
③ assemble τ_{AB} using H_S
τ_{AB} = ({0}, {1}, {2, 4}, {3}, {5}, {6}, {7})

Fig. 3 Workflow of the sorted partition product algorithm on τ_A and τ_B from Fig. 2

Where A has repeating values, B dominates the order of AB. Therefore, to build τ_{AB} we need to find the relative positions in τ_B of those tuple identifiers in τ_A, which are contained in an equivalence class of size greater than 1. Figure 2 depicts this scenario for the equivalence classes $\{5, 6\}$ and $\{2, 4, 3\}$ in τ_A. Since 5 is ranked before 6 in τ_B, the equivalence class $\{5, 6\}$ is split into two equivalence classes in τ_{AB}. The generation of, e.g., τ_{ABC} is simplified further by each of these splits of equivalence classes. The equivalence class $\{2, 4, 3\}$ shows another case (Case 2 in Fig. 2): Because 2 and 4 are in the same equivalence class in τ_B as well, this equivalence class of size 3 generates two equivalence classes in τ_{AB}, namely $\{2, 4\}$ and $\{3\}$. Since the tuple identified by 3 projected on B is strictly greater than the tuples identified by 2 and 4 projected on B, τ_{AB} ranks $\{3\}$ *after* $\{2, 4\}$.

The product of two sorted partitions can be efficiently implemented using a hash-join-like procedure as shown in Fig. 3. This algorithm for calculating the product of two sorted partitions consists of three steps. First, we build a hash table H_p mapping tuple identifiers t to the position in τ_A at which t resides. As explained above, equivalence classes of size 1 in τ_A are contained in τ_{AB} in the same order. Hence, H_p need only contain tuple identifiers that are part of an equivalence class of size greater than 1.

Second, we iterate over τ_B, building another hash table H_S mapping each position in τ_A that contains an equivalence class of size greater than 1 to a *list* of equivalence classes. This list is created for each $e_A \in \tau_A$ by (i) splitting e_A into equivalence classes e_{AB}, each containing tuple identifiers that are in the same equivalence class in τ_B and (ii) adding e_{AB} to the list corresponding to the order of the equivalence classes in τ_B.

Lines 2 to 11 in Algorithm 2 detail this approach: The algorithm iterates over each tuple identifier tid_{e_B} of each equivalence class $e_B \in \tau_B$. If tid_{e_B} is not in an equivalence class of size 1, i.e., $H_p(tid_{e_B})$ is mapped to an actual position in τ_A (line 5), tid_{e_B} is added to the current equivalence class in $H_S(pos)$ (line 9), where pos (the position, or list index in τ_A corresponding to tid_{e_B}) is determined by $H_p(tid_{e_B})$ (line 7).

Each iteration over e_B creates new *complete* equivalence classes, which are later added to τ_{AB}, i.e., a new complete

Algorithm 2 PARTITIONPRODUCT

```
Input: $\tau_A$, $\tau_B$, hash table $H_p$
Output: $\tau_{AB}$
1: $\tau_{AB} \leftarrow [\,]$
2: for each $e_B \in \tau_B$, $|\tau_B| > 1$
3:     $visited \leftarrow \emptyset$
4:     for each $tid_{e_B} \in e_B$
5:         if $H_p(tid_{e_B}) = \bot$
6:             continue
7:         $pos \leftarrow H_p(tid_{e_B})$
8:         add $pos$ to $visited$
9:         add $tid_{e_B}$ to last equivalence class in $H_S(pos)$
10:    for each $pos \in visited$
11:        append $\emptyset$ to $H_S(pos)$
12: for each index $i$ in $\tau_A$
13:    if $|\tau_A^i| = 1$
14:        append $e_A$ to $\tau_{AB}$
15:    else
16:        for each $e_{AB} \in H_S(i)$, $e_{AB} \neq \emptyset$
17:            append $e_{AB}$ to $\tau_{AB}$
```

equivalence class e_{AB} is created for each position in τ_A the tuple identifiers tid_{e_B} are mapped to by means of H_p. To guarantee that no tuple identifiers from different $e_B, e_{B'}$ are added to the same equivalence class in τ_{AB}, the tuple identifiers tid_{e_B} are only ever added to the last equivalence class in $H_S(pos)$ (line 9). In addition, for each pos that at least one of the tid_{e_B} has been mapped to, a new empty equivalence class is created and added to $H_S(pos)$ in line 11 after each iteration over the current e_B. This new empty equivalence class is then populated in one of the next iterations or stays empty and is ignored during the assembly step in line 16. In each iteration over e_B, the algorithm keeps track of the positions in τ_A that have been "visited" by tuple identifiers in e_B in the set $visited$. From a more abstract point of view, the above described part of the algorithm creates a connection (H_S) between equal values in A and the order of the corresponding values in B, so that the ordering of those tuples in AB with equal values in A can be established.

In a final assembly step (lines 12 to 17), we iterate over τ_A, putting each equivalence class of size 1 into τ_{AB} without further processing. For each position p of an equivalence class in τ_A of size greater than 1, we append $H_S(p)$ to τ_{AB}. Iterating over the last equivalence class e_B of size greater than 1, we added a redundant empty set to each $H_S(pos)$ with $pos \in visited$ (line 11). This empty set is not added to τ_{AB}.

5 Lattice traversal and pruning

In Sect. 2.6, we showed that the number of nodes in the candidate lattice that defines the search space for order dependency detection is $\lfloor |\mathcal{R}|! \times e \rfloor$. To tackle the resulting factorial complexity of the order dependency detection problem, we

Table 9 $A \not\rightarrow_< CD$. Hence, by pruning rule 1, $AB \not\rightarrow_< CD$

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

Table 10 $A \rightarrow_< BC$ is valid. Thus, by pruning rule 2, $A \rightarrow_< BCD$ is valid

A	B	C	D
1	5	8	2
1	4	7	1
3	6	1	5
4	6	2	5

employ a twofold strategy to systematically explore only parts of the candidate lattice.

First, we generate the nodes in the candidate lattice level by level from the bottom up making use of the general "Apriori" idea [3]. Apriori-like generation of candidates is also utilized by discovery algorithms for other types of constraints, such as *unique column combinations* [2], *inclusion dependencies* [4], and *functional dependencies* [10]. A concrete algorithm implementing this procedure for order dependency detection is presented in Sect. 6.3 as part of the algorithm ORDER. Second, we avoid enumerating all nodes by aggressively pruning the permutation lattice. To this end, we define pruning rules 1 to 4 that allow to skip many validity checks while guaranteeing that no (minimal or valid) order dependencies are missed.

Pruning rule 1 ***(Invalid under "<")***

$$\boldsymbol{X} \not\rightarrow_< \boldsymbol{Y} \Rightarrow \boldsymbol{XV} \not\rightarrow_< \boldsymbol{Y}$$

holds for any disjoint attribute lists $\boldsymbol{X}, \boldsymbol{Y}$*, and* $\boldsymbol{V}$ *over a schema* $\mathcal{R}$*, where only* $\boldsymbol{V}$ *may be empty.*

In Table 9, $A \not\rightarrow_< CD$. Therefore, by pruning rule 1, $AB \not\rightarrow_< CD$. The reason is a *merge* among (A, CD), which is still present when appending another column to the left-hand side A of the OD $A \not\rightarrow_< CD$.

Pruning rule 2 ***(Valid under "<")***

$$\boldsymbol{X} \rightarrow_< \boldsymbol{Y} \textit{ is valid} \Rightarrow \boldsymbol{X} \rightarrow_< \boldsymbol{YW} \textit{ is valid}$$

holds for any disjoint attribute lists $\boldsymbol{X}, \boldsymbol{Y}$*, and* $\boldsymbol{W}$ *over a schema* $\mathcal{R}$*, where only* $\boldsymbol{W}$ *may be empty.*

Table 10 shows a relation in which $A \rightarrow_< BC$ is valid. Therefore, by pruning rule 2, $A \rightarrow_< BCD$ is valid as well, because we cannot create a *merge* or *swap* among (A, BCD) if there is none among (A, BC).

Pruning rule 3 ***(Swap under "<")*** *Given* $\boldsymbol{X} \not\rightarrow_< \boldsymbol{Y}$ *is invalidated by a swap,*

$$\boldsymbol{X} \not\rightarrow_< \boldsymbol{Y} \Rightarrow \boldsymbol{XV} \not\rightarrow_< \boldsymbol{YW}$$

holds for any disjoint attribute lists $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{V}$*, and* $\boldsymbol{W}$ *over a schema* $\mathcal{R}$*, where only* $\boldsymbol{V}$ *and* $\boldsymbol{W}$ *may be empty.*

Table 11 $A \not\rightarrow_< C$ is invalidated by a *swap*

A	B	C	D
1	3	6	0
2	4	5	1
3	5	7	2
4	6	8	3

Hence, by pruning rule 3, $AB \not\rightarrow_< CD$ is also invalidated by a *swap*

Table 12 $A \rightarrow_< C$, A is unique

A	B	C	D
1	5	1	2
2	4	3	1
3	6	7	5
4	6	9	5

Thus, by pruning rule 12, $AB \rightarrow_< CD$ is valid

Table 11 shows a table to which we can apply pruning rule 3. $A \not\rightarrow_< C$ is invalidated by a *swap* among (A, C). By pruning rule 3, $AB \not\rightarrow_< CD$, because the order of the tuples in A and C does not change when we append any attribute (such as B and D, respectively). Thus, there is a *swap* among (AB, CD) as well.

Pruning rule 4 ***(Uniqueness under "<")*** *Given* $\boldsymbol{X}$ *is unique,*

$$\boldsymbol{X} \rightarrow_< \boldsymbol{Y} \textit{ is valid} \Rightarrow \boldsymbol{XV} \rightarrow_< \boldsymbol{YW} \textit{ is valid}$$

holds for any disjoint attribute lists $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{V}$*, and* $\boldsymbol{W}$ *over a schema* $\mathcal{R}$*, where only* $\boldsymbol{V}$ *and* $\boldsymbol{W}$ *may be empty.*

In Table 12, $A \rightarrow_< C$ is valid. Because A is unique, C is unique as well (refer to Lemma 6 for details) and by pruning rule 4, $AB \rightarrow_< CD$ is valid as well.

Note that the generation of candidates from the bottom up matches well to the presented pruning rules: All nodes of size l in the candidate lattice are prefixes of some node in level $l + 1$, and pruning of an OD candidate is done based on the prefixes of a candidate's left- and right-hand side attribute lists.

Moreover, the creation of *sorted partitions* needed in levels $l > 2$ can be created efficiently from the sorted partitions needed in a previous level, as we show in Sect. 4.3. A level-wise top-down traversal approach is not feasible, because it would require enumerating $|\mathcal{R}|!$ nodes already in the first level, namely for the top node in the lattice, which represents the entire attribute-set. Note that using the level-wise bottom-up traversal strategy, it may still be necessary to generate all lattice nodes, namely when none of the pruning rules apply.

6 The discovery algorithm

We present the algorithm ORDER (Algorithm 3), which efficiently discovers all n-ary lexicographical order dependencies under the operator "$<$" (and by Theorem 1, all ODs under "$\leq$") in a given table. ORDER traverses the lattice of all possible permutations of attributes described in Sect. 5 in a level-wise manner from the bottom up.

Algorithm 3 Algorithm ORDER, which finds all valid ODs $\mathbf{X} \rightarrow_< \mathbf{Y}$

```
Input: Relation $\mathcal{R}$, relational instance $r$
Output: valid, the set of all minimal valid order dependencies that are
    valid in $r$
1: $l \leftarrow 1$
2: $valid \leftarrow \emptyset$                    ▷ global variable: set of all valid ODs
3: $C_0([\,]) \leftarrow \emptyset$; $C_1([\,]) \leftarrow \{A \mid A \in \mathcal{R}\}$
4: $CS_0 \leftarrow \emptyset$; $CS_1 \leftarrow \emptyset$
5: $L_1 \leftarrow \{A \mid A \in \mathcal{R}\}$
6: while $L_l \neq \emptyset$
7:     $CS_l \leftarrow$ UPDATECANDIDATESETS($CS_{l-1}$)
8:     COMPUTEDEPENDENCIES($L_l, CS_l$)
9:     PRUNE($L_l, CS_l$)
10:    $L_{l+1} \leftarrow$ GENERATENEXTLEVEL($L_l$)
11:    $l \leftarrow l + 1$
12: output valid
```

The algorithm makes use of the Apriori approach [3] to generate candidates of size l from candidates of size $l-1$ in the function GENERATENEXTLEVEL (line 10, see also Algorithm 7 and the description in Sect. 6.3). In the call COMPUTEDEPENDENCIES (line 8, see also Algorithm 4), the OD candidates generated for the current level are checked for validity and added to *valid* if applicable. The knowledge gained during these checks is then used in the function PRUNE (line 9, see also Algorithm 6) to prune the search space before generating the next level. If there are no more candidates left (line 6), either because the lattice has been traversed fully or no candidates remain after pruning, ORDER terminates and all valid ODs in the table are returned (line 12).

The outline of the algorithm ORDER is structurally similar to the algorithm TANE [10], because TANE and ORDER both discover dependencies in a lattice of possible dependency candidates and traverse it bottom up. Both algorithms reduce the search space by applying pruning rules. Moreover, both use candidate sets to represent dependency candidates that need to be checked for validity. Still, ORDER and TANE differ in many details, e.g., the definition and application of pruning rules and the creation and management of candidate sets.

In the following, we explain each of the procedures that are called in the main loop of ORDER (Algorithm 3).

6.1 Dependency computation

In level l, ORDER checks $l-1$ order dependency candidates for validity for each node in the lattice. For example, in level 4, the node $CBDA$ generates the three OD candidates $C \rightarrow_< BDA$, $CB \rightarrow_< DA$, and $CBD \rightarrow_< A$. This trivial generation of OD candidates from a node in the candidate lattice is implemented by the function OBTAINCANDIDATES called in line 3 of the procedure COMPUTEDEPENDENCIES (Algorithm 4).

For every left-hand side $\mathbf{X}$ of an order dependency candidate in level l, ORDER maintains a *candidate set* $C_l(\mathbf{X})$. In level l, $C_l(\mathbf{X})$ contains all $\mathbf{Y}$, s.t. $\mathbf{X} \rightarrow_< \mathbf{Y}$ still needs to be checked for validity and $|\mathbf{X}| + |\mathbf{Y}| = l$. For instance in level 4, $C_4(AB)$ may contain CD, but not C or CDE. For non-minimal $\mathbf{X}$, $C_l(\mathbf{X}) = \emptyset$. For non-minimal $\mathbf{Y}$, $\mathbf{Y} \notin C_l(\mathbf{X})$ for *any* $\mathbf{X}$ over $\mathcal{R}$.

More formally, the candidate set $C_l(\mathbf{X})$ for a minimal $\mathbf{X}$ contains all those $\mathbf{Y}$, for which

1. $|\mathbf{X}| + |\mathbf{Y}| = l$,
2. $\mathbf{Y}$ is minimal,
3. there is no *swap* among $\mathbf{P}$ and $\mathbf{Q}$ for any prefixes $\mathbf{P}$ of $\mathbf{X}$ and $\mathbf{Q}$ of $\mathbf{Y}$,
4. $\mathbf{P} \not\rightarrow_< \mathbf{Q}$ for any prefixes $\mathbf{P}$ of $\mathbf{X}$ and $\mathbf{Q}$ of $\mathbf{Y}$ where $\mathbf{P}$ is unique (see Lem. 6 in Sec. 2.5),
5. $\mathbf{X} \not\rightarrow_< \mathbf{Q}$ for any prefix $\mathbf{Q}$ of $\mathbf{Y}$, and
6. $\mathbf{X}' \rightarrow_< \mathbf{Y}$ is invalidated by a *merge* (where $\mathbf{X}'$ is the longest prefix of $\mathbf{X}$), but $\mathbf{X}' \rightarrow_< \mathbf{Y}$ could not be merge-pruned (see Sec. 6.1.1 for details on merge-pruning).

Because ORDER traverses the candidate lattice of attribute lists from the bottom up, every node in level l is a prefix of some node in level $l+1$. That makes the gradual generation of the candidate sets easy, because the decision whether $\mathbf{Y} \in C_l(\mathbf{X})$ can be based on the prefixes of $\mathbf{X}$ and $\mathbf{Y}$. How candidate sets are created and extended is explained in detail in Sect. 6.1.2.

Every $\mathbf{Y} \in C_l(\mathbf{X})$ implies an order dependency candidate $\mathbf{X} \rightarrow_< \mathbf{Y}$, which may itself be valid or generate a minimal valid OD in a higher level. ORDER skips the validity check if $\mathbf{Y} \notin C_l(\mathbf{X})$ (line 4). By removing only those candidates $\mathbf{Y}$ from $C_l(\mathbf{X})$ for which the validity of all $\mathbf{XV} \rightarrow_< \mathbf{YW}$ is known from the pruning rules (see Sect. 5), ORDER guarantees that no ODs whose validity is unknown are missed. ORDER applies pruning by *swap* in line 11 by removing the right-hand side $\mathbf{Y}$ of the invalid OD $\mathbf{X} \not\rightarrow_< \mathbf{Y}$ from the candidate set $C(\mathbf{X})$. Thus, ODs of the form $\mathbf{XV} \not\rightarrow_< \mathbf{YW}$

Algorithm 4 ORDER: COMPUTEDEPENDENCIES

```
1: procedure COMPUTEDEPENDENCIES($L_l$, $CS_l$)
2:   for each $node \in L_l$
3:     for each $\mathbf{X}, \mathbf{Y} \in$ OBTAINCANDIDATES($node$)
4:       if $\mathbf{Y} \notin C_l(\mathbf{X})$
5:         continue
6:       if $\mathbf{X} \rightarrow_< \mathbf{Y}$ is valid
7:         add $\mathbf{X} \rightarrow_< \mathbf{Y}$ to $valid$
8:         if $\mathbf{X}$ is unique
9:           remove $\mathbf{Y}$ from $C_l(\mathbf{X})$
10:      else if $\mathbf{X} \not\rightarrow_< \mathbf{Y}$ invalidated by $swap$
11:        remove $\mathbf{Y}$ from $C_l(\mathbf{X})$
12:  ▷ merge-pruning:
13:  for each $C_l(\mathbf{X}) \in CS_l$ with $|\mathbf{X}| > 1$
14:    for each $\mathbf{Y} \in C_l(\mathbf{X})$
15:      $\mathbf{X}' \leftarrow$ MAXPREFIX($\mathbf{X}$)
16:      if $\mathbf{X}' \not\rightarrow_< \mathbf{Y}$ invalidated only by $merge$
17:        if $\nexists\, \mathbf{V} \in C_l(\mathbf{X}')$ MAXPREFIX($\mathbf{V}$) $= \mathbf{Y}$
18:          remove $\mathbf{Y}$ from $C_l(\mathbf{X})$
```

($\mathbf{V}, \mathbf{W} \neq [\ ]$) are never checked for validity, because they are known to be invalidated by a swap as well. In addition to the pruning applied in the first loop of COMPUTEDEPENDENCIES, we provide another slightly more complex pruning mechanism called *merge-pruning*, which is described in detail next.

6.1.1 Merge-pruning

In general, we cannot remove $\mathbf{Y}$ from $C_l(\mathbf{X})$ if $\mathbf{X} \not\rightarrow_< \mathbf{Y}$ is invalidated by a *merge*, but there is no *swap* among $(\mathbf{X}, \mathbf{Y})$. For example, let $A \not\rightarrow_< B$ be invalidated by a *merge*. We infer that $A\mathbf{X} \not\rightarrow_< B$ for any attribute list $\mathbf{X}$. However, there may exist some $\mathbf{Y}$, s.t. $A\mathbf{X} \rightarrow_< B\mathbf{Y}$ is valid. Table 13 illustrates this situation. But $A\mathbf{X} \rightarrow_< B\mathbf{Y}$ would not be generated if we had pruned B from $C_l(A)$. Hence, we cannot use *merges* to prune the search space in the same way we use *swaps*.

Instead, assume we know in addition to $A \not\rightarrow_< B$ being invalidated only by a *merge* that any $A \rightarrow_< B\mathbf{Y}$ ($\mathbf{Y} \neq [\ ]$) need not be checked. Then, we need not check any OD of the form $A\mathbf{X} \not\rightarrow_< B\mathbf{Y}$. This "merge-pruning" is performed in lines 13 to 18 of Algorithm 4 (COMPUTEDEPENDENCIES). Note that we need to combine the knowledge of validity of dependencies for two levels to apply merge-pruning. Merge-pruning is applied to candidate sets $C_l(\mathbf{X})$ with $|\mathbf{X}| > 1$ in every level. The function MAXPREFIX is used to obtain the longest prefix of an attribute list $\mathbf{X}$. Naturally, the longest prefix of an attribute list $\mathbf{X}$ is the attribute list with the first $|\mathbf{X}| - 1$ attributes in $\mathbf{X}$, e.g., MAXPREFIX($ABCD$) $= ABC$. If the OD MAXPREFIX($\mathbf{X}$) $\rightarrow_< \mathbf{Y}$ was invalidated only by a *merge* (this is knowledge gained in level $l-1$), we know that $\mathbf{X} \not\rightarrow_< \mathbf{Y}$ (and also, $\mathbf{XV} \not\rightarrow_< \mathbf{Y}$ for any $\mathbf{V}$). Then, if we cannot find in C_l(MAXPREFIX($\mathbf{X}$)) any $\mathbf{V}$ that starts with $\mathbf{Y}$, we need not examine any $\mathbf{X} \rightarrow_< \mathbf{YZ}$, because it is either invalid or not minimal. Combined, we now know that it is futile to extend the right- or the left-hand side of $\mathbf{X} \rightarrow_< \mathbf{Y}$. Therefore, we can remove $\mathbf{Y}$ from $C_l(\mathbf{X})$.

With merge-pruning, the algorithm can possibly terminate earlier. For example, let $C_2(A) = \{E\}$, with $A \not\rightarrow_< E$ invalidated only by a *merge*. $C_3(AB)$ is generated from $C_2(A)$ (line 17 in Algorithm 5), and therefore $C_3(AB) = \{E\}$. Analogously, we obtain $C_4(ABC) = \{E\}$, $C_5(ABCD) = \{E\}$, etc. Therefore, the candidates $C_l(A\mathbf{X})$ with $|A\mathbf{X}| = l - 1$ may contain E, even though $A\mathbf{X} \rightarrow_< E$ is known to be invalidated by a *merge*. This is necessary in general, because the OD $AB \rightarrow_< E$ invalidated by a *merge* may generate the possibly valid OD $AB \rightarrow_< EF$ in level 4, as we pointed out earlier by means of Table 13. Thus, we cannot remove E from $C_3(AB)$. If, however, it is known that $EK \notin C_3(A)$ (for any attribute $K \in \mathcal{R}$), because, e.g., $A \rightarrow_< EK$ are all invalidated by a *swap*, we would not need to examine $AB \rightarrow_< EK$ in level 4 and can therefore safely remove E from $C_3(AB)$ in level 3.

Because merge-pruning demands fully pruned candidate sets of one level, the earliest possible point in the algorithm to prune by *merge* is after all ODs of the current level have been checked, i.e., after termination of the first loop in COMPUTEDEPENDENCIES. Also, merge-pruning cannot happen before level 3, because level 3 is the first level to have more than one candidate set per node – for merge-pruning, we need *in one level* a candidate set $C_l(\mathbf{X})$ and the candidate set of its longest prefix C_l(MAXPREFIX($\mathbf{X}$)).

Table 13 $A \not\rightarrow_< B$ (*merge*), $AX \not\rightarrow_< B$ (*merge*), but $AX \rightarrow_< BY$ is valid

A	X	B	Y
1	1	1	4
2	1	1	5
3	2	2	6
3	3	3	6
4	4	4	6

6.1.2 Updating candidate sets

Unlike TANE [10], ORDER's candidate sets may increase in size during the execution of the algorithm. For example, with $\mathcal{R} = \{A, B, C, D\}$, consider the left-hand side A in level 2 and the corresponding candidate set $C_2(A) = \{B, C, D\}$. In level 3, $C_3(A) = \{BC, BD, CD, CB, DB, DC\}$, i.e., every $\mathbf{U} \in C_2(\mathbf{V})$ is *extended* with all $E \in \mathcal{R}$ for which $\mathbf{U}$, $\mathbf{V}$, and E are mutually disjoint. This guarantees that $|\mathbf{X}| + |\mathbf{Y}| = l$ for any OD candidate $\mathbf{X} \rightarrow_< \mathbf{Y}$ that is checked for validity in the current *level* l, and ultimately that all possible, completely non-trivial OD candidates are checked. A candidate set $C_l(\mathbf{X})$ is extended if $|\mathbf{X}| \neq l - 1$, i.e., in each level all candidate sets $C_l(\mathbf{X})$ with $1 \leq |\mathbf{X}| \leq l - 2$ are extended. This implies that before level 3, ORDER does not extend any of the candidate sets.

The procedure UPDATECANDIDATESETS shown in Algorithm 5 is responsible for the management of the candidate sets, i.e., to initialize and extend them. To this end, ORDER maintains a set of candidate sets CS_l in level l, which is populated using CS_{l-1}.

Algorithm 5 ORDER: UPDATECANDIDATESETS

```
1: procedure UPDATECANDIDATESETS($CS_{l-1}$)
2:   $CS_l \leftarrow \emptyset$
3:   for each $C_{l-1}(\mathbf{X}) \in CS_{l-1}$
4:     $C_l(\mathbf{X}) \leftarrow \emptyset$
5:     if $|\mathbf{X}| \neq l-1$                          ▷ extend $C_{l-1}(\mathbf{X})$
6:       for each $\mathbf{Y} \in C_{l-1}(\mathbf{X})$
7:         if $\mathbf{X} \rightarrow_< \mathbf{Y} \in valid$
8:           continue
9:         for each $\mathbf{U} \in$ EXTEND ($\mathbf{X}, \mathbf{Y}$)
10:          if $|\mathbf{X}| > 1$
11:            $\mathbf{X}' \leftarrow$ MAXPREFIX($\mathbf{X}$)
12:            if ($\mathbf{U} \notin C_{l-1}(\mathbf{X}')$) and
               ($\nexists$ $\mathbf{Q} \in$ PREFIXES($\mathbf{U}$) $s.t.$
                   $\mathbf{X}' \rightarrow_< \mathbf{Q} \in valid$)
13:              continue
14:          if $\mathbf{U}$ is not minimal
15:            continue
16:          add $\mathbf{U}$ to $C_l(\mathbf{X})$
17:    else                          ▷ $|\mathbf{X}| = l-1$: create new candidate set
18:      if $\mathbf{X}$ is minimal
19:        for $B \in C_{l-1}$(MAXPREFIX($\mathbf{X}$))
20:          if $\mathbf{X}$ and $B$ are disjoint
21:            add $B$ to $C_l(\mathbf{X})$
22:    if $C_l(\mathbf{X}) \neq \emptyset$
23:      add $C_l(\mathbf{X})$ to $CS_l$
24:  return $CS_l$
```

Extending attribute lists is implemented by the function EXTEND called in line 9. EXTEND takes the two parameters $\mathbf{X}$ and $\mathbf{Y}$, where $\mathbf{Y} \in C_{l-1}(\mathbf{X})$ is extended, and the extended attribute list may later be added to $C_l(\mathbf{X})$. $\mathbf{X}$ is needed so that only those extensions of $\mathbf{Y}$ are generated that are disjoint from $\mathbf{X}$.

Attribute lists $\mathbf{Y}$ in $C_{l-1}(\mathbf{X})$ for which $\mathbf{X} \rightarrow_< \mathbf{Y}$ is valid are not extended (lines 7 to 8), because the resulting OD candidates $\mathbf{X} \rightarrow_< \mathbf{Y}K$ (for any $K \in \mathcal{R}$) are not minimal, and neither are any $\mathbf{X} \rightarrow_< \mathbf{YW}$ (for any non-empty attribute list $\mathbf{W}$ over $\mathcal{R}$).

Lines 10 and 13 handle a case in which we need not add an extended right-hand side candidate $\mathbf{U}$ to $C_l(\mathbf{X})$. If $\mathbf{U} \notin C_{l-1}(\mathbf{X}')$ for the longest prefix $\mathbf{X}'$, then either (i) $\mathbf{U}$ is not minimal, (ii) $\mathbf{U}$ was pruned by the *swap* rule, (iii) $\mathbf{U}$ was pruned by the uniqueness rule, (iv) $\mathbf{U}$ was pruned by merge-pruning, or (v) $\mathbf{X} \rightarrow_< \mathbf{T}$ is valid for some prefix $\mathbf{T}$ of $\mathbf{U}$. In cases (i)–(iv), we need not add $\mathbf{U}$ to $C_l(\mathbf{X})$. However, in case (v) we need to add $\mathbf{U}$ to $C_l(\mathbf{X})$, because for any valid OD $\mathbf{X} \rightarrow_< \mathbf{Y}$, the OD $\mathbf{XV} \rightarrow_< \mathbf{YW}$ (for any non-empty attribute list $\mathbf{V}$ and any attribute list $\mathbf{W}$) may or may not be valid, i.e., all $\mathbf{XV} \rightarrow_< \mathbf{YW}$ still need to be checked.

If $\mathbf{U}$ was pruned from $C_{l-1}(\mathbf{X}')$ because case (v) applied, then there is a valid OD $\mathbf{X}' \rightarrow_< \mathbf{Q}$ ($\mathbf{Q} \in$ PREFIXES($\mathbf{U}$)). This condition is checked in line 12 of Algorithm 5. If we do not find any such valid OD, we do not need to add $\mathbf{U}$ to $C_l(\mathbf{X})$. Note that it is not necessary to actually check these ODs, because we keep track of all found valid ODs in the set *valid*. Furthermore, if the extended candidate is not minimal (line 14), we need not add it to $C_l(\mathbf{X})$, because a non-minimal attribute list cannot be a right-hand side to any minimal order dependency.

In level l, ORDER maintains all candidate sets for left-hand side attribute lists of size 1 to $l-1$. In each level, a new candidate set for some $\mathbf{Z}$ with $|\mathbf{Z}| = l-1$ is created from the previous level's longest prefix of $\mathbf{Z}$ (line 17). If, for example, $C_2(A) = \{B, C\}$, then we set $C_3(AB) = \{C\}$, excluding B of $C_2(A)$, because B is not disjoint from AB. Creating candidate sets $C_l(\mathbf{X})$ from C_{l-1}(MAXPREFIX($\mathbf{X}$)) is justified by the pruning rules presented in Sect. 5: For instance, let $D \notin C_3(AB)$ with $C_3(AB) \neq \emptyset$, i.e., AB is minimal. Then, we know that none of $AB\mathbf{X} \rightarrow_< D\mathbf{Y}$ need to be checked for validity, because D was pruned from $C_3(AB)$. Therefore, we also need not check $ABC \rightarrow_< D$, and consequentially, $D \notin C_4(ABC)$.

6.2 Pruning

It is only possible to prune a node from the permutation lattice if it is certain that this node is not needed for generating another node that contains an OD candidate whose validity is not already known. Each node n in level $|n|$ in the lattice contains $|n| - 1$ order dependency candidates and thus $|n| - 1$ candidate sets, one for each prefix of n. If a candidate set of some left-hand side $\mathbf{X}$ is empty, the search space for order dependencies with $\mathbf{X}$ as a left-hand side is exhausted, i.e., no more ODs with a prefix of $\mathbf{X}$ on their left-hand side need to be checked for validity. In the case that *all* $|n| - 1$ candidate sets are empty, the above is true for *all prefixes* of n. Thus, we can delete the node n from the lattice, which means that larger nodes with n as prefix are not generated. For example, for node $ABCD$ of level 4 in the permutation lattice, the candidate sets $C_4(A)$, $C_4(AB)$, and $C_4(ABC)$ are needed. If $C_4(A) = C_4(AB) = C_4(ABC) = \emptyset$, $ABCD$ (and all other nodes in level 4 with prefix ABC) are removed from the lattice and the candidate nodes $ABCD\mathbf{X}$ (for some non-empty $\mathbf{X}$) are not generated. The function PRUNE (Algorithm 6) checks in lines 3 to 9 if all candidate sets for the current node are empty and, if so, deletes the node from the current level's set of nodes L_l (line 11). Because empty candidate sets are not needed anymore, they are deleted from CS_l in line 14.

Algorithm 6 Function PRUNE

```
1: procedure PRUNE($L_l$, $CS_l$)
2:     for each $node \in L_l$
3:         $allEmpty \leftarrow false$
4:         for each $prefix \in$ PREFIXES ($node$)
5:             if $C_l(prefix) \neq \emptyset$
6:                 $allEmpty \leftarrow false$
7:                 break
8:             else
9:                 $allEmpty \leftarrow true$
10:        if $allEmpty = true$
11:            remove $node$ from $L_l$
12:    for each $C_l(\mathbf{X}) \in CS_l$
13:        if $C_l(\mathbf{X}) = \emptyset$
14:            remove $C_l(\mathbf{X})$ from $CS_l$
```

6.3 Level generation

In level l, the function GENERATENEXTLEVEL (Algorithm 7) generates nodes of size $l + 1$ in the candidate lattice by joining two nodes of size l with the same prefix of size $l - 1$. The procedure PREFIXBLOCKS is used in TANE as well [10]. It partitions the nodes of size l in L_l into lists of nodes with the same prefix of size $l - 1$ (line 3). Because, unlike TANE and other typical uses of the Apriori approach, we need to generate all *permutations* of attributes of size $l + 1$, we compute *all pairs* with distinct first and second components (lines 4 to 9) from each such prefix block.

Algorithm 7 ORDER: GENERATENEXTLEVEL

```
1: function GENERATENEXTLEVEL($L_l$)
2:     $L_{l+1} \leftarrow \emptyset$
3:     for each $prefixBlock \in$ PREFIXBLOCKS ($L_l$)
4:         for each $node \in prefixBlock$
5:             for each $joinNode \in prefixBlock$
6:                 if $node = joinNode$
7:                     continue
8:                 $joinedNode \leftarrow$ JOIN ($node, joinNode$)
9:                 add $joinedNode$ to $L_{l+1}$
10:    for each $node \in L_l$
11:        $C_l(node) \leftarrow \emptyset$                ▷ needed in the next level
12:        add $C_l(node)$ to $CS_l$
13:    return $L_{l+1}$
```

The procedure JOIN (line 8) is fairly simple: Given the pair of nodes (n_1, n_2) with each component of size l and the same prefix of size $l - 1$, JOIN creates a new node of size $l + 1$, with n_1 taking up the first l positions and the last attribute of n_2 at the last position. The joined node of size $l + 1$ is then added to L_{l+1} (line 9). Lines 10 to 12 initialize the candidate sets for the next level, which are then populated by UPDATECANDIDATESETS.

6.4 Algorithm extensions

There exist several other types of ODs, in particular bidirectional ODs [19,22] and partial ODs [5]. While these types of order dependencies are not the focus of this work, we discuss the changes needed in ORDER to facilitate their discovery.

Partial Order Dependencies are order dependencies that hold only for a subset of the data. To discover partial ODs, the order dependency check (Algorithm 1) has to be changed as to not terminate the check as soon as one tuple pair forming a swap is found. To establish the necessary notion of near-sortedness, we could make use of an algorithm to find the longest common subsequence between two sorted partitions, and compare its length to a given threshold (a deletion-based metric).

Bidirectional Order Dependencies allow mixed sort orders, i.e., an OD could be of the form $A \rightarrow_{\leq,\geq} B$, which states that for any two tuples r, s, if $r[A] \leq s[A]$, then $r[B] \geq s[B]$. The concept of a split can be directly transferred to bidirectional ODs. Swaps can be defined for bidirectional ODs as well by demanding a reverse order of the tuples of the left- and right-hand side of an OD. Thus, the only part of ORDER to change such that these bidirectional ODs are discovered is the dependency check (Algorithm 1), which then iterates the right-hand side sorted partition from back to front.

7 Evaluation

In this section, we evaluate the efficiency of the algorithm ORDER with respect to its ability to scale in the number of rows (Sect. 7.2) and columns (Sect. 7.3). We report the results of ORDER on ten real-world datasets and one synthetic dataset, which are described in Sect. 7.1. The algorithm was implemented for the Metanome data profiling framework [16]; all code and datasets are available at www.metanome.de.

All experiments were run on a Dell Poweredge R620 with CentOS 6.4, two Intel Xeon E5-2650 2GHz CPUs with 8 cores per CPU, and a total of 128 GB of main memory. There are eight hard disks in a RAID 6 configuration. The execution environment is a 64-bit OpenJDK in version 1.7.0_65, and the JVM heap space was limited to 100 GB.

As there exists no prior work on the discovery of order dependencies, we cannot perform a comparative analysis. Instead, we show that ORDER is able to detect order dependencies efficiently. For instance, ORDER finds the only OD in a dataset with 16 columns and 3 million rows in under 7 min and all three ODs in a dataset with 28 columns and 300 rows in under 30 s. While these number might not seem impressive, they are similar for instance to current state-of-the-art algorithms to detect functional dependencies on the same datasets [17], in the same execution environment (Metanome), and on the same server. The higher complexity of OD detection compared to FD detection is remedied by a considerably lower number of ODs than FDs found in

Table 14 Datasets and ORDER execution statistics

Dataset	Dataset properties				ORDER		
	$\|r\|$	$\|\mathcal{R}\|$	$\|\mathcal{OD}\|$	Max($\mathcal{OD}$)	#checks	Max(l)	Time
ABALONE	4,177	9	0	–	72	2	775 ms
ADULT	32,561	15	0	–	210	2	2031 ms
BRIDGES	108	13	0	–	156	2	517 ms
ECHO	132	13	12	2	156	2	529 ms
HEPATITIS	155	20	0	–	630	5	907 ms
HORSE	300	28	3	4	1,220	6	25.7 s
LETTER	20,000	17	0	–	272	2	1207 ms
WISCONSIN	699	11	0	–	110	2	592 ms
FLIGHTS	500,000	20	1,545†	6†	44,928†	6†	5 h†
NCVOTER	938,085	22	90†	4†	8,182†	4†	5 h†
LINEITEM	2,999,671	16	1	2	282	4	6.7 min

A "†" indicates that ORDER exceeded the time limit of 5 h ; the numbers are then intermediate results up to that level

the considered datasets – enabling more pruning possibilities during OD detection in ORDER.

7.1 Datasets

There are no publicly available datasets that have been checked for order dependencies. However, there are datasets that have previously been used to evaluate functional dependency detection algorithms. As the presence of functional dependencies in a dataset is a necessary condition for the presence of order dependencies, they constitute promising test data.

Table 14 lists the datasets and their properties. The first ten are real-world datasets, of which the first eight are published through the UCI Machine Learning Repository [11], FLIGHTS is a table containing data about US domestic flights, and NCVOTER is a dataset containing personal data of registered voters from North Carolina. The NCVOTER dataset originally contained 94 columns, resulting in an immensely high number of OD candidates, and is infeasible for ORDER to process. Thus, we excluded all columns that contain one or more `NULL` values, based on the assumption that the more interesting ODs originate from comparisons of non-`NULL` values. The result is the NCVOTER dataset with 22 columns listed in Table 14. We also include experiments on a synthetic dataset, namely the LINEITEM table of a sf = 0.5 TPC-H instance.

For each dataset, we report the number of rows ($|r|$), the number of columns without `NULL` values ($|\mathcal{R}|$), and the number of order dependencies ($|\mathcal{OD}|$). The size of the largest OD is given as max($\mathcal{OD}$), where the size of an OD is the sum of the sizes of its left- and right-hand sides. #checks is the number of dependency checks ORDER performed on the dataset, and max(l) indicates the highest level for which ORDER generated candidates.

7.2 Scalability in the number of rows

We expect ORDER to scale linearly with an increasing number of rows, because all operations on rows are performed on *sorted partitions* and have linear complexity, assuming constant insertion/deletion/search time on hash-based data structures. The mentioned operations on *sorted partitions* are the dependency check and the product of sorted partitions.

None of the UCI datasets has enough rows to draw profound conclusions about the scalability in the number of *rows* of ORDER. Thus, we concentrate on the real-world datasets FLIGHTS (500,000 rows), NCVOTER (938,085 rows), and the synthetic dataset LINEITEM with 2,999,671 rows. Because ORDER did not terminate on the full datasets, we consider only a subset of ten randomly chosen columns of the FLIGHTS and NCVOTER datasets.

For the experiments, we conduct several runs of ORDER per dataset with increasing number of randomly selected rows. Note that in principle, limiting the number of rows in a dataset could result in more ODs, resulting in a longer execution time of ORDER on the sample than on the full dataset. In practice, we did not encounter this situation; the initial sample sizes are chosen sufficiently large.

Figure 4 shows the results for the scalability experiment on the FLIGHTS dataset with up to 500,000 rows. Each 100,000 rows increases the overall execution time of ORDER by about 7 min. Figure 5 shows results for the LINEITEM table with up to 2,999,671 rows. Again, and as expected ORDER scales linearly in the number of rows. There is large difference in the execution times for the LINEITEM and FLIGHTS dataset: While processing 500,000 rows of the LINEITEM dataset takes just 50 s, processing the same number of rows on FLIGHTS takes 37 min. This difference is due to the much higher number of ODs in the FLIGHTS dataset (see Table 14), which causes ORDER to process the dataset up to level 9, while ORDER can terminate

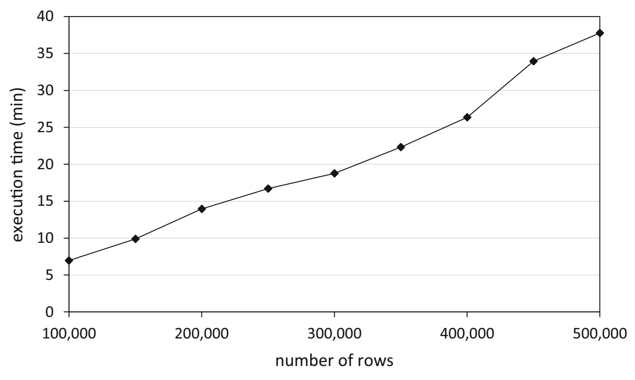

Fig. 4 Row scalability experiment on FLIGHTS (500,000 rows, 10 columns)

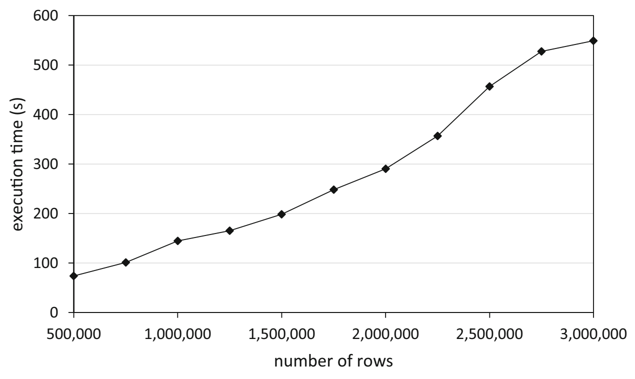

Fig. 5 Row scalability experiment on LINEITEM (2,999,671 rows, 16 columns)

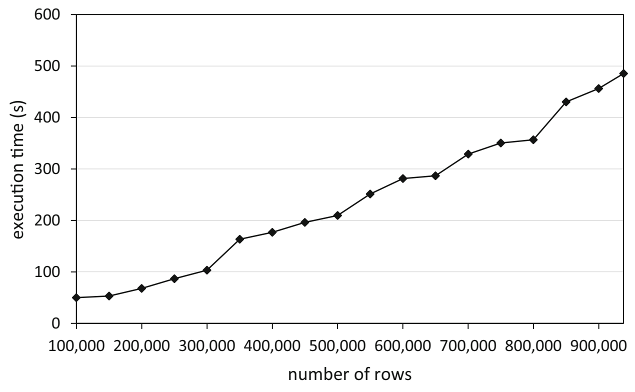

Fig. 6 Row scalability experiment on NCVOTER (938,085 rows, 10 columns)

on the LINEITEM table already after checking candidates in level 4. Finally, Fig. 6 confirms the linear scalability in the number of rows of ORDER on the NCVOTER dataset.

In summary, the experiments on all three datasets show similar results and support our initial claim that ORDER scales linearly with the number of rows. The experiments also show that ORDER is applicable to datasets with millions of rows, taking less than 7 min on the LINEITEM dataset with nearly 3,000,000 rows.

7.3 Scalability in the number of columns

In this section, we show how ORDER behaves on tables with an increasingly large number of columns. The UCI repository datasets HORSE and HEPATITIS contain only few rows, and both cause ORDER to generate OD candidates in higher levels (up to level 6 for HORSE and up to level 5 for HEPATITIS). Thus, these datasets are well suited for evaluation of the influence of an increasing number of columns on the execution time of ORDER. To better evaluate the performance impact of a larger number of columns, we limit the datasets FLIGHTS and NCVOTER to (randomly chosen) 1000 rows.

To evaluate ORDER on a dataset with an increasing number of columns, we take two columns from each dataset, and incrementally add more columns, until the number of columns in the dataset is reached.

To avoid skewing the results by choosing columns with especially high ordering impact, we generate k random 2-permutations of all columns. Each of the k 2-permutations is extended with another column, which is randomly chosen from the dataset, yielding k 3-permutations, and so on. ORDER is then executed on each of the k permutations of columns, and we report the execution time for c columns in the dataset as the average execution time over all k runs on c columns. The number of runs k is chosen per dataset, such that the set of experiments on each dataset could be run within 24 h.

Figure 7 shows the results of the column scalability experiment on the HORSE dataset, which contains three ODs. From considering 26 to considering 28 columns of the dataset, the execution time more than doubles from 10 to 25 s. This is due to the inherent complexity of OD detection: The candidate lattice for 28 columns has $\lfloor 28! \cdot e \rfloor \approx 8.29 \cdot 10^{29}$ nodes, and each node in level l in the lattice contains $l-1$ OD candidates. ORDER generates candidates up to level 6 in this lattice, which amounts to nearly 1.5 billion OD candidates (for 26 columns, the candidate lattice contains more than 860 million candidates up to level 6). Of course, because of pruning, ORDER enumerates only a small fraction of these candidates (namely 1220 of the 1.5 billion candidates) and thus is able to find all ODs within 10 and 25 s on 26 and 28 columns, respectively.

The column scalability experiment on the HEPATITIS dataset is depicted in Fig. 8. On this dataset, the number of dependency checks performed by ORDER grows quadratically in the number of columns. Interestingly, the graph does not clearly show such quadratic growth in execution time in the number of columns.

There are *swaps* among many of the columns in the HEPATITIS dataset and because of the small size of the dataset, the sorted partitions are small. Hence, the dependency check and calculation of the partition products run fast (less than 100 μs on average). If the dataset contained more rows, the quadratic increase in execution time could be observed more

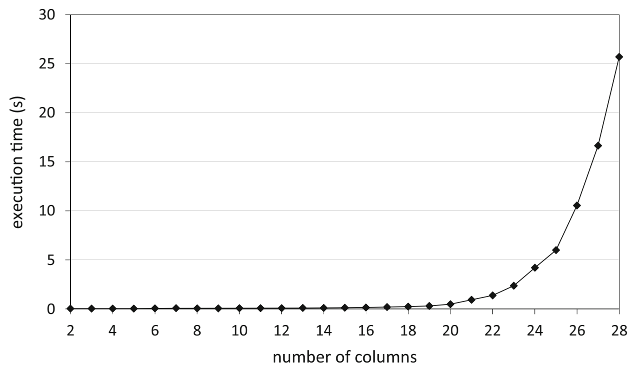

Fig. 7 Column scalability experiment on HORSE (28 columns, 300 rows, 3 ODs)

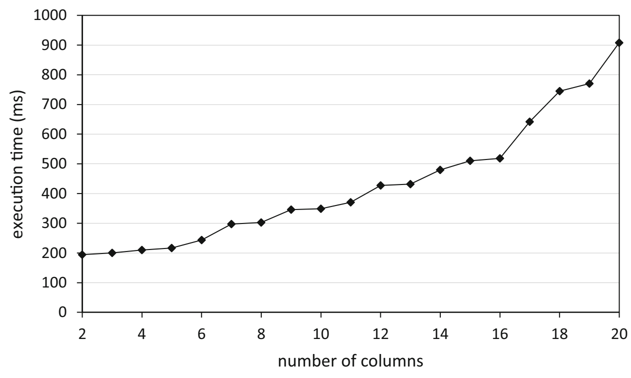

Fig. 8 Column scalability experiment on HEPATITIS (20 columns, 155 rows, 0 ODs)

clearly. Note that because ORDER can prune many candidates early on, the execution time is not exponential in the number of columns, as observed for the HORSE dataset.

Figure 9 shows the execution time of ORDER on an increasing number of columns on the FLIGHTS dataset alongside the number of found order dependencies.[1] Note the logarithmic scale on the vertical axes. On this dataset, we find more than 3,000 order dependencies when considering only 14 of the total of 20 columns.

Because ORDER needs to consider OD candidates that can be generated from valid ODs, the search space of open OD candidates (and thus, the overall execution time) grows rapidly with the number of columns. We aborted the experiment on 15 columns after 5 h. At that point, ORDER had found nearly 13,000 ODs. The large number of ODs in the FLIGHTS dataset originates from redundant data. For instance, FLIGHTS contains three columns that identify the origin airport of a flight (`Origin`, `OriginAirportID`, and `OriginAirportSeqID`), which results in ODs between all of these columns, and generates many more valid OD candidates in higher levels. A more interesting OD in FLIGHTS

[1] Figures 7 and 8 do not analogously show the number of ODs; there are too few, namely 3 and 0, respectively.

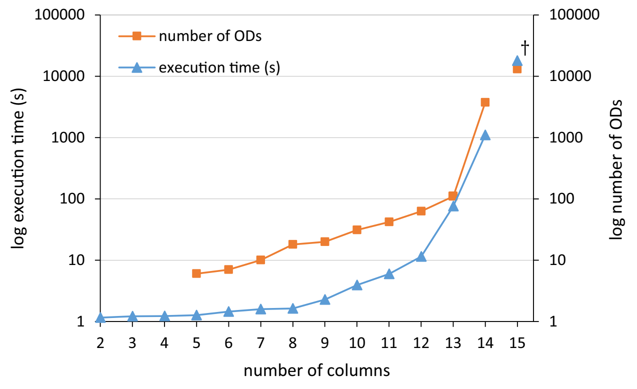

Fig. 9 Column scalability experiment on FLIGHTS (20 columns, 1000 rows). The *orange* (*squares*) *line* shows the number of found ODs, and the *blue* (*triangles*) *line* shows ORDER's execution time (note the logarithmic scale on the vertical axes). We aborted ORDER on its run on 15 columns after 5 h ("†")

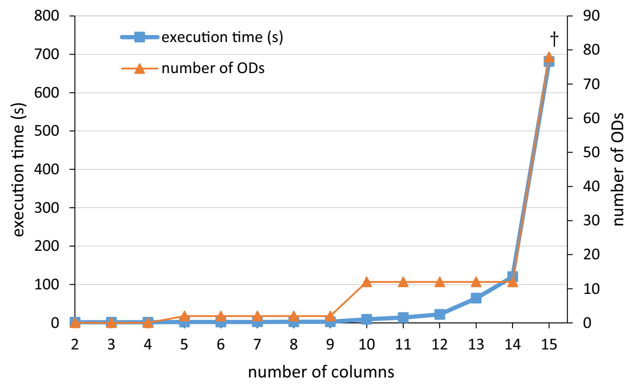

Fig. 10 Column scalability experiment on NCVOTER (22 columns, 1000 rows)

is `FlightDate` $\rightarrow_{\leq}$ `Month`; as in the given instance, all flight data are from the year 2012. In any instance, the OD `FlightDate` $\rightarrow_{\leq}$ `DayOfMonth, Month, Year` is valid and could be defined as a constraint. Note that the number of ODs in this experiment differs from the one reported in Table 14, because we consider only a small sample of the data, resulting in more found ODs than in the full dataset.

Finally, Fig. 10 tells a similar story for the NCVOTER dataset. As is the case for the FLIGHTS dataset, the larger number of ODs originates from columns that contain redundant data. On a subset of 14 columns of the dataset, ORDER terminates in about 100 s, finding all 12 ODs. As for the FLIGHTS dataset, this number of ODs differs from the one reported in Table 14, because we consider only a small sample of the data. Processing 15 columns takes about 700 ss and finds 78 ODs. We aborted ORDER on its run on 16 columns after 5 h ("†"). Again, we observe that the execution time of ORDER depends on the number of ODs found in the dataset. The NCVOTER dataset is sorted lexicographically by column `county`. Because each county is associated with an incrementing identifier in the column `county_id`, the order dependencies `county_id` $\rightarrow_{\leq}$ `county` and `county` $\rightarrow_{\leq}$ `county_id` are valid.

In summary, the column scalability experiments show that ORDER is applicable to datasets with tens of columns. We have also shown the limitations of the algorithm, when there are many valid ODs in the considered dataset. A comparative runtime analysis to show which pruning rules are most effective turned out to be intractable. For example, disabling the *swap* pruning rule leads to run times over 5 h (at that point we aborted the execution of the algorithm) on all considered datasets. In another similar experiment with the *merge*-pruning rule disabled, we terminated the execution of the algorithm on the LINEITEM table again after 5 h (note that with the *merge*-pruning rule enabled, the algorithm terminated in 6.7 min).

8 Conclusion and future work

Order dependencies are an interesting type of constraint with various use cases. While they are less common than, say, functional dependencies, they could lead to significant improvements of query optimization and aid in data cleansing. Order dependencies have been defined and treated theoretically, but this paper presented the first OD discovery algorithm and to this end some further theoretical results. The algorithm ORDER is efficient and scales well in light of similar results for functional dependency discovery algorithms.

As with any dependency discovery algorithm, several standard extensions can be made in the future: *Partial ODs* can be defined as order dependencies that hold for only a subset of the records in a relation, i.e., only some records violate the OD. Their discovery is in principle not difficult, as one can simply postpone the stopping criterion during validation. *Conditional ODs* are partial ODs for which a certain condition is known to select the non-violating records, i.e., the condition characterizes where a partial OD holds. Their discovery is difficult, as the search space is enormous, allowing conditions on various attributes and of various kinds.

Finally, and again as with all dependency discovery algorithms, ORDER discovers ODs that hold only for the current instance of the data. It is up to a human expert to promote a discovered OD to an actual constraint. The at times quite large results sets can be difficult to understand and interpret. Thus, we plan to develop methods to rank, visualize, and filter the set of observed dependencies.

Acknowledgments We thank Ziawasch Abedjan for his numerous helpful comments, which improved this work.

References

1. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB J. **24**(4), 557–581 (2015)
2. Abedjan, Ziawasch, Naumann, Felix: Advancing the discovery of unique column combinations. In: *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pp. 1565–1570, (2011)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: *Proceedings of the International Conference on Very Large Databases (VLDB)*, pp. 487–499, (1994)
4. De Marchi, F., Lopes, S., Petit, J.-M.: Unary and n-ary inclusion dependency discovery in relational databases. J. Intell. Inf. Syst. **32**(1), 53–73 (2009)
5. Dong, J., Hull, R.: Applying approximate order dependency to reduce indexing space. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, pp. 119–127, (1982)
6. Ginsburg, S., Hull, R.: Order dependency in the relational model. Theoret. Comput. Sci. **26**(1–2), 149–195 (1983)
7. Golab, L., Karloff, H.J., Korn, F., Saha, A., Srivastava, D.: Sequential dependencies. Proc. VLDB Endow. **2**(1), 574–585 (2009)
8. Halbeisen, L., Hungerbühler, N.: Number theoretic aspects of a combinatorial function. Notes Numb. Theory Discrete Math. **5**(4), 138–150 (1999)
9. Heise, A., Quiané-Ruiz, J.-A., Abedjan, Z., Jentzsch, A., Naumann, F.: Scalable discovery of unique column combinations. Proc. VLDB Endow. **7**(4), 301–312 (2013)
10. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100–111 (1999)
11. Lichman, M.: UCI machine learning repository. University of California, Irvine, School of Information and Computer Sciences (2013). http://archive.ics.uci.edu/ml. Accessed March 10, 2015
12. Liu, J., Li, J., Liu, C., Chen, Y.: Discover dependencies from data—a review. IEEE Trans. Knowl Data Eng. **24**(2), 251–264 (2012)
13. Naumann, F.: Data profiling revisited. SIGMOD Rec. **42**(4), 40–49 (2013)
14. Ng, W.: Ordered functional dependencies in relational databases. Inf. Syst. **24**(7), 535–554 (1999)
15. Northwestern University. WikiTables: Public Site (2015). http://downey-n1.cs.northwestern.edu/public. Accessed March 10, 2015
16. Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., Naumann, F.: Data profiling with Metanome. Proc. VLDB Endow. **8**(12), 1860–1871 (2015)
17. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J.-P., Schönberg, M., Zwiener, J., Naumann, F.: Functional dependency discovery: An experimental evaluation of seven algorithms. Proc. VLDB Endow. **8**(10), 1082–1093 (2015)
18. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences—A000522 (2015). http://oeis.org/A000522. Accessed March 10, 2015
19. Szlichta, J., Godfrey, P., Gryz, J.: Chasing polarized order dependencies. In: *Proceedings of the Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, pp. 168–179, (2012)
20. Szlichta, J., Godfrey, P., Gryz, J.: Fundamentals of order dependencies. Proc. VLDB Endow. **5**(11), 1220–1231 (2012)
21. Szlichta, J., Godfrey, P., Gryz, J., Ma, W., Qiu, W., Zuzarte, C.: Business-intelligence queries with order dependencies in DB2. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 750–761, (2014)
22. Szlichta, J., Godfrey, P., Gryz, J., Zuzarte, C.: Expressiveness and complexity of order dependencies. Proc. VLDB Endow. **6**(14), 1858–1869 (2013)