Chasing Polarized Order Dependencies

Jaroslaw Szlichta[1,2], Parke Godfrey[1,2], Jarek Gryz[1,2]

[1] York University, Toronto, Canada
[2] IBM Center for Advanced Studies, Toronto, Canada
{jszlicht, godfrey, jarek}@cse.yorku.ca

Abstract. Dependencies have played a significant role in database design for many years. They have also been shown to be useful in query optimization. In this paper, we discuss the new type of dependency for polarized lexicographically ordered sets of tuples. We introduce formally the concept of *polarized order dependencies* (PODs). We discuss their potential significance for database systems, and present a chase procedure for testing logical implication for them.

Keywords: Polarized Order Dependencies, Functional Dependencies, Chase

1 Introduction

Consider the following SQL query in Example 1.

EXAMPLE 1.

```
select year, quarter, month, sales
from Sales s, Dates d
where s.date_id = d.date_id
group by year, quarter, month, sales
order by year asc, quarter asc, month asc, sales desc
```

In the schema, `Dates` is a dimension table with a row per day, and `Sales` is a large *fact* table recording all individual sales. The column `date_id` is the primary key for `Dates`, each row describes a given day with explicit columns as `year`, `quarter`, `month`, and `day` that describe the natural date values.

Of course, `quarter` is logically redundant in the `group by`, as `month` (which follows it in the `group by`) functionally determines `quarter`. (First quarter encompasses the months of January, February, and March, second quarter, the months of April, May, and June, and so forth.) The query's author could not leave `quarter` out of the `group by`, because it is stated in the `select`. The query optimizer could, however, remove `quarter` to accomplish the `group by` on `year`, `quarter`, `month`, sales *if* it recognizes that `year`, `month` and `year`, `quarter`, `month` offer the same partition. This is done by query optimizers today – given the functional dependency (FD) information that `month` → `quarter` is available to the optimizer – by rewrite [16].

For the query above, the rewrite might still not be applied, however since the query also specifies the answers to be ordered by `year asc`, `quarter asc`, `month`

`asc, sales desc`. The FD that `month` → `quarter` is *not* logically sufficient to eliminate `quarter` from the `order by`, as it was to eliminate it from the `group by`. To see that the functional dependency does not suffice to eliminate `quarter` from the `order by`, imagine the values for `quarter` were the strings *first*, *second*, *third*, and *fourth*. Data would be lexicographically ordered as *first*, *fourth*, *second*, then *third*! Of course, we intend that values of `quarter` are, say, 1, 2, 3, and 4, so the data would order naturally as by date. It is unfortunate, then, that `quarter` is, in fact, redundant (in this query) in the `order by` also, but that the optimizer does not have the means to eliminate it. What is missing is the semantic information that `month asc` *orders* `quarter asc`, which is more than just that `month` *functionally determines* `quarter`. This states that as values *rise* from one tuple to another on `month`, they must *rise*, or stay the same, from the one tuple to the other on `quarter` (that is, the values do not descend from the one tuple to the other on `quarter`). The `order by` criteria can be a mix of `asc` and `desc` (as in Example 1). With this generalization we call it polarized order dependencies (POD). As best we know, polarized order dependencies have not been studied before.

Our objective is to bring a reasoning about PODs into the query optimizer. A query plan for the query above could then eliminate `quarter` from *both* the `order by` and the `group by` clauses. We are interested in PODs because polarized (`asc/desc`) lexicographical orders are part of SQL via `order by`, and it is how ordered tuple streams in a query are ordered. Therefore, PODs could be used to great effect in query optimization. A first question for any class of a dependencies is the inference problem: when does one POD logically follow from a collection of prescribed PODs? In this paper, we present a chase procedure for testing logical implication for PODs. Chase is a fixpoint algorithm enforcing satisfaction of data dependencies in databases [3, 11]. The chase algorithm is used to reason about the consistency and correctness of data design and in query optimization to rewrite queries.

Contributions. The main contributions of this paper follow. (Our work is the first work on PODs.)

1. *Applications.* We demonstrate the utility of the PODs for database systems.
   - We illustrate, the connection between PODs and the `order by` statement.
   - We show how polarized order dependencies – a new type of integrity constraint – can be used in query optimization.
2. *Chase procedure*. A chase procedure for polarized order dependencies for lexicographical orders.
   - We derive a chase procedure for polarized order dependencies over the sets of tuples ordered lexicographically, the first such in the literature.
   - We prove the soundness and completeness of chase rules for testing logical implication.

Outline. Section 2 provides background with our notational conventions and definitions. We motivate PODs, and discuss practical applications of them. The core of the paper is in Section 3, where we devise a chase procedure for testing logical implication. Related work is presented in Section 4. In Section 5, we make concluding remarks and discuss future work.

2 Background

We adopt the notational conventions below. We consider a relation $\boldsymbol{R}$ over the set of attributes $\mathcal{U}$. Let **r** be an arbitrary *table instance over* $\boldsymbol{R}$; thus, a *set* of tuples over $\boldsymbol{R}$'s schema with attributes $\mathcal{U}$. We assume a column (#) in $\boldsymbol{R}$ that takes a unique value per tuple, without loss of generality. This alleviates any need to work with a table instance as bags of tuples; we consider them as sets. This also removes the possibility that we might "lose" a row from **r** when we modify the value of one of its columns (besides #), since # will still distinguish it from other tuples. Our notational conventions are as follows.

- **Relations.** A capital letter in bold italics represents a *relation*: $\boldsymbol{R}$. Capital letters represent single *attributes*: A, B, C. A small letter in bold represent a *relational instance*: **r**. *Tuples* are marked with small letters in italics: *s, t*.
- **Sets.** Calligraphic letters stand for *sets of attributes*: $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$. Proximity is used for *union* of sets: $\mathcal{XY}$ is shorthand for $\mathcal{X} \cup \mathcal{Y}$. Likewise, A$\mathcal{X}$ or $\mathcal{X}$A, where $\mathcal{X}$ is a set of attributes and A is a single attribute, stands for $\mathcal{X} \cup \{A\}$. AB denotes {A, B}.
- **Lists.** Bold letters stand for *lists* of attributes: **X**, **Y**, **Z**. Note that list **X** could be the empty list, []. Square brackets denote a list: [A, B, C]. The notation [A | **Z**] denotes that A is the *head* of the list, and **Z** is the *tail* of the list, the remaining list with the first element removed. Proximity is used for concatenation of lists of attributes: **XY** is shorthand for **X** ∘ **Y**. Likewise, A**X** and **X**A stands respectively for [A] ∘ **X** and **X** ∘ [A], where **X** is list of attributes and A is a single attribute. AB denotes [A, B].

2.1 Definition of PODs

We consider *ascending* (`asc`) and *descending* (`desc`) order in the lexicographical ordering. (This is part of the SQL's standard). This also includes mixing of `asc` and `desc` (e.g., `order by X desc, Y asc`). Our work can be also easily extended to use of functions in the order directives (e.g., `order by -1*X asc, Y asc`).

Definition 1. (*marked attributes*) For each attribute A, the *marked attributes* of A are the formal symbols $\overline{\text{A}}$ and $\underline{\text{A}}$.

Marked attributes are used in the following way, giving us a *polarization* of attributes.

Definition 2. (*operator* $\preccurlyeq^{\text{PL}}$ and *operator* $\prec^{\text{PL}}$) Let **L** be a list of marked attributes, *s* and *t* be two tuples in relation instance **r**. Operator $\preccurlyeq^{\text{PL}}$ is defined as follows:

$s_{\mathbf{L}} \preccurlyeq^{\text{PL}} t_{\mathbf{L}}$ where $\mathbf{L} =$ [H | **Z**]
if (H is of the form $\overline{\text{A}}$ and $s_{\text{A}} < t_{\text{A}}$)
or if (H is of the form $\underline{\text{A}}$ and $s_{\text{A}} > t_{\text{A}}$)
or if (($s_{\text{A}} = t_{\text{A}}$) and (**Z** = [] or $s_{\mathbf{Z}} \preccurlyeq^{\text{PL}} t_{\mathbf{Z}}$))

Operator $\prec$ is defined as follows: $s_{\mathbf{X}} \prec t_{\mathbf{X}}$ iff $s_{\mathbf{X}} \preccurlyeq^{\text{PL}} t_{\mathbf{X}}$ and $t_{\mathbf{X}} \not\preccurlyeq^{\text{PL}} s_{\mathbf{X}}$.

We are now ready to define *polarized order dependency*.

Definition 3. (*polarized order dependency*) Let **X** and **Y** be a list of marked attributes. Call $\mathbf{X} \mapsto \mathbf{Y}$ a *polarized order dependency* over the relation $\boldsymbol{R}$ iff, for every pair of permissible tuples s and t in relation instance $\mathbf{r}$ over $\boldsymbol{R}$, $s_{\mathbf{X}} \preccurlyeq^{\mathrm{PL}} t_{\mathbf{X}}$ implies $s_{\mathbf{Y}} \preccurlyeq^{\mathrm{PL}} t_{\mathbf{Y}}$.

Whenever $\mathbf{X} \mapsto \mathbf{Y}$, we say that **X** *orders* **Y**. **X** and **Y** are *order equivalent iff* $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$. We denote this by $\mathbf{X} \leftrightarrow \mathbf{Y}$.

Table 1 Relation instance **r**.

#	A	B	C	D	E
s	1	4	4	6	3
t	2	3	4	6	4

The order of the attributes for FDs is not important. For PODs only particular permutations of attributes may hold as dependencies because they are built on lists. The order of the attributes is important as well as how they are polarized. PODs are prescriptive statements on the relation, as are FDs. That is, they can be used as a type of integrity constraint to prescribe which instances are admissible.

EXAMPLE 2. Let **r** be a relation instance over $\boldsymbol{R}$ with attributes {A, B, C, D, E} as shown in Table 1. Note $\mathbf{r} \vDash \overline{\mathrm{ACD}} \mapsto \overline{\mathrm{EB}}$ but $\mathbf{r} \nvDash \overline{\mathrm{ACD}} \mapsto \overline{\mathrm{BE}}$. Furthermore, $\mathbf{r} \vDash \underline{\mathrm{CA}} \mapsto \overline{\mathrm{B}}\underline{\mathrm{DE}}$ but $\mathbf{r} \nvDash \underline{\mathrm{CA}} \mapsto \overline{\mathrm{EB}}\underline{\mathrm{D}}$.

There is a relationship between PODs and FDs. We show that every POD infers a FD.

THEOREM 1. (relationship between PODs and FDs) *For every instance* **r** *of relation* $\boldsymbol{R}$*, if* **r** *satisfies OD* $\mathbf{X} \mapsto \mathbf{Y}$*, then* **r** *satisfies FD* $\mathcal{X} \rightarrow \mathcal{Y}$ (*in which* $\mathcal{X}$ *is the set of attributes in* **X***, and* $\mathcal{Y}$ *in* **Y**).

PROOF. Let $s, t \in \mathrm{r}$, such that $s_{\mathcal{X}} = t_{\mathcal{X}}$. Therefore, $s_{\mathbf{X}} \preccurlyeq^{\mathrm{PL}} t_{\mathbf{X}}$ and $t_{\mathbf{X}} \preccurlyeq^{\mathrm{PL}} s_{\mathbf{X}}$. By the definition of POD $s_{\mathbf{Y}} \preccurlyeq^{\mathrm{PL}} t_{\mathbf{Y}}$ and $t_{\mathbf{Y}} \preccurlyeq^{\mathrm{PL}} s_{\mathbf{Y}}$ as $\mathbf{X} \mapsto \mathbf{Y}$ is given, hence $s_{\mathcal{Y}} = t_{\mathcal{Y}}$.

THEOREM 2. (FD/POD correspondence) *For every instance* **r** *of relation* $\boldsymbol{R}$*,* $\mathbf{r} \vDash \mathcal{X} \rightarrow \mathcal{Y}$ *iff* $\mathbf{r} \vDash \mathbf{X} \mapsto \mathbf{XY}$*, for all lists* **X** *that order the attributes of* $\mathcal{X}$ *and all lists* **Y** *likewise for* $\mathcal{Y}$.

PROOF. **(IF)** If $\mathbf{X} \mapsto \mathbf{XY}$, then $\mathcal{X} \rightarrow \mathcal{XY}$ by Theorem 1. By Armstrong's axiom Reflexivity, $\mathcal{XY} \rightarrow \mathcal{Y}$ holds. Therefore, by Armstrong's axiom Transitivity, $\mathcal{X} \rightarrow \mathcal{Y}$ is true.

(ONLY IF) If $\mathbf{X} \mapsto \mathbf{XY}$ does not hold, there exists $s, t \in \mathbf{r}$, such that $s_{\mathbf{X}} \preccurlyeq^{\mathrm{PL}} t_{\mathbf{X}}$ but $s_{\mathbf{XY}} \not\preccurlyeq^{\mathrm{PL}} t_{\mathbf{XY}}$. This implies that $s_{\mathbf{X}} = t_{\mathbf{X}}$ and $t_{\mathbf{Y}} \prec^{\mathrm{PL}} s_{\mathbf{Y}}$. Therefore $s_{\mathcal{Y}} \neq t_{\mathcal{Y}}$ and $s_{\mathcal{X}} = t_{\mathcal{X}}$, $\mathcal{X} \rightarrow \mathcal{Y}$ is not true.

Note that the weakening rule $\mathbf{X} \mapsto \mathbf{XY}$ implies $\mathbf{X} \mapsto \mathbf{Y}$ does not hold and as such, FDs and PODs are distinct.

2.2 PODs in Databases

The concept of functional dependencies has come to have profound importance in databases, especially in schema design. While functional dependencies are a simple

notion in some ways, reasoning over them is, somewhat surprisingly, not nearly as simple. To gain insight into how sets of FDs behave, and to simplify the reasoning process over them, Armstrong provided an axiomatization for them [1]. Beyond layout and indexes, FDs play additional important roles in query optimization.

We have introduced PODs in analogy to FDs: FDs are to `group by` as PODs are to `order by`. Order plays pivotal roles on the physical side, in the physical database and in query optimization. Data is often stored sorted by a clustered (tree) index's key. In a query plan, an operator that takes as input the output stream of another operator can benefit in cases when the stream is sorted in a particular way. Given POD $\mathbf{X} \mapsto \mathbf{Y}$, if one has an SQL query with `order by` **Y**, one can rewrite the query with `order by` **X** instead, and meet the intent of the original query. However, the rewritten query is *not* semantically equivalent to the original (unless $\mathbf{X} \leftrightarrow \mathbf{Y}$)! One could not legally rewrite the query with `order by` **X** with `order by` **Y** instead. Strengthening the `order by` conditions is permitted, but weakening them is not. (This is true too inside query plans for ordered tuple streams.)

A POD can be declared as an *integrity constraint* to prescribe which instances are admissible. If one knows a collection of PODs, $\mathcal{M}$ – declared as integrity constraints over relation $\boldsymbol{R}$ – one might soundly infer additionally PODs that must be *true* for $\boldsymbol{R}$ and use them for query optimization. For example, if $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{Z}$ are *true*, then $\mathbf{X} \mapsto \mathbf{Z}$ is *true* also.

Polarized order dependencies are not just limited to the time domain as used in Example 1, however. They arise naturally in many other domains from the real-world semantics associated with given data. Consider Example 3, which concerns taxes.

EXAMPLE 3. Consider a table `Taxes` that includes columns for taxable `income`, tax `bracket`, and `taxes` on the income. The tax brackets are based on the level on income (the values of the tax brackets are: A, B, C, D and decrease with income level). Assume taxes go up with income. Then, from $[\overline{\text{income}}] \mapsto [\underline{\text{bracket}}]$ and $[\overline{\text{income}}] \mapsto [\overline{\text{taxes}}]$ it follows that $[\overline{\text{income}}] \mapsto [\underline{\text{bracket}}, \overline{\text{taxes}}]$. Assume the table has a tree (clustered) index on `income`. Given a query on the table with an `order by` on `bracket desc, taxes asc,` with the POD above, it could be evaluated using the index on `income` (for `order by income asc`), avoiding potentially an expensive sorting operation.

Instead of being columns with explicit data, `bracket` and `taxes` could be derived by functions or case expressions – say, if Taxes were a view – or generated columns in the table. In these cases, it would be possible for the database system to derive the polarized order dependency constraints above automatically. In [12], it was shown how to derive such monotonicity "constraints" from generated columns via algebraic expressions (in IBM DB2). Of course, one could prescribe the set of polarized order dependencies as check constraints directly to benefit by this technique.

In [16], the authors expounded on the important role of *order* in query optimization. They demonstrated numerous examples of how better *reasoning* over *interesting orders* in the query optimizer could lead to significantly better performing query plans. They introduced query rewrites in IBM DB2 that could replace one labeled interesting order by another, when it is known the two order in the same way (that is,

are *order equivalent*, as we have defined it). For that, they use notion of FDs. They showed how these rewrites could allow the optimizer to consider additional query plans that process `join`, `order by`, `group by`, and `distinct` operators more efficiently. However, they could not reduce the `order by year asc, quarter asc, month asc, sales desc` to `year asc, month asc, sales desc` as we did in Example 1, since their techniques do not employ the idea of PODs.

By recognizing that a tuple stream ordered with respect to some criteria is equivalently ordered with respect to other criteria, a sort on input can be removed for a sort-merge join. `Order by` and `group by` operators can be satisfied with no need for a sorting or partitioning operation more often. Likewise, as the `distinct` operator is exchangeable with `group by`, the need for a sorting or partitioning operation to satisfy `distinct` can be lessened. In [16], they introduced a rewrite algorithm for `order by` called *Reduce Order*. It sweeps the `order by` attribute list from right to left, seeking to eliminate attributes. At each iteration through the list, the prefix *set* with respect to the *current* attribute – that is, the set of attributes to the left of the current – is checked to see whether it *functionally determines* the current attribute. If so, the attribute is dropped from the list. We can augment that algorithm – call it *Reduce Order** – to do an additional step. At each iteration through the list, it can additionally be checked whether any postfix *list* with respect to the current attribute – that is, the list of attributes to the right of the current – *orders* the current attribute, where the `asc` and `desc` gives us a polarization.

3 A Chase Procedure for PODs.

A goal in any dependency theory is to develop algorithms for testing logical implication; that is, testing whether a dependency is satisfied based on a given set of dependencies. In this section, we show how to test logical implication for PODs using *chase* procedure.

Definition 4. (*equalize*) Let *s* and *t* be two tuples in relation instance **r**, and let A be a single attribute. Also let $x = \min(s_A, t_A)$. The operation equalize(**r**, A, *s*, *t*) returns a relational instance **r'**, with *s* and *t* modified in **r** so $s_A = x$ and $t_A = x$.

EXAMPLE 4. Consider Table 2 and Table 3 as an example of an operation equalize.

Table 2 Instance **r** = {s, t}

#	A
s	0
t	1

Table 3 Instance **r'** = equalize(**r**, A, *s, t*)

#	A
s	0
t	0

Now, we are going to introduce chase rules, which are applied to two rows in a relation instance with respect to set of PODs $\mathcal{M}$.

Definition 5. (*chase rules*) Let *s* and *t* be two tuples in relation instance **r**, and let **X** and **Y** be lists of marked attributes such that $\mathbf{X} \mapsto \mathbf{Y}$ is falsified in **r** ($s_\mathbf{X} \preccurlyeq^{PL} t_\mathbf{X}$ but $s_\mathbf{Y} \not\preccurlyeq^{PL} t_\mathbf{Y}$), let A be the first attribute in **X** such that $s_A \neq t_A$ (if such an attribute A exists) and let B be first attribute in **Y** such that $t_B \neq s_B$. Two chase rules are defined.

- **Split rule:** If $s_{\mathbf{X}} = t_{\mathbf{X}}$, then **r'** = equalize(**r**, B, *s, t*).
- **Swap rule:** If $s_{\mathrm{A}} \neq t_{\mathrm{A}}$ and $s_{\mathrm{B}} \neq t_{\mathrm{B}}$, then **r'** = equalize(**r**, B, *s, t*).

EXAMPLE 5. *Let* $\mathcal{M} = \{\overline{\mathrm{A}} \mapsto \overline{\mathrm{BC}}, \overline{\mathrm{B}} \mapsto \underline{\mathrm{C}}\}$ and let **r** be an instance over ***R*** with attributes {A, B, C}. From Table 4 to Table 5 demonstrates an example of applying the *split* rule ($\overline{\mathrm{A}} \mapsto \overline{\mathrm{BC}}$ is falsified in **r** in Table 4). From Table 5 to Table 6 demonstrates applying the *swap* rule ($\overline{\mathrm{B}} \mapsto \underline{\mathrm{C}}$ is falsified in **r'** in Table 5).

Let $\mathcal{M}$ be a set of prescribed PODs, and let $\mathbf{r} = \{t_1, \ldots, t_n\}$ be an instance relation over ***R***. The chase algorithm is as follows.

Algorithm 1 (chase(**r**, $\mathcal{M}$))

1. Current := **r**;
2. Previous := {}; //empty instance
3. **while** Current ≠ Previous {
4. Previous := Current;
5. **if** (∃ t,s ∈ Current, ∃ **X** ↦ **Y** in $\mathcal{M}$ such that $s_{\mathbf{X}} \preccurlyeq^{\mathrm{PL}} t_{\mathbf{X}}$ but t $s_{\mathbf{Y}} \not\preccurlyeq^{\mathrm{PL}} t_{\mathbf{Y}}$) {
6. Apply one of the chase rules (*split* or *swap* from Definition 5) to *s* and *t*,
7. assigning the table which is returned from *equalize* operation to Current.
8. }
9. } **return** Current;

EXAMPLE 6. *Let* $\mathcal{M} = \{\overline{\mathrm{A}} \mapsto \overline{\mathrm{BC}}, \overline{\mathrm{B}} \mapsto \underline{\mathrm{C}}\}$ be a set of PODs and **r** an instance over ***R*** with attributes {A, B, C}. The sequence from Table 4, Table 5, to Table 6 is an example of applying *Algorithm 1* (chase(**r**, $\mathcal{M}$)) to **r** as in Table 4. In Table 6, there is no *s* and *t* that matches (as in step 5 of Algorithm 1), so the procedure terminates with it.

Table 4 Table instance **r** ⊭ $\mathcal{M}$

#	A	B	C
s	1	1	2
t	1	1	1
u	3	3	3

Table 5 r' ⊭ $\mathcal{M}$, using *split* rule for rows s, *t*.

#	A	B	C
s	1	1	1
t	1	1	1
u	3	3	3

Table 6 chase(**r**, $\mathcal{M}$) ⊨ $\mathcal{M}$, using *swap* rule for rows *t*, *u*.

#	A	B	C
s	1	1	1
t	1	1	1
u	3	3	1

LEMMA 1. (termination and satisfaction) *Algorithm 1 terminates and the resulting table of Algorithm 1 satisfies set of PODs* $\mathcal{M}$.

PROOF. Consider a given relational instance **r**, and any relational instance **s**, over schema ***R***. Without loss of generality let all values in **r** and **s** be zero or greater. Let $\sum_{\mathbf{s}}$ be the sum of all the values of all the columns in **s**. Let there be an applicable chase rule – *split* or *swap* – on **s** with respect to $\mathcal{M}$, and **s'** be the result of its application. Instance **s'** has the same number of rows as **s**. Also $\sum_{\mathbf{s}'} < \sum_{\mathbf{s}}$ as the *equalize* replaced a value in some column of some row by a smaller value. (Note that *equalize* does not introduce new values.) Zero is the lower bound on the $\sum_{\mathbf{s}'}$. As a chase procedure is a finite sequence of such transformations starting with **r**, it must terminate.

The remaining step is to show that the resulting table of Algorithm 1 satisfies a set of PODs $\mathcal{M}$. The instance chase(**r**, $\mathcal{M}$) satisfies $\mathcal{M}$ as no *split* or *swap* with respect to $\mathcal{M}$ applies. If not the chase procedure would not have terminated at that point. □

THEOREM 3. *Let relation instance* **r** *be over* ***R*** *and let* $\mathcal{M}$ *be a set of PODs. Then* **r** ⊨ $\mathcal{M}$ *iff* **r** = chase(**r**, $\mathcal{M}$).

PROOF. **(IF):** If any *split* or *swap* applies to table instance **s** with respect to $\mathcal{M}$, for the resulting **s'**, $\sum_{s'} < \sum_{s}$. Then clearly **s** ≠ **s'**. Thus **r** = chase(**r**, $\mathcal{M}$) if no swap or split applies, meaning **r** ⊨ $\mathcal{M}$. **(ONLY IF):** From Definition 5 it follows that chase rules *split* or *swap* are only used if it breaks a dependency in $\mathcal{M}$. □

Let us define a *table template*.

Definition 6. (*table template*) *Let* ***R*** be relation schema with *n* attributes and *m* be a POD **X** ↦ **Y**, where list of marked attributes **X** contains attributes $[X_1, \dots, X_k]$. A table template for a POD *m*, denoted as $\mathbf{r}_m$, is a table consisting of two tuples *s* and *t*, such that is either equal to $\mathbf{r}_0$ or $\mathbf{r}_j$, for *j* in *1,..., k*. In $\mathbf{r}_0$ and $\mathbf{r}_j$, symbols p_i and q_i represents one of the following three cases, where the *ordering* of variables b_i and t_i is defined as $b_i < t_i$:

a) $p_i = b_i$ and $q_i = b_i$,
b) $p_i = b_i$ and $q_i = t_i$,
c) $p_i = t_i$ and $q_i = b_i$.

Table 7 Template $\mathbf{r_0}$

#	X_1		X_k	$R - \{X_1, \dots, X_k\}$		
s	b_1	…	b_k	p_{k+1}	…	p_n
t	b_1	…	b_k	q_{k+1}	…	q_n

Table 8 Template $\mathbf{r_j}$

#	X_1		X_{j-1}	X_j	$\boldsymbol{R} - \{X_1, \dots, X_j\}$		
s	b_1	…	b_{j-1}	b_j	p_{j+1}	…	p_n
t	b_1	…	b_{j-1}	t_j	q_{j+1}	…	q_n

Please note that we apply chase rules *split* and *swap* on table templates using *ordering* $b_i < t_i$ which is part of the Definition 6.

Definition 7. (*mapping* $\mathbf{r}_m$ *to* ö($\mathbf{r}_m$)) Let $\mathbf{r}_m$ be a table template from Definition 6. A mapping of $\mathbf{r}_m$ to ö($\mathbf{r}_m$) is *any* instance with values that satisfy the *ordering* from Definition 6.

EXAMPLE 7. Consider Table 9 as one of possible mappings from Definition 7. (In fact it can be any relation instance which satisfies the Definition 6 of *ordering* of variables).

Table 9 Table template $\mathbf{r_m}$.

#	A	B	C	D
s	b_1	b_2	b_3	h_4
t	b_1	b_2	t_3	b_4

Table 10 Instance ö($\mathbf{r_m}$).

#	A	B	C	D
s	1	4	0	8
t	1	4	1	7

LEMMA 2. *Let* $\mathbf{r}_m$ *be a table template from Definition 6, where m is a POD* **X** ↦ **Y**. *Then* $\mathbf{r}_m$ ⊨ $\mathcal{M}$ *iff* $\mathbf{r}_m$ = chase($\mathbf{r}_m$, $\mathcal{M}$).

PROOF. The proof follows directly from Theorem 3 by replacing $\mathbf{r}_m$ with $\mathbf{r}$ and applying chase rules *split* and *swap* on variables b_i, t_i using ordering defined in Definition 6. □

LEMMA 3. *Let* $\mathbf{r}_m$ *be a table template from Definition 6 and* ö($\mathbf{r}_m$) *be mapping from* $\mathbf{r}_m$ *(Definition 7). Then* $\mathbf{r}_m \vDash \mathbf{X} \mapsto \mathbf{Y}$ *iff* ö($\mathbf{r}_m$) $\vDash \mathbf{X} \mapsto \mathbf{Y}$.

PROOF. The proof follows from the definition of ordering of variables in Definition 6. Since ordering of values in ö($\mathbf{r}_m$) corresponds with ordering of variables in $\mathbf{r}_m$ respectively (Definition 7). □

Definition 8. (*tableaux* $\mathbf{T}_m$) Let *m* be a POD $\mathbf{X} \mapsto \mathbf{Y}$. We define $\mathbf{T}_m$ to be the set of all table templates $\mathbf{r}_m$, as defined in Definition 6.

Note that $\mathbf{T}_m$ is not just a single table template. It is a *set* of table templates (each consisting of two rows). The chase of $\mathbf{T}_m$ is defined as follows.

Definition 9. (*chase of tableaux* $\mathbf{T}_m$) The chase of $\mathbf{T}_m$ over a set of PODs $\mathcal{M}$, denoted as $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}}$ is defined by $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}}$ = {chase($\mathbf{r}_m$, $\mathcal{M}$) | $\mathbf{r}_m \in \mathbf{T}_m$ }. $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}}$ satisfies $\mathbf{X} \mapsto \mathbf{Y}$, denoted by $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathbf{X} \mapsto \mathbf{Y}$, if, for all $\mathbf{r}_m \in \mathbf{T}_m$, chase($\mathbf{r}_m, \mathcal{M}$) $\vDash \mathbf{X} \mapsto \mathbf{Y}$. $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}}$ satisfies $\mathcal{M}'$ denoted by $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathcal{M}'$, if for all $\mathbf{X} \mapsto \mathbf{Y} \in \mathcal{M}'$, $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathbf{X} \mapsto \mathbf{Y}$.

THEOREM 4. (chase procedure for PODs is sound and complete) *Let* $\mathcal{M}$ *be a set of PODs over* ***R*** *and m be a POD* $\mathbf{X} \mapsto \mathbf{Y}$. *Then* $\mathcal{M} \vDash \mathbf{X} \mapsto \mathbf{Y}$ *iff* $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathbf{X} \mapsto \mathbf{Y}$.

PROOF. **(IF):** Assume $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \nvDash \mathbf{X} \mapsto \mathbf{Y}$. By Definition 9, there exists $\mathbf{r}_m \in \mathbf{T}_m$, such that chase($\mathbf{r}_m, \mathcal{M}$) $\nvDash \mathbf{X} \mapsto \mathbf{Y}$. Note chase($\mathbf{r}_m, \mathcal{M}$) $\vDash \mathcal{M}$, by Lemma 1. Hence, there is a mapping ö to generate a relation instance ö(chase($\mathbf{r}_m, \mathcal{M}$)) and by Lemma 3, ö(chase($\mathbf{r}_m, \mathcal{M}$)) $\vDash \mathcal{M}$, but ö(chase($\mathbf{r}_m, \mathcal{M}$)) $\nvDash \mathbf{X} \mapsto \mathbf{Y}$. This implies that $\mathcal{M} \nvDash \mathbf{X} \mapsto \mathbf{Y}$ because we have found a relation instance which satisfies $\mathcal{M}$ but does not satisfy $\mathbf{X} \mapsto \mathbf{Y}$. Therefore, if $\mathcal{M} \vDash \mathbf{X} \mapsto \mathbf{Y}$ then $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathbf{X} \mapsto \mathbf{Y}$.

(ONLY IF): Assume $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathbf{X} \mapsto \mathbf{Y}$. Let *s, t* be *any* two tuples in any relation $\mathbf{r}$ such that $s_{\mathbf{X}} \preccurlyeq^{\text{PL}} t_{\mathbf{X}}$ and satisfying $\mathcal{M}$. We would like to present that $s_{\mathbf{Y}} \preccurlyeq^{\text{PL}} t_{\mathbf{Y}}$. Let $\mathbf{r}_m \in \mathbf{T}_m$, let $\mathbf{r}_m = \{p, q\}$ be the template relation such that ö(p) = s and ö(q) = t. It is possible always to find such a pair of tuples $\mathbf{r}_m$ since $\mathbf{T}_m$ considers all possibilities of two tuples which satisfy the condition $s_{\mathbf{X}} \preccurlyeq^{\text{PL}} t_{\mathbf{X}}$. Therefore, we have ö($\mathbf{r}_m$) = $\{s, t\}$ and ö($\mathbf{r}_m$) $\vDash \mathcal{M}$. By Lemma 3, it follows that $\mathbf{r}_m \vDash \mathcal{M}$. Therefore, it follows by Lemma 2 that $\mathbf{r}_m$ = chase($\mathbf{r}_m, \mathcal{M}$). Since we have assumed that $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathbf{X} \mapsto \mathbf{Y}$, we have chase($\mathbf{r}_m$, $\mathcal{M}$) $\vDash \mathbf{X} \mapsto \mathbf{Y}$. As ö($\mathbf{r}_m$) = ö(chase($\mathbf{r}_m$, $\mathcal{M}$)), it implies that ö($\mathbf{r}_m$) $\vDash \mathbf{X} \mapsto \mathbf{Y}$ by Lemma 3. So $s_{\mathbf{Y}} \preccurlyeq^{\text{PL}} t_{\mathbf{Y}}$. Hence, if $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \vDash \mathbf{X} \mapsto \mathbf{Y}$, then $\mathcal{M} \vDash \mathbf{X} \mapsto \mathbf{Y}$. □

THEOREM 5. (decidable) *The implication problem of the PODs is decidable.*

PROOF. Testing implication problem of the PODs is decidable as the chase procedure is a sound and complete inference algorithm for PODs (Theorem 4). □

THEOREM 6. (complexity of chase procedure for PODs) *The complexity of building templates is exponential for the PODs chase procedure.*

PROOF. According to Definition 6 there are 3^{n-k} templates for $\mathbf{r}_0$ and 3^{n-j} templates for each, $\mathbf{r}_j$. Therefore there are 3^{n-k} + (3^{n-k} +…3^{n-1}) templates in total. Because (3^{n-k} +…+ 3^{n-1}) is geometric progression this can be simplified to the form 3^{n-k} + 3^{n-k}(1 - 3^k)/(1-3) which is equal to (3^n+ 3^{n-k})/2. So the complexity of building the templates is shown to be $O(3^n)$. □

4 Related Work

Ordered sets and lattices have been a subject of research in mathematics. Our concept of polarized order dependency is equivalent to *order-preserving mappings* between ordered sets [6]. The work in mathematics has concentrated on investigating properties of, and relationships between, ordered sets rather than among the mappings. To the best of our knowledge, no inference system for describing relationships between mappings has been proposed.

Order dependencies were introduced for the first time in the context of database systems in [8]. However, the type of orders, hence the dependencies defined over them, were different from the ones we presented here. A dependency[1] $\mathcal{X} \rightsquigarrow \mathcal{Y}$ holds if order over the values of *each* attribute in $\mathcal{X}$ implies an order over the values of *each* attribute of $\mathcal{Y}$. This dependency is defined over the sets of attributes then, rather than lists. The distinction between these two types of dependencies was later aptly described as pointwise versus lexicographical order dependency [13]. An instance of a database satisfies a pointwise order dependency $\mathcal{X} \rightsquigarrow \mathcal{Y}$ if, for all tuples *s* and *t,* for every attribute A in $\mathcal{X}$, s[A] *op* t[A] implies that for every attribute B in $\mathcal{Y}$, s[B] *op* t[B], where *op* $\in \{<, =, >, \leq, \geq\}$. In [8], a sound and complete set of inference rules for such dependencies is defined together with an analysis of the complexity of determining logical implication. A practical application of the dependencies for an improved index design is presented in [7].

Dependencies defined over lexicographically ordered domains were introduced in [14] under the name *lexicographically ordered functional dependencies*. The order dependencies are defined as we do in this paper, but do not concern polarization (a mix of `asc` and `desc`). Call these *all-ascending* order dependencies. A set of inference rules (proved to be sound and complete) is introduced for pointwise dependencies (simpler than the one defined in [8]), but not for all-ascending order dependencies. A chase procedure is defined for the latter. In [18], we presented an axiomatization for all all-ascending order dependencies, and proved the axiomatization to be sound and complete. An interesting extension of relational algebra to ordered domains is presented in [14]. Our work is the first work on PODs.

Sorting is at the heart of many database operations: for instance sort-merge join, index generation, duplicate elimination and ordering the output through the `SQL order by` operator. The importance of sorted sets for query optimization and processing has been recognized very early on. The query optimizer of System R [15] paid particular attention to *interesting orders* by keeping track of all such ordered sets

[1] For simplicity, we use the arrow $\rightsquigarrow$ for any different type of orders.

throughout the process of query optimization. In more recent work, [9, 10] explored the use of sorted sets for executing nested queries. The importance of sorted sets has prompted the researchers to look *beyond* the sets that have been explicitly generated. Thus, [12] shows how to discover sorted sets created as generated columns[2] via algebraic expressions. For example, if column A is sorted, so is the generated column G defined as $G = A/100 + A - 3$ (that is, $A \rightsquigarrow G$). We show in [17] how to use relationships between sorted attributes discovered by reasoning over the physical schema. The chase presented here provides a formal way of discovering) previously unknown sorted sets. Based on this work, many other optimization techniques from relational query processing can also be adapted.

5 Conclusions

While order of tuples is purposely excluded in the relational model – an answer to a query is a *set* of tuples – *order* plays a vital role in the evaluation of queries and in real-world query languages as SQL with `order by`. (In other data/query models, order is part of semantics, as for XML/XQuery.) We provide a sound and complete chase procedure for the inference problem for PODs. The goal of this work was to develop a theory behind dependencies over polarized lexicographically ordered sets. To the best of our knowledge, this is the first attempt to develop a procedure for testing logical implication for PODs. We explored some correspondence between FDs and PODs. The story of PODs is not over. We plan next to pursue the following.

- We would like to extend our work for all-ascending order dependencies [18] into an axiomatization for polarized (`asc/desc`) order dependencies. Such an axiomatization can provide insight into how PODs behave to devise useful, logically sound rewrites rules for queries. An axiomatization also can provide basis for developing an efficient theorem prover (inference procedure) for PODs.
- Our chase procedure demonstrates the inference procedure is decidable (Theorem 5), but it is not efficient (Theorem 6). We would like an efficient *theorem prover* [1, 11]. Given a set of PODs $\mathcal{M}$ and an arbitrary dependency $\mathbf{X} \mapsto \mathbf{Y}$ we would like to *efficiently* decide whether $\mathcal{M}$ logically implies $\mathbf{X} \mapsto \mathbf{Y}$. Such a theorem prover would be a useful tool for the use of PODs in query optimization. (An axiomatization, as discussed above, could be instrumented for this.)
- Integrity constraints have been widely used in query optimization through *query rewrites*. For example, functional dependencies have been shown to be useful in simplifying queries with `distinct`, `order by`, and `group by` operations [16], whereas inclusion dependencies can be used to remove certain joins over primary and foreign keys [5]. Polarized order dependencies can be used in similar ways to simplify queries with the `order by` operation [18].
- It is possible PODs have a role in *database design* [4]. Functional dependencies are by far the most common integrity constraints in the real world. The notion of the key derived from a given set of FDs is a fundamental to the relational model. The

[2] In DB2, a generated column is a column that can be computed from other columns in the schema.

determination of polarized order dependencies might be an important part of designing databases in the relational model, too. It can be used in database normalization and denormalization. Polarized order dependencies can reveal redundancies that cannot be detected using functional dependencies alone. It would be an interesting research topic to extend the results obtained there to the design of relational databases.

Acknowledgments. We thank Calisto Zuzarte and Wenbin Ma from IBM laboratory in Toronto for their encouragement and helpful suggestions throughout the project.

6 References

1. Armstrong, W.W., 1974, Dependency structures of data base relationships. In *Proceedings of the IFIP Congress*, Stockholm, 580-583, North-Holland
2. Abiteboul, S., Hull, R., Vianu, V., 1995. *Foundations of Databases*. Addison-Wesley Publishing Company, Inc.
3. Aho, A.V., Sagiv, Y., Ullman, J.D., 1979. Equivalence of relational expressions. *SIAM J. Comptng*., 218-246.
4. Bernstein, P., 1976. Synthesizing third normal from relations. *ACM TODS*, 277-298.
5. Cheng, Q., Gryz. J., Koo, F., Leung, T.Y.C, Liu, L., Qian, X., Schiefer, K.B., 1999. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. *VLDB*.
6. Davey, B.A., Priestley, H.A., 2002. *Introduction to Lattices and Order* (2. ed.). Cambridge University Press, 1-298.
7. Dong, J., Hull, R., 1982. Applying Approximate Order Dependency to Reduce Indexing Space. *SIGMOD Conference*, 119-127.
8. Ginsburg, S., Hull, R., 1981.Ordered Attribute Domains in the Relational Model. XP2 Workshop on Relational Database Theory.
9. Graefe, G., 2003. Executing Nested Queries. Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. *BTW-Konferenz*, 58-77.
10. Guravannavar, R., Ramanujam, H.S., Sudarshan, S, 2005. Optimizing Nested Queries with Parameter Sort Orders. *VLDB*, 481-492.
11. Maier, D., Mendelzon A.O., Sagiv, Y., 1979. Testing implication of data dependencies. *ACM Transactions on Database Systems*, 455-469.
12. Malkemus, M., Padmanabhan, S., Bhattacharjee, B., Cranston, L, 2005. Predicate Derivation and Monotonicity Detection in DB2 UDB. *ICDE*, 939-947.
13. Ng, W., 1999. Lexicographically Ordered Functional Dependencies and Their Application to Temporal Relations. *IDEAS* 279-287.
14. Ng, W., 2001. An extension of the relational data model to incorporate ordered domains. *ACM Trans. Database Syst*., 344-383.
15. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G., 1979. Access Path Selection in a Relational Database Management System. *SIGMOD* Conference, 23-34.
16. Simmen, D.E, Shekita, E.J., Malkemus, 1996. Fundamental Techniques for Order Optimization. *SIGMOD*, 57-67.
17. Szlichta, J., Godfrey, P., Gryz, J., Ma, W., Pawluk, P., Zuzarte, C., 2011, Queries on dates: fast yet not blind. *EDBT* 497-502.
18. Szlichta, J., Godfrey, P., Gryz, J., 2012, Fundamentals of Order Dependencies. *VLDB*.
19. Ullman, J.D., 1988. *Principles of Database and Knowledge-Base Systems*, Vol. I, 378-379, Computer Science Press, Rockville, MD, 376-423.