FUNDAMENTALS AND APPLICATIONS OF ORDER DEPENDENCIES

JAROSLAW SZLICHTA

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
JULY 2013

FUNDAMENTALS AND APPLICATIONS OF ORDER DEPENDENCIES

by **Jaroslaw Szlichta**

a dissertation submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

FUNDAMENTALS AND APPLICATIONS OF ORDER DEPENDENCIES

by **Jaroslaw Szlichta**

By virtue of submitting this document electronically, the author certifies that this is a true electronic equivalent of the copy of the dissertation approved by York University for the award of the degree. No alteration of the content has occurred and if there are any minor variations in formatting, they are as a result of the coversion to Adobe Acrobat format (or similar software application).

Examination Committee Members:

1. Professor Aijun An
2. Professor Leopoldo Bertossi
3. Professor Radu Campeanu
4. Professor Nick Cercone
5. Professor Parke Godfrey
6. Professor Jarek Gryz

Abstract

Business-intelligence queries often involve SQL functions and algebraic expressions. There can be clear semantic relationships between a column's values and the values of a function over that column. A common property is *monotonicity*: as the column's values ascend, so do the function's values (or the other column's values). This we call an *order dependency* (OD). Queries can be evaluated more efficiently when the query optimizer uses order dependencies. They can be run even faster when the optimizer can also reason over known ODs to infer new ones.

Order dependencies can be declared as *integrity constraints*, and they can be detected automatically for many types of SQL functions and algebraic expressions. We present optimization techniques using ODs for queries that involve *join*, *order by*, *group by*, *partition by*, and *distinct*. Essentially, ODs can further exploit *interesting orders* to eliminate or simplify potentially expensive sorts in the query plan. We evaluate these techniques over our prototype implementation in IBM® DB2® using the TPC-DS® benchmark schema and some customer inspired queries. Our experimental results demonstrate a

significant performance gain.

Dependencies have played an important role in database theory. We study the theoretical aspects of *order dependencies*—and *unidirectional* order dependencies (UODs), a proper sub-class of ODs—which describe the relationships among *lexicographical* orderings of sets of tuples. We investigate the *inference problem* for order dependencies. We establish the following: (i) a sound and complete axiomatization for UODs which is sound for ODs; (ii) a hierarchy of order dependency classes; (iii) a proof of co-NP-completeness of the inference problem for ODs and for the subclass of UODs; (iv) a proof of co-NP-completeness of the inference problem of functional dependencies (FDs) from ODs in general, but demonstrate linear time complexity for the inference of FDs from UODs; (v) a sound and complete elimination procedure for testing logical implication over ODs; and (vi) a sound and complete *polynomial* inference algorithm for sets of UODs over *natural* domains.

Acknowledgements

First, I would like to express my deepest appreciation to my supervisor, Professor Jarek Gryz and co-supervisor Professor Parke Godfrey. They have provided a constant guidance, advice and patience for my research and thesis. I am grateful to them for their spirit of adventure, continuing interest, encouragement and support.

I would like to thank my committee members, Professor Aijun An, Professor Leopoldo Bertossi, Professor Radu Campeanu and Professor Nick Cercone for their patience in reviewing my thesis and valuable suggestions they made.

I thank IBM Centre for Advanced Studies (CAS) for providing me the opportunity to learn the internals of DB2. In particular, I thank Calisto Zuzarte for his inspiration, foresight and instruction on my thesis. I am grateful to Wenbin Ma for sharing his knowledge about DB2 database and Weinan Qiu for helping with evaluating experiments over TPC-DS benchmark. I thank Joanna Ng for creating a warm and friendly research environment for graduate students at CAS.

Thanks also go to those people who have helped me and inspired me: Professor

Reinhard Pichler, Divesh Srivastava. I really enjoyed the conversations I had with them about my research.

Last, but not the least, I thank Krzysztof, Teresa and Grzegorz, my lovely family, for always being supportive and understanding. Without such a support, this thesis would never have been possible.

Table of Contents

List of Tables

List of Figures

Abbreviations

[BI] Business Intelligence

[BOD] Bidirectional Order Dependency

[CAS] Centre for Advanced Studies

[DW] Data Warehouse

[ETL] Extract, Transform and Load

[FD] Functional Dependency

[OD] Order Dependency

[UOD] Unidirectional Order Dependency

1 Introduction

Understanding the semantics of data is important, both for data quality analysis and knowledge discovery [7, 10]. While the relational data model is *set* based and does not concede the concept of *order*, ordered streams nonetheless play important roles in relational systems. SQL allows one to specify by its order-by clause that the answer "set" be returned in the specified order. Ordered streams are prevalent in query plans to provide efficient evaluation. A query optimizer must reason extensively over *interesting orders* while building efficient query plans [37].

Order for a tuple stream can be semantically specified via the attributes as by SQL's order-by clause. The *order specification* "order by year desc, name asc" requires that the tuple stream be sorted by year in descending order and, within each year group (with the same value for year), by name in ascending order. This is a *lexicographical* ordering, a nested sort. (Note there could be many ways to order the tuples that satisfy this specification: the tuples within any year-name partition may be ordered freely.)

Order dependency (OD) then is the semantic relationship amongst order specifica-

tions. An *order dependency* states a relationship between two order specifications. Say that we knew the OD that id asc *orders* year asc, name asc. Then we would be assured that any tuple stream ordered by id asc would also necessarily be ordered by year asc, name asc. (Note the converse is *not* necessarily assured: if the stream were ordered by year asc, name asc, it still might not be ordered by id asc. This is because the tuples within a given partition of year-name might fail to be ordered by id asc.)

The concept of order dependency is closely related to that of *functional dependency*. Indeed, we shall show that order dependency *subsumes* functional dependency. If id asc *orders* year asc, name asc, then the functional dependency (FD) that id *functionally determines* year and name must hold. ODs convey additional semantic information, of course: that of order. (In fact, any OD is inherently also an FD, but not vice versa.) Furthermore, working with ODs is more complex than working with FDs, because the *sequence* of the attributes in order specifications matters. ODs are specified with respect to *lists* of attributes, whereas FDs are specified with respect to *sets* of attributes.

Order dependency has been studied before with respect to lexicographical orders [34, 39], and with respect to other order definitions (pointwise) [16, 17]. Our focus is on lexicographical orders. In [39], we have established a sound and complete *axiomatization* for lexicographical ODs. The *inference problem* is to answer whether an OD is logically entailed by a set of ODs. While lexicographical order dependency has been studied before, it has not been well understood. (The *complexity* of the inference problem for

(lexicographical) order dependency has not been known. We address this in this work.)

ODs can be declared as an integrity constraint and use to eliminate joins [40]. Even if a database has no declared ODs, OD-optimization techniques are still relevant. Local ODs can arise from algebraic expressions and SQL functions, and can be used for query optimization [41]. Our experimental results in [40] and [41] over the TPC-DS benchmark have shown the usefulness of (lexicographical) ODs for query optimization. Lexicographical ODs are specified with respect to *lists* of attributes, whereas (pointwise) ODs are specified with respect to *sets* of attributes. Working with (lexicographical) ODs as defined in this work is much more useful for query optimization than working with (pointwise) ODs [16, 17], because the *sequence* of the attributes in order specifications (interesting orders) [37] as in the order-by statement matters.

As business-intelligence (BI) applications have become more complex and data volumes grow [26, 45], so have the analytic queries needed to support them. The increasing complexity raises performance issues and numerous challenges for query optimization. Worse, traditional optimization methods often fail to apply when logical subtleties in database schemas and in queries circumvent them. For example, data-warehouse schemas will use surrogate keys, while predicates in analytic queries will use natural values (sale_date = '2010-07-01'). Real world queries will use *SQL functions* (such as year(d_date)) and *algebraic expressions* (such as d_date + 30 days).

These subtleties cause the optimizer to miss opportunities to use indexes, partition

elimination and pipeline operations, and to add potentially expensive operations, such as sort, even when the data is already sorted appropriately. This is because *semantic relationships* between the functions and expressions the queries use and the data in the database—and between data themselves in the schema, as between surrogate and natural keys—are opaque. If these relationships could be discovered and used, more efficient query plans would result.

The relationship on which we focus in this work is *order*. If the rows of a table were ordered by its date column d_date, they would also necessarily be ordered by d_date + 30 days. Indeed, the *function* (over d_date) of d_date + 30 days is *monotonically increasing* with respect to d_date. For this, we say d_date *orders* d_date + 30 days.[1] If an index on d_date could be used to provide results ordered by d_date, then the same index would provide the results ordered by d_date + 30 days, since this is the same order. This semantic relationship of order is a type of *dependency*, and we call it an *order dependency*.

While it will be readily obvious to any reader that d_date orders d_date + 30 days, this observation is not for free for the optimizer. It would need explicit techniques to recognize the dependency. While this particular order dependency rightfully seems trivial, we shall see there are many that are not. Then "when" and "how" to exploit such

[1] In this case, d_date + 30 days *orders* d_date also. We then say the two are *order equivalent*. However, "orders" is not inherently symmetric. Consider year(d_date) and d_date. In this case, d_date orders year(d_date), but year(d_date) does not order d_date.

dependencies in query planning is far from trivial too. This work is about this aspect of query optimization.

Query 1 Plus thirty days.

```
select D.d_date + 30 days,
       max(S.ws_ext_sales_price) as most
from date_dim D, web_sales S
where S.ws_sold_date_sk = D.d_date_sk
and
      D.d_date between
          date('1998-01-01') and
          date('2002-01-01')
group by D.d_date + 30 days
order by D.d_date + 30 days;
```

Consider the SQL query in Query 1 over the TPC-DS[2] schema. In the schema, date_dim is a *dimension* table with the primary key d_date_sk with one row per day. (The attribute d_date_sk is a sequential number.) The table has columns d_date, d_month, d_quarter, and d_day, and additional columns that qualify the day (such as whether it is the weekend, a holiday, and, if so, the name of the holiday). The table web_sales is a large *fact* table recording all individual sales, with ws_sold_date_sk as a foreign key referencing date_dim on d_date_sk.

[2]http://www.tpc.org

Let there be a tree index for date_dim on d_date. The optimizer will miss that the index could be used in evaluating Query 1 to accomplish both the group-by and the order-by. How might the query be rewritten manually to resolve this?

- group by d_date + 30 days and order by d_date:

  This is not legal SQL; the attribute in the order-by is not listed in the group-by (as such).

- group by d_date and order by d_date + 30 days:

  This is accepted by DB2; derived attributes—functions and algebraic expressions derived over the attributes listed in the group-by (which may include derived attributes itself)—can be used in the select and order-by clauses.

  However, this does not resolve the inefficiency. The query plan still explicitly sorts to "satisfy" the order-by.

- group by d_date and order by d_date:

  This does work! The index can now be employed to implement the group-by and to satisfy the order-by.

Of course, it is not the responsibility of the SQL programmer to write queries painstakingly—or of an automated BI report system that generates SQL queries in the back-end—in such a way to assure the optimizer will handle it well. This would violate the declarative principle of SQL. Even if we tried to put the onus on programmers to be careful, they cannot be expected to know what is problematic and what is not. While

a clever SQL programmer can sometimes skirt such pitfalls by careful composition (as here), more often it is not possible. So, we have to fix it. The optimizer needs to recognize that d_date and d_date + 30 days are semantically equivalent for order, thus skipping the superfluous sorting step, regardless of how the query is written.

Query 2 Eliminating trimester and quarter.

```
select D.year, D.trimester, D.quarter,
       D.month, D.day, sum(S.sales) as total
       count(*) as quantity
from date_dim D, sales S
where S.date_id = D.date_id and
      D.year between 2001 and 2004
      and sum(S.sales) > 10000
group by D.year, D.trimester, D.quarter,
         D.month, D.day
order by D.year, D.trimester, D.quarter,
         D.month, D.day;
```

Next consider the SQL query in Query 2 over a data warehouse schema as in [24]. The fact table sales has a foreign key S.date_id which references the dimension table date_dim. *Date* is captured in a hierarchical manner by attributes year, quarter or trimester, month, and day. The values of the attribute quarter divide year into four three-month periods, while those of trimester divide it into three four-month periods.

Let there be a B+ tree index for date_dim on year, month, day. The query optimizer may not employ this index to evaluate either the group-by or the order-by for the query in Query 2, because their specifications do not *match* the index's search key.

Of course, it is clear that month *functionally* determines quarter and trimester. So partitioning by year, trimester, quarter, month, day is the same as just by year, month, day. In fact, optimizers today would eliminate trimester and quarter from the group-by via reasoning over the relevant FDs [37], and then employ the index for the group-by operation.[3] Note that primary and unique keys are usually declared; this provides much FD information to the optimizer. If the schema is normalized, most FDs will have been thus captured. This is exploited in [37] for optimization.

The FD month $\rightarrow$ quarter, trimester is not logically sufficient to optimize the order-by operation, however. One would need the additional semantic information of an OD that year, month, day *orders* year, trimester, quarter, month, day. This and similar subtleties cause the optimizer to miss opportunities to use indexes and to pipeline operations. Expensive operations as sort are added to a query plan, even when the data is already sorted properly. By incorporating reasoning over ODs into the optimizer—as has already been done for reasoning over FDs [37]—many new optimizations would be possible [40, 41]. The ordered stream by year, month, day then could satisfy *both* the group-by and the order-by operations *on-the-fly*.

[3] IBM DB2 incorporates such rewrites.

Our axioms for ODs help us explore beneficial query rewrites. We show how ODs can be cast as a new type of integrity constraint to be used in query optimization. We derive theorems based on our axioms, which illustrate surprising inferences and equivalences over ODs, and which can provide for powerful query rewrites. Working with ODs is more involved than with FDs because the order of the attributes matters. Thus, we must work with lists of attributes instead of with sets. This necessarily complicates our axioms—compared with Armstrong's axioms for FDs—and the proofs of our theorems.

In our work we pay special attention to *time dimension*. (However, we do not limit ourselves to time and consider other dimensions.) The time dimension is a significant aspect of data warehouses. Data warehouses are often designed to assist in analysis of business data over a historical period [24]. Tables often include attributes which refer to time. This makes it possible for business analysts to detect the existence of temporal patterns. For example, keeping the date of a sale enables analysis over different quarters or months of the year, and allows for the comparison of corresponding quarters or months over various years.

The amount of historical data grows quickly. For example, consider an organization in the telecommunications industry tracking phone calls over different cities and countries. Integration of data is a complex extract-load-transform (ETL) process that uses multiple sources [35]. In order to make the storage of data efficient for analysis, the data is often aggregated in a warehouse. It is important to balance the degree of aggregation

to support various types of drill down and roll up queries. OLAP functions such as sum, count, max, min and avg support the process of granulating data. As a warehouse continually grows, it is important to design scalability from the very beginning, so query performance is not sacrificed. We present how the time dimension is modeled in the data warehouse, and describe optimization techniques of queries which involve the time dimension.

The outline of the thesis is as follows.

In Chapter 2 (*Fundamentals*), we introduce a theoretical framework for ODs. In Chapter 2.2, we introduce *unidirectional order dependencies*. We present a common multi-dimensional model in Chapter 2.4. We demonstrate how FDs can be used effectively in optimization in Chapter 2.3. This includes putting ODs into a canonical form to enable reasoning over ODs in the query optimizer. We discuss how ODs arise in Chapter 2.5.

The contributions of this thesis described in Chapters 3 through 4 are as follows.

1. *Optimizing with Order Dependencies.*
   (a) In Chapter 3.1.1, we go into further depth how ODs are used to optimize queries.
   (b) In Chapter 3.1.2 we present two algorithms (Reduce Order OD and Homogenize Order OD) and show where and how they are invoked in the optimizer. *Reduce Order OD* puts ODs into a canonical form for matching against *inter-*

esting orders. Homogenize Order OD discovers equivalent columns, order-wise. We discuss the utility of these algorithms.

2. *Detecting Order Dependencies.* In Chapter 3.2, we show how ODs between columns and functions over columns (SQL functions and algebraic expressions) can be automatically detected by the optimizer [41].
   (a) These techniques have been implemented within DB2.
   (b) We present a suite of real-world IBM customer queries over the TPC-DS benchmark that illustrate the issues, which are then used in Chapter 3.5 for an experimental performance evaluation. The optimizer automatically infers the associated OD information and uses it to produce the improved query plans.
3. *Declaring Order Dependencies.* In Chapter 3.4, we consider how OD information can be declared, and what types of natural ODs occur in today's schemas [40].
   (a) Order dependencies can be explicitly declared in our prototype implementation in DB2 as a type of integrity constraint.
   (b) We demonstrate how ODs between surrogate and natural keys can be used for strong performance improvement.
4. *Axiomatization for UODs.*

   A sound and complete axiomatization for UODs [39], analogous to Armstrong's axiomatization for FDs [2]. This provides a formal framework for reasoning about

ODs. There are two reasons for one to pursue an axiomatization:

(a) The axioms provide insight into how dependencies behave – and patterns for how dependencies logically follow from others – that are not easily evident reasoning from first principles.

(b) A *sound* and *complete* axiomatization is the first necessary step to designing an efficient *inference procedure*.

5. A hierarchy of OD classes (Chapter 4.2) [43].

   (a) Lexicographical ODs as defined in this thesis are a proper sub-class of point-wise ODs. (Chapter 4.2.2) The latter is defined in [16].

   (b) UODs form a proper sub-class of ODs. (Chapter 4.2.3)

   (c) FDs form a proper sub-class of UODs. (Chapter 4.2.4)

6. *Decidability of the inference problem for ODs.*

   Decidability follows from that our class of ODs is a (proper) subclass of pointwise ODs (Chapter 4.2.2), for which a sound and complete axiomatization has been established [16], and also from the sound and complete *elimination procedure* for ODs. (Chapter 4.4.1)

7. *Complexity.* (Chapter 4.3.) [42]

   (a) The inference problem for UODs is co-NP-complete. (The inference problem for OD is co-NP-complete too.) (Chapter 4.3.1)

   (b) The inference problem of inferring FDs from ODs is also co-NP-complete,

but that it is only linear for the case of FDs over UODs. (Chapter 4.3.2)

8. *Inference Procedures.* (Chapter 4.4.)

   (a) A sound and complete *elimination procedure* for inference over ODs, for which the complexity is exponential [42]. This complexity is with respect to schema (number of unique attribute in the set of prescribed ODs over relation), not with respect to data. Therefore, it can be used in practice. We have implemented elimination procedure in IBM DB2. (Chapter 4.4.1)

   (b) We present in Chapter 4.4.2 *chase procedure* for testing logical implication for ODs with exponential complexity [38]. This complexity is with respect to the size of relation **R**. (This is an alternative approach to elimination procedure.)

   (c) A restricted, *natural* domain, the *order-compatible transitive domain*, which makes reasoning over ODs simpler [42].[4] (Chapter 4.4.3)

   (d) An efficient, polynomial time inference procedure for testing implication of ODs over the transitive domain that is sound and complete. We have implemented this procedure in IBM DB2. (Chapter 4.4.4)

In Chapter 6, we discuss related work. In Chapter 5, we conclude and consider future work.

[4]A domain is *restricted* if an additional order property is guaranteeed over the schema. The *order-compatible transitive domain* is quite natural in that it is intuitive, it holds for all real-world business domains that we have encountered, and it can easily be verified whether it holds.

This work, we believe, opens exciting venues for future work to develop a powerful new family of query optimization techniques in database systems.

2 Fundamentals

First, we establish notational conventions and definitions for ODs and UODs. We demonstrate how ODs, similarly as FDs [37], can be used effectively in optimization. Next we describe a common multi-dimensional model based on a schema with fact and dimension tables. Finally, we discuss how ODs arise in the multi-dimensional model.

2.1 Framework

We adopt the notational conventions specified in Table 2.1. We consider a relation **R** with a schema set of attributes $\mathcal{U}$. Let **r** be an arbitrary table instance over **R**; thus a set of tuples under **R**'s schema with attributes $\mathcal{U}$. We limit table instances to sets in our definitions, to keep our definitions simpler and easier to follow. However, this could be changed to *multi-sets* easily, with no consequences to our axiomatization and inference problem results.

Whenever it is necessary we assume a column (#) in **R** that takes a unique value per tuple, without loss of generality. This alleviates any need to work with a table instance as

Table 2.1: Notational conventions.

- **Relations.**
  - A capital letter in bold italics represents a *relation*: $\mathbf{R}$.
  - A small letter in bold: $\mathbf{r}$ denotes a specific *relation instance* (*a table*).
  - We use capital letters to represent *single attributes*: A, B and C.
  - Additionally, s and t denote *tuples* and t_A denotes the value of attribute A in tuple t.
- **Sets.**
  - Calligraphic letters $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ denote *sets*.
  - $t_{\mathcal{X}}$ denotes the *projection* of tuple t on $\mathcal{X}$.
  - $\mathcal{XY}$ is shorthand for $\mathcal{X} \cup \mathcal{Y}$.
- **Lists**
  - **X**, **Y** and **Z** denote *lists*. (Note **X** could represent the empty list, [].)
  - List [A, B, C] denotes an explicit list.
  - [A | **T**] denotes a list with *head* A and *tail* **T**, the remaining list with the first element removed.
  - **XY** is shorthand for **X** ∘ **Y** (**X** concatenate **Y**).
  - Set $\mathcal{X}$ denotes the *set* of elements in *list* **X**. Anyplace a set is expected but a list appears, the list is *cast* to a set; e.g., $t_{\mathbf{X}}$ denotes $t_{\mathcal{X}}$.
  - $\mathbf{X}'$ denotes some other permutation of elements of list **X**.

bags of tuples; we consider them as sets. It removes the possibility that we might "lose" a row from **r** when we modify the value of one of its columns (beside #), since # will still distinguish it from other tuples.

We model *order specification* as provided by SQL's order-by clause for specifying lexicographical orderings.

Definition 1 (order specification)

An order specification *is a list of* directionality-marked attributes *(or* marked *attributes, for short).*

There are two directionality operators: asc *and* desc, *indicating* ascending *and* descending, *respectively. Each operator is unary, applies over an attribute, and is written postfix; e.g.,* A asc *and* B desc. *As shorthand notation, we write* $\overrightarrow{\mathsf{A}}$ *and* $\overleftarrow{\mathsf{A}}$ *for* A asc *and* A desc, *respectively.*

In any context an order specification *is expected but a list of (unmarked) attributes appears, the list is cast to the order specification with each attribute marked as* asc; *e.g.,* $[\mathsf{A}, \mathsf{B}, \mathsf{C}]$ *is cast to* $[\overrightarrow{\mathsf{A}}, \overrightarrow{\mathsf{B}}, \overrightarrow{\mathsf{C}}]$.[5]

The order specification **X** defines an algebraic relation '$\preceq_{\mathbf{X}}$'. The operator '$\preceq_{\mathbf{X}}$' defines a *weak total order* over any set of tuples.

[5] *Ascending* is the default for SQL in order-by for any attributes for which directionality is not explicitly indicated.

Definition 2 (algebraic relation '$\preceq_{\mathbf{X}}$')

Let $\mathbf{X}$ *be a list of marked attributes. For two tuples* r *and* s *(over a schema containing the attributes in* $\mathbf{X}$*),* $r \preceq_{\mathbf{X}} s$ iff

- $\mathbf{X} = [\overrightarrow{A} \mid \mathbf{T}]$ *and* $r_A < s_A$*; or*
- $\mathbf{X} = [\overleftarrow{A} \mid \mathbf{T}]$ *and* $r_A > s_A$*; or*
- $\mathbf{X} = [\overrightarrow{A} \mid \mathbf{T}]$ *or* $\mathbf{X} = [\overleftarrow{A} \mid \mathbf{T}]$*,* $r_A = s_A$*, and* $r \preceq_{\mathbf{T}} s$*;* or
- $\mathbf{X} = []$.

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

We now define *order dependencies*.

Definition 3 (order dependency)

Let $\mathbf{X}$ *and* $\mathbf{Y}$ *be lists of marked attributes.* $\mathbf{X} \mapsto \mathbf{Y}$ *denotes an* order dependency *(OD), read as* $\mathbf{X}$ orders $\mathbf{Y}$*. Let* $\mathbf{R}$ *be a relation (over a schema that contains the attributes that appear in* $\mathbf{X}$ *and* $\mathbf{Y}$*), and let* $\mathbf{r}$ *be a relation instance of* $\mathbf{R}$*. Table* $\mathbf{r}$ satisfies $\mathbf{X} \mapsto \mathbf{Y}$ *(*$\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$*)* iff*, for all* $s, t \in \mathbf{r}$*,* $r \preceq_{\mathbf{X}} s$ *implies* $r \preceq_{\mathbf{Y}} s$*. The OD* $\mathbf{X} \mapsto \mathbf{Y}$ *is said to* hold *for* $\mathbf{R}$ *(*$\mathbf{R} \models \mathbf{X} \mapsto \mathbf{Y}$*)* iff*, for each admissible relational instance* $\mathbf{r}$ *of* $\mathbf{R}$*, table* $\mathbf{r}$ *satisfies* $\mathbf{X} \mapsto \mathbf{Y}$*. A dependency* $\mathbf{X} \mapsto \mathbf{Y}$ *is* trivial iff*, for all* $\mathbf{r}$*,* $\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$*. (This is written as* $\models \mathbf{X} \mapsto \mathbf{Y}$*.) We write* $\mathbf{X} \leftrightarrow \mathbf{Y}$*, read as* $\mathbf{X}$ *and* $\mathbf{Y}$ *are* order equivalent, iff $\mathbf{X}$ orders $\mathbf{Y}$ *and* $\mathbf{Y}$ orders $\mathbf{X}$.

Table 2.2: Relational instance **r**.

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

Example 1

Let **r** *be a relation instance over* **R** *with attributes* A, B, C, D, *and* E, *as shown in Table 2.2. Note* $\mathbf{r} \models [\overrightarrow{A}, \overrightarrow{C}, \overrightarrow{D}] \mapsto [\overrightarrow{E}, \overrightarrow{B}]$, *but* $\mathbf{r} \not\models [\overrightarrow{A}, \overrightarrow{C}, \overrightarrow{D}] \mapsto [\overrightarrow{B}, \overrightarrow{E}]$. *Also note* $\mathbf{r} \models [\overleftarrow{C}, \overleftarrow{A}] \mapsto [\overrightarrow{B}, \overleftarrow{D}, \overleftarrow{E}]$, *but* $\mathbf{r} \not\models [\overleftarrow{C}, \overleftarrow{A}] \mapsto [\overrightarrow{E}, \overrightarrow{B}, \overleftarrow{D}]$. *Furthermore,* $\mathbf{r} \models [\overrightarrow{C}] \mapsto [\overrightarrow{D}]$, *but* $\mathbf{r} \not\models [\overrightarrow{C}] \mapsto [\overrightarrow{E}]$

Order dependencies can be prescriptive statements on the relation, as can functional dependencies. That is, they can be used as a type of integrity constraint to prescribe which instances are admissible.

There is a strong relationship between ODs and FDs. Any OD implies an FD, modulo lists and sets, but not vice versa. In fact, this remains true when we limit to UODs.

Lemma 1 *(relationship between ODs and FDs)*

For every instance **r** *of relation* **R**, *if a UOD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$ *holds, then the FD* $\mathcal{X} \rightarrow \mathcal{Y}$ *is true.*

Proof

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

We introduce one additional order relation, *order compatibility*.

Definition 4 (order compatible)

Order specifications $\boldsymbol{X}$ *and* $\boldsymbol{Y}$ *are* order compatible *(independently from an instance), denoted as* $\boldsymbol{X} \sim \boldsymbol{Y}$, iff $\boldsymbol{XY} \leftrightarrow \boldsymbol{YX}$.

The empty order specification [], is order compatible with *any* order specification. Interestingly, order compatibility does not add expressiveness over order dependencies as already introduced. Indeed, we can define it directly as an OD of a specific form. Because the concept proves invaluable for our theoretical framework described in Section 4, however, for reasoning about ODs, we introduce it explicitly (Definition 4) for this purpose.

$\mathbf{X}$ and $\mathbf{Y}$ may be order compatible without either $\mathbf{X} \mapsto \mathbf{Y}$. At first glance, this might seem surprising. A degenerate case as we show in demonstrates this quickly, however.

2.2 Unidirectional ODs

We accommodate bidirectionality (asc and desc) in order specifications for generality's sake, and because SQL's order-by clause does. Of course, marking attributes adds complication. It is reasonable to ask whether this bidirectionality adds expressiveness and if so, whether the added expressiveness is useful.

One can consider a simplified version of ODs for which we remove this bidirectionality. Call a set of ODs *unidirectional* in which any given attribute appears in the ODs

either marked as all asc or all as desc, but not both. Without loss of generality, one can consider just sets of ODs in which all attribute occurrences are marked as asc. Call an individual OD in which all attributes are marked as asc a *unidirectional* order dependency (UOD). This restriction to just ascending has the advantage that one can verify that a set of ODs is unidirectional by verifying in isolation that each OD is unidirectional.

Definition 5 (unidirectional order dependency)

An order dependency is unidirectional *when all attributes within it are marked* asc. *In contrast, call an order dependency which has both attributes marked as* asc *and as* desc *a* bidirectional *order dependency (BOD).*

UODs are a sub-class of ODs, by definition.

2.3 Functional Dependencies

That attribute A *functionally determines* attribute B (with respect to a relational instance **r**) means that knowing values are the same for two rows of A *tells* one the values for the same two rows for Balso has to be the same. That is, if A has the value V in tuple $s \in \mathbf{r}$ and B has the value W in s, then for any other tuple $t \in \mathbf{r}$ in which A has value V, B again has value W.

This generalizes to sets of attributes easily. A set of attributes $\mathcal{X}$ functionally determines a set of attributes $\mathcal{Y}$ (with respect to a relational instance **r**) if any value for

$\mathcal{X}$—that is, given values for each attribute in $\mathcal{X}$—is associated with a single value for each attribute of $\mathcal{Y}$ in **r**.

We write $\mathcal{X} \rightarrow \mathcal{Y}$ to denote $\mathcal{X}$ functionally determines $\mathcal{Y}$, and we refer to $\mathcal{X} \rightarrow \mathcal{Y}$ as a *functional dependency* (FD). An FD can be prescriptive; that is, we posit it as an integrity constraint. Then, only instances **r** of **R** for which the FD is valid are permitted.

This very simple notion, of course, came to have profound importance in databases, especially in schema design. Primary keys are functional dependencies. In design, reasoning over functional dependencies is needed. Given a set of FDs prescribed on **R**– that is, guaranteed to be true with respect to any admissible instance **r** – will another given FD necessarily be *true*?

While functional dependency is a very simple notion, reasoning over them (inference procedure) is, somewhat surprisingly, not nearly as simple [3, 22]. To gain insight into how sets of functional dependencies behave, and to simplify the reasoning process over them, Armstrong provided an axiomatization for them [2].

Armstrong proved *soundness* of his axioms, and his axiomatization has been proved *complete* for *logical* inference over functional dependencies. Armstrong's axioms simplified reasoning over FDs, which allowed for the evolution of database design theory and normalization.

Beyond layout and indexes, FDs play additional important roles in query optimization. (This is exploited in [13].) Knowledge about prescribed FDs on the schema are

1. *Reflexivity.*

$$\frac{\mathcal{X} \subseteq \mathcal{Y}}{\mathcal{X} \rightarrow \mathcal{Y}}$$

2. *Augmentation.*

$$\frac{\mathcal{X} \rightarrow \mathcal{Y}}{\mathcal{X}\mathcal{Z} \rightarrow \mathcal{Y}\mathcal{Z}}$$

2. *Transitivity.*

$$\frac{\mathcal{X} \rightarrow \mathcal{Y} \quad \mathcal{Y} \rightarrow \mathcal{Z}}{\mathcal{X} \rightarrow \mathcal{Z}}$$

Figure 2.1: Axioms for FDs.

used in the query-rewrite phase of optimization potentially to eliminate predicates. They are used in the cost-based phase to do better cardinality estimation. They are also used to recognize partitioning equivalences of result streams within query plans.

Functional dependency specifically aligns with the notion of *partitioning*, which manifests itself explicitly in SQL in group by. Lastly, FDs are pertinent to people when writing queries, as well. One must be familiar with the *semantics* of the database to be able to write queries that mean what is intended.

2.4 Multidimensional Model

The TPC-DS benchmark is for *decision-support*, and its schema typifies that of data warehouses (DWs) designed to aid analysis of business over a historical period.

The schema is a common multi-dimensional model based on a schema with fact and dimension tables. (See Appendix A for the details.) Fact tables will have many rows capturing measures or events over time [28], such as sales. Dimension tables model the entities such as the customers and products. Date is often made an explicit dimension table, because the designers need to keep specific data about given dates. Date dimension tables, apart from storing the date as a column, can also keep descriptions of the date which can be filtering fields and labels for business reporting. Columns in the date table can include, for example, day of week, the day number in a calendar month, the month number in year, month name, and fiscal periods. We can similarly include a holiday indicator, weekday indicator, the name of a holiday (Easter, Thanksgiving, Valentines Day), the name of special events (Back to School, Super Bowl), and so forth. In the TPC-DS schema, the date dimension has a granularity of day. (See Figure 2.3.)

We distinguish date and time dimensions. A date dimension refers to a table that is granulated by day, whereas a time dimension represents the time of day. Such a dimension is practically universal as it appears in any data warehouse that is a historical repository of data [24]. It is often recommended to separate the time of the day from

the date dimension in order to keep the date dimension small. As the time description is not usually required in a data warehouse, it is a good practice to keep the time attribute out of the date table in the fact table. Time of day might be represented as the number of milliseconds, seconds or minutes since midnight. For optimization reasons, the granularity of the date in the data dimension table is sometimes even further aggregated to the level of week or month. It is also a common practice to keep the data from previous years aggregated with higher granularity. The time dimension is traditionally used for tracking changes over measures. A model which allows a conceptual representation of time-varying levels, attributes and hierarchies is described in [30]. In that model, the time dimension can be used additionally to track changes in the other dimensions. In our work, we focus on the time dimension as the entry point (by join) to the fact table.

Data kept about the entities is factored out into the dimension tables (and out of the fact table) based on good principles of design (*normalization*), but also for practical reasons. Because the fact table will be very large row-wise, it is important to keep the size of rows small. A measure is taken at the intersection of the fact table and dimensions such as date, item or location. All measurement rows in the fact table are at the same granularity. Fact measures are usually numeric and additive. Dimension tables are a fundamental part of a data warehouse that contain the description of the data [9]. The dimension tables are necessary to understand the data in the fact tables. The fact tables can have a very large number of records, but are compact in terms of the number of

columns. It is typical that the fact tables take about 90% of the entire space of the data warehouse. Conversely, the dimension tables are shallow in their number of records, but have many columns for the descriptive attributes.

Designing date dimensions is a challenge. There are two main methods to represent the date in a data warehouse. The first approach is to keep the date dimension in the fact table, as shown in Figure 2.2. If the date attribute is explicitly in a fact table, we can make a direct SQL query involving date that will not necessitate a join. Filtering based on date can avoid a join with a date dimension table (which could be quite expensive), so that the query is evaluated solely over the fact table. The second technique is to create a separate date dimension table. There are good reasons why the second method is used more often. SQL date functions do not assist in filtering based on weekdays, weekends, holidays, major events and fiscal periods. Since there are no such built-in functions, it is better to store these as data in a dimensional table. Also, most database systems do not support index calculations using functions (e.g., month, day).

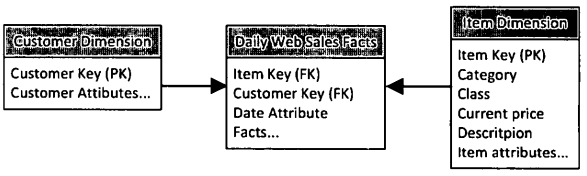

Figure 2.2: Date in fact table.

Designers sometimes replace a date dimension table by representing time via buckets

in the fact table [24]. This solution is not commonly used though, because it is not flexible. With a predefined number of buckets which represent month one, month two, and so on, at some point the table has to be altered in order to add a bucket for a new month, or to shift all the buckets. This may not be the best choice as the month first on the list will be lost. A second disadvantage of this approach is that it is not possible to keep the description of date as specified in Chapter 2.2 so there is no way to get information regarding what the date refers to.

A common design question for DWs is whether to use *surrogate* keys [24]. Surrogate keys are unique identifiers usually generated as sequential integers. A join between the fact table and a dimension table is based on surrogate keys if we use surrogate keys for dimension tables. It is considered good practice to use surrogate keys instead of operational keys (derived from external names such as production codes) which may have built-in dependencies. Avoiding using an operational key as the primary key of the table is a good idea because our expectations might be invalidated over the time. In business organizations, operational codes such as product codes are reassigned after some period of time. Surrogate keys offer the data warehouse a mechanism to distinguish between two different instances of the same product with different operational codes. Developing the data warehouse with operational keys might be faster, but implementing surrogate keys can bring benefits in the long run. The main advantage of surrogate keys is that they enable keeping track of changes independently from the key. Even if the operational

key changes in the next ETL cycle, it can still stay in the data warehouse with the same surrogate key. Also, extracting data from multiple sources may be easier using surrogate keys as they allow integrating data from multiple source systems even if they do not have well-matched source keys. Using INTEGER (four bytes) for a surrogate key is adequate for a dimension table as it provides over two billion possible values (2^{32-1}).

SQL's *date* data type would be a good *natural* key in the date table. However, it used to be common in database systems—and still is in some—for the *date* data type to take eight bytes. (Note that DB2 uses a compact four byte DATE: two for year, one for month and one for the day.) Given a fact table with a billion rows, each additional byte per row is a gigabyte of storage. And this storage is not necessarily inexpensive. To achieve high performance, many providers of data warehouse systems require expensive, proprietary hardware. Because of the cost of the disk, which must be efficient and reliable, database schema designers must pay attention to saving space. More importantly, most queries scan rows from the fact table. By making the fact table more compact, this reduces the I/O expense of evaluating these queries. (Even though there is no space savings in DB2 as there might be in other database systems, designers often still use integer surrogate keys to be consistent with other dimension surrogate keys.) So instead, a four-byte integer could be used as a surrogate key in the date table; then, the fact table's foreign-key field referencing table date is smaller. This is the design choice in TPC-DS's schema.

Furthermore, the date dimension table can be populated at the time the DW is created.

It does not undergo regular updates. Its surrogate key can be generated via increment, in the order of the date values. Therefore, there will be an order equivalence between the surrogate key and the date's day.

While the use of the surrogate key helps reduce the size of the fact table, it does introduce costs at query time. Queries will often have predicates involving (natural) date values to access data from the fact table. This necessitates a potentially expensive join between the fact and date tables.

Time dimension is often a central component of a data warehouse. Our observations with customer queries at IBM have shown that almost all queries involve time attributes. Therefore, being able to optimize queries with respect to data warehouse's time dimensions could offer large returns. We observe it to be quite common that the selectivity (filter factor) for the time predicates in queries to be the greatest filtering of all the predicates. This means the date dimension table is often joined first with the fact table in the query plan. We show that this *blind* join can be replaced with a pair of *fast probes*.

Large fact tables are often partitioned to speed up evaluation and for easier roll-in and roll-out of data. The date surrogate key which is a date sequence number enables physical partitioning of the fact table on the date key (foreign key from the date dimension). Partitioning the fact table on the date key is well-kept throughout the changes as date is normally unchanged. We have observed cases when the optimizer cannot exploit partition elimination in order to reduce I/O. The entire fact table may need to be scanned

rather than just the relevant partitions for the query. We need to ensure that the optimizer can take advantage of the partitioning. We describe this problem and offer a solution in Chapter 3.4.

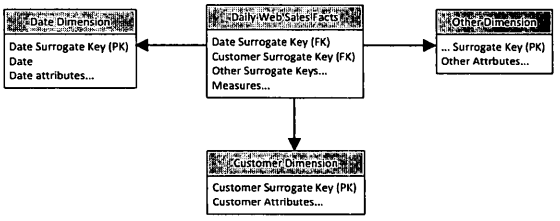

Figure 2.3: Standard TPC-DS schema.

IBM recommends to its business customers to use a natural key for the date table (using the *date* data type). IBM DB2 manages to store the *date* data type in a compact four bytes. The advantages of using a surrogate key in this case are nullified. For this reason, we consider also a variation of the TPC-DS schema as in Figure 2.4 which uses the natural date key in the date table and in the fact table for the foreign key.

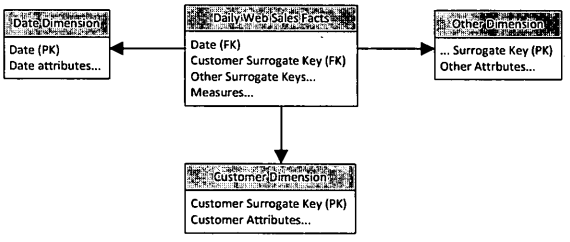

Figure 2.4: Alternative TPC-DS schema with natural date key.

For our performance study (Chapter 3.5) of the technique described in Chapter 3.2,

Table 2.3: An instance of the table date_dim.

date_id	date	year	month	day	quarter	trimester
8300	20100830	2010	08	30	3	2
8301	20100931	2010	09	31	3	3
8302	20110105	2011	01	05	1	1
8303	20110106	2011	01	06	1	1
8304	20110401	2011	04	01	2	1

we test six queries under the unmodified TPC-DS schema (Figure 2.3), and three queries under the alternative schema (Figure 2.4). The queries are motivated and presented in the next chapter. The two schemas allow us to illustrate different optimizations using ODs that can be accomplished by detecting monotonicity. It also shows that we achieve good performance improvements in both, so our technique do not require specific schema designs.

2.5 Ordered Domains

Example 2 and Table 2.3 refer to the *date* domain described in Chapter 1 and Chapter 2.4.

Example 2 (ODs over an instance of the date_dim.)

Order dependencies

$[\text{year}, \text{month}, \text{day}] \mapsto [\text{date}]$ *and*

$[\text{date\_id}] \mapsto [\text{year}, \text{month}, \text{day}]$

are satisfied in Table 2.3. However, order dependencies

$[\text{date\_id}] \mapsto [\text{year}, \text{day}, \text{month}]$ *and*

$[\text{year}, \text{month}] \mapsto [\text{date}]$

are falsified *by Table 2.3.*

While order is not part of the relational model (Chapter 3.1), per se, ordered value domains are of key importance for most databases, and most queries. Many types of ODs are apparent in the semantics of databases (even though these ODs are not declared explicitly). Perhaps the most important of these ordered domains in practice is time. *Time* and *date* (time at a coarser granularity) are supported in the SQL standard in a rich manner. The popular TPC-DS benchmark consists of 99 queries. Of these, 85 involve date operators and predicates and five involve time operators and predicates. Even if the concept of ODs was only applied to date and time, it could still be of great use for query optimization, as shown in Query 2. However, ordered domains are not only limited to date and time. They arise in many other domains from business semantics, such as sequence numbers, surrogate keys, measured values such as sales, stock prices and taxes (Example 3).

Figure 2.5 represents possible ODs, in which the left-hand side of a dependency is *time* and the right-hand side is one of the paths through the diagram. Each node is an equivalent class of the list of attributes leading up to it, with respect to the starting point. Theorem 12 proves that any list appearing on the left side can be suffixed by attributes appearing along an equivalent path. This is shown in Example 29.

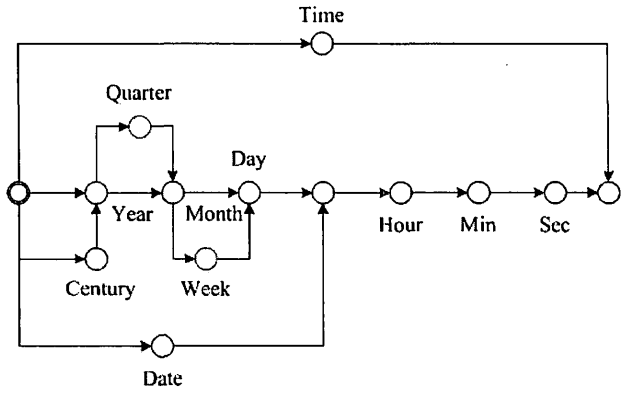

Figure 2.5: Time diagram.

As discussed earlier, order dependencies are not limited to date and time. They commonly arise in many other domains.

Example 3 (Taxes) *Consider table* taxes *in Table 2.4, which has columns for the taxable* salary, *tax* group, *tax* subgroup, taxes *on the salary, and the tax's* percent *of the salary. The tax groups are based on the level of salary and, therefore, increase with the salary. (The tax subgroup increases for the same group as the salary goes up, but oscillates within a group.) Assume that the taxes go up with income and are calculated by as a percentage. Thus, we can declare*

[salary] $\mapsto$ [taxes],

[salary] $\mapsto$ [percent], *and*

[salary] $\mapsto$ [group, subgroup].

It logically follows from these ODs that

Table 2.4: Table taxes.

id	salary	percent	taxes	group	subgroup
100	5000	19%	950	A	II
101	6000	19%	1140	A	III
102	3000	19%	570	A	I
103	20000	30%	6000	B	I
104	50000	40%	20000	C	I

[salary] $\mapsto$ [taxes, percent, group, subgroup].

This OD was derived automatically using our inference procedure for ODs described below.

Let the table taxes in Table 2.4 have a clustered index on salary. A query with order by taxes, percentage, group, subgroup given the three ODs as declared in Example 3 could then be evaluated using the index on salary, as the inference procedure could infer that

[salary] $\mapsto$ [taxes, percent, group, subgroup].

Obviously, the database administrator could have declared that OD too; but that is unlikely.

In Chapter 3.4, we had assumed that [date_sk] $\mapsto$ [date]was declared. Instead, however, we may have had the following ODs:

[date_sk] $\mapsto$ [year, month, day], and

[year, month, day] $\mapsto$ [d_date].

From these, [date_sk] $\mapsto$ [date] can be concluded.

The optimizer needs the means to discover ODs that logically follow from known ODs to benefit most from our techniques. We present an inference procedure as a formal means to do this.

We are interested in ODs because asc and desc bidirectional lexicographical orders are part of SQL standard. Consider a modification of Query 2 with order by year asc, quarter asc, month asc, day asc, sum(S.sales) desc. A query plan could then also eliminate quarter from the order-by clause as

$$[\overrightarrow{\text{year}}, \overrightarrow{\text{quarter}}, \overrightarrow{\text{month}}, \overrightarrow{\text{day}}, \overleftarrow{\text{sum(sales)}}] \mapsto$$

$$[\overrightarrow{\text{year}}, \overrightarrow{\text{month}}, \overrightarrow{\text{day}}, \overleftarrow{\text{sum(sales)}}]$$

is satisfied.

3 Business-Intelligence Queries

In Chapter 3.1, we discuss how current query optimizers can be extended to use ODs. In Chapter 3.1.1, we delve more in depth into how *order* is presently used in optimization, and how ODs extend the effectiveness of this. In Chapter 3.1.2 we present two algorithms, *Reduce Order OD* and *Homogenize Order OD*, that extend broadly two existing algorithms in DB2 for matching interesting orders [37] further to accommodate ODs. Of course, for these techniques to add benefit, ODs must exist and be known. ODs can arise from three sources: they can be detected in the context of the query, they may be declared for the database as integrity constraints (as SQL check constraint); and they may be inferred from known ODs.

Local order dependencies can arise within a query's scope, due to the query's semantics and constructs. For instance, if there is a predicate A = B in the where clause, then clearly the OD [A] $\leftrightarrow$ [B] is satisfied in the query's scope but is not necessarily satisfied generally in the database). (Local order dependencies can be also derived for views in this sense.) Local ODs also arise through a query's *derived attributes* via SQL func-

tions and algebraic expressions, as motivated in Chapter 1. The optimizer must *detect* automatically local ODs to use them. In Chapter 3.2, we demonstrate the types of local ODs we have instrumented DB2 to detect and use, and we illustrate these with customer-motivated, real-world queries over the TPC-DS schema. (We generalize it in Algorithm 1.) These queries are used in our performance study, presented in Chapter 3.5.

In Chapter 3.3, we consider how to embed information of the natural date value into the date surrogate key. This preserves most of the benefits of using surrogate keys, and it lets us rewrite the query for optimization. There are certain disadvantages to using embedded surrogate keys, however, so this does not offer a general solution. We introduce universal solution that considers ODs *declared* on a database as *integrity constraints*. In Chapter 3.4, we address this.

Even with the ODs declared for the database and the local ODs deduced within the scope of the query, the optimizer might miss opportunities. There may be an OD that logically follows from the declared and local ODs that would allow for a better plan, while none of the declared or local ODs match directly. For instance, again assume there is a predicate $A = B$ in the where clause. If we also knew the declared OD $[A] \mapsto [Z]$, within the query's scope, OD $[B] \mapsto [Z]$ is also satisfied (by *transitivity* of ODs [39]). Therefore, the optimizer has a need to *infer* ODs from others. In Chapter 4.4, we show when and where the optimizer would invoke OD-inference procedures (two of which were presented in Chapter 3.1.2 as algorithm, for canonicalization), and we develop such

procedures. We have demonstrated that dramatic gain in query performance can be had in business-intelligence queries in Chapter 3.5.

To the best of our knowledge, we are the first to bring reasoning over order dependencies into the query optimizer of a relational database system.

3.1 Using ODs in the Optimizer

We motivate *ODs* in analogy to *FDs*: FDs are to group-by as ODs are to order-by. Order is an additional property over a partition that plays important roles in databases.[6] ODs can be used to great advantage in query processing, just as FDs have been [37].

3.1.1 Order in Optimization

On the one hand, order is irrelevant in the relational model on the *logical* side. Relational instances are *sets* of attributes, and a schema is a *set* of attributes. So there is no notion of order. (For different data models such as XML, order is an integral part of the model itself.) SQL concedes a single order-by clause to be appended to a query to order the result set, as a convenience, given that people usually want to see the results organized in a given way. (The SQL extension of window aggregation provides this too.)

[6]The group-by and order-by clauses in SQL are syntactically the same, excepting the optional sort-direction directives—ascending (asc) and descending (desc)—in order-by. Their meanings are quite similar too; order-by results in an ordered list (*ordering*) of the answer tuples, which matches the *partitioning* of the tuples with respect to an equivalent group-by. Of course, group-by's are over sets of attributes (as are FDs), while order-by's are over lists of attributes (as are ODs), as the order of attributes is important to specify the lexicographical order at the data. This difference makes working with ODs harder.

On the other hand, order plays an important role on the physical side, in storage, indexes, and optimization.

- *indexes.*

  Data is often referenced by (clustered) tree indexes, which provides ordered access.

- *pipelining.*

  In a query plan tree, *pipelining* is a prevalent technique. This is when a parent operator can *pull* its input streams from its child operators as they produce their (output) streams. The operator's procedure may need its input sorted in a given way, as does a *merge* join. An operator such as group-by or order-by can be handled very efficiently *on-the-fly* when its input stream is ordered appropriately. Pipelining between operators also saves processing and possibly I/O since the results of the child operator do not have to be fully *materialized, spilled* to disk with expensive I/O overhead.

- *interesting orders.*

  Some access paths and procedures will result in the operator's output stream being ordered. It may be that a procedure can be chosen for the parent operator which relies on this ordered stream for input and which is less expensive than the alternative choices.

  This enables pipelining between the operators, and may also be less expensive as it allows the optimizer to forgo inserting an expensive operation in between.

For example, the operator in the tree under a group-by might provide its output ordered in such a way the group-by's partitioning can be done *on-the-fly*. If not, an expensive partitioning or sort operation has to be inserted into the tree. Interesting orders can be effective for *join*, *order-by*, *group-by*, *partition-by*, and *distinct*.

Say that we know the database satisfies $\mathbf{X} \mapsto \mathbf{Y}$. Given a query with order by **Y**, we can rewrite it instead with order by **X**. Note that, unless $\mathbf{X} \leftrightarrow \mathbf{Y}$, the original and rewritten query are not "semantically" equivalent! This is an important property for query plans with ordered tuple streams. It means order *equivalences* are not required for valid query rewrites; directional order dependencies (that is, $\mathbf{X} \mapsto \mathbf{Y}$ instead of $\mathbf{X} \leftrightarrow \mathbf{Y}$) suffice. This provides us with much versatility for rewrites. The rewritten query satisfies the intent of the original (but, perhaps, not vice versa). *Strengthening* the order-by conditions is allowed, but *weakening* them is not.

One does not need order equivalences then to accomplish useful query rewrites. Directional order dependencies (e.g., $\mathbf{X} \mapsto \mathbf{Y}$, but not $\mathbf{Y} \mapsto \mathbf{X}$) suffice. This makes ODs that much more versatile for rewrites. Notice this differs from the use of FDs for query rewrites, for instance, to simplify group-by's. To replace year, quarter, month by year, month in the group-by for the query in the example in Chapter 1, one should know the two are functionally equivalent. One could not replace it by year, month, day, for example, even though {year, month, day} $\rightarrow$ {year, quarter, month}.

Sorting is an expensive operator. The key goal of our OD-optimization is to opti-

mize or eliminate sorting operations in query plans whenever possible. Our techniques are built upon the seminal techniques for order optimization from [37] as described in Chapter 3.1.2.

Query 3 Template of the query.

```
select year(a.y),...
from a, b
where a.x = b.x
group by year(a.y)
order by year(a.y);
```

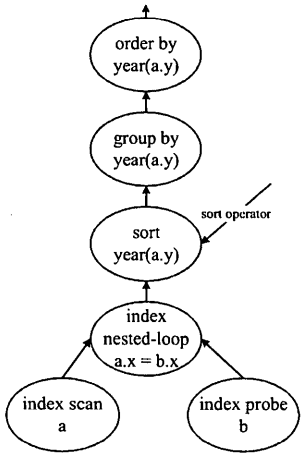

Figure 3.1: Query access plan.

Query 3 is a query sketch of a common pattern seen in analytic queries. It employs the SQL function year. Figure 3.1 illustrates a query plan one would expect for the

query in Query 3. Let the index employed as the access path on table a be on its column y, a date type. A sort operator is placed under the group-by and order-by operators, regardless, as the optimizer does not recognize that a.y *orders* year(a.y). Our work is to recognize these order dependencies, to "remove" the sort operator from access plan.

3.1.2 Canonical Form

The critical role of *interesting orders* was recognized quite early [36]. Because we are interested in ordered streams between operators in the query plan (to allow for pipelining, selecting more efficient procedures, and eliminating intermediate sort and partitioning steps), the optimizer needs to track which stream orders are possible to generate by alternative sub-plans. The ones that the optimizer tracks during query plan construction are called *interesting orders*.

The optimizer needs to determine which orders that sub-plans can produce are "interesting"; an order is not interesting if it is of no potential benefit to any other operator. In DB2, interesting orders are generated in a top-down scan of the graph of the query prior to the planning phase. (This is called the *order scan*.) Interesting orders are considered for join, order-by, group-by, partition-by and distinct operators. They are represented as lists of attributes. Interesting orders are *pushed down* and DB2 optimizer tries to combine interesting orders whenever possible. As interesting orders are pushed down, they can turn into sort-ahead orders. (Sort-ahead allows a sort for something like order-by to be

pushed into a join tree.) This process enables multiple interesting orders to be satisfied by one sort. Different alternative plans are tried and the least expensive is chosen.

Generating interesting orders (order specifications) is a complex task. Many different orders could apply for a given operation. For example, group by $A_0, \ldots, A_{n-1}$ can be accommodated on-the-fly by an input stream ordered by $[A_{p_0}, \ldots, A_{p_{n-1}}]$, for *any* one of the $n!$ permutations of $\{p_0, \ldots, p_{n-1}\} \subseteq \{0, \ldots, n-1\}$. Matching order specifications is also a complex task. For instance, group by A_1, A_2, A_3 can be done on-the-fly with an input stream ordered by $[A_3, A_1, A_2, A_4]$, but not by $[A_3, A_1, A_4, A_2]$.

On the one hand, the number of orders deemed "interesting" must be contained because of the sheer number of possibilities. On the other hand, we want to label more of those orders as "interesting" which would offer more planning options. In particular, we should recognize any order that is *order equivalent* with, or that *orders*, any order already marked as interesting.

One of the most basic operations used by order optimization is *reduction* [37]. Reduction is the method of rewriting an order specification in a simple *canonical form*. This includes substituting each column in the specification by an equivalent one, and then removing all redundant columns. Reduction is fundamental for simplifying an order specification $\mathbf{O} = [O_0, O_1, ..., O_{n-1}]$.

In [37], the authors explored the important role of *order* for optimizing queries. They introduced query rewrites in IBM DB2 that could exchange one interesting order by

another, when it is known that the orders were *order equivalent* (as defined in this work). They employ functional dependencies for this very task. The group by A_1, A_2, A_3 can be done on-the-fly with the input A_3, A_1, A_4, A_2, if $\{A_1, A_2, A_3\} \rightarrow \{A_4\}$. In that case, orders $[A_3, A_1, A_4, A_2]$, and $[A_3, A_1, A_2]$ are order equivalent.

Definition 6 (generated attribute)

A generated attribute is an attribute computed from other column using algebraic expressions and SQL functions.

Example 4 (attribute generated based on date and month)

Let G = year(d_date)*100 + month(d_month). *Thus,* G *is a* generated attribute.

We extend further the techniques of [37] by also employing order dependencies to recognize more order optimization techniques. (Their rewrites rely on FD information available to the optimizer, but do not use *order dependencies*.)

They introduced an algorithm, *Reduce Order*, which traverses the interesting-order list of attributes from right to left, that checks to eliminate attributes. This is for putting interesting orders into a canonical form.[7] We extend this algorithm[8] by iterating through the list, additionally checking following.

[7]The main body of their Reduce Order algorithm are lines 4 and 9–10 from Algorithm 1.

[8]In [39] we focused on order optimization techniques based on an axiomatization of ODs. In each iteration through the list, algorithm additionally checks whether any postfix *list* with respect to the current attribute – that is, a list of attributes to the right of the current – *orders* the current attribute. If so, the attribute is dropped from the list. In this work, we base the reduction on testing logical implication and algebraic expressions. This is more general and more efficient.

Algorithm 1 Reduce Order OD

Input: A set of ODs $\mathcal{M}$ and

order specification $\mathbf{O} = [O_0, O_1, ..., O_{n-1}]$, where O_i is an attribute.

Output: The reduced version of $\mathbf{O}$.

1: **for** $i \leftarrow 0$ **to** $n-1$ **do**

2: **if** O_i is a generated attribute from algebraic expression G and $O_i \leftrightarrow G$ **then**

3: $\mathbf{O} = [O_0, ..., O_{i-1}, G, O_{i+1}, ..., O_{n-1}]$

4: Rewrite $\mathbf{O}$ in terms of each column's equivalence class with a designated representative (called the equivalence class head) [37].

5: **for** $i \leftarrow n-1$ **to** 0 **do**

6: Let $B = \{O_0, ..., O_{i-1}\}$

7: **if** order specification is a single ($\mathbf{O} = [O_0]$), generated attribute from algebraic expression G and $G \mapsto O_0$ **then**

8: $\mathbf{O} = [G]$

9: **else if** $B \rightarrow O_i$ **then**

10: Remove O_i from $\mathbf{O}$ [37]

11: **else if** $[O_0, ..., O_{i-1}, O_{i+1}, ..., O_{n-1}] \mapsto [O_0, ..., O_{n-1}]$ **then**

12: Remove O_i from $\mathbf{O}$

13: **return** $\mathbf{O}$

- Whether the currently considered attribute O_i is a generated attribute from G and $O_i \leftrightarrow G$. If so, the attribute O_i is replaced by the attribute G in the list, $\mathbf{O} = [O_0, ..., O_{i-1}, G, O_{i+1}, ..., O_{n-1}]$.
- Whether the order specification is a single, generated attribute from an attribute G. (Call this attribute O_0.) If, $G \mapsto O_0$, the attribute O_0 is replaced by the attribute G in the list, $\mathbf{O} = [G]$. Detecting monotonicity property for generated attributes is described in detail in Chapter 3.2.
- Whether the *list* without the attribute being currently considered *orders* the full list. If so, the attribute is dropped from the current list.

With this, we can optimize queries such as Query 2 in Chapter 1. We infer

[d_year, d_month, d_day] $\mapsto$

[d_year, d_quarter, d_month, d_day]

from the set of declared ODs. Then, both the order-by and group-by can be reduced from d_year, d_month, d_quarter, d_day to just d_year, d_month, d_day. (Similarly, we can optimize Query 1 by detecting that d_date $\mapsto$ d_date + 30 days.)

Example 5 (Canonical form with dependencies)

Consider statement group by d_date + 30 days, d_year, d_month. *The canonical form is necessary to recognize that this is equivalent to* group by d_date, d_year, d_month. *It is only after that could reductions by FDs* — d_date $\rightarrow$ d_year *and* d_date $\rightarrow$ d_month — *be applied to reduce this even further to* group by d_date.

Reduce Order OD is correct for reducing an the order specification because removing O_i from the list using a FD $\mathsf{B} \rightarrow \mathsf{O}_i$ is part of *Reduce Order* algorithm described in [37] (Algorithm 1, lines 9–10). Given an order dependency $\mathbf{X} \mapsto \mathbf{Y}$, the clause order by $\mathbf{Y}$, can be rewritten with order by $\mathbf{X}$, as strengthening the order-by conditions is allowed as described in Chapter 3.1 (Algorithm 1, lines 7–8 and Algorithm 1, lines 11–12). It is also sound to replace order equivalent attributes ($\mathsf{O}_i \leftrightarrow \mathsf{G}$, Algorithm 1 lines 2–3).

Note that, when sorting is required, the simplified version of $\mathbf{O}$ provides a reduced number of sorting columns. This is important for minimizing sort costs. It may also happen that because of a reduced $\mathbf{O}$, an index can be matched, eliminating the need for a sort operator altogether.

In DB2, some columns might be substituted with *equivalent columns*. For instance, columns can be substituted with the ones on which an index is declared. This process is called *homogenization*. The *Homogenize Order* algorithm is described in [37]. It uses *equivalent classes* to substitute columns in an interesting order, $\mathbf{O} = [\mathsf{O}_0, \mathsf{O}_1, ..., \mathsf{O}_{n-1}]$, from a target order $\mathbf{T}$. We extend the algorithm as *Homogenize Order OD* to account for ODs.

The algorithm Homogenize Order OD is correct for substituing with equivalent columns because, given an OD $\mathsf{A} \leftrightarrow \mathsf{B}$, if the data are ordered by A, they are also ordered by B (and vice versa), so A can be substituted by B in an interesting order $\mathbf{O}$ (Algorithm 2 lines 3–4).

Algorithm 2 Homogenize Order OD

Input: A set of ODs $\mathcal{M}$,

an interesting order $\mathbf{O} = [O_0, O_1, ..., O_{n-1}]$, and

a target order $\mathbf{T} = [T_0, T_1, ..., T_{m-1}]$

Output:

$\mathbf{O}$ homogenized to $\mathbf{T}$ marked as $\mathbf{O_T}$ or

returned "false" indicating that $\mathbf{O_T}$ cannot be found.

1: Reduce $\mathbf{O}$

2: Using the set of ODs $\mathcal{M}$ try to substitute each column in $\mathbf{O}$ from $\mathbf{T}$

3: **if** for each A in $\mathbf{O}$ there exists B in $\mathbf{T}$ such that $A \leftrightarrow B$ **then**

4: **return** $\mathbf{O_T}$

5: **else**

6: **return** "false"

3.2 Detecting Order Dependencies

Analytic queries often use functions, algebraic expressions, and case expressions. Order dependencies can be derived from built-in SQL functions, and from case expressions [31]. For example, the SQL function *year* extracts the year component (the leading component) of the *date*. Thus, [date] $\mapsto$ [year(date)]. Let the table date_dim have an index on its d_date column. If it could detect the OD that d_date orders year(d_date),

the optimizer could accomplish order by year(d_date) in a query by using an index scan over the d_date index to provide a correct "interesting" order, with no need to employ sort operation.

We describe our techniques using monotonicity properties. These techniques have been implemented in DB2. The following rewrites were performed via this implementation. We describe how the monotonicity detection algorithm in IBM DB2 [31] allows for rewrites in the case of queries with order-by. We then show the value of this technique when combined with indexing in DB2. The algorithm detects monotonicity in algebraic expressions and SQL functions. It maintains a monotonicity state as the input expression is traversed. Given the parse tree of the expression to be checked, it answers whether the expression is monotonic (with respect to the attributes over which it is defined). It employs a transition table, scanning the left and right operands. For example, if the left side of the operand of the sum operator is monotonic and right operator is a constant, the result is is also monotonic. (See [31] for details.)

In [31], they showed how to use this for predicate derivation which led to it being implemented in DB2. They offered no performance study, though. We demonstrate the value for SQL queries via interesting orders. In our implementation, the monotonicity detection algorithm is called during the query rewrite phase, when processing statements which involve *join*, *order-by*, *group-by*, *partition by*, and *distinct* (Reduce Order OD). This is useful for improving access methods and for improving cardinality estimation.

Monotonicity can be also detected for a variety of SQL built-in functions. Ones we demonstrate here include the following. Each is monotonic with respect to its input.

- year(): Returns the year of the date.
- substr(): Returns a sub-string of the string input. If the starting position of the requested substring in the string is one, the result is monotonic. For example substr(s_zip, 1, 2) is monotonic.
- concat(): Returns the concatenation of two strings. (When the first string is always the same length and the second string is a constant, the function is monotonic.)

Monotonicity is detected for a wide range of functions: functions that refer to time dimensions, such as day() and hour(); mathematical functions, such as log(), ceil(), and sqrt(); and type conversions, such as int() and float().

Query 4 Substring with group-by.

```
select substr(P.s_zip, 1, 2) as area,
       count(distinct P.s_zip) as cnt,
       sum(S.ss_net_profit) as net
from store_sales S, store P
where S.ss_store_sk = P.s_store_sk
group by substr(P.s_zip, 1, 2);
```

Query 4 employs the substring function in its group-by. Recall Query 1 in Chapter 1. We saw that a clever programmer could recompose it to avoid the performance problem

that the use of the algebraic expression in the group-by and the order-by could cause. In this case, however, the programmer could not rewrite this to avoid the issue, since the substring changes the partition of the group-by.

Let there be an index on s_zip in table store. It is obvious that the column s_zip orders the derived column substr(s_zip, 1, 2). Given the optimizer detects this OD, it can choose to do an index scan using the index on s_zip to accomplish the group-by on-the-fly, and no partitioning or sort operator would be needed.

Query 5 With string conversion and concatenation.

```
select I.i_item_desc,
       to_char(D.d_date,'YYYYMMDD')
           || ' 12:00:00' as when,
       sum(W.ws_sales_price) as total
from web_sales W, item I, date_dim D
where W.ws_item_sk = I.i_item_sk and
      I.i_category = 'Children' and
      W.ws_sold_date_sk = D.d_date_sk
group by I.i_item_desc,
         to_char(D.d_date,'YYYYMMDD')
             ||'12:00:00'
order by to_char(D.d_date,'YYYYMMDD')
             ||'12:00:00';
```

Let there be an index on d_date in the date_dim table. In Query 5, the data are ordered by d_date converted to char and *concatenated* with a *time* constant, '12:00:00'. This is a type of query commonly used in business-intelligence reporting. The monotonicity detection algorithm works across the type conversion, and then over the string concatenation as the first string is known to be of a constant length. This makes the OD

[d_date] $\mapsto$ [to_char(d_date,'YYYYMMDD')||'12:00:00'], visible to the optimizer.

Query 6 can be effectively rewritten by the optimizer to the form in Query 7. An evaluation of the rewritten query then uses the index on d_date. The two constants 1998 and 2002 are used in Query 6 as a filter predicate in its where clause. The optimizer could not use the index on d_date, however, since the predicate is over year(D.d_date). In the query rewrite, the filter is set on d_date, to be on the range between date('1998-01-01') and date('2002-12-31'). Then, the optimizer uses index on d_date in the query plan. The method above generalizes to a query rewrite technique. Note that this is not captured by canonical form, as formed by Reduce Order OD algorithm. This demonstrates a need for rewrite techniques with ODs beyond those that the canonicalization provides. It can be used for SQL functions (besides year() of course) or user defined functions. For example, let there be an index on an attribute A. Assuming, there is known OD [A] $\mapsto$ [sql_function(A)] (or [A] $\mapsto$ [user_defined_function(A)]), the predicate can be set accordingly over the attribute A instead of the calculated sql_function (or user_defined function) on A.

Query 6 With the predicate *year*.

```
select I.i_item_desc, I.i_category,
       I.i_class, I.i_current_price,
       sum(W.ws_ext_sales_price) as revenue
from web_sales W, item I, date_dim D
where W.ws_item_sk = I.i_item_sk and
      W.ws_sold_date_sk = D.d_date_sk and
      year(D.d_date) between 1998 and 2002
group by I.i_item_id, I.i_item_desc,
         I.i_category, I.i_class,
         I.i_current_price
order by I.i_category, I.i_class,
         I.i_item_id;
```

Query 7 Rewrite of Query 6.

```
select ... from ... where ... and
       d_date between
           date('1998-01-01') and
           date('2002-12-31')
group by ... order by ...;
```

Query 8 is similar to Query 4, but with a filter predicate in its where clause. (This version does not contain a group-by clause, so the order-by could be rewritten manually. If a group-by were done on substr(H.w_warehouse_name,1,10) also, it could not be as substr function changes partitions. It illustrates our OD techniques the same, though, in either case.)

Query 8 Substring variation with order-by.

```
select substr(H.w_warehouse_name,1,10)
from web_sales W, warehouse H
where W.ws_warehouse_sk = H.w_warehouse_sk
      and W.ws_quantity > 90
order by substr(H.w_warehouse_name,1,10);
```

Query 9 OLAP query.

```
select count(*) as count
       over (partition by
          year(S.ws_sold_date)*100
          + month(S.ws_sold_date))
from web_sales S;
```

Query 9 is an OLAP query that uses a *partition-by* clause over modified TPC-DS schema as in Figure 2.4. The query plan can employ the index on ws_sold_date, if the optimizer detects via the monotonicity detection algorithm that OD

$$[\text{ws\_sold\_date}] \mapsto [\text{year(ws\_sold\_date)}*100 + \text{month(ws\_sold\_date)}]$$

holds, which effectively partitions by year concatenated with month.

Query 10 year(d_date) variation with order-by.

```
select year(D.d_date), M.sm_type, S.web_name,
       sum(case when
               (W.ws_ship_date_sk
                   - W.ws_sold_date_sk <= 30)
           then 1 else 0 end) as "30 days",
               ⋮
       sum(case when
               (W.ws_ship_date_sk
                   - W.ws_sold_date_sk > 120)
           then 1 else 0 end) as ">120 days"
from web_sales W, warehouse H, ship_mode M,
     web_site S, date_dim D
where W.ws_ship_date_sk = D.d_date_sk and ... group by
year(D.d_date), M.sm_type,
     S.web_name
order by year(D.d_date), M.sm_type,
     S.web_name;
```

Our work on order dependencies can be combined with notion of *near–sortedness*.

If a stream is sorted by A, it may be *nearly sorted* on, say, year(A), B, C. If there is an index on A and it is known that every partition of year(A) is small, the stream could be produced by an index scan, and thus be ordered by A. Given [A] $\mapsto$ [year(A)] and that the year(A)– blocks are small, each year(A)–block can be re–sorted in main memoery on–the–fly. This removes the need for an external sort operator. This technique also extends beyond the canonicalization.[9] As an example, consider Query 10 with a case expression. The monotonicity detection algorithm is triggered due to the order-by. It detects that [d_date] $\mapsto$ [year(d_date)]. Hence, the optimizer can then take advantage of the index on d_date, speeding up the sort operator in the plan, to accomplish the order-by.

3.3 Embedding the Information into the Key

Our next consideration is to embed information about other attributes (from the same table as the surrogate key) into the surrogate key. While this would be problematic for surrogate keys generally, this can work for time because the date dimension table is static. The representation of the date does not change over time, so keys might be assigned in a meaningful way. By embedding information into the surrogate key, a query can directly filter over the fact table avoiding an expensive join (assuming all information for the

[9]Similarly, ODs and near-sortedness can be used when using SQL functions such as concat(), when the first string is of fixed length and the second string is not constant. Without the second string a constant as in Query 5, we can still use the index on the first string to provide a *prefix order* and then use "mini-sorts" on–the–fly to avoid spilling to the disk.

query which is necessary is embedded in the key). A typical query with a join between a fact table and a dimension table taken from TPC-DS benchmark has a form like query Query 11 (Q12 from TPC-DS benchmark).

A critical factor in making the technique efficient and useful is to find a good function which generates the key. A function which can be used is one converting the date into a number which consists of the year, month and day. The date 22th February 1999 would be converted into the key of the date dimension table: 1999022. This is shown in the Query 12. A more sophisticated function can be established for generating the key. Assume the surrogate key is a 4-byte integer. The date takes eight digits so there are still free digits. These can be used for indicators for weekday, holiday, current day, or other fields which appear often. We can also use a binary representation to make the key with embedded information more compressed. With embedding information in the key, it is possible to keep historical information as well as some years into the future in the dimension table, because the function used is known in advance. One hundred fifty years of daily records is around 54,750 rows, which is a small table compared to fact tables which are counted in terabytes. An ETL process benefits from filling the date dimension table in advance. Whenever the records are inserted to the fact table, a join between the fact table and the date dimension table is not needed to find the surrogate key.

Query 11 With an expensive join.

```
select I.i_item_desc, I.i_category,
       I.i_class, I.i_current_price,
       sum(W.ws_ext_sales_price) as itemrevenue,
       sum(W.ws_ext_sales_price)*100
           /sum(sum(W.ws_ext_sales_price)) over
            (partition by I.i_class) as revenueratio
from web_sales W, item I, date_dim D where W.ws_item_sk =
I.i_item_sk and
       I.i_category
          in ('Sports', 'Books', 'Home') and
       W.ws_sold_date_sk = D.d_date_sk and
       D.d_date between
           cast('1999-02-22' as date) and
           (cast('1999-02-22' as date)
                + 30 days)
group by I.i_item_id, I.i_item_desc, I.i_category,
       I.i_class, I.i_current_price
order by I.i_category, I.i_class, I.i_item_id
       I.i_item_desc, revenueratio;
```

Query 12 Embedding the information.

```
select ...
from web_sales W, item I, date_dim D, where ... and
      W.ws_sold_date_sk between
          19990222 and 19990324;
group by ... order by ...;
```

3.4 Declaring Order Dependencies

In [40], we demonstrated that dramatic gains in query performance can be had in queries by recognizing ordering correspondences between attributes. Our techniques looked promising to generalize to many more types of the queries, which lead to the work here.

As discussed in Chapter 2, there are often compelling reasons in a data warehouse to have an explicit dimension table for date, and, furthermore, one that uses surrogate keys. This is standard practice. There are disadvantages to this, however. A prevalent class of queries accesses a fact table, and has predicates on date that mention natural date values. Their evaluation will blindly join the fact table to the dimension table. This can be quite expensive. Often, a fact table will be partitioned over date. Data warehouse systems use partitioning to accommodate very large tables, making it easier to administer (back up data, re-organize data, roll-in new data, and roll-out old data) and to improve query performance. In this case, however, all partitions of the fact table will have to be scanned,

because the natural date values in the query cannot be used to establish a range to scan just the relevant partitions, which are partitioned on the surrogate keys.

We seek to optimize such queries by avoiding the join to the date table, and selecting just the relevant partitions of the fact table. We introduce a query-rewrite technique that achieves this. We present a universal solution that rewrites the query within the optimizer considering order dependency. We note that in the implementation of almost all data warehouses that use surrogate keys for date, the order of surrogate keys is precisely the same as the order of the natural date values. That is to say, the surrogate keys and natural date values are monotone with respect to each other. We can use this order dependency to rewrite queries.

Below, our technique using order dependencies is described. This technique was used in our prototype which has been implemented in DB2 V9.7. The following rewrites were considered as part of developing the prototype within the optimizer.

The surrogate (date) keys in the date dimension table are ordered in the same way as natural date values in the dimension table, however. So there is a known order dependency between them. This can be declared as a check constraint in DB2.[10] This can be done whenever the database administrator knows of relevant order dependencies that are essentially a part of the *semantics* of the database. Declaring an OD as an integrity constraint (as SQL check constraint) gives a guarantee that the database will satisfy it.

[10]In our prototype in DB2 in [40], we had a way to express ODs internally, whereas in this work we propose to declare such dependencies as formal integrity constraints.

If in the table there is order dependency between the surrogate key and another column, a small number of lookups can be used as part of the query. Two probes can be made into the dimension table in order to calculate the range of the surrogate keys to be used as a filter over the fact table.

Query 13 Rewrite of Query 11.

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

Query 11, with the observation of the order dependency between the date surrogate

key and the natural date values, can be rewritten to the form shown below (Query 13). The two probes are selected from date table: mindate and maxdate surrogate key. These two probes provide us with minimum and maximum surrogate values of the key which are used to set the filter in the where clause.

In our first attempt, we have rewritten the query to use subqueries for these probes in the where clause. Based on our experiments and analyzing the access paths, we discovered that the optimizer does not treat it as a predicate on the same field, which meant it was not giving the optimal access plan. Our next try was to put the subqueries for the probes into the from clause. This is successful. The optimizer then guarantees the same access plans as it does for queries with constants in a between range predicate.

The following conditions are required in order to implement an efficient automatic rewrite:

1. **Foreign key relationship (condition for optimization).** The primary key from table A matches a surrogate foreign key from table B in the predicate (B.fk = A.pk) and relationship between table A (tpcds.date_dim in Query 11) and B (tpcds.web_sales in Query 11) is one to many.
2. **Partitioning or index key (condition for optimization).** The foreign key B.fk is a range partitioning key, or there exists an index on it which can be used by the index manager for the subquery range predicate as a start and stop key.
3. **Local predicate (condition for optimization).** Table A has a simple local pred-

icate in the form of: A.col <relational operator> literal,where the relational operator is one of $\geq$, $\leq$, $<$, $>$, $=$ and A.col is not A.pk. In our implementation, we consider queries with a binary relationship predicate. This can be extended to queries with more complex expressions such as the in operator. This kind of query also exists in the TPC-DS benchmark.

4. **Monotonic dependency (semantically required condition).** There exists an order dependency declaration between the primary key of table A, column A.pk, and A.col in the local predicate in the query. At present in our prototype in DB2, we have a way to express this increasing or decreasing order dependency. We implemented a mechanism to declare such dependencies as formal integrity constraints. Note that maintenance of such a dependency declaration is not expensive. When a new row is inserted in table A, only the rows with immediately preceding and succeeding values of A.pk by order need to be checked to ensure that the new A.col value maintains the monotone condition. Given there will be an index on the primary key of A, this check is inexpensive. The dependency between attributes d_date_sk and d_date is strict monotonic.

5. **No select on the A dimension table (condition for optimization).** Table A is not involved in the select. The fact that A has no column output from the current select is optional. If there is no column output from A, the join is eliminated. Otherwise, we can still do the rewrite and take advantage of partitioning.

In the propc

guarantees that

2.3 the fill rate

the query rewr

we would be a

relational opera

Note that tl

tables in the T

but they do nc

of the partitior

is based on th

rewrite algorit

matches the p

tables is highl

data to be load

the data based

our rewritten c

The secon

TPC-DS benc

scription of th

:d query rewrite, the function min and max is used because there are no
ıe fill rate in the date dimension table is 100%. (For example in Table
ɔr column date is not 100%.) There is currently no method to inform
: engine about this. With the possibility of passing this information,
ıe to simplify the query by removing min and max and replacing the
ırs by an offset equation so fewer IOs would be used for the probes.

rewritten query makes appropriate use of the partitioning. All the fact
C-DS benchmark include the surrogate key from the date dimension,
include the actual date. So the original query cannot take advantage
ıg with the surrogate key from the date dimension because the filter
natural date value. The situation is different with the proposed query
ı. The filter uses the surrogate key from the date dimension so it perfectly
itioning key which is kept in the fact table. Partitioning for large fact
effective because it allows old data to be removed gracefully and new
and indexed without disturbing the rest of the fact table. Also selecting
ı partitioning key is more efficient. We have checked the access plan for
ery, and the partition key based on the date surrogate key is used.

query from the TPC-DS benchmark we considered is Query 14 (Q7 in
nark) which joins the fact table with the date dimension and uses a de-
date as a filter. In the TPC-DS schema, there is an additional column

which keeps the year (d_year). In the Query 14, the filter is set to the year 2000. We chose this query because it selects over many more days (one year) whereas Query 11 was only over 30 days. This query involves a select on a column in the date dimension table. We wanted to check how the proposed query rewrite behaves with a higher selection of data from the database.

The rewrite can be done with the same technique as for Query 11, but this time with an equality operator (filtering for year 2000). The rewrite cannot be applied if instead of a predicate on d_date there is a predicate on d_month. Here the order dependency is only local and there is no order dependency between d_date_sk and d_month defined. On the other hand, if there is predicate on both columns: d_year and d_month then the suggested strategy to optimize the query can still be used. In Query 15 if we would be able to pass information that the fill rate is 100%, and assuming there is check constraint d_year = year(d_date), the rewritten query could have just used a filter on a d_date. At present, we have not addressed this. Therefore, we can use min and max in the same way we did for Query 11.

The automatically rewritten query then with the filter on year is shown as in Query 15.

We performed experiments over TPC-DS in our implementation in DB2 to demonstrate the efficiency of the approach. Thirteen of TPC-DS's queries matched for the rewrite. Each benefited, with an average performance gain of 48%. The details of the

Query 14 With filter set to the year.

```
select I.i_item_id,
    cast(avg(cast(S.ss_quantity as decfloat))
        as decimal(10,6)) agg1,
   cast( avg(S.ss_list_price)
       as decimal(10,6)) agg2,
   cast(avg(S.ss_coupon_amt)
       as decimal(10,6)) agg3,
   cast(avg(S.ss_sales_price)
       as decimal(10,6)) agg4
from store_sales S, customer_demographics C
    date_dim D, item I, promotion P
where S.ss_sold_date_sk = D.d_date_sk and
      S.ss_item_sk = I.i_item_sk and
      S.ss_cdemo_sk = C.cd_demo_sk and
      S.ss_promo_sk = P.p_promo_sk and
      C.cd_gender = 'M' and
      C.cd_marital_status = 'S' and
      C.cd_education_status = 'College' and
      (P.p_channel_email = 'N' or
            P.p_channel_event = 'N') and
      D.d_year = 2000
group by I.i_item_id
order by I.i_item_id;
```

Query 15 Rewrite of Query 14.

```
select ...
from store_sales S, customer_demographics C
     item I, promotion P
     (select min(d_date_sk) as mindate
         from date_dim
         where d_year = 2000
         as A,
     (select max(d_date_sk) as maxdate
         from date_dim
         where d_date = 2000
         as Z
where ... and
      S.ss_sold_date_sk between
          A.mindate and Z.maxdate...
group by ... order by ...;
```

experimental results are in Chapter 3.5. Many more than 13 of the 99 queries in TPC-DS involve date predicates.

We know that we can extend our rewrite rules to cover many more of the cases seen in TPC-DS queries (for instance, to cover the case of sub-queries in an IN predicate). This is a matter of further implementation. These techniques can be extended, we believe, to cover more dimensions in data warehouses. Order dependencies exist elsewhere too,

specifically with geo-spatial information. Thus, these techniques should extend well for other common data types.

3.5 Experiments

We demonstrate that dramatic gain in query performance can be had in business-intelligence queries by detecting (Chapter 3.5.1) and declaring (Chapter 3.5.2) order dependencies. All test queries show a significant performance gain using the OD-extended optimizer.

Our techniques look promising to generalize many more types of queries that involve order dependencies.

3.5.1 Detecting ODs in Algebraic Expressions

We have implemented and tested in DB2 V10 the query rewrites described in Chapter 3.2. This chapter reports the performance of the six queries described above. Three more queries were run against a corresponding alternative database, based on the alternative schema (with the natural date key in the date table) as shown in Figure 2.4. This included variations of Queries 1 and 5 (Q1′ and Q5′, respectively), modified to match the alternative schema, and Query 9.

The experiments were performed on a performance testing machine with the operating system AIX 6.1 TL6 SP5 with four processors (Intel(R) Xeon(R) CPU) and 1GB of memory. Results were obtained on a ten-GB TPC-DS database.

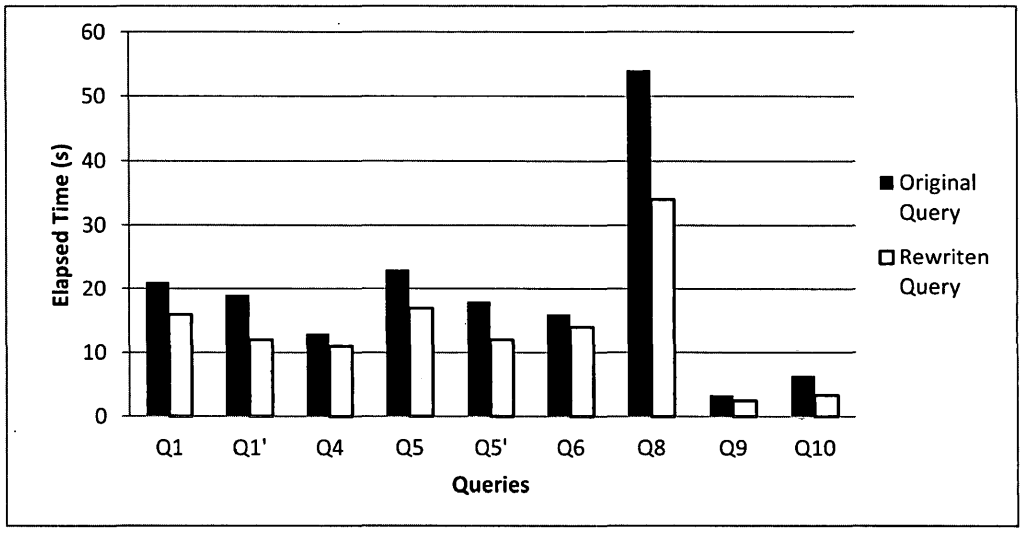

Figure 3.2: The effect of detecting ODs.

Figure 3.2 shows the execution times for the nine queries modified from the TPC-DS benchmark and expressing real world IBM customer scenarios, executed in two modes, with and without the OD-rewrites in the optimizer. Each query was run three times in both modes. (We repeat the tries in order to eliminate noise, including cold runs.) As shown in Figure 3.2, the results for the OD-optimized queries are significantly better. The performance improvement is, on average, a 30% improvement on elapsed time. Each of the nine queries benefited from the OD-rewrites.

The improved efficiency is dependent on the precise nature of the database tables, the generated columns and the query itself. Our techniques eliminate and optimize expensive operations such as sort, which are super-linear, and which begin to dominate the

execution costs as the database size increases.

3.5.2 Declaring ODs.

The query transformations described in Chapter 3.4 have been implemented in IBM DB2 V9.7 The experiments were performed on a performance testing machine with operating system SUSE Linux Enterprise Server SP1 with 4 processors (Intel(R) Xeon(R) CPU 5160 @ 3.00GHz), 22GB memory using TPC-DS benchmark with size 1GB.

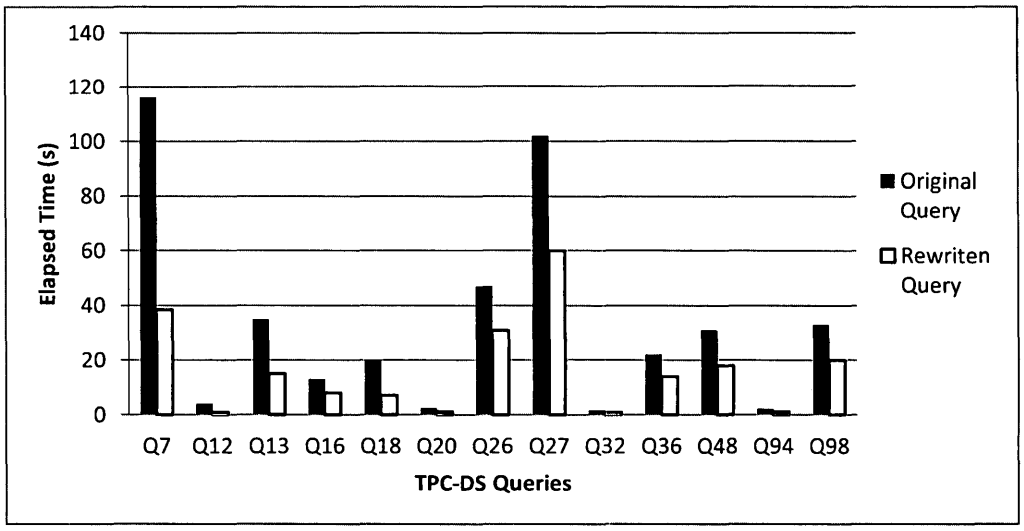

Figure 3.3: The effect of eliminating join with ODs.

The performance test results are shown in bar chart Figure 3.3. In our experiments, we measured the performance of the 13 queries from the TPC-DS benchmark that use dates in predicates and that match our rewrite rules. Identifiers of the queries follow the

convention used in TPC-DS benchmark.

As expected a significant reduction of the execution time is achieved. The optimization is achieved primarily by avoiding the join between the fact table and the date table by selecting two probes to calculate the range of surrogate keys from the dimension table. Also, as we have mentioned, the cardinality reduction due to the selection on the date table is greater than due to the selections on other tables, so the first join is done between the fact table and the dimension table. Eliminating this first join from the access plan brings significant benefits. An index on the date foreign key in the fact table is enough for efficient evaluation. Note that more substantial performance improvements could be achieved if the date foreign key in the fact table is also a partitioning key.

Figure 3.3 shows the execution times for the 13 queries from the benchmark executed with and without our rewrite. The results demonstrate significant performance improvement, on average a reduction of 48% in elapsed time. The other queries in TPC-DS were not affected as they were not rewritten. (There is an additional optimization cost because of the additional rewrite rules, but it is marginal.)

For the queries discussed in Chapter 3.4, the reduction was from 116.34 seconds to 38.98 seconds (66%) for Query 14 (Query Q7 from TPC-DS benchmark) and for the query with smaller filter, Query 11 (Q12 from TPC-DS benchamrk), from 4.26 seconds to 2.11 seconds (50%).

4 Axiomatization and Complexity of OD Inference

One of the key issues in dependency theory is the development of algorithms for testing logical implication [20, 29]. In addition to developing algorithms for determining logical implication, the second fundamental theme in dependency theory is the development of inference rules (axiomatization), which can be used to generate proofs of logical implication. Although the inference rules do not typically yield the most efficient mechanisms for deciding logical implication, in many cases they capture concisely the essential properties of the dependencies under study. The study of inference rules of dependencies is especially intriguing because there are several classes of dependencies for which there is no finite set of inference rules that characterizes logical implication.

Inference rules and algorithms for testing implication provide alternative approaches to showing logical implication between dependencies. In general, the existence of a finite set of inference rules for a class of dependencies is a stronger property than the existence of an algorithm for testing implication.

1. The existence of a finite set of inference rules for a class of dependencies implies

the existence of an algorithm for testing logical implication; and

2. There are dependencies for which there is no finite set of inference rules but for which there is an algorithm to test logical implication.

The inference rules are used to form proofs about logical implication between UODs, in a manner analogous to the proofs found in mathematical logic. It will be shown that the resulting proof system is *sound* and *complete* for UODs.

4.1 Axiomatization

A key concern in dependency theory is developing the algorithms for testing logical implication. Developing inference rules is an approach to show logical implication between dependencies.

4.1.1 Axioms

Definition 7 (A proof of θ from $\mathcal{M}$)

Let $\mathcal{M}$ be a set of prescribed UODs. A proof of UOD θ from $\mathcal{M}$ with the set of inference rules $\mathcal{I}$ is a sequence $\theta = \theta_1, ..., \theta_n$, where $n \geq 1$ such that for $k \in [1, n]$ either $\theta_k \in \mathcal{M}$, or there exists a substitution for some rule $\theta \in \mathcal{I}$, such that θ_k is consequence of φ, and such that for each unidirectional order dependency in the predecessor of θ the corresponding unidirectional order dependency is in the set $\{\theta_i \mid 1 \leq i < k\}$.

OD 1. *Reflexivity.*

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

OD 2. *Prefix.*

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

OD 3. *Normalization.*

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

OD 4. *Transitivity.*

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

OD 5. *Suffix.*

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

OD 6. *Chain.*

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

Figure 4.1: Axioms for UODs.

The UOD θ is provable from $\mathcal{M}$ using axioms $\mathcal{I}$ (relative to set of attributes $\mathcal{U}$), denoted $\mathcal{M} \vdash_{\mathcal{I}} \theta$, if there is a proof of θ from $\mathcal{M}$ using $\mathcal{I}$.

We now introduce axioms (inference rules) for UODs. In [39], we studied UODs and provided a *sound* and *complete* axiomatization for them. The inference rules of the axiomatization are shown in Figure 4.1.

Two of our axioms generate *trivial* dependencies:

- *Reflexivity.* If the right side is a prefix of the left side, it forms dependency.
- *Normalization.* Repetitive attributes following the ones already on the list can be removed.

We define the closure of the set of UODs $\mathcal{M}$, denoted $\mathcal{M}^+$, to be the set of UODs that are logically implied by $\mathcal{M}$.

Definition 8 (closure of $\mathcal{M}$ using $\mathcal{I}$)

Let $\mathcal{I}$ = *OD1-OD6, then* $\mathcal{M}^+ = \{\mathbf{X} \mapsto \mathbf{Y} \mid \mathcal{M} \vdash_{\mathcal{I}} \mathbf{X} \mapsto \mathbf{Y}\}$.

Definition 9 (equivalent sets of UODs)

Let $\mathcal{M}$ *and* $\mathcal{M}'$ *be sets of UODs. We say that* $\mathcal{M}$ *and* $\mathcal{M}'$ *are equivalent iff* $\{\mathbf{X} \mapsto \mathbf{Y} \mid \mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}\} = \{\mathbf{X} \mapsto \mathbf{Y} \mid \mathcal{M}' \models \mathbf{X} \mapsto \mathbf{Y}\}$, *where* $\mathcal{M} \models \theta$ *denotes that an OD* θ *is true with respect to set of ODs* $\mathcal{M}$.

Example 6 (Axiomatization)

1. Reflexivity.

$$[\text{year, month, week}] \mapsto [\text{year, month}]$$

2. Prefix.

$$\frac{[\text{month}] \mapsto [\text{quarter}]}{[\text{day, month}] \mapsto [\text{day, quarter}]}$$

3. Normalization.

$$[\text{year, month, day}] \leftrightarrow [\text{year, month, day, month}].$$

4. Transitivity.

$$\frac{\begin{array}{l}[\text{time}] \mapsto [\text{date}] \\ [\text{date}] \mapsto [\text{century}]\end{array}}{[\text{time}] \mapsto [\text{century}]}$$

5. Suffix.

$$\frac{[\text{date}] \mapsto [\text{year, month, day}]}{[\text{date}] \leftrightarrow [\text{year, month, day, date}]}$$

6. Chain.

$$\frac{\begin{array}{l}[\text{time}] \sim [\text{century}] \\ [\text{century}] \sim [\text{year}] \\ [\text{year}] \sim [\text{date}] \\ [\text{year, time}] \sim [\text{year, date}] \\ [\text{century, time}] \sim [\text{century, date}]\end{array}}{[\text{time}] \sim [\text{date}]}$$

4.1.2 Soundness

In this chapter, we show that our UOD axioms are sound.

Definition 10 (soundness of UOD axioms)

Let $\mathcal{I}$ be a set of inference rules OD1-OD6. Then $\mathcal{I}$ is sound for logical implication of UODs if $\mathbf{X} \mapsto \mathbf{Y}$ is deduced from $\mathcal{M}$ using axioms $\mathcal{I}$ ($\mathcal{M} \vdash_{\mathcal{I}} \mathbf{X} \mapsto \mathbf{Y}$), then $\mathbf{X} \mapsto \mathbf{Y}$ is true in any relation in which the dependencies of $\mathcal{M}$ are true ($\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$).

Let **r** be a relation over **R**. The following lemmas are true.

Lemma 2 (soundness of Reflexivity)

Reflexivity is sound.

Proof

Let s and $t \in \mathbf{r}$, such that $s \preceq_{\mathbf{XY}} t$. From the definition of operator $\preceq$ it follows that (1) $s_{\mathcal{X}} = t_{\mathcal{X}}$ and $s \preceq_{\mathbf{Y}} t$ or (2) $s \prec_{\mathbf{X}} t$. (1) and (2) imply that $s \preceq_{\mathbf{X}} t$.

$\forall\, \mathbf{r}\ \mathbf{XY} \mapsto \mathbf{X}$. □

Lemma 3 (soundness of Prefix)

Prefix is sound.

Proof

Let s and $t \in \mathbf{r}$, such that $s \preceq_{\mathbf{ZX}} t$. This implies (1) $s \prec_{\mathbf{X}} t$ or (2) $s_{\mathcal{Z}} = t_{\mathcal{Z}}$ and $s \preceq_{\mathbf{X}} t$. For

(1) $s \preceq_{\mathbf{ZY}} t$ holds as $s \prec_{\mathbf{Z}} t$. In the second scenario (2), $s \preceq_{\mathbf{X}} t$ implies $s \preceq_{\mathbf{Y}} t$ ($\mathbf{X} \mapsto \mathbf{Y}$ is given.) Hence, as $s_{\mathcal{Z}} = t_{\mathcal{Z}}$ it is true that $s \preceq_{\mathbf{ZY}} t$.

$\forall$ **r** $\mathbf{X} \mapsto \mathbf{Y}$ implies $\mathbf{ZX} \mapsto \mathbf{ZY}$. □

Lemma 4 (soundness of Normalization)

Normalization is sound.

Proof

IF Let s and $t \in$ **r**, such that $s \preceq_{\mathbf{WXYV}} t$. This implies that: (1) $s_{\mathcal{W}\mathcal{X}\mathcal{Y}} = t_{\mathcal{W}\mathcal{X}\mathcal{Y}}$ and $s \preceq_{\mathbf{V}} t$ or (2) $s \prec_{\mathbf{WXY}} t$. In (1) $s_{\mathcal{X}} = t_{\mathcal{X}}$ as $s_{\mathcal{W}\mathcal{X}\mathcal{Y}} = t_{\mathcal{W}\mathcal{X}\mathcal{Y}}$. Therefore, we can suffix **WXY** by list **X** and $s_{\mathcal{W}\mathcal{X}\mathcal{Y}\mathcal{X}} = t_{\mathcal{W}\mathcal{X}\mathcal{Y}\mathcal{X}}$ holds. Hence, $s \preceq_{\mathbf{WXYXV}} t$ as we know that $s \preceq_{\mathbf{V}} t$. Scenario (2), as $s \prec_{\mathbf{WXV}} t$ implies that we can suffix list **WXY** by **XV**and $s \preceq_{\mathbf{WXYXV}}$ holds. t

ONLY IF Let s and $t \in$ **r**, such that $s \preceq_{\mathbf{WXYXV}} t$. This implies that: (1) $s_{\mathcal{W}\mathcal{X}\mathcal{Y}} = t_{\mathcal{W}\mathcal{X}\mathcal{Y}}$ and $s \preceq_{\mathbf{XV}} t$ or (2) $s \prec_{\mathbf{WXY}} t$. In (1) $s_{\mathcal{X}} = t_{\mathcal{X}}$ as $s_{\mathcal{W}\mathcal{X}\mathcal{Y}} = t_{\mathcal{W}\mathcal{X}\mathcal{Y}}$. Hence, $s \preceq_{\mathbf{V}} t$ as we know that $s \preceq_{\mathbf{XV}} t$. Therefore, $s \preceq_{\mathbf{WXYV}} t$. Scenario (2), as $s \prec_{\mathbf{WXY}} t$ implies that we can suffix list **WXY** by **V** and $s \preceq_{\mathbf{WXYV}} t$ holds.

$\forall$ **r** $\mathbf{WXYXV} \leftrightarrow \mathbf{WXYV}$. □

Lemma 5 (soundness of Transitivity)

Transitivity is sound.

Proof

Let s and $t \in$ **r**, such that $s \preceq_{\mathbf{X}} t$. By $\mathbf{X} \mapsto \mathbf{Y}$, which is given $s \preceq_{\mathbf{Y}} t$, which implies $s \preceq_{\mathbf{Z}} t$

and it ends the proof.

$\forall$ **r** $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{Z}$ implies $\mathbf{X} \mapsto \mathbf{Z}$. □

Lemma 6 (soundness of Suffix)

Suffix is sound.

Proof

IF Let s and $t \in \mathbf{r}$, such that $s \preceq_{\mathbf{X}} t$. Therefore, $s \preceq_{\mathbf{X}} t$ as $\mathbf{X} \mapsto \mathbf{Y}$ is given, which implies that $s \preceq_{\mathbf{YX}} t$ ($\mathbf{X} \mapsto \mathbf{YX}$).

ONLY IF Let s and $t \in \mathbf{r}$, such that $s \preceq_{\mathbf{YX}} t$. Therefore, (1) $s_{\mathcal{Y}} = t_{\mathcal{Y}}$ and $s \preceq_{\mathbf{X}} t$ or (2) $s \prec_{\mathbf{Y}} t$ is true. Scenario (1) directly implies that $s \preceq_{\mathbf{X}} t$ ($\mathbf{YX} \mapsto \mathbf{X}$). Scenario (2) where $s \prec_{\mathbf{Y}} t$ implies that $s \prec_{\mathbf{X}} t$. This is because $s \not\prec_{\mathbf{X}} t$ implies $t \prec_{\mathbf{X}} s$ which implies $t \prec_{\mathbf{Y}} s$. Hence $s \not\prec_{\mathbf{Y}} t$. This ends the proof as $s \preceq_{\mathbf{X}} t$ ($\mathbf{YX} \mapsto \mathbf{X}$).

$\forall$ **r** $\mathbf{X} \mapsto \mathbf{Y}$ implies $\mathbf{X} \leftrightarrow \mathbf{YX}$. □

Lemma 7 (soundness of Chain)

Chain is sound.

Proof

Without loss of generality, assume that the lists in the axiom are single attributes. Let $\mathbf{X} = \mathsf{A}$, $\mathbf{Y}_1 = \mathsf{B}_1$, ..., $\mathbf{Y}_n = \mathsf{B}_n$ and $\mathsf{Z} = \mathsf{C}$. This simplification makes it easier to extend the rule to lists. The proof is by contradiction. Assume that A and C are order incompatible. Then there are two tuples for which there is a swap of the values between A and C. Also

Table 4.1: Order incompatible attributes.

A	B_1	B_2	...	B_n	C
0	0	0	0	0	1
1	1	1	1	1	0

the two tuples disagree on attribute B_i for all i. Otherwise condition number 4 would not be true. As $A \sim B_1$, the values for B_1 follow A, so does the rest of attributes B_i because of the condition (2). This means the two rows look in the way shown in Figure 4.1. But then B_n is order incompatible with C, which we assumed not to be the case. We conclude with contradiction. □

Theorem 1 *(soundness of axioms)*

OD1-OD6 axioms are sound for logical implication of UODs.

Proof

In order to prove the soundness of $\mathcal{I}$ we have to prove that each of the rules is sound (Lemma 2−Lemma 7). □

Theorem 2 *(soundness over ODs)*

The set of the axioms from Figure 4.1 is sound over ODs.

Proof

It is straightforward to show that inference rules in Figure 4.1 remain *sound* over (bidirectional) ODs. □

4.1.3 Additional Inference Rules

We introduce additional inference rules as they will be used throughout the thesis.

Union shows what can be inferred from two or more dependencies which have the same lists on the left side.

Theorem 3 *(Union)*

$$\frac{\begin{array}{l}1.\ \mathbf{X} \mapsto \mathbf{Y} \\ 2.\ \mathbf{X} \mapsto \mathbf{Z}\end{array}}{\mathbf{X} \mapsto \mathbf{YZ}}$$

Proof

3. $\mathbf{YX} \mapsto \mathbf{YZ}$ [Prefix(2)]

4. $\mathbf{X} \leftrightarrow \mathbf{YX}$ [Suffix(1]

5. $\mathbf{X} \mapsto \mathbf{YZ}$ [Transitivity(3, 4)]

□

Example 7 (Union)

$$\frac{\begin{array}{l}1.\ [\text{time}] \mapsto [\text{year, month, day}] \\ 2.\ [\text{time}] \mapsto [\text{century}]\end{array}}{[\text{time}] \mapsto [\text{year, month, day, century}]}$$

Theorem 4 states that we may augment the left side by any list of the attributes.

Theorem 4 *(Augmentation)*

$$\frac{1.\ \mathbf{X} \mapsto \mathbf{Y}}{\mathbf{XZ} \mapsto \mathbf{Y}}$$

Proof

2. $\mathbf{XZ} \mapsto \mathbf{X}$ [Reflexivity]

3. $\mathbf{XZ} \mapsto \mathbf{Y}$ [Transitivity(1,2)]

□

Example 8 (Augmentation)

$$\frac{1.\ [\text{date}] \mapsto [\text{year, month, day}]}{[\text{date, minute}] \mapsto [\text{year, month, day}]}$$

Example 9 (Augmentation)

$$\frac{1.\ [\text{date}] \mapsto [\text{year, month, day}]}{[\text{date, week}] \mapsto [\text{year, month, day}]}$$

Example 10 (Augmentation)

$$\frac{1.\ [\text{time}] \mapsto [\text{century, year, quarter}]}{[\text{time, day}] \mapsto [\text{century, year, quarter}]}$$

Example 11 (Augmentation)

$$\frac{1.\ [\text{time}] \mapsto [\text{century, year, quarter}]}{[\text{time, day, hour, minute}] \mapsto [\text{century, year, quarter}]}$$

The Shift rule is an extension of the Prefix rule. It tells us that equivalent lists of attributes can be shifted on the right side by **X** and **Y** if the order dependency between two lists holds.

Theorem 5 *(Shift)*

$$\frac{\begin{array}{l}1.\ \mathbf{W} \leftrightarrow \mathbf{V} \\ 2.\ \mathbf{X} \mapsto \mathbf{Y}\end{array}}{\mathbf{WX} \mapsto \mathbf{VY}}$$

Proof

3. $\mathbf{VX} \mapsto \mathbf{W}$ [Augmentation(1)]
4. $\mathbf{VVX} \mapsto \mathbf{VW}$ [Prefix(3)]
5. $\mathbf{VX} \leftrightarrow \mathbf{VW}$ [Normalization]
6. $\mathbf{VVX} \leftrightarrow \mathbf{VX}$ [Normalization]
7. $\mathbf{VX} \mapsto \mathbf{VW}$ [Transitivity(4,5)]
8. $\mathbf{VX} \leftrightarrow \mathbf{VWVX}$ [Suffix(6)]
9. $\mathbf{VWX} \leftrightarrow \mathbf{VWVX}$ [Normalization] □
10. $\mathbf{VX} \mapsto \mathbf{VWX}$ [Transitivity(7,8)]
11. $\mathbf{WX} \mapsto \mathbf{V}$ [Augmentation(1)]
12. $\mathbf{WX} \mapsto \mathbf{VWX}$ [Suffix(10)]
13. $\mathbf{WX} \mapsto \mathbf{VX}$ [Transitivity(9,11)]
14. $\mathbf{VX} \mapsto \mathbf{VY}$ [Prefix(2)]
15. $\mathbf{WX} \mapsto \mathbf{VY}$ [Transitivity(12,13)]

Example 12 (Shift)

1. [year] $\leftrightarrow$ [century, year]
2. [time] $\mapsto$ [date]

[year, time] $\mapsto$ [century, year, date]

Example 13 (Shift)

1. [date] $\leftrightarrow$ [year, month, day]
2. [month] $\mapsto$ [quarter]

[date, month] $\mapsto$ [year, month, day, quarter]

Example 14 (Shift)

1. [date] $\leftrightarrow$ [year, month, day]
2. [month, day] $\mapsto$ [quarter]

[date, month, day] $\mapsto$ [year, month, day, quarter]

Example 15 (Shift)

1. [time] $\leftrightarrow$ [year, month, day, hour, minute, second]
2. [year] $\mapsto$ [century]

[time, year] $\mapsto$ [year, month, day, hour, minute, second, century]

Example 16 (Shift)

1. [time] $\leftrightarrow$ [year, month, day, hour, minute, second]
2. [year, quarter] $\mapsto$ [century]

[time, year, quarter] $\mapsto$ [year, month, day, hour, minute, second, century]

Decomposition shows the method of decomposing an unidirectional order dependency given the prefix of the left side of the dependency.

Theorem 6 *(Decomposition)*

$$\frac{1.\ \mathbf{X} \mapsto \mathbf{ZY}}{\mathbf{X} \mapsto \mathbf{Z}}$$

Proof

2. $\mathbf{ZY} \mapsto \mathbf{Z}$ [Reflexivity]

3. $\mathbf{X} \mapsto \mathbf{Z}$ [Transitivity(1,2)]

□

Example 17 (Decomposition)

$$\frac{1.\ [\text{date}] \mapsto [\text{year, quarter}]}{[\text{date}] \mapsto [\text{year}]}$$

Example 18 (Decomposition)

$$\frac{1.\ [\text{date}] \mapsto [\text{year, quarter, month, week, day}]}{[\text{date}] \mapsto [\text{year, quarter, month}]}$$

Example 19 (Decomposition)

$$\frac{1.\ [\text{date}] \mapsto [\text{year, quarter, month, week, day}]}{[\text{date}] \mapsto [\text{year, quarter}]}$$

The following theorem is helpful to prove Eliminate, Left Eliminate and Drop.

Theorem 7 *(Replace)*

$$\frac{1.\ \mathbf{M} \leftrightarrow \mathbf{N}}{\mathbf{XMZ} \leftrightarrow \mathbf{XNZ}}$$

Proof

2. $\mathbf{Z} \mapsto \mathbf{Z}$ [Reflexivity]

3. $\mathbf{MZ} \mapsto \mathbf{NZ}$ [Shift(1,2)]

4. $\mathbf{NZ} \mapsto \mathbf{MZ}$ [Shift(1,2)]

5. $\mathbf{XMZ} \mapsto \mathbf{XNZ}$ [Prefix(3)]

6. $\mathbf{XNZ} \mapsto \mathbf{XMZ}$ [Prefix(4)]

7. $\mathbf{XMZ} \leftrightarrow \mathbf{XNZ}$ [Transitivity(5,6)]

□

Example 20 (Replace)

$$\frac{1.\ [\text{date}] \leftrightarrow [\text{year, month, day}]}{[\text{century, date, week}] \leftrightarrow [\text{century, year, month, day, week}]}$$

Example 21 (Replace)

$$\frac{1.\ [\text{date}] \leftrightarrow [\text{year, month, day}]}{[\text{century, date}] \leftrightarrow [\text{century, year, month, day}]}$$

Theorem 8 *(Eliminate)*

$$\frac{1.\ \mathbf{X} \mapsto \mathbf{Y}}{\mathbf{MXNYW} \leftrightarrow \mathbf{MXNW}}$$

Proof

2. $\mathbf{X} \mapsto \mathbf{YX}$ [Suffix(1)]

3. $\mathbf{XX} \mapsto \mathbf{XYX}$ [Prefix(2)]

4. $\mathbf{X} \mapsto \mathbf{XX}$ [Normalization]

5. $\mathbf{XY} \leftrightarrow \mathbf{XYX}$ [Normalization]

6. $\mathbf{X} \leftrightarrow \mathbf{XY}$ [Transitivity(3-5)] □

7. $\mathbf{MXYNW} \leftrightarrow \mathbf{MXNYW}$ [Replace(6)]

8. $\mathbf{MXYNYW} \leftrightarrow \mathbf{MXYNW}$ [Normalization]

9. $\mathbf{MXYNW} \leftrightarrow \mathbf{MXNW}$ [Replace(6)]

10. $\mathbf{MXNYW} \leftrightarrow \mathbf{MXNW}$ [Transitivity(7-9)]

Example 22 (Eliminate)

$$\frac{1.\ [\text{month}] \mapsto [\text{quarter}]}{[\text{date, month, quarter, day}] \mapsto [\text{date, month, day}]}$$

Example 23 (Eliminate)

$$\frac{1.\ [\text{month}] \mapsto [\text{quarter}]}{[\text{date, month, day, quarter}] \mapsto [\text{date, month, day}]}$$

Theorem 9 *(Left Eliminate)*

$$\frac{1.\ \mathbf{X} \mapsto \mathbf{Y}}{\mathbf{VYXZ} \leftrightarrow \mathbf{VXZ}}$$

Proof

2. $\mathbf{X} \leftrightarrow \mathbf{YX}$ [Suffix(1)]

3. $\mathbf{VYXZ} \mapsto \mathbf{VXZ}$ [Replace(1,2)]

□

Example 24 (Left Eliminate)

$$\frac{1.\ [\text{month}] \mapsto [\text{quarter}]}{[\text{century, quarter, month, week}] \mapsto [\text{century, month, week}]}$$

Example 25 (Left Eliminate)

$$\frac{1.\ [\text{month}] \mapsto [\text{quarter}]}{[\text{year, month, day}] \leftrightarrow [\text{year, quarter, month, day}]}$$

Example 26 (Left Eliminate)

$$\frac{1.\ [\text{month}] \mapsto [\text{quarter}]}{[\text{year, month, day}] \mapsto [\text{year, quarter, month, day}]}$$

Example 27 (Left Eliminate)

$$\frac{1.\ [\text{year}] \mapsto [\text{century}]}{[\text{year, month, day}] \mapsto [\text{century, year, month, day}]}$$

Theorem 10 shows that dropping elements on the right side of the list in the dependency is possible.

Theorem 10 *(Drop)*

1. $\mathbf{X} \mapsto \mathbf{VYZW}$
2. $\mathbf{X} \leftrightarrow \mathbf{V}$

$\mathbf{X} \mapsto \mathbf{VZ}$

Proof

3. $\mathbf{VYZW} \mapsto \mathbf{XYZW}$ [Replace(2)]
4. $\mathbf{X} \mapsto \mathbf{XYZW}$ [Transitivity(1,3)]
5. $\mathbf{X} \mapsto \mathbf{XY}$ [Decomposition(4)]
6. $\mathbf{XZW} \mapsto \mathbf{XY}$ [Augmentation(5)]
7. $\mathbf{XZW} \leftrightarrow \mathbf{XYXZW}$ [Suffix(6)]
8. $\mathbf{XYXZW} \leftrightarrow \mathbf{XYZW}$ [Normalization] □
9. $\mathbf{XZW} \leftrightarrow \mathbf{XYZW}$ [Transitivity(7,8)]
10. $\mathbf{X} \mapsto \mathbf{XZW}$ [Transitivity(4,9)]
11. $\mathbf{XZW} \mapsto \mathbf{VZW}$ [Replace(2)]
12. $\mathbf{X} \mapsto \mathbf{VZW}$ [Transitivity(10,11)]
13. $\mathbf{X} \mapsto \mathbf{VZ}$ [Decomposition(12)]

Example 28 (Drop)

1. [date] $\mapsto$ [year, month, day, quarter, century]
2. [date] $\leftrightarrow$ [year, month, day]

[date] $\mapsto$ [year, month, day, century]

Theorem 11 *(Path)*

1. $\mathbf{X} \mapsto \mathbf{YV}$
2. $\mathbf{Y} \leftrightarrow \mathbf{VMN}$

$\mathbf{X} \mapsto \mathbf{YMW}$

Proof

3. $\mathbf{X} \mapsto \mathbf{Y}$ [Decomposition(1)]

4. $\mathbf{X} \mapsto \mathbf{VMN}$ [Transitivity(2,3)]

5. $\mathbf{X} \mapsto \mathbf{YVMN}$ [Union(3,4)]

6. $\mathbf{YVMN} \mapsto \mathbf{YVMNM}$ [Normalization]

7. $\mathbf{X} \mapsto \mathbf{YVMNM}$ [Transitivity(5,6)] □

8. $\mathbf{X} \mapsto \mathbf{YM}$ [Elimination(2,7)]

9. $\mathbf{X} \mapsto \mathbf{YMYW}$ [Union(1,8)]

10. $\mathbf{YMYW} \mapsto \mathbf{YMW}$ [Normalization]

11. $\mathbf{X} \mapsto \mathbf{YMW}$ [Transitivity(1,10)]

Example 29 (Path)

1. [time] $\mapsto$ [year, month, day, hour]
2. [date] $\leftrightarrow$ [year, month, day]

[time] $\mapsto$ [date, hour]

Chain axiom is used to prove following two theorems.

Theorem 12 *(Partition)*

1. $\mathbf{X} \mapsto \mathbf{Y}$
2. $\mathbf{X} \mapsto \mathbf{Z}$
3. $\text{set}(\mathbf{Y}) = \text{set}(\mathbf{Z})$

$\mathbf{Y} \leftrightarrow \mathbf{Z}$

Proof

4. $\mathbf{X} \leftrightarrow \mathbf{YX}$ [Suffix(1)]
5. $\mathbf{X} \leftrightarrow \mathbf{X}$ [Reflexivity]
6. $\mathbf{X} \mapsto \mathbf{XY}$ [Union(1,5)]
7. $\mathbf{X} \leftrightarrow \mathbf{XYX}$ [Suffix(6)]
8. $\mathbf{X} \leftrightarrow \mathbf{XY}$ [Normalization(7)]
9. $\mathbf{XY} \leftrightarrow \mathbf{YX}$ [Transitivity(4,8)]
10. $\mathbf{XZ} \leftrightarrow \mathbf{ZX}$ [(2,4-9)]
11. $\mathbf{X} \sim \mathbf{Y}$ [(9)]
12. $\mathbf{X} \sim \mathbf{Z}$ [(10)]
13. $\mathbf{XYZ} \leftrightarrow \mathbf{XYZ}$ [Reflexivity]
14. $\mathbf{XY} \leftrightarrow \mathbf{XZ}$ [Elimination(1,2,13)]
15. $\mathbf{Y} \sim \mathbf{Z}$ [Chain(11-14)]
16. $\mathbf{YZ} \leftrightarrow \mathbf{ZY}$ [(15)]
17. $\mathbf{Y} \leftrightarrow \mathbf{Z}$ [Normalization(3,16)]

□

Example 30 (Partition)

1. [date] $\mapsto$ [year, month, quarter]
2. [date] $\mapsto$ [year, quarter, month]
3. {year, month, quarter, day} = {year, quarter, month, day}

[year, month, quarter] $\leftrightarrow$ [year, quarter, month]

Example 31 (Partition)

1. [date] $\mapsto$ [year, month, quarter, day]
2. [date] $\mapsto$ [year, quarter, month, day]
3. {year, month, quarter, day} = {year, quarter, month, day}

[year, month, quarter, day] $\leftrightarrow$ [year, quarter, month, day]

Example 32 (Partition)

1. [time] $\mapsto$ [date, year, month, day]
2. [time] $\mapsto$ [year, month, day, date]
3. {year, month, quarter, day} = {year, quarter, month, day}

[year, month, day, date] $\leftrightarrow$ [date, year, month, day]

Example 33 (Partition)

1. [time] $\mapsto$ [date, century]
2. [time] $\mapsto$ [century, date]
3. {year, month, quarter, day} = {year, quarter, month, day}

[century, date] $\leftrightarrow$ [date, century]

Theorem 13 *(Downward Closure)*

$$\frac{1.\ \mathbf{XY} \sim \mathbf{ZV}}{\mathbf{X} \sim \mathbf{Z}}$$

Proof

2. $\mathbf{ZVXY} \mapsto \mathbf{Z}$ [Reflexivity]

3. $\mathbf{XYXV} \mapsto \mathbf{Z}$ [Transitivity(1,2)]

4. $\mathbf{XYZV} \mapsto \mathbf{X}$ [Reflexivity]

5. $\mathbf{XYZV} \mapsto \mathbf{XZ}$ [Union(3,4)]

6. $\mathbf{XYZV} \mapsto \mathbf{ZX}$ [Union(3,4)]

7. $\mathbf{X} \sim \mathbf{Z}$ [Partition(5,6)]

□

Example 34 (Downward Closure)

$$\frac{1.\ [\text{time, quarter}] \sim [\text{date, month}]}{[\text{time}] \sim [\text{date}]}$$

Example 35 (Downward Closure)

$$\frac{1.\ [\text{time, quarter}] \sim [\text{date, month}]}{[\text{time, quarter}] \sim [\text{date}]}$$

Example 36 (Downward Closure)

$$\frac{1.\ [\text{year, quarter, month}] \sim [\text{year, month, quarter}]}{[\text{year, quarter}] \sim [\text{year, month}]}$$

4.1.4 Sketch of Completeness Proof

We sketch the important elements of the proof for completeness of our UOD axiomatization. The formal proof for completeness appears in the following two subchapters (Chapter 4.1.5 and Chapter 4.1.6). Our proof is constructive. To prove the axiomatization is complete, it suffices to demonstrate, for any set of UODs $\mathcal{M}$, a table **t** can be constructed that satisfies (Lemma 14) and is complete (Lemma 15) with respect to, $\mathcal{M}$ using $\mathcal{I}$, the axiomatization.

Definition 11 (table **t** satisfies $\mathcal{M}$)

A table **t** *satisfies* $\mathcal{M}$ *iff no UOD that is derivable over* $\mathcal{M}$ *using* $\mathcal{I}$ *(thus, in* $\mathcal{M}^+$*) is falsified by the table* **t**.

Definition 12 (a table **t** is complete with respect to $\mathcal{M}$)

A table **t** *is complete with respect to* $\mathcal{M}$ *iff every UOD that is constructible over the attributes that appear in* $\mathcal{M}$ *that is not derivable over* $\mathcal{M}$ *using* $\mathcal{I}$ *(thus, is not in* $\mathcal{M}^+$*) is falsified by the table* **t**.

In Theorem 1, we proved the soundness of $\mathcal{I}$. Thus, any table that satisfies each UOD in $\mathcal{M}$ satisfies $\mathcal{M}^+$, and no table that satisfies $\mathcal{M}$ can falsify any UOD in $\mathcal{M}^+$. Any UOD $\mathbf{X} \mapsto \mathbf{Y}$ can be falsified in just two ways by a table. We name these two ways split and swap (See Definition 13 and Definition 14 respectively).

Definition 13 (split)

A split *with respect to an OD* $\mathbf{X} \mapsto \mathbf{XY}$ *is a pair of tuples* s *and* t *such that* $s_{\mathcal{X}} = t_{\mathcal{X}}$ *but* $s_{\mathcal{Y}} \neq t_{\mathcal{Y}}$. *This says that* $\mathcal{X}$ *does not functionally determine* $\mathcal{Y}$.

Definition 14 (swap)

A swap *with respect to an OD* $\mathbf{XY} \leftrightarrow \mathbf{YX}$ *is a pair of tuples* s *and* t *such that* s $\prec_{\mathbf{X}}$ t, *but* t $\prec_{\mathbf{Y}}$ s; *i.e.,* s *comes before* t *in any stream satisfying* `order by` $\mathbf{X}$, *but* t *comes before* s *in any stream satisfying* `order by` $\mathbf{Y}$. *Thus, the swap falsifies* $\mathbf{X} \sim \mathbf{Y}$. *(Consequently,* $\mathbf{X} \mapsto \mathbf{Y}$ *is falsified, too.)*

The table **t** that we construct for the set of unidirectional order dependencies $\mathcal{M}$ will consist of two parts: split($\mathcal{M}$) and swap($\mathcal{M}$). We shall construct these two parts of **t** – the first half of the table, split($\mathcal{M}$) and the second half, swap($\mathcal{M}$) – in such a way that **t** satisfies $\mathcal{M}$. The purpose of split($\mathcal{M}$) will be to falsify every UOD of the form $\mathbf{X} \mapsto \mathbf{XY}$ not in $\mathcal{M}^+$. The purpose of swap($\mathcal{M}$) will be to falsify every UOD of the form $\mathbf{X} \mapsto \mathbf{Y}$, $\mathbf{XY} \leftrightarrow \mathbf{YX}$ not in $\mathcal{M}^+$ but for which $\mathbf{X} \mapsto \mathbf{XY}$ is in $\mathcal{M}^+$. (So $\mathbf{X} \mapsto \mathbf{Y}$ is not in $\mathcal{M}^+$ by Theorem 18).

Definition 15 (split($\mathcal{M}$))

Split($\mathcal{M}$) is a table that demonstrates for each $\mathbf{X} \mapsto \mathbf{XY}$ *which is not in* $\mathcal{M}^+$ *that* $\mathbf{X} \mapsto \mathbf{XY}$ *is falsified by split (and so, falsifies* $\mathbf{X} \mapsto \mathbf{Y}$, *too).*

Definition 16 (swap($\mathcal{M}$))

Swap($\mathcal{M}$) is a table that demonstrates for each $\mathbf{XY} \leftrightarrow \mathbf{YX}$ *which is not in* $\mathcal{M}^+$ *that* $\mathbf{X} \mapsto \mathbf{XY}$ *is falsified by split (and so, falsifies* $\mathbf{X} \mapsto \mathbf{Y}$*, too).*

In the table **t** that we construct, we shall use integer values for the cells. (A cell is a given column entry of a given row.) We construct table **t** by adding splits and swaps. We have to make sure that these pieces combined together do not interfere. That is why we formalize the notion *append.* When we append two tables $\mathbf{t}_1$ and $\mathbf{t}_2$, we shall ensure that the resulting table cannot introduce any splits (except $[] \mapsto \mathbf{X}$) or swaps beyond those that appear in $\mathbf{t}_1$ and in $\mathbf{t}_2$ alone (Lemma 9).

Definition 17 (append)

Appending two sub-tables $\mathbf{t}_1$ *and* $\mathbf{t}_2$ *is accomplished by following steps.*

1. *Find the minimum value, x, over all cells of* $\mathbf{t}_1$*. Subtract x from all cells in* $\mathbf{t}_1$*. (Now its minimum value is zero.)*
2. *Find the maximum value, y, over all cells of* $\mathbf{t}_1$*. Add y + 1 to all cells in* $\mathbf{t}_2$*. The resulting table of the append is the union of* $\mathbf{t}_1$ *and* $\mathbf{t}_2$ *as adjusted in steps 1 and 2.*

Table 4.2 is an example of an operation append.

The table **t** we construct will be split($\mathcal{M}$) and swap($\mathcal{M}$) (which we call split-swap form.) We shall construct split($\mathcal{M}$) in a way analogous to the construction in Ullman's proof of the completeness of Armstrong's axiomatization for FDs in [44]. This proves our axiomatization for UODs is sound and complete over FDs.

Table 4.2: Operation append.

A	B	C	D
0	0	0	0
0	0	1	1

(a) Table $\mathbf{t}_1$.

A	B	C	D
0	1	0	0
1	0	0	0

(b) Table $\mathbf{t}_2$.

A	B	C	D
0	0	0	0
0	0	1	1
2	3	2	2
3	2	2	2

(c) Table $\mathbf{t}_1$ append $\mathbf{t}_2$.

We shall construct swap($\mathcal{M}$) in a way to falsify each UOD $\mathbf{X} \mapsto \mathbf{Y}$ not in $\mathcal{M}^+$ (but for which $\mathbf{X} \mapsto \mathbf{XY}$ is in $\mathcal{M}^+$). This construction will be more complex than for split($\mathcal{M}$). For each pair of attributes A and B from $\mathcal{M}$ we determine whether there needs to be a swap between A and B – a pair of tuples s and t such that $t \prec_{\mathsf{A}} s$, but $s \prec_{\mathsf{B}} t$ – and, if so, the *context* in which swap between A and B need to occur.

Definition 18 (constant)

A marked attribute A *is called a* constant *with respect to* $\mathcal{M}$ iff $\mathcal{M} \models [] \mapsto \mathsf{A}$. *Call an attribute a non-constant, otherwise.*

If an attribute is a constant, it means in any table that satisfies $\mathcal{M}$, it can have only a single value occurring in the table.

Definition 19 (context)

A set of non-constant attributes $\mathcal{X}$ with respect to $\mathcal{M}$ is a context of a swap t, s *iff* $t_{\mathcal{X}} = s_{\mathcal{X}}$. *We say swap t, s is in the context of $\mathcal{X}$ iff $t_{\mathcal{X}} = s_{\mathcal{X}}$. (Note that a context for a swap t, s is not unique.)*

Constructing table swap($\mathcal{M}$) is not straightforward. We are able to simplify the construction via structural induction. the hypothesis is as follows.

Hypothesis 1 *(hypothesis)*

For some fixed integer K, for any set of UODs $\mathcal{M}$ composed over attributes $\{E_1, ..., E_k\}$, there exists a table **t** *in split-swap form that satisfies, and is complete with respect to, $\mathcal{M}$.*

We prove the base of this for $K \leq 2$ (in Lemma 11). We hypothesize this is true for any $\mathcal{M}$ with $K + 1$ attributes. We then prove that for any $\mathcal{M}$ with $K + 1$ attributes that the hypothesis remains true (Theorem 17). Proof of the induction hypothesis in essence completes the overall proof.

Induction provides us with a powerful mechanism within the proof. Consider any $\mathcal{M}$ with $K + 1$ attributes. In the first case, if any of the attributes are constants with respect to $\mathcal{M}$ we can reduce the problem. We effectively *project out* those constant attributes from $\mathcal{M}$. This means we simply remove all occurrences of the attributes in the UODs. For example, if we are projecting out B and E, ABC $\mapsto$ DEF becomes AC $\mapsto$ DF. Call

the result $\mathcal{M}'$. Then, $\mathcal{M}'$ is over K or fewer attributes. By the induction hypothesis there is a table $\mathbf{t}'$ which satisfies, and is complete with respect to, $\mathcal{M}'$. We can show easily how to construct a table $\mathbf{t}$ from $\mathbf{t}'$ which must satisfy, and be complete with respect to, $\mathcal{M}$. This is established by Lemma 8.

Lemma 8

Let $\mathbf{r}$ *be a table that satisfies, and is complete with respect to,* $\mathcal{M}$. *Let* Z *be an attribute not in* $\mathcal{M}$. *Construct table* $\mathbf{r}'$ *as* $\mathbf{r}$ *with an extra column* Z, *and the same single value for* Z *in each row. Then* $\mathbf{r}'$ *satisfies, and is complete with respect to,* $\mathcal{M} \cup \{[] \mapsto \mathsf{Z}\}$.

Proof

It is straightforward that $\mathbf{r}'$ satisfies $\mathcal{M} \cup \{[] \mapsto \mathsf{Z}\}$ because Z is a constant in $\mathbf{r}'$ and Z does appear in $\mathcal{M}$. Clearly, $\mathbf{r}'$ falsifies each $\mathbf{X} \mapsto \mathbf{Y}$ that does not mention Z that $\mathbf{r}$ falsifies. For any $\mathbf{X} \mapsto \mathbf{Y}$ that mentions Z, it is equivalent to some UOD that does not mention Z by the Replace rule, which has already been established. Thus, $\mathbf{r}'$ satisfies, and is complete with respect to, $\mathcal{M} \cup \{[] \mapsto \mathsf{Z}\}$. □

In the second case, we may assume $\mathcal{M}$ contains no constant attributes. when considering the pair A and B, if we find they require a swap in non-empty context $\mathcal{X}$, we can "freeze" the attributes of $\mathcal{X}$ to a single value. This is true, for any table that satisfies $\mathcal{M}'$ $= \mathcal{M} \cup \{[] \mapsto \mathsf{X}_1, ..., [] \mapsto \mathsf{X}_n\}$, where $\mathcal{X} = \{\mathsf{X}_1, ..., \mathsf{X}_n\}$. Now, we have an instance with K or fewer non-constants attributes. By our induction hypothesis, there exists a table $\mathbf{t}'$ in split-swap form that satisfies and is complete with respect to $\mathcal{M}'$. Note that

$\mathcal{M}'^{+} \supseteq \mathcal{M}^{+}$. Thus, $\mathbf{t}'$ does not falsify any UODs in $\mathcal{M}^{+}$. We append $\mathbf{t}'$ to the table $\mathbf{t}$ that we are constructing. (Appending these is safe, since $\mathcal{M}$ has no constants.) Our table swap($\mathcal{M}$) therefore is a recursive appending of (sub)tables.

There is a case of attributes A and B such that $\mathcal{M}$ dictates they must have a swap, but in the empty context $\{\}$. This time, we cannot use the induction hypothesis to construct the tuples for us $\mathbf{r}'$, that do the job. For this case, however, we can construct two tuples directly that introduce a swap for A and B, but that do not introduce swaps between any other pair of attributes that would falsify any UOD in $\mathcal{M}^{+}$. (The soundness of this step is established in Lemma 12.)

For the latter, we must show that, for each $\mathbf{X} \mapsto \mathbf{Y}$ not in $\mathcal{M}^{+}$ such that $\mathbf{X} \mapsto \mathbf{XY}$ is in $\mathcal{M}^{+}$, some sub-table in swap($\mathcal{M}$) by our construction does falsify it. This is done by proving there always is an attribute A in $\mathcal{X}$, an attribute B in $\mathbf{Y}$, and a swap between A and B in some context $\mathcal{W}$, which falsifies $\mathbf{X} \mapsto \mathbf{Y}$. (This is part of Lemma 15.) That completes the proof. These pieces are formally proved in the next two subchapters (Chapter 4.1.5 and Chapter 4.1.6).

4.1.5 Completeness over FDs

We show completeness of our axioms over FDs. This result is then used toward showing completeness over UODs.

Theorem 14 *(correspondence between FD and OD)*

For relation **R**, *for every instance* **r** *of it,* $\mathcal{X} \rightarrow \mathcal{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$, *for all list* $\mathbf{X}$ *that order the attributes of* $\mathcal{X}$ *and all list* $\mathbf{Y}$ *likewise for* $\mathcal{Y}$.

Proof

IF: Assume an OD $\mathbf{X} \mapsto \mathbf{XY}$ does not hold. This means, there exists s and $t \in \mathbf{r}$, such that $s \preceq_{\mathbf{X}} t$ but $s \not\preceq_{\mathbf{XY}} t$ by Definition 5. Therefore, $s_{\mathcal{X}} = t_{\mathcal{X}}$ and $s \prec_{\mathbf{Y}} t$. Also $s \prec_{\mathbf{Y}} t$ implies that $s_{\mathcal{Y}} \neq t_{\mathcal{Y}}$. Therefore, $\mathcal{X} \rightarrow \mathcal{Y}$ is not satisfied.

ONLY IF: By Lemma 1 if $\mathbf{X} \mapsto \mathbf{XY}$, then $\mathcal{X} \rightarrow \mathcal{XY}$. The FD $\mathcal{XY} \rightarrow \mathcal{Y}$ holds by Armstrong axiom Reflexivity [2]. Hence by Armstrong axiom Transitivity, $\mathcal{X} \rightarrow \mathcal{Y}$. □

Theorem 15 *(Permutation)*

$$\frac{1.\ \mathbf{X} \mapsto \mathbf{XY}}{\mathbf{X}' \mapsto \mathbf{X}'\mathbf{Y}'}$$

Proof

Let $\mathbf{Y} = [\mathsf{Y}_1, \mathsf{Y}_2, ..., \mathsf{Y}_n]$, $\forall k \in [1, n]$

2. $\mathbf{X} \mapsto \mathbf{X}\mathsf{Y}_1...\mathsf{Y}_k$ [Decomposition(1)]

3. $\mathbf{X}'\mathbf{X} \mapsto \mathbf{X}'\mathbf{X}\mathsf{Y}_1...\mathsf{Y}_k$ [Prefix(2)]

4. $\mathbf{X}'\mathbf{X} \leftrightarrow \mathbf{X}'$ [Normalization]

5. $\mathbf{X}'\mathbf{X}\mathsf{Y}_1...\mathsf{Y}_k \leftrightarrow \mathbf{X}'\mathsf{Y}_1...\mathsf{Y}_k$ [Normalization] □

6. $\mathbf{X}' \mapsto \mathbf{X}'\mathsf{Y}_1...\mathsf{Y}_k$ [Transitivity(3-5)]

7. $\mathbf{X}' \leftrightarrow \mathbf{X}'$ [Reflexivity]

8. $\mathbf{X}' \mapsto \mathbf{X}'\mathsf{Y}_k$ [Drop(6,7)]

9. $\mathbf{X}' \mapsto \mathbf{X}'\mathbf{Y}'$ [Union(8)]

Example 37 (Permutation)

$$\frac{\begin{array}{l}1.\ [\text{year, month, day}] \mapsto [\text{year, month, day, century, week}] \\ 2.\ [\text{month, day, year}] \mapsto [\text{month, day, year, week, century}]\end{array}}{[\text{time}] \mapsto [\text{year, month, day, century}]}$$

Theorem 16 *(completeness over FDs)*

Given the set of UODs $\mathcal{M}$, UOD axioms are sound and complete over functional dependencies.

Proof

Soundness is by Theorem 1, because of the correspondence between FDs and UODs (Theorem 14). The remaining step is to prove completeness over FDs, if $\mathcal{M} \models \mathcal{X} \rightarrow \mathcal{Y}$ then $\mathcal{M} \vdash_{\mathcal{I}} \mathcal{X} \rightarrow \mathcal{Y}$. This is equivalent to say if $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{XY}$ then $\mathcal{M} \vdash_{\mathcal{I}} \mathbf{X} \mapsto \mathbf{XY}$ for all lists $\mathbf{X}$ that order the attributes of $\mathcal{X}$ and all lists $\mathbf{Y}$ for $\mathcal{Y}$ by Theorem 14 and Permutation.

Firstly, we show that axioms for UODs imply Armstrong's axioms for FDs. We can do it because of soundness of axioms.

FD_1 Reflexivity:

$\mathcal{Y} \subseteq \mathcal{X}$ implies $\mathcal{X} \rightarrow \mathcal{Y}$.

1. We are given that $\mathcal{Y}$ is a subset of $\mathcal{X}$.
2. Therefore, the normalization rule implies that an UOD $\mathbf{X} \leftrightarrow \mathbf{XY}$ holds, for some list $\mathbf{X}$ that order the attributes of $\mathcal{X}$ and some list $\mathbf{Y}$ likewise for $\mathcal{Y}$.
3. Hence, Permutation and Theorem 14 implies that FD $\mathcal{X} \rightarrow \mathcal{Y}$ holds.

FD_2 Augmentation:

$\mathcal{X} \rightarrow \mathcal{Y}$ implies $\mathcal{Z}\mathcal{X} \rightarrow \mathcal{Z}\mathcal{Y}$.

1. Since we are given $\mathcal{X} \rightarrow \mathcal{Y}$, Theorem 14 tells us $\mathbf{X} \mapsto \mathbf{Y}$, for all lists $\mathbf{X}$ that order

the attributes of $\mathcal{X}$ and all lists **Y** likewise for $\mathcal{Y}$.

2. By Reflexivity we can infer **Z** $\leftrightarrow$ **Z**, for all list **Z** that order the attributes of $\mathcal{Z}$. Hence, by Prefix rule **ZX** $\mapsto$ **ZXY** holds.
3. By Suffix **ZX** $\leftrightarrow$ **ZXYZX**. **ZXYZX** may be normalized (**ZXYZX** $\leftrightarrow$ **ZXYX**).
4. By Transitivity **ZX** $\mapsto$ **ZXYZ**. Therefore, by Permutation and Theorem 14 FD $\mathcal{Z}\mathcal{X} \rightarrow \mathcal{Z}\mathcal{Y}$ holds.

FD_3 Transitivity:

$\mathcal{X} \rightarrow \mathcal{Y}$ and $\mathcal{Y} \rightarrow \mathcal{Z}$ implies $\mathcal{X} \rightarrow \mathcal{Z}$.

However, this proves that axiom system comprising of inference rules $\mathcal{I}$ is sound and complete for the set of FDs $\mathcal{F}$. We would like to show it is true for set of UODs $\mathcal{M}$.

Let $\mathcal{M}' = \{\mathbf{X} \mapsto \mathbf{XY}, \mathbf{XY} \leftrightarrow \mathbf{YX} \mid \mathbf{X} \mapsto \mathbf{XY} \in \mathcal{M}'\}$. Based on Theorem 18 $\mathcal{M}$ and $\mathcal{M}'$ are equivalent. Also let $\mathcal{F} = \{\mathcal{X} \rightarrow \mathcal{Y} \mid \mathbf{X} \mapsto \mathbf{XY} \in \mathcal{M}^+\}$

Based on Permutation rule and Theorem 14 we know that any relation instance satisfying dependencies in $\mathcal{F}$ satisfies dependencies in $\mathcal{M}'$ and vice versa. Let $\mathcal{X}^+$ [44], the closure of $\mathcal{X}$ (with respect to $\mathcal{F}$) be the set of attributes A such that $\mathcal{X} \rightarrow$ A can be deduced from $\mathcal{F}$ by Armstrong's axioms. We consider the relational instance **r** with two rows shown in figure below.

Based on Ullman's [44] proof of soundness and completeness of Armstrong's axioms, relation instance **r** shows that if $\mathcal{F}$ is the given set of dependencies, and $\mathcal{X} \rightarrow \mathcal{Y}$ cannot be proved by Armstrong's axioms, then **r** is a relation in which the dependencies

Table 4.3: Table r showing soundness and completeness over FDs.

$\mathcal{M}^+$ Attributes	Other Attributes
0...0	0...0
0...0	1...1

of $\mathcal{F}$ hold but $\mathcal{X} \rightarrow \mathcal{Y}$ does not. That is, $\mathcal{F}$ does not logically imply $\mathcal{X} \rightarrow \mathcal{Y}$. This means the inference rules are sound and complete over $\mathcal{F}$. As there is no swaps in **r**, we do not falsify anything in $\mathcal{M}'$, therefore $\mathcal{M}$, too. This ends the soundness and completeness proof for FDs over set of $\mathcal{M}$. □

4.1.6 Completeness of the UOD Axiomatization

As discussed before an UOD can be falsified by a split or a swap. Using this, our proof for completeness is by case. If $\mathbf{X} \mapsto \mathbf{XY}$ is not in $\mathcal{M}^+$, there will be a split in the sub-table split($\mathcal{M}$) that we construct that falsifies $\mathbf{X} \mapsto \mathbf{XY}$, and so that falsifies $\mathbf{X} \mapsto \mathbf{Y}$ also. If $\mathbf{X} \mapsto \mathbf{Y}$ is not in $\mathcal{M}^+$, but $\mathbf{X} \mapsto \mathbf{XY}$ is, there will be a swap in sub-table swap($\mathcal{M}$) that falsifies $\mathbf{X} \mapsto \mathbf{Y}$.

Lemma 9

There is not split in $\mathbf{t}_1$ *append* $\mathbf{t}_2$ *that is between rows from* $\mathbf{t}_1$ *and* $\mathbf{t}_2$*, respectively, besides* $[] \mapsto \mathbf{X}$ *for any* $\mathbf{X}$*. There is no swap in* $\mathbf{t}_1$ *append* $\mathbf{t}_2$ *that is between rows from* $\mathbf{t}_1$ *and* $\mathbf{t}_2$*, respectively.*

Proof

Let t be a tuple in $\mathbf{t}_1$ and s be a tuple in $\mathbf{t}_2$. Since all values in t are less that all values in s, it is impossible for there to be a split (except $[] \mapsto \mathbf{X}$) or swap introduced between $\mathbf{t}_1$ and $\mathbf{t}_2$ within $\mathbf{t}_1$ *append* $\mathbf{t}_2$. □

We construct table $\mathbf{t}$ to satisfy, and to be complete with respect to, $\mathcal{M}$. Table $\mathbf{t}$ will be split($\mathcal{M}$) append swap($\mathcal{M}$), as introduced above. Note that by Theorem 18 these are the only two scenarios.

Table split($\mathcal{M}$) is constructed by appending two rows to the table, as in Table 4.3 for each subset of attributes of $\mathcal{X}$ from $\mathcal{M}$.

Lemma 10 *(split($\mathcal{M}$) satisfies $\mathcal{M}$)*

For any $\mathcal{M}$ with no constants, split($\mathcal{M}$) does not falsify any UOD in $\mathcal{M}$.

Proof

The relational instance split($\mathcal{M}$) we have constructed contains splits, but no swaps. Therefore $\mathbf{X} \mapsto \mathbf{Y}$ could be only falsified by split. (Consequently, $\mathbf{X} \mapsto \mathbf{XY}$ is falsified, too.) But we know that we are sound and complete over set over FDs by Theorem 16 and by Lemma 9 appending of the tables does not introduce additional splits (except $[] \mapsto \mathbf{X}$) or swaps, therefore this is not possible.

Table split($\mathcal{M}$) is based on table we constructed for $\mathcal{M}$in the proof of Theorem 16, which establishes that UODs subsume FDs; that is, split($\mathcal{M}$) satisfies $\mathcal{M}$and it is com-

plete with respect to the UOD of the form $\mathbf{X} \mapsto \mathbf{XY}$ – which are equivalent to FD statement (Theorem 14) – in that it falsifies each $\mathbf{X} \mapsto \mathbf{Y}$ not in $\mathcal{M}^+$ but which is composable over the attributes in $\mathcal{M}$. As constructed, split($\mathcal{M}$) introduces no swaps.

For swap($\mathcal{M}$) a natural approach would seem to be to construct the table incrementally, to falsify each UOD not in $\mathcal{M}^+$, in turn, while ensuring we do not also falsify any OD in $\mathcal{M}^+$, in each step. This would be similar to how we constructed split($\mathcal{M}$). However, how to do this by a straightforward construction is not apparent. When considering how to falsify $\mathbf{X} \mapsto \mathbf{Y}$, which attributes from $\mathbf{X}$ and from $\mathbf{Y}$, respectively, should have a swap appear in the table? And how do we ensure that this swap does not falsify any UOD in $\mathcal{M}^+$? Instead, we consider every pair of attributes, A and B, from the set of attributes in $\mathcal{M}$. We determine the relevant contexts, if any, in which a swap with respect to A and B must occur in swap($\mathcal{M}$).

The set($\mathbf{XY}$) is a context for A, B with respect to $\mathcal{M}$ *iff* $\mathbf{XA} \sim \mathbf{Y}$ and $\mathbf{X} \sim \mathbf{YB}$ are in $\mathcal{M}^+$, $\mathbf{XA} \sim \mathbf{YB}$ but is not in $\mathcal{M}^+$. If there exists such a context for A, B, this indicates there should be a swap between A and B (to falsify $\mathbf{XA} \sim \mathbf{YB}$). It also indicates the "context" of the swap, as the swap must not falsify $\mathbf{XA} \sim \mathbf{Y}$ or $\mathbf{X} \sim \mathbf{YB}$. One could imagine constructing a swap – a pair of rows t and s for this – by having $t_{\mathcal{XY}} = s_{\mathcal{XY}}$. That way, the swap t, s would not falsify $\mathbf{XA} \sim \mathbf{Y}$ or $\mathbf{X} \sim \mathbf{YB}$. But what should the values of t and s be outside of $\mathbf{XY}$? We cannot construct t and s simply, ensuring the swap s, t does not falsify anything in $\mathcal{M}^+$. Instead, we use structural induction. Consider for now

Table 4.4: A relation instance for $K+1$ non-constants attributes.

Attributes of **XY**	Other Attributes			
0..0	$a_{1,1}$	$a_{1,2}$	...	$a_{1,i}$
...	...	...	...	...
0..0	$a_{j,1}$	$a_{j,2}$	...	$a_{j,i}$

that **XY** is non-empty. If we added $[] \mapsto$ **XY** to $\mathbb{M} -$ call the result $\mathcal{M}' -$ **XY** can only have a single value in any table that satisfies $\mathcal{M}'$. Recall the hypothesis from Hypothesis 1. We adopt this as our induction hypothesis. Assume our present $\mathcal{M}$ contains $K+1$ attributes. Then $\mathcal{M}'$ contains K or fewer attributes since $[] \mapsto$ **XY**. By our induction hypothesis, there is a table $\mathbf{t}'$ (see Table 4.4) that satisfies, and is complete with respect to $\mathcal{M}'$. As **XA** $\sim$ **YB** is not in $\mathcal{M}^+$, it is not in $\mathcal{M}'^+$ either. Thus $\mathbf{t}'$ falsifies **XA** $\sim$ **YB**. □

Which context for A, B should we do this for? Not for all of them. It is the maximal contexts that are relevant. **X**, **Y** is a maximal context for A, B *iff* it is a context for A, B and there is no other context **X**$'$, **Y**$'$ such that set(**X**$'$**Y**$'$) $\supset$ **XY**.

Since we use induction in the proof, we need to prove a base case of the induction hypothesis. We prove it for the cases of $\mathcal{M}$ with 0, 1, and 2 non−constant attributes in the following Lemma.

Lemma 11 *(Induction, base $K \leq 2$)*

For at most $K \leq 2$ attributes there exists a table **t** *in split-swap form that satisfies and is complete with respect to $\mathcal{M}$.*

Proof

This can be directly shown by enumerating through all the possibilities. □

We have assumed so far that the (maximal) contexts, if any, for A, B are non-empty. There is the case where A, B has a single maximal context $\{\}$, the empty context. In this case, we cannot appeal to the induction hypothesis. Fortunately, such pair A, B will have special properties by virtue of the fact they have swapped orders only in the empty context. In fact, our sixth axiom schema speaks directly to this very case. (We likely would never have had the insight for the sixth axiom (schema) Chain had we not encountered this case while attempting to prove completeness.) In this case, we will be able to construct a two-row swap for A, B directly that does not falsify anything in $\mathcal{M}^+$.

Lemma 12 *(empty context)*

There exists a swap for A, B *with the empty maximal context that satisfies* $\mathcal{M}$ *while falsifying* $\mathsf{A} \sim \mathsf{B}$.

Proof

We construct a two-row swap with values 0 and 1 that falsifies $\mathsf{A} \sim \mathsf{B}$ but cannot falsify anything in $\mathcal{M}^+$ as shown in Figure 4.5. For the latter, it suffices to prove that the swap does not falsify any $\mathsf{C} \sim \mathsf{D}$ in $\mathcal{M}^+$. For A and B, they have opposite values in each row in the swap. For any C such that $\mathsf{A} \sim \mathsf{C}$ is in $\mathcal{M}^+$, C must have the same value as A in each row. (Otherwise, A and C would have swapped values – 0 and 1 – between the two rows.) Likewise for B. And for any D such that $\mathsf{C} \sim \mathsf{D}$ is in $\mathcal{M}^+$, D must have the

Table 4.5: Swap for attributes with the empty maximal context.

A	B	A's group			B's group				Remaining attributes			
0	1	0	...	0	1	1	1	0	0	...	0
1	0	1	...	1	0	0	0	1	1	...	1

same value as C (and so the same as A) in each row. And so forth. Of course, it would be impossible to construct our two rows if there is a chain connecting A and B through order-compatibility: $\mathsf{A} \sim \mathsf{E}_1 \sim ... \sim \mathsf{E}_n \sim \mathsf{B}$. If there were, we would need to set the value of each $\mathsf{E}_1 \sim ... \sim \mathsf{E}_n$ the same as A's value and the same as B's value in each row. But A's and B's values differ. The Chain axiom schema (OD6) ensures there is no such chain from A to B. $\mathsf{E}_i\mathsf{A} \sim \mathsf{E}_i\mathsf{B}$ is in $\mathcal{M}^+$, for each E_i, since the maximal context for A, B is $[]$. If there were a chain $\mathsf{A} \sim \mathsf{E}_1 \sim ... \sim \mathsf{E}_n \sim \mathsf{B}$ such that $\mathsf{A} \sim \mathsf{E}_1$ is in $\mathcal{M}^+$, $\mathsf{E}_i \sim \mathsf{E}_{i+1}$ is in $\mathcal{M}^+$ for each i on $1, .., n-1$, and E_n B is in $\mathcal{M}^+$, then $\mathsf{A} \sim \mathsf{B}$ is in $\mathcal{M}^+$ also, by the Chain axiom. Since we know that $\mathsf{A} \sim \mathsf{B}$ is not in $\mathcal{M}^+$, there is no such Chain. Thus, our two rows are constructible. We can partition the attributes into three groups: those that must have the same values as A, those the same as B, and those for which it does not matter. Table 4.5 shows the construction.

For attributes that do not match A or B, it is important we do not introduce swaps between them, as this could falsify something in $\mathcal{M}^+$. It suffices to use the same value for these in each row. Call the two-row swap in Table 4.5 **r**. Table **r** satisfies $\mathcal{M}$. Assume otherwise: for $\mathbf{X} \mapsto \mathbf{Y} \in \mathcal{M}$, **r** falsifies it. Let $\mathbf{X} \mapsto \mathbf{Y}$ be over non-constants attributes, without loss of generality. Let E be the first element of **X**, and F of **Y**. If both E and

F are from A, A's group or the remaining group attributes (as in Table 4.5), or they are both from B or B's group attributes, then **X** and **Y** order the two tuples of **r** the same way. Therefore, E must be from one group, and F from the other. Since $\mathbf{X} \mapsto \mathbf{Y} \in \mathcal{M}^+$, $\mathbf{X} \sim \mathbf{Y} \in \mathcal{M}^+$ by Theorem 18. By the Downward Closure rule $\mathsf{E} \sim \mathsf{F} \in \mathcal{M}^+$. Contradiction. □

Our proof obligation for swap($\mathcal{M}$), that it does not falsify any UOD in $\mathcal{M}^+$ is proved in the following Lemma.

Lemma 13 *(swap($\mathcal{M}$) satisfies $\mathcal{M}$)*

Assuming Hypothesis 1, for all $\mathcal{M}$ of K or fewer non-constants attributes, swap($\mathcal{M}$) does not falsify any UOD in $\mathcal{M}$.

Proof

Hypothesis 1 is the key in proving that A, B does not falsify any UOD in $\mathcal{M}^+$. When we consider pair A and B which requires a swap in non-empty context **X** we obtain $\mathcal{M}' = \mathcal{M} \cup \{[] \mapsto \mathsf{X}_1, ..., [] \mapsto \mathsf{X}_n\}$, where $\mathcal{X} = \{\mathsf{X}_1, ..., \mathsf{X}_n\}$. By our hypothesis, there exists a table $\mathbf{t}'$ in split-swap form that is satisfied and complete with respect to $\mathcal{M}'$. As $\mathcal{M}'^+ \supseteq \mathcal{M}^+$, therefore any UODs in $\mathcal{M}^+$ is not falsified.

None of the sub-tables falsifies any UOD in $\mathcal{M}^+$ by the hypothesis in non-empty context and soundness of base cases (empty context and $K \leq 2$). As the table swap($\mathcal{M}$) is append-normalized, swap($\mathcal{M}$) does not falsify any UOD in $\mathcal{M}^+$. □

Lemma 14

Every UOD that is derivable with respect to the axiomatization over $\mathcal{M}$ is not falsified by the table **t**.

Proof

The sub-tables split($\mathcal{M}$) and swap($\mathcal{M}$), as we construct them, are satisfied with respect to $\mathcal{M}$ (Lemma 10 and Lemma 13 respectively). If neither split($\mathcal{M}$) nor swap($\mathcal{M}$) falsifies any UOD in $\mathcal{M}^+$, then **t** as split($\mathcal{M}$) append swap($\mathcal{M}$) cannot falsify any UOD in $\mathcal{M}^+$ either (See Lemma 9). □

Lemma 15

Assuming Hypothesis 1 for all $\mathcal{M}$ constructed over K or fewer attributes, given any $\mathcal{M}$ constructed over $K + 1$ attributes and none is a constant with respect to $\mathcal{M}$, the table **t** *= split($\mathcal{M}$) append swap($\mathcal{M}$) is complete with respect to $\mathcal{M}$.*

Proof

Assume $\mathbf{X} \mapsto \mathbf{Y}$ over only non-constant attributes, is in the complement of $\mathcal{M}^+$ ($\mathbf{X} \mapsto \mathbf{Y} \notin \mathcal{M}^+$). Theorem 18 tells us that unidirectional order dependency $\mathbf{X} \mapsto \mathbf{Y}$ holds iff $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

Case 1

$\mathbf{X} \mapsto \mathbf{XY} \notin \mathcal{M}^+$. We have already proven that for the scenario with $\mathbf{X} \mapsto \mathbf{XY}$ (FD) we are always complete (Theorem 16).

Case 2

$\mathbf{X} \mapsto \mathbf{Y} \notin \mathcal{M}^+$, but $\mathbf{X} \mapsto \mathbf{XY} \in \mathcal{M}^+$. By Theorem 18 $\mathbf{X} \sim \mathbf{Y} \notin \mathcal{M}^+$. Find longest $\mathbf{P}$A prefixing $\mathbf{X}$ such that:

1. $\mathbf{P} \sim \mathbf{Y} \in \mathcal{M}^+$
2. $\mathbf{PA} \sim \mathbf{Y} \notin \mathcal{M}^+$

   Find longest $\mathbf{Q}$B prefixing $\mathbf{Y}$ such that:

3. $\mathbf{PA} \sim \mathbf{Q} \in \mathcal{M}^+$
4. $\mathbf{PA} \sim \mathbf{QB} \notin \mathcal{M}^+$
5. $\mathbf{P} \sim \mathbf{Q} \in \mathcal{M}^+$ [Downward Closure(1)]
6. $\mathbf{P} \sim \mathbf{QB} \in \mathcal{M}^+$ [Downward Closure(1)]
7. $\mathbf{PAQB} \leftrightarrow \mathbf{QPAB} \in \mathcal{M}^+$ [Shift(3, B $\leftrightarrow$ B)]
8. $\mathbf{PAQB} \leftrightarrow \mathbf{PQAB} \in \mathcal{M}^+$ [Replace(5)]
9. $\mathbf{QBPA} \leftrightarrow \mathbf{PQBA} \in \mathcal{M}^+$ [Shift(6, A $\leftrightarrow$ A)]
10. $\mathbf{PAQB} \leftrightarrow \mathbf{QBPA} \notin \mathcal{M}^+$ [(4)]
11. $\mathbf{PQAB} \leftrightarrow \mathbf{PQBA} \notin \mathcal{M}^+$ [Transitivity(8,9,10)]
12. $\mathbf{PQA} \leftrightarrow \mathbf{PQB} \notin \mathcal{M}^+$ [(11)]

A and B have a swap within the context, $\mathcal{W}$ = set($\mathbf{PQ}$). In constructing swap($\mathcal{M}$), we considered all maximal contexts for A, B for which a swap is needed. Hence, we considered some superset $\mathcal{V} \supseteq \mathcal{W}$. If $\mathcal{V} \neq [\,]$, a sub-table that satisfies, and is complete with respect to $\mathcal{M} \cup \{[\,] \mapsto \mathsf{V}_1, ..., [\,] \mapsto \mathsf{V}_n\}$, where $\mathcal{V} = \{\mathsf{V}_1, ..., \mathsf{V}_n\}$ is appended in

swap($\mathcal{M}$). This falsifies $\mathbf{WA} \sim \mathbf{WB}$, for all lists $\mathbf{W}$ that order the attributes of $\mathcal{W}$ (thus, falsifies $\mathbf{X} \mapsto \mathbf{Y}$). Else if $\mathcal{V} = [\,]$, we appended a swap s, t as in Figure 4.5 which falsifies $\mathsf{A} \sim \mathsf{B}$ ($[\,]\mathsf{A} \sim [\,]\mathsf{B}$). □

Theorem 17 *(completeness)*

The set of the UOD axioms $\mathcal{I}$ = OD1-OD6 is complete.

Proof

Base case:

$\mathcal{M}$ with $K \leq 2$ attributes proved by Lemma 11. Assume Hypothesis 1 for all $\mathcal{M}$ composed over K or fewer attributes.

Induction step:

Consider an $\mathcal{M}$ over $K + 1$ attributes.

Case 1.

$\mathcal{M}$ contains constants attributes (Definition 24). Let $\mathcal{M}'$ be $\mathcal{M}$ with these constants attributes removed. $\mathcal{M}'$ has K or fewer attributes. By the induction hypothesis (Hypothesis 1) there is $\mathbf{r}'$ which satisfies, and is complete with respect to $\mathcal{M}'$. Lemma 8 guarantee we can construct $\mathbf{r}$ from $\mathbf{r}'$ that satisfies, and is complete with respect to $\mathcal{M}$.

Case 2.

$\mathcal{M}$ contains no constants attributes. Lemma 15 establishes there exists an $\mathbf{r}$ that satisfies, and is complete with respect to $\mathcal{M}$.

□

4.2 A Hierarchy of OD Classes

One can define *classes* of ODs, as we have done with our class of lexicographical ODs and the sub-class of UODs. Likewise, we can show that the class of functional dependencies is a sub-class of the class of UODs. In the literature, a number of variations of order dependencies have been studied. (We review these in Chapter 6.) The seminal work of [16] presented the especially general class of *pointwise order dependencies*. We establish the relationships among these classes.

Of course, the syntax for FDs and ODs is different. We need to say formally what it means for one class of dependencies to *generalize* another (or that one class is a *sub-class* of another).

Definition 20 *(*Class **A** generalizes class **B**.*)*

Let mapping σ map dependencies from class **B** *into sets of dependencies in class* **A**. *Mapping σ is* semantically preserving iff, *for any table* **r**, *for any B of class* **B**, $\mathbf{t} \models B \iff \mathbf{t} \models \bigwedge \sigma(B)$. *(Additionally, mapping σ is polynomial* iff *there is a polynomial-time algorithm that implements it.)*

Dependency class **A** generalizes *dependency class* **B** iff *there is a* semantically preserving mapping *of any arbitrary dependency of class* **B** *into a set of dependencies of class* **A**.

Class **A** strictly generalizes *class* **B** iff **A** *generalizes* **B** *but* **B** *does not generalize* **A**.

If **A** *(strictly) generalizes* **B**, *we also say then that* **B** *is a (proper) sub-class of* **A**.

In Chapter 4.2.1, we characterize the inference problem for classes of dependencies of this type in general, for our (lexicographical) OD class, and decompose this into its fundamental pieces. (We employ these definitions and concepts in the proofs that follow.) We then establish a strict hierarchy of classes of ODs: in Chapter 4.2.2, we establish that our class of lexicographical ODs is, in fact, a proper sub-class of the class of pointwise order dependencies; in Chapter 4.2.3, we prove that UODs form a proper sub-class of the (lexicographical) ODs; and, in Chapter 4.2.4, we prove that FDs form a proper sub-class of UODs.

4.2.1 Violations

We are considering dependencies of the form $X \Rightarrow Y$ for which X and Y are predicate conditions that can be evaluated over an ordered pair of tuples. A table *satisfies* the dependency *iff*, for every possible (ordered) pair of tuples from the table, if the pair satisfies X, then the pair also satisfies Y. Thus, if table **t** does not *satisfy* a dependency, then there exists a *pair* of tuples from **t** that *violates* (*falsifies*) the dependency. This suffices to define functional and order dependencies.

The inference problem for any such class is whether, given a collection of dependencies, a target dependency is logically entailed by the collection. This is defined in Definition 21 for our class of lexicographical ODs.

Definition 21 ($\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$)

The problem of testing logical implication *for ODs is, given a set of ODs* $\mathcal{M}$ *and an OD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$*, to decide whether* $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$.

Since an ordered pair suffices to represent a violation, one can rewrite the two tuples with just the values 0 and 1, while preserving the relative order between columns's values between the two tuples, without loss of generality. Thus, to answer a question of logical implication (as by Definition 21), it would suffice to evaluate every pair of tuples composable over 0 and 1. For every such pair, if the pair does not violate any dependency in the collection, it also does not violate the target dependency, then the target is logically entailed.

Consider ODs. Given n attributes over the collection, there are 2^{2n} such ordered tuple pairs composable over 0 and 1. This sets a lower bound for this brute force approach for checking of $O(4^n)$. (Checking each pair additionally requires the expense of evaluating, for each OD $\mathbf{X} \mapsto \mathbf{Y}$ in the collection and the target itself, $\mathbf{X}$ against the pair and $\mathbf{Y}$ against the pair.) This procedure directly establishes that the inference problem for ODs is *decidable*.

For any class of dependencies, the inference problem is likewise decidable, of course, if i) a violation is a pair of tuples, ii) the domain of pairs that need to be checked is finite with respect to the dependencies, and iii) the check for violation given a pair and dependency is itself decidable.

Table 4.6: Table representing a split and a swap.

N	V	A	B	M	W
0	0	0	1	0	0
0	0	1	0	1	1

An OD $\mathbf{X} \mapsto \mathbf{Y}$ can be violated (*falsified*) in two ways, as by Theorem 18: by *splits* and *swaps*.

Theorem 18 *(decomposition)*

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

Proof

IF: Suppose $\mathbf{X} \mapsto \mathbf{Y}$. By the Suffix rule $\mathbf{X} \leftrightarrow \mathbf{YX}$. By Prefix and Normalization $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

ONLY IF: Assume that $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{XY} \leftrightarrow \mathbf{YX}$. By Transitivity, $\mathbf{X} \mapsto \mathbf{YX}$. By Reflexivity and Transitivity, $\mathbf{X} \mapsto \mathbf{Y}$. □

Example 38 (split and swap)

There is a split in Table 4.6 with respect to an OD $[\overleftarrow{\mathsf{N}}, \overleftarrow{\mathsf{V}}] \mapsto [\overleftarrow{\mathsf{N}}, \overleftarrow{\mathsf{V}}, \overrightarrow{\mathsf{A}}, \overrightarrow{\mathsf{B}}]$ *and a swap in Table 4.6 with respect to an OD* $[\overleftarrow{\mathsf{N}}, \overrightarrow{\mathsf{A}}, \overrightarrow{\mathsf{M}}] \sim [\overleftarrow{\mathsf{V}}, \overrightarrow{\mathsf{B}}, \overrightarrow{\mathsf{W}}]$.

4.2.2 Pointwise generalizes Lexicographical

The class of pointwise order dependencies was proposed in the context of database systems in [16]. The type of dependency looks rather different than the lexicographical

ODs we have presented. The pointwise order dependency $\mathcal{X} \rightsquigarrow \mathcal{Y}$ holds if order over the values of *each* attribute of $\mathcal{X}$ implies order over the values of *each* attribute of $\mathcal{Y}$. Both $\mathcal{X}$ and $\mathcal{Y}$ are *sets* of order conditions. Let us restrict our interest to domains for which values are comparable.

Definition 22 *Each order condition is a marked attribute* A^{op} *for which* $\mathrm{op} \in \{<, >, \leq, \geq, =\}$. *For any table* $\mathbf{r}$, $\mathbf{r}$ *satisfies* $\mathcal{X} \rightsquigarrow \mathcal{Y}$ iff, *for any tuples* $\mathsf{s}, t \in \mathbf{r}$, *if, for each* A^{op} *in* $\mathcal{X}$, s_A op t_A, *then, for each* B^{op} *in* $\mathcal{Y}$, s_B op t_B.

While lexicographical ODs have been studied since (see Chapter 6), it has never been established how they are related with pointwise. We show thought the class of pointwise ODs strictly generalize the class of lexicographical ODs. (The mapping requires a quadratic number of pointwise ODs in the size of the lexicographical OD.)

The following two lemmas prove Theorem 19. Lemma 16 establishes that there are pointwise order dependencies that cannot be mapped in any semantically preserving way into lexicographical ODs. Lemma 17 shows a semantically preserving mapping of a lexicographical OD into a set of pointwise ODs. Together, these establishes that the class of lexicographical ODs is a proper sub-class of that of pointwise ODs.

Lexicographical ODs can be expressed by pointwise ODs.

Lemma 16

There exists a semantically preserving, polynomial mapping (Definition 20) of a lexico-

graphical OD into a set of pointwise ODs.

Proof

Algorithm 3 provides a polynomial mapping of an arbitrary lexicographical OD into a set of $\mathcal{E}$ of pointwise ODs. Any split or swap that violates the lexicographical OD $\mathbf{X} \mapsto \mathbf{Y}$ violates some pointwise OD in $\mathcal{E}$, and vice versa. □

Algorithm 3 Translation

Input: Lexicographical OD $\mathbf{X} \mapsto \mathbf{Y}$, where $\mathbf{X} = [X_0, ..., X_{m-1}]$ and $\mathbf{Y} = [Y_0, ..., Y_{n-1}]$

Output: A set of pointwise ODs $\mathcal{E}$ semantically equivalent to lexicographical OD $\mathbf{X} \mapsto \mathbf{Y}$.

1: $\mathcal{E} = \{A_0^{=}, ..., A_{m-1}^{=} \rightsquigarrow A_0^{=}, ..., A_{n-1}^{=}\}$

2: **for** $i \leftarrow 0$ **to** $m - 1$ **do**

3: **for** $j \leftarrow 0$ **to** $n - 1$ **do**

4: **if** $X_i = \overrightarrow{A}_i$ and $Y_j = \overrightarrow{B}_j$ **then**

5: $\mathcal{E} = \mathcal{E} \cup \{A_0^{=}, ..., A_i^{>} \rightsquigarrow B_0^{=}, ..., B_j^{\geq}\}$

6: **else if** $X_i = \overrightarrow{A}_i$ and $Y_j = \overleftarrow{B}_j$ **then**

7: $\mathcal{E} = \mathcal{E} \cup \{A_0^{=}, ..., A_i^{>} \rightsquigarrow B_0^{=}, ..., B_j^{\leq}\}$

8: **else if** $X_i = \overleftarrow{A}_i$ and $Y_j = \overrightarrow{B}_j$ **then**

9: $\mathcal{E} = \mathcal{E} \cup \{A_0^{=}, ..., A_i^{<} \rightsquigarrow B_0^{=}, ..., B_j^{\geq}\}$

10: **else if** $X_i = \overleftarrow{A}_i$ and $Y_j = \overleftarrow{B}_j$ **then**

11: $\mathcal{E} = \mathcal{E} \cup \{A_0^{=}, ..., A_i^{<} \rightsquigarrow B_0^{=}, ..., B_j^{\leq}\}$

Lemma 17

There exists a pointwise OD that cannot be mapped in a semantically preserving way (Definition 20) into a set of lexicographical ODs.

Proof

Consider the following table **t**.

Table 4.7: Table **t**.

#	A	B	C
a	0	0	0
b	0	1	1
c	2	2	2
d	3	2	3
e	4	4	4
f	5	5	4
g	6	6	6
h	7	6	6
i	8	8	8
j	8	9	8
k	10	10	10
l	10	10	11
m	12	12	13
n	12	13	12
o	14	14	15
p	15	14	14
q	16	17	16
r	17	16	16

Pointwise OD $A^{>}B^{>} \rightsquigarrow C^{>}$ is satisfied by table **t**. However, it is straightforward to show that table **t** that we construct consists of all possible *splits* (Definition 13, rows a–l) and *swaps* (Definition 14, rows a–f and m–r) defined for ODs over marked attributes $\overleftarrow{A}$,

$\overrightarrow{A}$, $\overleftarrow{B}$, $\overrightarrow{B}$, $\overleftarrow{C}$ and $\overrightarrow{C}$ are falsified by table **t**. □

Theorem 19

The class of pointwise ODs strictly generalizes the class of lexicographical ODs.

Proof

There exists semantically preserving, polynomial mapping for any set of lexicographical ODs to a set of pointwise ODs (Lemma 16). Additionally, the class of pointwise ODs are more expressive than lexicographical ODs (Lemma 17). □

In [16], the authors demonstrated that the inference problem for pointwise ODs in general is co-NP-complete. By Theorem 19 and the fact that the mapping is polynomial this sets a ceiling for the inference problem for lexicographical ODs. However, the problem for lexicographical ODs is just as hard, as we prove in Chapter 4.3.

4.2.3 ODs generalize UODs

Next, we investigate whether the addition of bidirectioal ODs (BODs, Definition 5) add expressive power over UODs. This is equivalent to the question of whether the class of ODs strictly generalizes UODs.

With BODs, ODs are more expressive than UODs. (Given that UODs are a syntactic sub-class of ODs, it follows that the class of ODs strictly generalizes the class of UODs.)

Theorem 20

The class of ODs strictly generalizes the class of UODs.

Proof

UODs are a proper sub-class of ODs by Definition 5. □

In fact, we can easily prove that the sound and complete axiomatization for the class of UODs from Figure 4.1 is *not* complete for the class of ODs.

Lemma 18

(incomplete for ODs) The set of the axioms from Figure 4.1 is not complete over ODs.

Proof

Consider the set $\mathcal{M}$ of $[\overrightarrow{A}] \mapsto [\overrightarrow{B}]$ and $[\overrightarrow{A}] \mapsto [\overleftarrow{B}]$. From first principles, it is simple to show that $\mathcal{M} \models [] \mapsto [\overrightarrow{B}]$. None of the axioms *reduce* the left-hand side of an OD (besides *Normalization*, which does not apply here). $\mathcal{M} \models [] \mapsto [\overrightarrow{B}]$ cannot be proved from the axioms. □

The next question we might ask is whether there is a cost for this extra expressiveness. Is the inference problem harder for the class of ODs than for the class of UODs? Surprisingly perhaps, the answer is yes. This is shown in Chapter 4.3.

4.2.4 UODs generalize FDs

UODs are more expressive than FDs.

Table 4.8: Table **t** falsifying FDs.

#	A	B
a	0	0
b	0	1
c	2	2
d	3	2

Lemma 19

The class of UODs is more expressive than the class of FDs.

Proof

Consider the table **t** in Table 4.8. The UOD [A] $\sim$ [B] is satisfied in table **t**. However, in table **t**, all possible non-trivial FDs over attributes A and B are falsified. □

Theorem 21

The class of UODs strictly generalizes the class of FDs.

Proof

By Theorem 14 and Lemma 19. □

Since the class of ODs generalize the class of FDs the inference problem for UODs with INDs is *undecidable*.

Corollary 22 *(Undecidable for ODs with INDs)*

Testing logical implication for ODs with INDs is undecidable.

Proof

This follows from that implication for FDs and INDs is undecidable and Theorems 20 and 21 that ODs generalize UODs and that UODs generalize FDs, respectively. □

4.3 Complexity

We show that the inference problem for UODs (and ODs) is co-NP-complete [42]. Additionally, we show that inference problem for UODs and the inference problem of FDs from ODs are co-NP-complete. FD inference from UODs, a restricted case, is polynomially decidable, however.

4.3.1 OD Inference

We introduce first the notation which permits us to translate instances of 3-SAT into instances of the decision problem for testing logical implication for ODs. We assume the reader is familiar with NP-completeness in general, with the 3-SAT problem, and with reducibility [14].

Definition 23

Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ *be a set of propositional variables for an arbitrary finite* n*, and let* $\overline{\mathcal{P}} = \{\neg p_1, \ldots, \neg p_n\}$*. Let* $\mathcal{F}$ *be a formula written over the propositional variables in* $\mathcal{P}$ *and their negations in conjunctive normal form with* k *clauses, each a disjunction of*

length three, for an arbitrary finite k.

For $i \in \{1, \ldots, k\}$, let $\mathsf{V}_{i,1} \vee \mathsf{V}_{i,2} \vee \mathsf{V}_{i,3}$ represent clause i such that

$\mathsf{V}_{i,1} \in (\mathcal{P} \cup \overline{\mathcal{P}})$,

$\mathsf{V}_{i,2} \in (\mathcal{P} \cup \overline{\mathcal{P}}) - \{\mathsf{V}_{i,1}\}$, *and*

$\mathsf{V}_{i,3} \in (\mathcal{P} \cup \overline{\mathcal{P}}) - \{\mathsf{V}_{i,1}, \mathsf{V}_{i,2}\}$,

without loss of generality.

Call any such $\mathcal{F}$ a 3-SAT candidate. *Call any such* 3-SAT candidate *$\mathcal{F}$ for which there exists a truth assignment over $\mathcal{F}$'s $\mathcal{P}$ which satisfies $\mathcal{F}$ a* 3-SAT instance.

3-SAT *is the collection of 3-SAT instances.*

Lemma 20 [14] **3-SAT** *is in NP-complete.*

Lemma 21

Given a set $\mathcal{M}$ of UODs and UOD $[\mathsf{A}] \sim [\mathsf{B}]$, deciding whether $\mathcal{M} \models [\mathsf{A}] \sim [\mathsf{B}]$ is co-NP-complete.

Proof

Candidate and instance. Given a 3-SAT candidate $\mathcal{F}$ (Definition 23), we construct an UODI candidate $\langle \mathcal{M}_{\mathcal{F}}, [\mathsf{T}] \sim [\mathsf{F}] \rangle$.

Let $\langle \mathcal{M}, \mathbf{X} \mapsto \mathbf{Y} \rangle$ be an arbitrary pair of a finite set $\mathcal{M}$ of UODs and a *target* UOD $\mathbf{X} \mapsto \mathbf{Y}$ constructed over the attributes that appear in $\mathcal{M}$.

Call any such $\langle \mathcal{M}, \mathbf{X} \mapsto \mathbf{Y} \rangle$ an *UODI candidate*. Call any such $\langle \mathcal{M}, \mathbf{X} \mapsto \mathbf{Y} \rangle$ for which $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ an *UODI instance*.

UODI is the collection of UODI instances. This is the set-theoretic characterization of *the inference decision problem for UODs*.

Reduction from 3-SAT. Construction.

$\mathcal{M}_{\mathcal{F}}$ is constructed as follows. For each p_i, $i \in \{1, \ldots, n\}$, from $\mathcal{F}$, we introduce four attributes to appear in $\mathcal{M}_{\mathcal{F}}$: $\mathsf{P}_{i,t}$, $\mathsf{P}_{i,f}$, $\mathsf{Q}_{i,t}$, and $\mathsf{Q}_{i,f}$. (Our intent is that $[\mathsf{P}_{i,t}, \mathsf{P}_{i,f}]$ will mirror the truth value of p_i from $\mathcal{F}$ in a given truth assigment, and $[\mathsf{Q}_{i,t}, \mathsf{Q}_{i,f}]$ will mirror the truth value of $\neg p_i$ in that truth assigment.)

For $i \in \{1, \ldots, n\}$, add the following order dependencies for $\mathsf{P}_{i,t}$ and $\mathsf{P}_{i,f}$ to $\mathcal{M}_{\mathcal{F}}$:

1. $[\mathsf{P}_{i,t}] \sim [\mathsf{T}]$
2. $[\mathsf{P}_{i,f}] \sim [\mathsf{F}]$
3. $[\mathsf{P}_{i,t}] \sim [\mathsf{P}_{i,f}]$
4. $[\mathsf{P}_{i,t}, \mathsf{P}_{i,f}, \mathsf{T}] \sim [\mathsf{P}_{i,t}, \mathsf{P}_{i,f}, \mathsf{F}]$

Likewise, for $i \in \{1, \ldots, n\}$, symmetrically add the "same" order dependencies for $\mathsf{Q}_{i,t}$ and $\mathsf{Q}_{i,f}$ to $\mathcal{M}_{\mathcal{F}}$:

5. $[\mathsf{Q}_{i,t}] \sim [\mathsf{T}]$
6. $[\mathsf{Q}_{i,f}] \sim [\mathsf{F}]$
7. $[\mathsf{Q}_{i,t}] \sim [\mathsf{Q}_{i,f}]$
8. $[\mathsf{Q}_{i,t}, \mathsf{Q}_{i,f}, \mathsf{T}] \sim [\mathsf{Q}_{i,t}, \mathsf{Q}_{i,f}, \mathsf{F}]$

For $i \in \{1, \ldots, n\}$, add to $\mathcal{M}_{\mathcal{F}}$:

9. $[\mathsf{P}_{i,t}, \mathsf{Q}_{i,t}, \mathsf{T}] \sim [\mathsf{P}_{i,t}, \mathsf{Q}_{i,t}, \mathsf{F}]$

10. $[\mathsf{P}_{i,f}, \mathsf{Q}_{i,f}, \mathsf{T}] \sim [\mathsf{P}_{i,f}, \mathsf{Q}_{i,f}, \mathsf{F}]$

Next, we encode the clauses. For each clause, $i \in \{1, \ldots, k\}$, from $\mathcal{F}$, we introduce three attributes: $\mathsf{V}_{i,1}$, $\mathsf{V}_{i,2}$, and $\mathsf{V}_{i,3}$. For $i \in \{1, \ldots, k\}$, $j \in \{1, \ldots, 3\}$, add one OD to $\mathcal{M}_{\mathcal{F}}$ as follows. If $V_{i,j} = p_l$ (for a given $l \in \{1, \ldots, n\}$) in $\mathcal{F}$, add to $\mathcal{M}_{\mathcal{F}}$:

11. $[\mathsf{V}_{i,j}] \sim [\mathsf{P}_{l,t}, \mathsf{P}_{l,f}]$

Else, $V_{i,j} = \neg p_l$ (for a given $l \in \{1, \ldots, n\}$) in $\mathcal{F}$; add to $\mathcal{M}_{\mathcal{F}}$:

12. $[\mathsf{V}_{i,j}] \sim [\mathsf{Q}_{l,t}, \mathsf{Q}_{l,f}]$

Finally, for each clause $i \in \{1, \ldots, k\}$ in $\mathcal{F}$, we introduce an attribute C_i, and we add to $\mathcal{M}_{\mathcal{F}}$:

13. $[\mathsf{C}_i] \mapsto [\mathsf{T}]$

14. $[\mathsf{C}_i] \mapsto [\mathsf{V}_{i,1}, \mathsf{V}_{i,2}, \mathsf{V}_{i,3}, \mathsf{F}]$

Polynomial reduction.

The translation procedure above of a 3-SAT candidate into an UODI candidate is clearly polynomial in the size of $\mathcal{F}$.

Witness.

We can build a counter-example for a given UODI candidate to demonstrate that it is *not* an UODI instance, in **UODI**. A pair of tuples is necessary and sufficient to falsify $[\mathsf{T}] \sim [\mathsf{F}]$. Therefore, $\mathcal{M}_{\mathcal{F}} \not\models [\mathsf{T}] \sim [\mathsf{F}]$ *iff* we can construct a two-tuple table **t** over

the attributes appearing in $\mathcal{M}_{\mathcal{F}}$ that falsifies $[\mathsf{T}] \sim [\mathsf{F}]$, but that does not falsify any order dependency in $\mathcal{M}_{\mathcal{F}}$ (thus *satisfies* $\mathcal{M}_{\mathcal{F}}$). Between the two tuples in $\mathbf{t}$, T will have different values, F will have different values, and the values of T and F will be anti-monotonic. Let the two values for T and for F in $\mathbf{t}$ be 0 and 1, without loss of generality. We write the tuples in $\mathbf{t}$ in a fixed order in our discussion such that $t_{\mathsf{T},\mathsf{F}} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, without loss of generality. Conceptually, a transition from 0 to 1, as in $t_{\mathsf{T}} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$, encodes *true*; a transition from 1 to 0, as in $t_{\mathsf{F}} = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$, represents *false*.

We can always build a two-tuple table $\mathbf{t}$ such that $t_{\mathsf{T},\mathsf{F}} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ (which is necessary and sufficient to falsify $[\mathsf{T}] \sim [\mathsf{F}]$) which satisfies ODs 1–13 of $\mathcal{M}_{\mathcal{F}}$. Let us construct such a $\mathbf{t}$. Because of OD 1, $t_{\mathsf{P}_{i,t}} = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$. (If only a single value appears in $\mathbf{t}$ for an attribute, we can assume that value is 0, without loss of generality.) Because of OD 2, $t_{\mathsf{P}_{i,f}} = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$. Because of OD 3, $t_{\mathsf{P}_{i,t},\mathsf{P}_{i,f}} \neq \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$. Because of OD 4, $t_{\mathsf{P}_{i,t},\mathsf{P}_{i,f}} \neq \left[\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right]$. (Otherwise, OD 4 would be falsified by $\mathbf{t}$, since $t_{\mathsf{T},\mathsf{F}} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$.) Therefore, $t_{\mathsf{P}_{i,t},\mathsf{P}_{i,f}} = \left[\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right]$.

From ODs 5–8, it symmetrically follows that $t_{\mathsf{Q}_{i,t},\mathsf{Q}_{i,f}} = \left[\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right]$.

From ODs 9–10, it further follows that

$$t_{\mathsf{P}_{i,t},\mathsf{P}_{i,f},\mathsf{Q}_{i,t},\mathsf{Q}_{i,f}} = \left[\begin{smallmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{smallmatrix}\right]$$

or

$$t_{\mathsf{P}_{i,t},\mathsf{P}_{i,f},\mathsf{Q}_{i,t},\mathsf{Q}_{i,f}} = \left[\begin{smallmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{smallmatrix}\right].$$

Thus, $\mathbf{t} \models [\mathsf{P}_{i,t}, \mathsf{P}_{i,f}] \not\sim [\mathsf{Q}_{i,t}, \mathsf{Q}_{i,f}]$.

For any $\mathsf{V}_{i,j}$ such that $V_{i,j} = p_l$ for a given l in $\mathcal{F}$, so OD 11 is in $\mathcal{M}_{\mathcal{F}}$ for i, we know the following:

- if $t_{\mathsf{P}_{l,t},\mathsf{P}_{l,f}} = \left[\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right]$, then $t_{\mathsf{V}_{i,j}} = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$;
- else $t_{\mathsf{P}_{l,t},\mathsf{P}_{l,f}} = \left[\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right]$ and $t_{\mathsf{V}_{i,j}} = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$.

For any $\mathsf{V}_{i,j}$ such that $V_{i,j} = \neg p_l$ for a given l in $\mathcal{F}$ *instead*, so OD 12 is in $\mathcal{M}_{\mathcal{F}}$ for i, we know the following:

- if $t_{\mathsf{P}_{l,t},\mathsf{P}_{l,f}} = \left[\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right]$, then $t_{\mathsf{Q}_{l,t},\mathsf{Q}_{l,f}} = \left[\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right]$ and $t_{\mathsf{V}_{i,j}} = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$;
- else $t_{\mathsf{P}_{l,t},\mathsf{P}_{l,f}} = \left[\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right]$, $t_{\mathsf{Q}_{l,t},\mathsf{Q}_{l,f}} = \left[\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right]$ and $t_{\mathsf{V}_{i,j}} = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$ *or* $\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$.

To satisfy ODs 13, for $i \in \{1, \ldots, k\}$, it must be that $t_{\mathsf{C}_i} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$ since $t_{\mathsf{T}} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$.

In coNP.

It is not always possible further to set values for the $\mathsf{V}_{i,j}$'s in such a way that **t** also satisfies the ODs 14, for $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, 3\}$, and so satisfies $\mathcal{M}_{\mathcal{F}}$ completely. When we can also set values for the $\mathsf{V}_{i,j}$'s so that **t** also satisfies the ODs 14 too, then **t** suffices as a *witness* that $\langle \mathcal{M}_{\mathcal{F}}, [\mathsf{T}] \sim [\mathsf{F}] \rangle \notin$ **UODI**.

Correspondence.

$\mathcal{F} \in$ **3-SAT** *iff* $\langle \mathcal{M}_{\mathcal{F}}, [\mathsf{T}] \sim [\mathsf{F}] \rangle \notin$ **UODI**.

Consider two-tuple tables **t** that satisfy the ODs 1–10 and 13 from $\mathcal{M}_{\mathcal{F}}$, but that falsify $[\mathsf{T}] \sim [\mathsf{F}]$. There is a one-to-one mapping between truth assigments over the p_i, for $i \in \{1, \ldots, n\}$, in $\mathcal{F}$ and settings for $\mathsf{P}_{i,t}$ in such **t**. For $i \in \{1, \ldots, n\}$, if $p_i = \mathit{true}$ in the truth assignment, set

$$t_{\mathsf{P}_{i,t},\mathsf{P}_{i,f},\mathsf{Q}_{i,t},\mathsf{Q}_{i,f}} = \left[\begin{smallmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{smallmatrix}\right];$$

else ($p_i = \mathit{false}$), set

$$t_{\mathsf{P}_{i,t},\mathsf{P}_{i,f},\mathsf{Q}_{i,t},\mathsf{Q}_{i,f}} = \left[\begin{smallmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{smallmatrix}\right].$$

IF: There is some truth assigment over $p_1, \ldots, p_n$ that satisfies $\mathcal{F}$.

We construct a two-tuple table **t** based on this truth assignment that satisfies $\mathcal{M}_{\mathcal{F}}$ for ODs 1–13, and that falsifies [T] $\sim$ [F], as above (in the *Witness* part). For $i \in \{1, \ldots, n\}$, assign values for $\mathsf{P}_{i,t}$, $\mathsf{P}_{i,f}$, $\mathsf{Q}_{i,t}$, and $\mathsf{Q}_{i,f}$ according to the truth assignment mapping above.

To satisfy further ODs 14, we must be able to assign values to the $\mathsf{V}_{i,j}$'s that suffice. For $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, 3\}$, if $V_{i,j} = \mathit{true}$, set $t_{\mathsf{V}_{i,j}} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$. This satisfies the OD 11 or 12 added to $\mathcal{M}_{\mathcal{F}}$ for i, given how we assigned $\mathsf{P}_{i,t}$, $\mathsf{P}_{i,f}$, $\mathsf{Q}_{i,t}$, and $\mathsf{Q}_{i,f}$ based on p_i's truth value. Otherwise ($V_{i,j} = \mathit{false}$), set $t_{\mathsf{V}_{i,j}} = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$. This satisfies either the OD 11 or 12 for i, j, vacuously.

Since, for each $i \in \{1, \ldots, k\}$, at least one of $V_{i,1}$, $V_{i,2}$, and $V_{i,3}$ is *true* in the truth assignment, at least one of $t_{\mathsf{V}_{i,1}}$, $t_{\mathsf{V}_{i,2}}$, or $t_{\mathsf{V}_{i,3}}$ is $\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$. Thus, **t** as constructed satisfies ODs $1-14$, and so all of $\mathcal{M}_{\mathcal{F}}$.

ONLY IF: There is no truth assignment that satisfies $\mathcal{F}$.

For any arbitrary truth assignment, we can build a two-tuple table **t** that falsifies [T] $\sim$ [F] based on the truth assignment mapping that satisfies ODs 1–13, as done in the *if* part. We next try to assign values to the $\mathsf{V}_{i,j}$'s in such a way that **t** satisfies ODs 14.

Since the truth assignment does not satisfy $\mathcal{F}$, there is some clause i such that $V_{i,1}$, $V_{i,2}$, and $V_{i,3}$ are each *false*. The OD 14 for i will be falsified. For each $V_{i,j}$, as either the OD 11 or 12 is satisfied accordingly, $t_{\mathsf{V}_{i,j}} = [\begin{smallmatrix}0\\0\end{smallmatrix}]$ *or* $[\begin{smallmatrix}1\\0\end{smallmatrix}]$.

If, for any $V_{i,j}$, $t_{\mathsf{V}_{i,j}} = [\begin{smallmatrix}1\\0\end{smallmatrix}]$, OD 14 is falsified since $t_{\mathsf{C}_i} = [\begin{smallmatrix}0\\1\end{smallmatrix}]$. If instead, for all $V_{i,j}$, $t_{\mathsf{V}_{i,j}} = [\begin{smallmatrix}0\\0\end{smallmatrix}]$, OD 14 is still falsified, since $t_{\mathsf{F}} = [\begin{smallmatrix}1\\0\end{smallmatrix}]$.

No two-tuple table $\mathbf{t}$ that falsifies $[\mathsf{T}] \sim [\mathsf{F}]$ can be constructed that satisfies $\mathcal{M}_{\mathcal{F}}$. Any table $\mathbf{t}$ therefore either satisfies $[\mathsf{T}] \sim [\mathsf{F}]$ or falsifies $\mathcal{M}_{\mathcal{F}}$. □

Theorem 23 *(single OD)*

$\boldsymbol{X} \sim \boldsymbol{Y}$ *holds iff* $\boldsymbol{XY} \mapsto \boldsymbol{Y}$.

Proof

IF: By Reflexivity axiom, OD $\mathbf{YX} \mapsto \mathbf{Y}$ is true. Therefore, by Transitivity, $\mathbf{XY} \mapsto \mathbf{Y}$.

ONLY IF: By Suffix axiom, $\mathbf{XY} \leftrightarrow \mathbf{YXY}$ is true. Therefore, by Normalization and Transitivity, $\mathbf{XY} \sim \mathbf{YX}$. □

Theorem 24

Given a set $\mathcal{M}$ of UODs and UOD $\boldsymbol{X} \mapsto \boldsymbol{Y}$, deciding whether $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$ is co-NP-complete.

Proof

By Theorem 23, order compatible $\mathbf{X} \sim \mathbf{Y}$ is equivalent to a single UOD. Therefore, by Lemma 21, deciding whether $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ is co-NP-hard.

Witness

Any counter example for a given UOD $\mathbf{X} \mapsto \mathbf{Y}$is a pair of tuples (that can be checked in polynomial time). This is necessary and sufficient to falsify $\mathbf{X} \mapsto \mathbf{Y}$, by the definitions of split and swap (Definitions 13 and 14).

Thus, deciding $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ is co-NP-complete. □

4.3.2 FD inference over ODs

Corollary 25

Given a set $\mathcal{M}$ of ODs and OD $\mathbf{X} \mapsto \mathbf{Y}$, deciding whether $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ is co-NP-complete.

Proof

Hardness, follows directly from Theorem 24 as a class of UODs is a proper sub-class of ODs. (Any *witness* that $\mathcal{M} \not\models \mathbf{X} \mapsto \mathbf{Y}$ is a pair of tuples (that can be checked in polynomial time) by definitions of split and swap (Definitions 13 and 14). □

Functional-dependency inference for UODs is polynomial. In fact, it can be done in linear time (Theorem 26). This does not contradict Theorem 24; the type of inference that is *hard* for unidirectional ODs is order compatibility, $\mathbf{X} \sim \mathbf{Y}$ (as shown in Lemma 21 and Theorem 24).

Let the length of the representation of $\mathcal{M}$, the string of concatenated left-hand and right-hand sides of the ODs, be denoted by $|\mathcal{M}|$.

Theorem 26 *(FDs over UODs)*

Let $\mathcal{M}$ be a set of UODs. Testing whether $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{XY}$ ($\mathcal{M} \models \mathcal{X} \rightarrow \mathcal{Y}$) can be accomplished in $O(|\mathcal{M}|)$ time. (This includes finding the closure for FDs, $\mathcal{X}^+$.)

Proof

Assume $\mathcal{M}' = \{\mathbf{X} \mapsto \mathbf{XY}, \mathbf{XY} \leftrightarrow \mathbf{YX} \mid \mathbf{X} \mapsto \mathbf{Y} \in \mathcal{M}\}$. In [39], we have shown that $\mathcal{F} = \{\mathcal{X} \rightarrow \mathcal{Y} \mid \mathbf{X} \mapsto \mathbf{Y} \in \mathcal{M}'\}$ is a set of FDs which enables one to compute the closure for FDs $\mathcal{X}^+$ over the set of UODs $\mathcal{M}$. Testing logical implication of a FD $\mathcal{X} \rightarrow \mathcal{Y}$ over a set of prescribed FDs has already been shown to be linear in [3]. This implies that testing $\mathcal{M} \models \mathcal{X} \rightarrow \mathcal{Y}$ can be also accomplished in $O(|\mathcal{M}|)$. The same applies to $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{XY}$ by Theorem 14. □

This is not the same case, however, for bidirectional order dependencies. Both the inference problems for functional dependencies (embedded within the ODs), $\mathbf{X} \mapsto \mathbf{XY}$, and for order compatibility, $\mathbf{X} \sim \mathbf{Y}$, are *hard.*

We call an attribute a *constant* if, for any table that satisfies the set of ODs $\mathcal{M}$, it can have only a single value occurring in the table.

Definition 24 (constant)

A marked attribute A *is called a* constant *with respect to* $\mathcal{M}$ iff $\mathcal{M} \models [] \mapsto \mathsf{A}$.

Lemma 22

Given a set $\mathcal{M}$ of ODs and a functional dependency $\{\} \rightarrow \mathsf{A}$, deciding whether $\mathcal{M} \models$

$\{\} \rightarrow \mathsf{A}$ *is co-NP-complete.*

Proof

Let $\langle \mathcal{M}, \mathbf{X} \mapsto \mathbf{Y} \rangle$ be an arbitrary pair of a finite set $\mathcal{M}$ of ODs and an *target* UOD $\mathbf{X} \mapsto \mathbf{Y}$ constructed over the attributes that appear in $\mathcal{M}$.

Call any such $\langle \mathcal{M}, \mathbf{X} \mapsto \mathbf{Y} \rangle$ an *ODI candidate.* Call any such $\langle \mathcal{M}, \mathbf{X} \mapsto \mathbf{Y} \rangle$ for which $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ an *ODI instance.*

It suffices to show that any 3-SAT candidate (Definition 23) can be reduced to an ODI candidate of the form $\langle \mathcal{M}, \mathbf{X} \mapsto \mathbf{XY} \rangle$. We reduce any 3-SAT candidate to an ODI candidate of the form $\langle \mathcal{M}, [\,] \mapsto [\overrightarrow{\mathsf{T}}] \rangle$.[11]

ODI is the collection of ODI instances. This is the set-theoretic characterization of *the inference decision problem for ODs.*

Construct an ODI candidate from a given 3-SAT candidate in the very way the UODI candidate—also an ODI candidate—is constructed in Lemma 21. (Recall every unmarked attribute in ODs 1–14 is ascending, by default.) Add one more OD to $\mathcal{M}$:

15. $[\overrightarrow{\mathsf{F}}] \mapsto [\overleftarrow{\mathsf{T}}]$.

Witness

A two-tuple table $\mathbf{t}$ is a necessary and sufficient *witness* that $\mathcal{M} \not\models [\,] \mapsto [\overrightarrow{\mathsf{T}}]$. Let

[11] A simpler reduction from 3-SAT to ODI is possible which suffices. A single pair of attributes can be forced to be anti-monotonic: $[\overrightarrow{\mathsf{P}}] \leftrightarrow [\overleftarrow{\mathsf{Q}}]$, just as with T and F. Then, P_i and Q_i can be used to encode the truth assignment of p_i. Given that we preceded with Theorem 24, however, the proof given here is more concise.

Table 4.9: Table template.

#	X_1	...	X_k	attributes($\mathcal{M}$) − {X_1,...,X_k}		
s	b_1	...	b_k	p_{k+1}	...	p_n
t	b_1	...	b_k	q_{k+1}	...	q_n

(a) Template $\mathbf{r}_0$.

#	X_1	...	X_{j-1}	X_j	attributes($\mathcal{M}$) − {X_1,...,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

(b) Template $\mathbf{r}_j$.

$t_{\mathsf{T}} = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$, without loss of generality. Then, $t_{\mathsf{F}} = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$, given $\mathbf{t}$ satisfies OD 15.

The rest of the proof then proceeds the same as for Lemma 21. □

Theorem 27

Given a set $\mathcal{M}$ of ODs and a functional dependency $\mathcal{X} \rightarrow \mathcal{Y}$, deciding whether $\mathcal{M} \models \mathcal{X} \rightarrow \mathcal{Y}$ is co-NP-complete.

Proof

By Theorem 14 $\mathcal{X} \rightarrow \mathcal{Y}$ iff $\mathbf{X} \mapsto \mathbf{XY}$, for any list $\mathbf{X}$ that orders the attributes of $\mathcal{X}$ and any list $\mathbf{Y}$ that orders for $\mathcal{Y}$. Therefore, by Theorem 22, deciding whether $\mathcal{M} \models \mathcal{X} \rightarrow \mathcal{Y}$ is co-NP-complete as we can always construct a *witness* (that can be checked in polynomial time) that $\mathcal{M} \not\models \mathcal{X} \rightarrow \mathcal{Y}$, by the definition of split (Definition 13). □

Table 4.10: Mapping.

#	A	B	C
s	b_1	b_3	t_4
t	b_1	t_3	b_4

(a) $\mathbf{r}_m$.

#	A	B	C
s	5	0	8
t	5	1	7

(b) Instance $\varphi(\mathbf{r}_m)$.

4.4 Inference Procedures

A goal in any dependency theory is to develop algorithms for testing logical implication. We show how to test logical implication for ODs with an *elimination* procedure.[12] An elimination procedure is a reduction procedure for testing satisfaction of a data dependencies in a database. It is used to reason about the consistency and correctness of data design and in query optimization.

We next introduce an inference procedure for testing logical implication for a *restricted domain* for UODs. The additional order property to be guaranteed over the schema is intuitive, holds for all real-world business domains that we have encountered, and can easily be verified whether it holds for a given table. Thus, it is a natural restriction on the domain. We develop an inference procedure for UODs which is sound and

[12]In preliminary work [38] (see Appendix), we focused on *fixing* the table templates (Definition 25) with a *chase* procedure, whereas here, our technique is based on *detecting* with an *elimination procedure* the table templates which falsify the set of ODs $\mathcal{M}$. Therefore, this revised elimination procedure is simpler and more efficient. This also means the proof that follows is much more concise.

complete when applied over a database that satisfies the property. Our inference procedure is efficient, with a reasonable polynomial complexity bound with respect to schema size.

4.4.1 Elimination Procedure

We establish a sound and complete elimination procedure for ODs for *testing logical implication*, for which the complexity is exponential. This complexity is with respect to schema (Therefore, it can be used in practice.)

We define a *table template* over variables with respect to a given OD. We use these table templates to enumerate through all the possible cases where the OD can be falsified by splits and swaps.

Definition 25 (table template)

Let set of ODs $\mathcal{M}$ over relation **R** *have* n *unique attributes and* m *be an OD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$, *where* $\boldsymbol{X}$ *is over attributes* $\mathsf{X}_1, ..., \mathsf{X}_k$. *A table template for OD* m, *denoted as* $\mathbf{r}_m$, *is a table consisting of two tuples* s *and* t, *such that it is either* $\mathbf{r}_0$ *(Table 4.9a) or* $\mathbf{r}_j$ *(Table 4.9b), for* j *in* $[1, ..., k]$. *In* $\mathbf{r}_0$ *and* $\mathbf{r}_j$, *symbols* p_i *and* q_i *represent one of the following three cases, where the* ordering *of variables* b_i *and* t_i *is defined as* $b_i < t_i$:

1. $p_i = b_i$ *and* $q_i = b_i$;
2. $p_i = b_i$ *and* $q_i = t_i$; *and*
3. $p_i = t_i$ *and* $q_i = b_i$.

Example 39 presents how to apply a *mapping* (Definition 26) to a table template.

Definition 26 (mapping $\mathbf{r}_m$ to $\varphi(\mathbf{r}_m)$)

Let $\mathbf{r}_m$ *be a table template from Definition 25. A mapping of* $\mathbf{r}_m$ *to* $\varphi(\mathbf{r}_m)$ *is* any *instance with values that satisfy the* ordering *from Definition 25.*

Example 39

Consider Tables 4.10a and 4.10b as one of the possible mappings from Definition 26. In fact, it can be any relational instance which satisfies the Definition 25 for ordering *of the variables. (The* ordering *of variables* b_i *and* t_i *is defined as* $b_i < t_i$.*)*

Lemma 23

Let $\mathbf{r}_m$ *be a table template (Definition 25) and* $\varphi(\mathbf{r}_m)$ *be a mapping from* $\mathbf{r}_m$ *(Definition 26). Then* $\mathbf{r}_m \models \boldsymbol{X} \mapsto \boldsymbol{Y}$ *iff* $\varphi(\mathbf{r}_m) \models \boldsymbol{X} \mapsto \boldsymbol{Y}$.

Proof

By Definition 26, ordering of values in $\varphi(\mathbf{r}_m)$ corresponds to the ordering of variables in $\mathbf{r}_m$, respectively. □

Definition 27 (tableaux $\mathbf{T}_m$)

Let m *be an OD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$. *We define* $\mathbf{T}_m$ *to be the set of all table templates* $\mathbf{r}_m$, *as we defined in Definition 25.*

Note that $\mathbf{T}_m$ is a *set* of table templates, each consisting of two rows. The *elimination* of $\mathbf{T}_m$ is defined as follows.

Definition 28 (elimination of tableaux $\mathbf{T}_m$)

The elimination of $\mathbf{T}_m$ *over a set of order dependencies* $\mathcal{M}$ *denoted as* $ELIM_{\mathbf{T}_m,\mathcal{M}}$ *is defined by* $ELIM_{\mathbf{T}_m,\mathcal{M}} = \{\mathbf{r}_m \mid \mathbf{r}_m \in \mathbf{T}_m \wedge \mathbf{r}_m \models \mathcal{M}\}$. *Furthermore,* $ELIM_{\mathbf{T}_m,\mathcal{M}}$ *satisfies* $\boldsymbol{X} \mapsto \boldsymbol{Y}$, *denoted by* $ELIM_{\mathbf{T}_m,\mathcal{M}} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$, iff, *for all* $\mathbf{r}_m \in ELIM_{\mathbf{T}_m,\mathcal{M}}$, $\mathbf{r}_m \models \boldsymbol{X} \mapsto \boldsymbol{Y}$. $ELIM_{\mathbf{T}_m,\mathcal{M}}$ *satisfies the set of ODs* $\mathcal{M}'$, *which is denoted as* $ELIM_{\mathbf{T}_m,\mathcal{M}} \models \mathcal{M}'$, iff, *for all* $\boldsymbol{X} \mapsto \boldsymbol{Y} \in \mathcal{M}'$, $ELIM_{\mathbf{T}_m,\mathcal{M}} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$.

Theorem 28 *(elimination procedure for ODs is sound and complete)*

Let $\mathcal{M}$ *be a set of ODs over* $\mathbf{R}$ *and m be an OD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$. *Then* $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$ *iff* $ELIM_{\mathbf{T}_m,\mathcal{M}} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$.

Proof

IF: Assume $\text{ELIM}_{\mathbf{T}_m,\mathcal{M}} \not\models \mathbf{X} \mapsto \mathbf{Y}$. By Definition 28, there exists $\mathbf{r}_m \in \text{ELIM}_{\mathbf{T}_m,\mathcal{M}}$ such that $\mathbf{r}_m \not\models \mathbf{X} \mapsto \mathbf{Y}$. By Definition 28, $\mathbf{r}_m \models \mathcal{M}$. Hence, there is a mapping φ to generate a relation instance $\varphi(\mathbf{r}_m)$. By Lemma 23, $\varphi(\mathbf{r}_m) \models \mathcal{M}$, but in addition $\varphi(\mathbf{r}_m) \not\models \mathbf{X} \mapsto \mathbf{Y}$. We have found a relation instance which satisfies $\mathcal{M}$ but does not satisfy $\mathbf{X} \mapsto \mathbf{Y}$, which implies that $\mathcal{M} \not\models \mathbf{X} \mapsto \mathbf{Y}$.

ONLY IF: Assume $\text{ELIM}_{\mathbf{T}_m,\mathcal{M}} \models \mathbf{X} \mapsto \mathbf{Y}$. Let s and t be *any* two tuples in *any* relation $\mathbf{r}$ such that $s \preceq_{\mathbf{X}} t$ and that satisfies the set of ODs $\mathcal{M}$. We would like to present that $s \preceq_{\mathbf{Y}} t$. Let $\mathbf{r}_m \in \mathbf{T}_m$. Let $\mathbf{r}_m = \{p, q\}$ be the template relation such that $\varphi(p) = s$ and $\varphi(q) = t$. 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 condition $s \preceq_{\mathbf{X}} t$. Therefore, we have $\varphi(\mathbf{r}_m) = \{s, t\}$ and

$\varphi(\mathbf{r}_m) \models \mathcal{M}$. By Lemma 23, it follows that $\mathbf{r}_m \models \mathcal{M}$. It follows by Definition 28 that $\mathbf{r}_m \in \text{ELIM}_{\mathbf{T}_m,\mathcal{M}}$. Since we assumed that $\text{ELIM}_{\mathbf{T}_m,\mathcal{M}} \models \mathbf{X} \mapsto \mathbf{Y}$, we have $\mathbf{r}_m \models \mathbf{X} \mapsto \mathbf{Y}$. This implies that $\varphi(\mathbf{r}_m) \models \mathbf{X} \mapsto \mathbf{Y}$ by Lemma 23. Hence, $s \preceq_{\mathbf{Y}} t$. □

Proof

By Theorem 28, testing logical implication problem of ODs is decidable as the elimination procedure is a sound and complete inference algorithm for ODs. □

Theorem 29 *(complexity of elimination)*

The complexity of building templates for the ODs elimination procedure is $\mathrm{O}(3^n)$. *(Note this is schema complexity.)*

Proof

By Definition 25, there are 3^{n-k} templates for $\mathbf{r}_0$ and 3^{n-j} templates for each $\mathbf{r}_j$. Therefore, there are $(3^n + 3^{n-k})/2$ templates in total. By geometric progression: $O(3^n)$. □

We have implemented elimination procedure in IBM DB2. Our experiments have shown that the time of running it for real world business domains for which the number of unique attributes in the set of prescribed ODs is from 5 to 12 is marginal.

4.4.2 Chase Procedure

We show how to test logical implication for ODs using *chase* procedure. Chase is a fixpoint algorithm enforcing satisfaction of data dependencies in databases. The chase

algorithm is used to reason about consistency and correctness of data design and in query optimization to rewrite queries.

Definition 29 (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 40

Consider Table 4.11a and Table 4.11b as an example of an operation equalize.

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

Definition 30 (chase rules)

Let s and t be two tuples in relational instance **r**, *and let* ***X*** *and* ***Y*** *be lists of marked attributes such that* $\boldsymbol{X} \mapsto \boldsymbol{Y}$ *is falsified in* **r** ($s_{\boldsymbol{X}} \preceq t_{\boldsymbol{X}}$ *but* $s_{\boldsymbol{Y}} \not\preceq t_{\boldsymbol{Y}}$). *Also, 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* $s_B \neq t_B$ *Two chase rules are defined.*

4. ***Split rule:*** *if* $s_{\boldsymbol{X}} = t_{\boldsymbol{X}}$, *then* **r**′ = *equalize(***r**, B, *s, t).*

5. ***Swap rule:*** *if* $s_A \neq t_A$ *and* $s_B \neq t_B$, *then* **r**′ = *equalize(***r**, B, *s, t).*

Table 4.11: Operation equalize.

#	A
s	0
t	1

(a) $\mathbf{r} = \{s, t\}$.

#	A
s	0
t	0

(b) $\mathbf{r}' = \text{equalize}(\mathbf{r}, \mathsf{A}, s, t)$.

Example 41

Let $\mathcal{M} = \{\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}},\ \overrightarrow{\mathsf{B}} \mapsto \overleftarrow{\mathsf{C}}\}$ *and let* **r** *be an instance over* **R** *with attributes* $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. *From Table 4.12a to Table 4.12b demonstrates an example of applying the split rule (*$\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}}$ *is falsified in* **r** *in Table 4.12a). From Table 4.12b to Table 4.12c demonstrates applying the swap rule (*$\overrightarrow{\mathsf{B}} \mapsto \overleftarrow{\mathsf{C}}$ *is falsified in* $\mathbf{r}'$ *in Table 4.12b).*

The chase algorithm is as follows (Algorithm 4).

Example 42

Let $\mathcal{M} = \{\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}},\ \overrightarrow{\mathsf{B}} \mapsto \overleftarrow{\mathsf{C}}\}$ *be a set of ODs and* **r** *be an instance over* **R** *with attributes* $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. *The sequence from Table 4.12a, Table 4.12b, to Table 4.12c is an example of applying chase algorithm (Algorithm 4) to* **r** *as in Table . In Table 4.12c, there is no s and t that matches (as in step 5 of Algorithm 4), so the procedure terminates with it.*

Algorithm 4 Chase procedure.

Input: A set of prescribed ODs $\mathcal{M}$ and an instance relation **r** over **R**.

Output: Table $Current$

1: $Current \leftarrow \mathbf{r}$

2: $Previous \leftarrow \{\}$

3: **while** $Current \neq Previous$ **do**

4: $Current \leftarrow Previous$

5: **if** $\exists\, s, t \in Current, \exists\, \mathbf{X} \mapsto \mathbf{Y} \in \mathcal{M}$ such that $s_{\mathbf{X}} \preceq t_{\mathbf{X}}$ but $s_{\mathbf{Y}} \not\preceq t_{\mathbf{Y}}$ **then**

6: Apply one of the chase rules (split or swap from Definition 30) to s and t, assigning the table which is returned from *equalize* operation to $Current$.

7: **return** $Current$

Lemma 24 *(*termination and satisfaction*)*

Algorithm 4 terminates and the resulting table of Algorithm 4 satisfies set of ODs $\mathcal{M}$.

Proof

Consider a given relation 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 $\Sigma_{\mathbf{s}}$ 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 $\Sigma_{\mathbf{s}'} < \Sigma_{\mathbf{s}}$ as the *equalize* replaced a value in some column of

Table 4.12: Chase algorithm.

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

(a) $\mathbf{r} \not\models \mathcal{M}$.

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

(b) $\mathbf{r} \not\models \mathcal{M}$, using split rule for rows s and t.

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

(c) chase($\mathbf{r}$, $\mathcal{M}$) $\models \mathcal{M}$, using swap rule for rows t and u.

some row by a smaller value. (Note that *equalize* does not introduce new values.) Zero is the lower bound on the $\Sigma_{s'}$. As a chase procedure is a finite sequence of transformation starting with **r**, it must terminate.

The remaining step is to show that resulting table of Algorithm 4 satisfies set of ODs $\mathcal{M}$. 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 30

Let relation instance **r** *be over* **R** *and let* $\mathcal{M}$ *be a set of ODs. Then* $\mathbf{r} \models \mathcal{M}$ *iff* $\mathbf{r}$ = *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**′, $\Sigma_{s'} < \Sigma_s$. Then clearly $\mathbf{s} \neq \mathbf{s}'$. Thus, $\mathbf{r} = \text{chase}(\mathbf{r}, \mathcal{M})$ if no swap or split applies, meaning $\mathbf{r} \models \mathcal{M}$.

ONLY IF: From Definition 30 it follows that chase rules *split* and *swap* are only used if they break a dependency in $\mathcal{M}$. □

Please note that we apply chase rules *split* and *swap* on table templates using *ordering* from Definition 25 (Section 4.2).

Lemma 25

Let $\mathbf{r}_m$ *be a table template from Definition 25, where m is an OD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$. *Then,* $\mathbf{r}_m \models \mathcal{M}$ *iff* $\mathbf{r}_m = chase(\mathbf{r}_m, \mathcal{M})$.

Proof

The proof follows directly from Theorem 30 by replacing $\mathbf{r}_m$ with **r** and applying chase rules *split* and *swap* on variables, using ordering defined in Definition 25. □

Lemma 26

Let $\mathbf{r}_m$ *be a table template from Definition 25 and* $\varphi(\mathbf{r}_m)$ *be mapping from* $\mathbf{r}_m$*(Definition 26). Then,* $\mathbf{r}_m \models \boldsymbol{X} \mapsto \boldsymbol{Y}$ *iff* $\varphi(\mathbf{r}_m) \models \boldsymbol{X} \mapsto \boldsymbol{Y}$.

Proof

The proof follows from the definition of ordering of variables in Definition 25. Since

ordering of values in $\varphi(\mathbf{r}_m)$ corresponds with ordering of variables in $\varphi(\mathbf{r}_m)$ respectively (Definition 26). □

Definition 31 (tableaux $\mathbf{T}_m$)

Let m be an OD $\boldsymbol{X} \mapsto \boldsymbol{Y}$. *We define* $\mathbf{T}_m$ *to be the set of all table templates* $\mathbf{r}_m$, *as defined in Definition 25.*

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 32 (chase of tableaux $\mathbf{T}_m$)

The chase of $\mathbf{T}_m$ *over a set of ODs* $\mathcal{M}$, *denoted as* $CHASE_{\mathbf{T}_m,\mathcal{M}}$ *is defined by* $CHASE_{\mathbf{T}_m,\mathcal{M}} = \{chase(\mathbf{r}, \mathcal{M}) \mid \mathbf{r}_m \in \mathbf{T}_m$. *Furthermore,* $CHASE_{\mathbf{T}_m,\mathcal{M}}$ *satisfies* $\boldsymbol{X} \mapsto \boldsymbol{Y}$, *denoted by* $CHASE_{\mathbf{T}_m,\mathcal{M}} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$, *iff, for all* $\mathbf{r}_m \in CHASE_{\mathbf{T}_m,\mathcal{M}}$, *chase*(**r**, $\mathcal{M}$) $\models \boldsymbol{X} \mapsto \boldsymbol{Y}$. $CHASE_{\mathbf{T}_m,\mathcal{M}}$ *satisfies the set of ODs* $\mathcal{M}'$, *which is denoted as* $CHASE_{\mathbf{T}_m,\mathcal{M}} \models \mathcal{M}'$, iff, *for all* $\boldsymbol{X} \mapsto \boldsymbol{Y} \in \mathcal{M}'$, $CHASE_{\mathbf{T}_m,\mathcal{M}} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$.

Theorem 31 *(chase procedure for ODs is sound and complete)*

Let $\mathcal{M}$ *be a set of ODs over* **R** *and m be an OD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$. *Then* $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$ *iff* $CHASE_{\mathbf{T}_m,\mathcal{M}} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$.

Proof

IF: Assume $\text{CHASE}_{\mathbf{T}_m,\mathcal{M}} \not\models \mathbf{X} \mapsto \mathbf{Y}$. By Definition 32, there exists $\mathbf{r}_m \in \mathbf{T}_m$, such

that chase($\mathbf{r}_m$, $\mathcal{M}$) $\not\models \mathbf{X} \mapsto \mathbf{Y}$. Note chase($\mathbf{r}_m$, $\mathcal{M}$) $\models \mathcal{M}$ by Lemma 24. Hence, there is a mapping φ to generate a relation instance $\varphi(chase(\mathbf{r}_m, \mathcal{M}))$ and by Lemma 26, $\varphi(chase(\mathbf{r}_m, \mathcal{M})) \models \mathcal{M}$, but $\varphi(chase(\mathbf{r}_m, \mathcal{M})) \not\models \mathbf{X} \mapsto \mathbf{Y}$. This implies that $\mathcal{M} \not\models \mathbf{X} \mapsto \mathbf{Y}$ because we have found a relation instance which satisfies $\mathcal{M}$ but does not satisfy $\mathbf{X} \mapsto \mathbf{Y}$. Therefore, if $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ then CHASE$_{\mathbf{T}_m,\mathcal{M}} \models \mathbf{X} \mapsto \mathbf{Y}$.

ONLY IF: Assume CHASE$_{\mathbf{T}_m,\mathcal{M}} \models \mathbf{X} \mapsto \mathbf{Y}$. Let s, t be *any* two tuples in relation $\mathbf{r}$ such that $s \preceq_{\mathbf{X}} t$ and that satisfies the set of ODs $\mathcal{M}$. We would like to present that $s \preceq_{\mathbf{Y}} t$. Let $\mathbf{r}_m \in \mathbf{T}_m$. Let $\mathbf{r}_m = \{p, q\}$ be the template relation such that $\varphi(p) = s$ and $\varphi(q) = t$. 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 condition $s \preceq_{\mathbf{X}} t$. Therefore, we have $\varphi(\mathbf{r}_m) = \{s, t\}$ and $\varphi(\mathbf{r}_m) \models \mathcal{M}$. By Lemma 26, it follows that $\mathbf{r}_m \models \mathcal{M}$. Therefore, it follows by Lemma 25 that $\mathbf{r}_m$ = chase($\mathbf{r}_m$, $\mathcal{M}$). Since we assumed that CHASE$_{\mathbf{T}_m,\mathcal{M}} \models \mathbf{X} \mapsto \mathbf{Y}$, we have chase($\mathbf{r}_m$, $\mathcal{M}$) $\models \mathbf{X} \mapsto \mathbf{Y}$. As $\varphi(\mathbf{r}_m) = \varphi(chase(\mathbf{r}_m, \mathcal{M}))$, it implies that $\varphi(\mathbf{r}_m) \models \mathbf{X} \mapsto \mathbf{Y}$ by Lemma 26. Hence, $s \preceq_{\mathbf{Y}} t$. □

4.4.3 A Natural Domain

In [32], an axiomatization for UODs (defined as we have in this thesis) over a restricted domain is presented. The authors call these *temporal functional dependencies* (TFDs), focusing on the time domain, and they provide axiomatization for TFDs. A TFD $\mathbf{X} \rightarrow \mathcal{Y}$ means that $\forall$ A $\in \mathcal{Y}$. $\mathbf{X} \mapsto$ [A], in which $\mathbf{X}$ describes time and the attributes in $\mathcal{Y}$

are time variants. The domain is too restricted, unfortunately, to be of use to us. It effectively restricts one to ODs of the form with just a single attribute on the right-hand side (e.g., $\mathbf{X} \mapsto$ [A]). In many of our examples, in particular, in Examples 2 and 3, we need UODs with lists of multiple attributes on the right-hand side. Thus, TFDs do not suffice. Furthermore, no inference procedure for TFDs was defined.

We can take the same tactic to find a natural property by which we can restrict our database domains to make the inference problem tractable, but still cover real-world domains. Our axiomatization in Chapter 2 yields us insight into how this can be done.

We observe that a relation satisfying the OD $\mathbf{X} \leftrightarrow \mathbf{Y}$ satisfies the OD $\mathbf{X} \sim \mathbf{Y}$, but conversely a relation which satisfies the OD $\mathbf{X} \sim \mathbf{Y}$ may not necessarily satisfy the OD $\mathbf{X} \leftrightarrow \mathbf{Y}$. The following example helps to illustrate that point.

Example 43 (Difference between two lists being order compatible and order equivalent.) *The order dependency* [month] $\sim$ [quarter] *is satisfied in table* date_dim. *On the other hand, order dependency* [month] $\leftrightarrow$ [quarter] *is falsified by table* date_dim.

It is surprising initially that the *order-compatibility* relation '$\sim$' (Definition 4) is *not* transitive as shown in Example 44. (By *Transitivity* axiom (Figure 4.1) the *order* relation ('$\mapsto$') is.)

Example 44 (Order compatibility is not transitive.)

Assume $\mathcal{M} = \{[A] \sim [B], [B] \sim [C]\}$. *The Table 4.13 satisfies the set of UODs* $\mathcal{M}$.

Table 4.13: Showing Lack of Transitivity.

A	B	C
0	0	1
1	0	0
2	2	2

However, it falsifies [A] $\sim$ [C]. *This demonstrates that the order-compatibility relation is* not *transitive.*

If we restrict our domains to have a property that guarantees a limited form of transitivity over *order-compatibility*, then we can make an efficient inference procedure for UODs.[13] The property we prescribe is *transitivity of order compatibility over single attributes*. Let us call a domain a *transitive* domain if it satisfies this property.

Definition 33 (transitivity of order compatibility)

A domain (relation schema) has the property of transitivity of order compatibility iff *it can be guaranteed that, for each relation* **R** *in the schema, for any three attributes* A, B, *and* C *where* B *is not a constant, if* [A] $\sim$ [B] *and* [B] $\sim$ [C], *then* [A] $\sim$ [C]. *If a domain has this property, we call it a* transitive *domain.*

Example 45 (Transitivity over order compatibility.)

[quarter] $\sim$ [month] *and* [month] $\sim$ [trimester] *are satisfied in Table 2.3 by Date*

[13] It is this lack of transitivity over the order-compatible relation generally that is at the heart of the high complexity for the inference problem over general domains.

domain. Also, so is [quarter] $\sim$ [trimester] *Hence, the transitivity property holds over order compatibility for this.*

All of the real-world business domains we have explored including the TPC-DS schema, and the examples which are used in this thesis are *transitive*, as by Definition 33. One can argue that breaking the underlying property in data can be only done by contrivance. Thus, this restriction is quite *natural*, as it covers the cases one sees in practice. Domains can be tested if they are transitive in a straightforward way, by enumeration.

4.4.4 An Inference Procedure for UODs over Natural Domains

Let $\mathcal{M} = \{\sigma_0, ..., \sigma_{n-1}\}$ be a set of UODs defined over the set of attributes $\mathcal{U} = \{\mathsf{A}_0, ..., \mathsf{A}_{m-1}\}$. The set $\mathcal{M}$ is represented as a string of pairs, each pair representing an UOD (the left-hand and right-hand sides of the dependency). Each side is a list of attributes. Let the length of the representation of $\mathcal{M}$, the string of concatenated left-hand and right-hand sides, be denoted by $|\mathcal{M}|$.

We first present the key elements of the algorithm for testing logical implication for transitive domains of UODs. Lastly, we establish that the algorithm is sound and complete in Theorem 32.

The algorithm *TestOrderDependency* (Algorithm 5) tests logical implication for transitive domains of UODs. It invokes algorithms *TestFunctionalDependency* and *TestOrder-*

Compatible (Algorithm 6). Algorithm *TestFunctionalDependency* performs a test whether $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{XY}$ which by Theorem 14 implies an FD, $\mathcal{M} \models \mathcal{X} \rightarrow \mathcal{Y}$. Algorithm *TestOrderCompatible* tests whether $\mathcal{M} \models \mathbf{XY} \leftrightarrow \mathbf{YX}$ ($\mathcal{M} \models \mathbf{X} \sim \mathbf{Y}$). These parts combine to complete the proof of soundness and completeness of our inference procedure for UODs over transitive domains. Since by Theorem 18 $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

Theorem 26 states that testing whether $\mathbf{X} \mapsto \mathbf{XY}$, which corresponds to an FD $\mathcal{X} \rightarrow \mathcal{Y}$ (Theorem 14), can be achieved in linear time. Notice that we assume there is a *TestFunctionalDependency* algorithm which finds a closure of a given set of attributes $\mathcal{X}$, as in [5] which provides a *linear* algorithm for finding closures over FDs.

Testing if $\mathbf{X} \sim \mathbf{Y}$ is more involved and complex. We observe that $\mathcal{M} \not\models \mathbf{X} \sim \mathbf{Y}$ *iff* we are able to construct a table $\mathbf{t}$ that *satisfies* set of UODs $\mathcal{M}$ and consists of two rows which have a *swap* (see Definition 14 and Example 38 in Chapter 4.4.1) with respect to $\mathbf{X} \sim \mathbf{Y}$. In the table $\mathbf{t}$ that we construct, we shall use integer values for the *cells* without lost of generality. (A cell is a given column entry of a given row.)

We test if $\mathbf{X} \sim \mathbf{Y}$ in the algorithm *TestOrderCompatible* (Algorithm 6). For each pair of attributes A in $\mathbf{X}$ and B in $\mathbf{Y}$, we test in an algorithm *TestSingleOrderCompatible* (Algorithm 7) whether we can construct a table $\mathbf{t}$ described above with a swap with respect to [A] $\sim$ [B] with attributes prefixing A and B, in lists $\mathbf{X}$ and $\mathbf{Y}$, respectively, being *constants* (Definition 24) within table $\mathbf{t}$, such that table $\mathbf{t}$ satisfies the set of ODs $\mathcal{M}'$. (Let

Algorithm 5 TestOrderDependecy

Input: A set $\mathcal{M}$ of n UODs on attributes $\{\mathsf{A}_0, ..., \mathsf{A}_{m-1}\}$ and an UOD $\mathbf{X} \mapsto \mathbf{Y}$.

Output: "true" if $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$; "false" otherwise.

Global data structures:

a. Attributes are represented by integers between 0 and m-1.

b. UODs in $\mathcal{M}$ are represented by integers between 0 and n-1.

c. LS[0:$n-1$], RS[0:$n-1$] are arrays of lists, containing the attributes in the left and right side of each UOD.

d. DEPEND[0:$m-1$] is an array of attributes found to be *functionally dependent* on given set of attributes.

e. OC[0:$n-1$; 0:1] is a two dimensional array of *order compatible* dependencies with *single attribute* on the left and right side.

f. LX and LY are lists of attributes represented by integers, corresponding to $\mathbf{X}$ and $\mathbf{Y}$ respectively.

1: DEPEND $\leftarrow$ *TestFunctionalDependency*($\mathcal{M}$, $\mathcal{X}$)

2: **if** exists i in LY such that DEPEND[i] = "false" **then**

3: $result \leftarrow$ "false"

4: **return** $result$

5: **else**

6: $result \leftarrow$ *TestOrderCompatible*

7: **return** $result$

P be the concatenated attributes prefixing A and B. We consider $\mathcal{M}' = \mathcal{M} \cup \{[] \mapsto \mathbf{P}\}$.) $[] \mapsto \mathbf{P}$ is a way of forcing each attribute C in list **P** to be a *constant*. Note that any table which satisfies $\mathcal{M}'$ satisfies $\mathcal{M}$. Once we find a swap, we halt in Algorithm 6.

Algorithm 6 TestOrderCompatible

Output: A result which states if **X** and **Y** are order compatible.

```
1: for i ← 0 to |X| − 1 do
2:   for j ← 0 to |Y| − 1 do
3:     result ← TestSingleOrderCompatible(i, j)
4:     if !result then
5:       return "false"
6: return "true"
```

Based on Definition 33, order compatibility for single attributes (over the attributes which are non-constant) is transitive for transitive domains. Therefore, we test if there is a *path* between A and B in a graph consisting of the first non-constant attributes from the left-hand side and a right-hand side of each UOD from $\mathcal{M}'$. We find this graph in Algorithm *FindOrderCompatibleGraph* (Algorithm 8). Finding a path by the transitivity property over order-compatibility means that $[\mathsf{A}] \sim [\mathsf{B}]$ holds.

We assume Algorithm *TestExistPath* which tests if there exists a path between two nodes. The problem of testing if there *exists* a path is simple. One can track the visited

edges during the process of traversing the nodes. We can guarantee that each edge is visited only once. Hence, we can check the existence of the path in linear time. Note there is an edge per OD in $\mathcal{M}'$, so the number of edges (plus number of nodes) is $O(|\mathcal{M}|)$.

Algorithm 7 TestSingleOrderCompatible

Input: Attributes indexes i and j.

Output: "true" if single attributes LX[i] and LY[j] are order compatible, "false" otherwise.

1: **if** LX[i] = LY[j] **then**
2: **return** "true"
3: **else**
4: List **P** is a concatenation of lists LX.subList(0, $i-1$) and LY.subList(0, $j-1$)
5: $\mathcal{M}' \leftarrow \mathcal{M} \cup \{[] \mapsto \mathbf{P}\}$
6: DEPEND $\leftarrow$ *TestFunctionalDependency*([ ], $\mathcal{M}'$)
7: **if** DEPEND[LX[i]] || DEPEND[LY[j]] **then**
8: **return** "true"
9: **else**
10: OC $\leftarrow$ *FindOrderCompatibleGraph*
11: **return** *TestExistPath*(LX[i], LY[j], OC)

Theorem 32 *(soundness and completeness) Algorithm 5 for testing logical implication* $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$ *for transitive domains of UODs is sound and complete.*

Algorithm 8 FindOrderCompatibleGraph

```
1: initialize two dimensional array OC with [0; 0]
2: for l ← 0 to n − 1 do
3:   for k ← 0 to LS[l].size() − 1 do
4:     if DEPEND[LS[l][k]] = "false" then
5:       a ← DEPEND[LS[l][k]]
6:       for s ← 0 to RS[l].size() − 1 do
7:         if DEPEND[RS[l][s]] = "false" then
8:           b ← DEPEND[RS[l][s]]
9:           OC[l][0] ← a, OC[l][1] ← b
10:          break
11:      break
```

Proof

Theorem 18 states that order dependency $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

Case 1 $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{XY}$. We have already proven that testing $\mathbf{X} \mapsto \mathbf{XY}$ is sound and complete (Theorem 26).

Case 2 $\mathcal{M} \models \mathbf{X} \sim \mathbf{Y}$. This step is tested in algorithm *TestOrderCompatible* (Algorithm 6). If $\mathcal{M} \models \mathbf{X} \sim \mathbf{Y}$ is falsified, then we show that we are always able to construct a two-tuple table $\mathbf{t}$ which satisfies set of UODs $\mathcal{M}$ and has a *swap* (Definition 14) with respect to an UOD $\mathbf{X} \sim \mathbf{Y}$. The main body of this algorithm is a double-nested for-loop

Table 4.14: Table **t**.

Constants	A	B	Group A			Remaining attributes		
0	0	1	0	...	0	1	...	1
0	1	0	1	...	1	0	...	0

runs $|\mathbf{X}||\mathbf{Y}|$ times (and *terminates*). Inside, it invokes Algorithm 7 each time.

Algorithm 7 tests for each pair of attributes A and B from **X** and **Y**, respectively, if it is possible to construct the described table **t** which has a swap between A and B, and which also satisfies $\mathcal{M}'$. $\mathcal{M}' = \mathcal{M} \cup \{[] \mapsto \mathbf{P}\}$, where **P** is a conjunction of lists prefixing A and B from lists **X** and **Y**, respectively. Note that any table which satisfies $\mathcal{M}'$ satisfies $\mathcal{M}$. Therefore, as by Definition 14 we enumerate through all the possible cases in the columns where an UOD $\mathbf{X} \sim \mathbf{Y}$ can be falsified by a *swap*. (It cannot be falsified by a *split*, Definition 13.)

We construct the table **t** (see Table 4.14) with values 0 and 1 only if both A and B are not *constants*. Attributes are partitioned into three groups: those that have the same values as A (and consequently swapped values to B); those that are constants; and the remaining attributes which have swapped attributes to A. Group A is the set of attributes which have a path with A in a data structure OC (defined in Algorithm 5, the values are assigned in Algorithm 8). We use the transitivity property for single attributes over order-compatibility. Algorithm *TestExistPath* tests if there exists a path.

Table **t** satisfies $\mathcal{M}'$. Assume otherwise: for $\mathbf{M} \mapsto \mathbf{N} \in \mathcal{M}'$, **t** falsifies it. We do not introduce splits in table **t** that falsify $\mathbf{M} \mapsto \mathbf{MN}$ because by Theorem 26 the algorithm is

sound and complete for inferring FDs. (Note that it applies also to constants which can be expressed as FDs.)

Consider $\mathbf{M} \sim \mathbf{N}$. Breaking this is the other way of falsifying UOD $\mathbf{M} \mapsto \mathbf{N}$ by Theorem 18. Let E be the first element which is not a constant from **M** and F from **N**, respectively. If both E and F are from group A with A or they are both from remaining attribute group with B, then **M** and **N** order the tuples of **t** the same way. Therefore, E must be from one group and F from the other. Since the transitivity property holds over order-compatibility, we would detect this. Contradiction. □

Theorem 33 *(complexity for transitive domains)*

Testing logical implication for transitive domains of UODs, (that is whether $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$*) is solvable in polynomial time,* $O(|\boldsymbol{X}||\boldsymbol{Y}||\mathcal{M}|)$.

Proof

By Theorem 18, $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

Case 1 Testing the logical implication that $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{XY}$ can be done in $O(|\mathcal{M}|)$ time by Theorem 26.

Case 2 Algorithm 6 tests logical implication, that $\mathcal{M} \models \mathbf{X} \sim \mathbf{Y}$. In the main body of this algorithm, the double-nested for-loop runs $|\mathbf{X}||\mathbf{Y}|$ times invoking each time Algorithm 7. Algorithm 7 tests if each A from **X** is order compatible with each B from **Y**. This is done by checking if there is a path in a graph. We keep track of visited edges. Hence,

Table 4.15: Table Employee_salary.

id	position	rank	grade	hire_date	salary	years
100	Manager	70	87%	20010112	80K	11
150	Secretary	30	90%	20050112	40K	7
200	Manager	70	90%	20060817	50K	6
202	Director	90	50%	20080817	200K	4
203	Director	90	95%	20080818	200K	4

we can check if there is a path in linear time over $O(|\mathcal{M}|)$. Therefore the complexity of Algorithm 6 is $O(|\mathbf{X}||\mathbf{Y}||\mathcal{M}|)$. □

We illustrate the use of ODs, and how they can be used to support SQL functions and user-defined functions. Consider the human-resource table employee_salary in Table 4.15.

Example 46 (Human Resource)

A table employee_salary *has the following attributes:* id*(employee identifier),* position *(corporate title),* rank *(ranking of the employee)*[14], grade *(employee evaluation),* hire_date *(date hired),* salary *(employee's salary),* years *(years of service).*

Salary rises as ranking, years of service, and grade rise, in that lexicographical order. That is, [rank, years, grade] $\mapsto$ [salary]. *Moreover, the date when the person was hired is monotonic with respect to the identifier:* [id] $\mapsto$ [date_hire]. *Assume these order dependencies are declared as check constraints. The first constraint which expresses that* [rank, years, grade] $\mapsto$ [salary] *may be used to check the consistency of the database.*

[14]The higher the position, the higher the rank. (For example, for the secretary, the rank is 30, while for the director, it is 90.)

(It would detect errors in assigning the salary, according to the business logic.) Furthermore, assume the table has a clustered index on hire_date. *Given a business query with* order by date_hire, *it could be evaluated using the index on* id. *Note also that an OD* [id] $\mapsto$ [date_hire] *can be used to save disk space, since no index on* date_hire *is needed.*

In this example, all of the inferences can be automatically done by our OD-inference procedure.

5 Conclusions

5.1 In Summary

Ordering permeates databases, to such an extent that we take it for granted. We expect it to be exploited wisely in query plans. It appears in many queries and is relatively expensive to perform. Queries that involve order by, group by, join, partition by and distinct statement with SQL functions and algebraic expressions are common in real business scenarios. Identifying an order dependency between the attributes of such queries can remove or simplify potentially expensive operators such as sort. One of the thesis's key contributions is an algorithm for reducing an *interesting order* into a *canonical form* by using *declared*, *detected*, and *inferred* ODs.

We built a prototype of this in IBM DB2, (as a branch of the code base). The techniques described in this thesis, although implemented in IBM DB2, are general enough to be used in any query optimizer. These techniques should apply to a wide range of BI queries.

Our experiments show the viability of our proposed solutions. We ran the benchmark

experiments over TPC-DS to demonstrate the efficiency of our approach using the order dependency. The nine queries described in the thesis (Section 3.2) benefited with an average gain of 30% on a a ten-GB database.

Of TPC-DS's 99 queries, 13 matched our rewrite technique described in Section 3.4. Every one of the 13 benefited, with an average performance gain of 48%. The other queries in TPC-DS were not affected as they were not rewritten. (There is an additional optimization cost because of the additional rewrite rules, but it is marginal.) Queries that involve predicates over time and date are exceedingly common for most data warehouses. For many design reasons, date is often represented explicitly as a dimension table of its own, with the primary key of the date table done as a surrogate key. While this design can have compelling advantages, the surrogate key can cause problems for queries. A majority of queries are over a fact table. A query often uses natural date values in predicates. However, date in the fact table is recorded by surrogate key. This necessitates a potentially quite expensive join between the fact table and the date dimension table when the query is evaluated. There is an additional problem when a fact table has been partitioned by date, as it is common practice in data warehouse systems in order to accommodate very large tables (e.g. in distributed systems). Since the date range (surrogate values) over the fact table cannot be determined from the query (natural values) all the partitions of the fact table must be scanned.

We have sought to optimize such queries involving date in data warehouses by re-

moving this join, and by choosing just the relevant partitions of the fact table when the table is distributed. We explored two solutions. The first is to encode date information into the so-called surrogate key. In some situations, this is acceptable, and it preserves most of the advantages of using a surrogate key. Such an embedded surrogate key, however, makes it possible to execute many queries without a join to the date table. However, embedded surrogate keys are not always acceptable. Our second solution is more general. We introduce the notion of order dependency, and show that surrogate data keys will be monotone with respect to the natural data values they represent in most any data warehouse design. By making this monotone dependency known to the query optimizer, queries with date predicates can often be rewritten to avoid a join with the date table, and to select just the relevant partitions of the fact table.

One of the main goals of this thesis was to develop a theory behind the complexity of ODs. To the best of our knowledge, this is the first attempt to study the complexity of such dependencies.

We devise an elimination procedure for testing logical implication for ODs and show hierarchy of order dependencies classes. We present that testing logical implication for UODs is co-NP-complete, as well as FDs over ODs. However, we demonstrated that testing logical implication of FDs over UODs is linear. We have also shown our inference rules for UODs are sound and complete. Finally, we have also shown a transitive domain over which the inference problem for UODs is more efficient. We designed a

polynomial algorithm for testing logical implication over this domain. We have implemented the elimination procedure and the inference procedure for transitive domains in IBM DB2. Our experiments have shown that the time of running the elimination procedure for real world business domains, for which the number of unique attributes in the set of prescribed ODs is from 5 to 12 is marginal. (The time of running the inference procedure for transitive domains is marginal even for large domains.)

5.2 Future Work

There is more that can be done, and that we plan to do. Future work in this area should pursue two lines of research: further investigation of the theoretical questions; and, applications of the theoretical framework in a practical database setting.

In future work, we plan to pursue following.

- We plan to work on extending our work axiomatization for UODs [39] into an axiomatization for ODs, which allow the mix of ascending and descending orders. Such an axiomatization might provide insight into how ODs behave, and provide input for useful query rewrites.
- One of the practical applications on which we are currently working is a *sound* theorem prover. We prove in this work that testing logical implication for ODs is co-NP-complete. However, it is the lack of transitivity over the order-compatibility that is at the heart of the complexity. That is why the *Chain* axiom is necessary for

a complete axiomatization of UODs (Figure 4.1). We would like to investigate if there is a polynomial algorithm for reasoning over the first five axioms, excluding the Chain axiom (Figure 4.1). Such a theorem prover would be a useful tool in query optimization and an alternative approach to what we proposed in this work, defining a domain property that makes reasoning over ODs efficient.

- Integrity constraints have been widely used in query optimization through *query rewrites*. For example FDs have been shown to be useful in simplifying queries with `distinct`, `join`, `order by` and `group by` operations [37] whereas inclusion dependencies can be used to remove certain joins over primary and foreign keys [9]. We would like to extend our work of optimizing business-intelligence queries with UODs [41], in order to cover more scenarios including nested queries [21]. This includes monotonicity in case expressions and optimization of queries such as Query 16 where there is an order dependency between the customer_id and the output of the then statement.
- We are working on introducing a framework for discovering *conditional order dependencies*. The problem of extending dependencies with conditions was studies in [8, 13]. Conditional sequential dependencies were proposed in [17]. A conditional order dependency can be represented as a pair ($\mathbf{X} \mapsto \mathbf{Y}$, $\mathbf{T}_r$), where $\mathbf{X} \mapsto \mathbf{Y}$, referred to as the embedded OD, and $\mathbf{T}_r$ is a range pattern tableau defining over which rows the dependency applies. It would provide a novel integrity

Query 16 Categories by case.

```
select ..., sum(S.quantity),
      (case
       when S.customer_id between 1 and 10
           then 1
                ⋮
       when S.customer_id between 91 and 100
           then 10
       end)
from sales S, ...
where ...
group by (case ...)
order by(case ...);
```

constraint allowing one to express, that an OD date $\mapsto$ salary holds for a given employee_id.

- We are trying to improve the integration of our order constraints with the cost-based optimizer to improve cardinality estimation. For example, when we know there is an order equivalence between columns, such as between d_date_id and d_date, a surrogate and natural key, *and* we know there is a one-to-one mapping between them, then the cardinality of a range predicate on one could be estimated using the other. This could improve the performance even more beyond what we have already gained with our query-rewrite techniques.

- If ABC $\mapsto$ D holds but not AB $\mapsto$ D, is ordering by AB useful if we need a stream sorted by D? If the stream is sorted by AB, it may be *nearly sorted* on D. If it were known that every partition of AB is small, each AB-partition could be sorted on-the-fly in main memory, removing the need for an external sort operator. We believe the work of [4] and this work on order dependencies could be combined to formalize the concept of nearly sorted.
- In the process of merging data from various sources, it is often the case that small variations occur. For example, one movie site might report the movie *Gone with the Wind* as having a running time of 222, while site two reports 238 minutes for it. The FD that *movie* $\rightarrow$ *length* would be violated. In [27], they define a metric over *FDs* to allow for such small variations. Likewise, we would like to define metric ODs to generalize both ODs as in this thesis and metric FDs. We would like to devise algorithms for determining whether a given metric OD holds for a given relation, and to investigate the use of these as data cleaning techniques as in [6] for *matching dependencies*.
- We want to improve the integration of our order constraints with the cost-based optimizer to improve cardinality estimation. For example, when we know there is an order equivalence between columns, such as between d_date_sk and d_date, a surrogate and natural key, *and* we know there is a one-to-one mapping between them, then the cardinality of a range predicate on one could be estimated using the

other. This could improve the performance beyond what we have already gained with our query-rewrite techniques.

- We believe that order dependencies can also be identified in *geo-spatial* dimensions of data warehouses. Similar optimization techniques which we have described in this thesis can be applied there, too.
- We are exploring the use of ODs for *database design* [3]. The concept of functional dependency lies at the heart of database design and the relational model. Order dependency extends functional dependency in a quite natural way to include also semantics of order over the data. ODs can reveal redundancies that cannot be detected using FDs alone. This leads one to wonder about the concept of normalization modulo ODs. It would be an interesting research topic to extend the results obtained there to the design of relational databases.

6 Related Work

Ordered sets and lattices have been researched in mathematics [11]. In fact, our concept of order dependencies is equivalent to *order-preserving mapping* between two *ordered* sets. The work in mathematics has concentrated on investigating the properties of, and relationships between, ordered sets rather than among the mappings. To the best of our knowledge, no complexity study for investigating relationship between mappings of ODs has been proposed.

Sorting is at the heart of many database operations: sort-merge join, index generation, duplicate elimination, ordering the output through the SQL order-by operator, etc. The importance of sorted sets for query optimization and processing was recognized very early on. Right from the start, the query optimizer of System R [36] paid particular attention to *interesting orders* by keeping track of all such ordered sets throughout the process of query optimization. In [21] authors explored the use of sorted sets for executing nested queries. In System R, interesting orders were primarily employed to prevent subplans that satisfy some useful order from being pruned by less expensive but

unordered subplans during bottom-up plan generation. A later paper on the System R optimizer [1] shows how to combine interesting orders when possible from order-by, group-by, and distinct statements, so a single sort can be used more often to satisfy more than one operator. An other, strongly related paper is [36]. Its main contribution was a set of fundamental operations for use in order optimization by exploiting functional dependencies.

The importance of sorted sets has prompted the researchers to look *beyond* the sets that have been explicitly generated. Thus, [31] shows how to discover sorted sets created as generated columns via algebraic expressions. (In DB2, a generated column is a column that can be computed from other columns in the schema.)

For example, if column A is sorted, so is the generated column G defined as $\mathsf{G} = \mathsf{A}/100 + \mathsf{A} - 3$ (that is, $\mathrm{A} \rightsquigarrow \mathrm{G}$).[15] We show in [40] how to use relationships between sorted attributes discovered by reasoning over the physical schema. The testing logical implication system presented here provides a formal way of reasoning (hence discovering) previously unknown or hidden sorted sets. Based on this work, many other optimization techniques from relational query processing can also be adapted.

Order dependencies were introduced for the first time in the context of database systems in [15]. However, the type of orders, hence the dependencies defined over them, were different from the ones we presented here. A dependency $\mathcal{X} \rightsquigarrow \mathcal{Y}$ holds if order

[15] We use the arrow "$\rightsquigarrow$" for simplicity for different type of orders, regardless.

over the values of each attribute in $\mathcal{X}$ implies an order over the values of each attribute of $\mathcal{Y}$. In other words, the dependency is defined over the sets of attributes rather than lists. The distinction between these two types of dependencies was later [32] aptly described as pointwise versus lexicographical order dependency. Formally, an instance satisfies a pointwise order dependency $\mathcal{X} \rightsquigarrow \mathcal{Y}$ if, for all tuples s and t, for every attribute A^{op} in $\mathcal{X}$, s_{A} *op* t_{A}, implies that for every attribute B^{op} in $\mathcal{Y}$, s_{B} *op* t_{B}.

In [16] a sound and complete set of inference rules for such dependencies is defined with demonstrating that determining logical implication is co-NP-complete. A practical application of the dependencies for an improved index design is presented in [12].

Dependencies defined over lexicographically ordered domains were introduced in [32] under the name *lexicographically ordered functional dependencies*. (We called these UODs.) Two other papers [33, 34] by the same author develops a theory behind both lexicographical as well as pointwise dependencies. (The latter were simpler than the dependencies defined in [16].) A set of inference rules (proved to be sound and complete) is introduced for pointwise dependencies, but –interestingly – not for the lexicographical dependencies in this work. Only a chase procedure is defined for the latter, for which the order dependencies are defined as here. the complexity of testing logical implication for ODs has also not been studied, which is the subject of our work. An extension of relational algebra to ordered domains is presented in [34]. Recently, in [39], we presented a sound and complete axiomatization for UODs. UODs do not consider bidirectionality

(a mix of `asc` and `desc`) as do ODs which we introduced for the first time in [38].

Traditional integrity constraints have been adapted to apply *conditionally* on the data to be able to capture the semantics of, and errors commonly found, in real data. *Conditional functional dependencies* were proposed in [7] and *conditional inclusion dependencies* were proposed in [8]. In a similar vein a novel integrity constraint for ordered data, sequential dependencies which defined also over *sets* of attributes was introduced in [17]. For example, a sequential dependency $\mathsf{sequence\_id} \rightsquigarrow_{[5,6]} \mathsf{time}$ means that time *gaps* between consecutive sequence numbers are between 5 and 6. The authors present a framework for *discovering* which subsets of the data obey prescribed sequential dependencies.

The problem of *discovering* dependencies in data was first studied in [23, 25] where the goal was to find antecedent and consequent attributes from among different subsets of attributes in the schema satisfying an FD. The problem of a conditional functional dependency discovery, given an embedded FD, was introduced in [18]. The *range tableaux* proposed in [18] specify that the FD independently holds on each subset of tuples that agree on the antecedent attributes such that the value of the ordered attribute is within the range. In [17], a tableau pattern denotes that the underlying Sequential Dependency holds in the entire interval.

The problem of estimating the sortedness of data stream was studies in [19]. Many times, relations prove to be *nearly-sorted*; most of tuples are close to their place in the

order. An interesting study of establishing whether a given stream is sufficiently nearly-sorted was described in [4].

7 Copyrights

Bibliography

[1] G. Antoshenkov. Query processing in dec rdb: Major issues and future challanges. In *IEEE Bulletin on the Technical Comittee on Data Engineering*, 1993.

[2] W. Armstrong. Dependency Structures of Database relationships. In *Proceedings of the IFIP Congress, 580-583*, 1974.

[3] C. Beeri and P. Bernstein. Computional Problems Related to the Design of Normal Form Relational Schemas. *TODS 4(1):30-59*, 1979.

[4] S. Ben-Moshe, Y. Kanza, E. Fischer, A. Matsliah, M. Fischer, and C. Staelin. Detecting and exploiting near-sortedness for efficient relational query evaluation. In *ICDT, 256-267*, 2011.

[5] P. Bernstein. Synthesing third normal form relations from functional dependencies. *TODS, 1(4): 277-298*, 1976.

[6] L. Bertossi, S. Kolahi, and V. Laks Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT, 268-279*, 2011.

[7] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional funcional dependencies for data cleaning. In *ICDE, 746-755*, 2007.

[8] L. Bravo, W. Fan, and L. Yang. Extending dependencies with conditions. In *VLDB, 243-253*, 2007.

[9] Q. Cheng, J. Gryz, F. Koo, T. Leung, L. Liu, X. Qian, and K. Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB, 687-698*, 1999.

[10] F. Chiang and R. Miller. Discovering data quality rules. *PVLDB, 1(1): 1166-1177*, 2008.

[11] B. Davey and H. Priestley. Introduction to Lattices and Order. In *Cambridge University Press, 1-50*, 2002.

[12] J. Dong and R. Hull. Applying Approximate order dependency to Reduce Indexing Space. In *SIGMOD, 119-127*, 1982.

[13] W. Fan, F. Geerts, L. Lakshmanan, and M. Xiong. Discovering conditional functional dependencies. In *ICDE, 481-492*, 2009.

[14] M. Garey and D. Johnson. A Guide to NP-completness. In *Freeman*, 1979.

[15] S. Ginsburg and R. Hull. Ordered Attribute Domains in the Relational Model. In *XP2 Workshop on Relational Database Theory*, 1981.

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

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

[18] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yo. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB, 1(1): 376-390*, 2008.

[19] P. Gopalan, T. Jayram, R. Krauthgamer, and R. Kumar. Estimating the sortedness of a data stream. In *SODA, 318-327*, 2007.

[20] S. Guo, W. Sun, and M. Weiss. Solving satisfiability and implication problem in database systems. *TODS, 21(2): 270-293*, 1996.

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

[22] P. Honeyman. Testing Satisfaction of Functional Dependencies. *Journal of the ACM, 668-677*, 1982.

[23] Y. Huhtala, P. Karkkainen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal 42(2), 100-111*, 1999.

[24] R. Kimball and M. Ross. The Data Warehouse Toolkit Second Edition. The Complete Guide to Dimensional modeling. In *John Wiley and Sun, 217-227*, 2012.

[25] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *TCS 149(1), 129-149*, 1995.

[26] F. Korn, S. Muthukrishnan, and Y. Zhu. Checks and balances: monitoring data quality problems in network traffic databases. In *VLDB, 536-547*, 2003.

[27] N. Koudas, A. Saha, , A. Saha, and V. S. Srivastava, D. Metric Functional Dependencies. In *ICDE, 1291-1294*, 2009.

[28] N. Lorentzos. DBMS Support for Time and Totally Ordered Compound Data Types. *Information Systems 17(5), 347-358*, 1992.

[29] D. Maier, A. Mendelzon, and Y. Sagiv. Testing Implication of Data Dependencies. *TODS, 1-16*, 1979.

[30] E. Malinowski and E. Zimanyi. A conceptual solution for representing time in data warehouse dimension. In *APCCM, 45-54*, 1982.

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

[32] W. Ng. Lexicographically Ordered Functional Dependencies and Their Application to Temporal Relations. In *IDEAS, 279-287*, 1999.

[33] W. Ng. Ordered Functional Dependencies in Relational Databases. *Informaiton Systems, 535-554*, 1999.

[34] W. Ng. An Extension of the Relational data model to incorporate ordered domains. *TODS, 26(3) 344-383*, 2001.

[35] M. Riedewald, A. Agrawal, and A. Abbadi. Efficient Integration and Aggregation of Historical Information. In *SIGMOD, 13-24*, 1982.

[36] P. Selinger and M. Astrahan. Access Path Selection in a Relational Database Management System. In *SIGMOD, 23-34*, 1979.

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

[38] J. Szlichta, P. Godfrey, and J. Gryz. Chasing Polarized Order Dependencies. In *AMW, 168-179*, 2012.

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

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

[41] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, W. Qiu, and C. Zuzarte. Business-Intelligence Queries in DB2 with Order Dependencies. Technical report, York University, 2012. Submitted for review. www.cse.yorku.ca/techreports/2012/CSE-2012-04.pdf.

[42] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. Expressiveness and Complexity of Order Dependencies. *PVLDB*, 2013.

[43] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. The Axiomatic System for Order Dependencies. In *AMW*, 2013.

[44] J. Ullman. Principles of database and knowledge-base systems, vol. 1. In *Computer Science Press, 378-379*, 1988.

[45] Y. Zhu and D. Shasha. Statistical monitoring of thousands of data streams in real time. In *VLDB, 358-369*, 2002.

Appendix

A TPC-DS Benchmark Table Definitions

The TPC-DS benchmark is a decision support benchmark. It consists of a suite of business oriented queries. The queries and the data populating the database have been chosen to have a broad industry–wide relevance while maintaining a sufficient degree of ease of implementation. This benchmark illustrates decision support systems that:

- Examine large volumes of data,
- Execute queries with a high degree of complexity,
- Give answers to critical business questions.

The minimum database required to run the benchmark holds business data of the size 1GB. Compliant benchmark implementations may also use one of the larger permissible database populations(e.g. 10GB).

Below we show table definitions used in experimental evaluation of our prototype.

A1 Fact Table Definitions

Each row in *web_sales* table represents a single lineitem for a sale made through the web channel and recorded in the web_sales fact table.

Each row in *store_sales* table represents a single lineitem for a sale made through the store channel and recorded in the store_sales fact table.

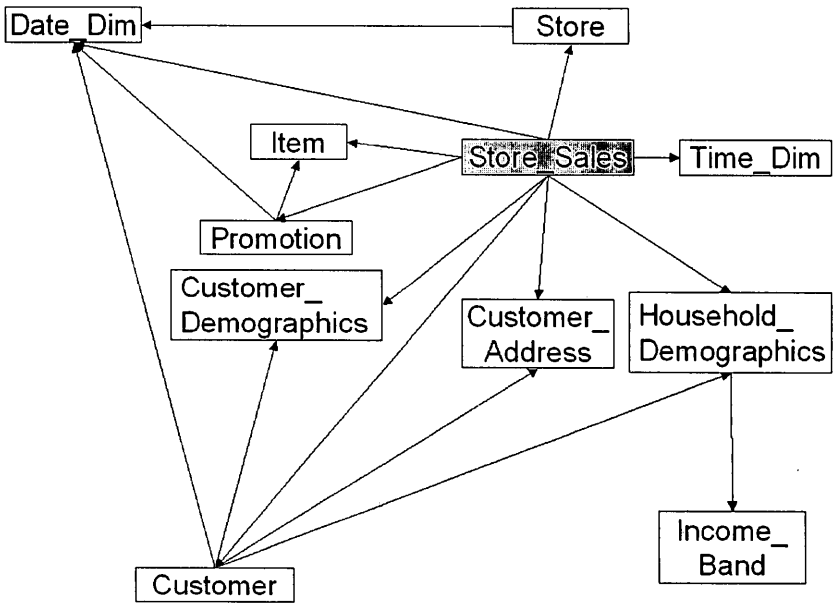

Figure A1: Store_sales ER-Diagram.

A2 Dimension Table Definitions

The *customer_demographics* table contains one row for each unique combination of customer demographic information.

Each row in *date_dim* table represents one calendar day. The surrogate key (d_date_sk) for a given row is derived from the julian date being described by the row.

Each row in *time_dim* table represents one second.

Each row in *item* table represents a unique product formulation (e.g., size, color, manufacture, etc.).

Each row in *promotion* table represents details of a specific product promotion (e.g.,

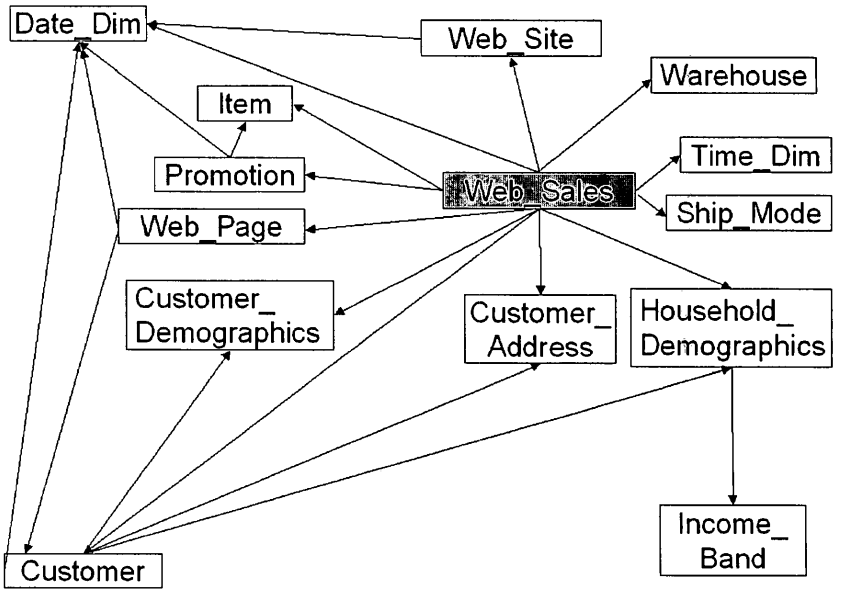

Figure A2: Web_sales ER-Diagram.

advertising, sales, PR).

Each row in *store* dimension table represents details of a store.

Each row in *warehouse* dimension table represents a warehouse where items are stocked.

Each row in *ship_mode* represents a shipping mode.

Table A1: Web_sales column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
ws_sold_date_sk	identifier			d_date_sk
ws_sold_time_sk	identifier			t_time_sk
cr_item_sk	identifier	N	Y	i_item_sk,cs_item_sk
cr_refunded_customer_sk	identifier			c_customer_sk
cr_refunded_cdemo_sk	identifier			cd_demo_sk
cr_refunded_hdemo_sk	identifier			hd_demo_sk
cr_refunded_addr_sk	identifier			ca_address_sk
cr_returning_customer_sk	identifier			c_customer_sk
cr_returning_cdemo_sk	identifier			cd_demo_sk
cr_returning_hdemo_sk	identifier			hd_demo_sk
cr_returning_addr_sk	identifier			ca_address_sk
cr_call_center_sk	identifier			cc_call_center_sk
cr_catalog_page_sk	identifier			cp_catalog_page_sk
cr_ship_mode_sk	identifier			sm_ship_mode_sk
cr_warehouse_sk	identifier			w_warehouse_sk
cr_reason_sk	identifier			r_reason_sk
cr_order_number	identifier	N	Y	cs_order_number
cr_return_quantity	integer			
cr_return_amount	decimal(7,2)			
cr_return_tax	decimal(7,2)			
cr_return_amt_inc_tax	decimal(7,2)			
cr_fee	decimal(7,2)			
cr_return_ship_cost	decimal(7,2)			
cr_refunded_cash	decimal(7,2)			
cr_reversed_charge	decimal(7,2)			
cr_store_credit	decimal(7,2)			
cr_net_loss	decimal(7,2)			

Table A2: Sales column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
ss_sold_date_sk	identifier			d_date_sk
ss_sold_time_sk	identifier			t_time_sk
ss_item_sk	identifier	N	Y	i_item_sk,sr_item_sk
ss_customer_sk	identifier			c_customer_sk
ss_cdemo_sk	identifier			cd_demo_sk
ss_hdemo_sk	identifier			hd_demo_sk
ss_addr_sk	identifier			ca_address_sk
ss_store_sk	identifier			s_store_sk
ss_promo_sk	identifier			p_promo_sk
ss_ticket_number	identifier	N	Y	sr_ticket_number
ss_quantity	integer			
ss_wholesale_cost	decimal(7,2)			
ss_list_price	decimal(7,2)			
ss_sales_price	decimal(7,2)			
ss_ext_discount_amt	decimal(7,2)			
ss_ext_sales_price	decimal(7,2)			
ss_ext_wholesale_cost	decimal(7,2)			
ss_ext_list_price	decimal(7,2)			
ss_ext_tax	decimal(7,2)			
ss_coupon_amt	decimal(7,2)			
ss_net_paid	decimal(7,2)			
ss_net_paid_inc_tax	decimal(7,2)			
ss_net_profit	decimal(7,2)			

Table A3: Customer_demographics column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
cd_demo_sk	identifier	N	Y	
cd_gender	char(1)			
cd_marital_status	char(1)			
cd_education_status	char(20)			
cd_purchase_estimate	integer			
cd_credit_rating	char(10)			
cd_dep_count	integer			
cd_dep_employed_count	integer			
cd_dep_college_count	integer			

Table A4: Date_dim column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
d_date_sk	identifier	N	Y	
d_date_id	char(16)	N		
d_date	date			
d_month_seq	integer			
d_week_seq	integer			
d_quarter_seq	integer			
d_year	integer			
d_dow	integer			
d_moy	integer			
d_dom	integer			
d_qoy	integer			
d_fy_year	integer			
d_fy_quarter_seq	integer			
d_fy_week_seq	integer			
d_day_name	char(9)			
d_quarter_name	char(6)			
d_holiday	char(1)			
d_weekend	char(1)			
d_following_holiday	char(1)			
d_first_dom	integer			
d_last_dom	integer			
d_same_day_ly	integer			
d_same_day_lq	integer			
d_current_day	char(1)			
d_current_week	char(1)			
d_current_month	char(1)			
d_current_quarter	char(1)			
d_current_year	char(1)			

Table A5: Time_dim column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
t_time_sk	Identifier	N	Y	
t_time_id	char(16)	N		
t_time	Integer			
t_hour	Integer			
t_minute	Integer			
t_second	Integer			
t_am_pm	char(2)			
t_shift	char(20)			
t_sub_shift	char(20)			
t_meal_time	char(20)			

Table A6: Item column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
i_item_sk	identifier	N	Y	
i_item_id (B)	char(16)	N		
i_rec_start_date	date			
i_rec_end_date	date			
i_item_desc	varchar(200)			
i_current_price	decimal(7,2)			
i_wholesale_cost	decimal(7,2)			
i_brand_id	integer			
i_brand	char(50)			
i_class_id	integer			
i_class	char(50)			
i_category_id	integer			
i_category	char(50)			
i_manufact_id	integer			
i_manufact	char(50)			
i_size	char(20)			
i_formulation	char(20)			
i_color	char(20)			
i_units	char(10)			
i_container	char(10)			
i_manager_id	integer			
i_product_name	char(50)			

Table A7: Item column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
i_item_sk	identifier	N	Y	
i_item_id (B)	char(16)	N		
i_rec_start_date	date			
i_rec_end_date	date			
i_item_desc	varchar(200)			
i_current_price	decimal(7,2)			
i_wholesale_cost	decimal(7,2)			
i_brand_id	integer			
i_brand	char(50)			
i_class_id	integer			
i_class	char(50)			
i_category_id	integer			
i_category	char(50)			
i_manufact_id	integer			
i_manufact	char(50)			
i_size	char(20)			
i_formulation	char(20)			
i_color	char(20)			
i_units	char(10)			
i_container	char(10)			
i_manager_id	integer			
i_product_name	char(50)			

Table A8: Promotion column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
p_promo_sk	identifier	N	Y	
p_promo_id	char(16)	N		
p_start_date_sk	identifier			d_date_sk
p_end_date_sk	identifier			d_date_sk
p_item_sk	identifier			i_item_sk
p_cost	decimal(15, 2)			
p_response_targ et	integer			
p_promo_name	char(50)			
p_channel_dmail	char(1)			
p_channel_email	char(1)			
p_channel_catalog	char(1)			
p_channel_tv	char(1)			
p_channel_radio	char(1)			
p_channel_press	char(1)			
p_channel_event	char(1)			
p_channel_demo	char(1)			
p_channel_details	varchar(100)			
p_purpose	char(15)			
p_discount_active	char(1)			

Table A9: Store column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
s_store_sk	identifier	N	Y	
s_store_id	char(16)	N		
s_rec_start_date	date			
s_rec_end_date	date			
s_closed_date_sk	identifier			d_date_sk
s_store_name	varchar(50)			
s_number_employees	integer			
s_floor_space	integer			
s_hours	char(20)			
s_manager	varchar(40)			
s_market_id	integer			
s_geography_class	varchar(100)			
s_market_desc	varchar(100)			
s_market_manager	varchar(40)			
s_division_id	integer			
s_division_name	varchar(50)			
s_company_id	integer			
s_company_name	varchar(50)			
s_street_number	varchar(10)			
s_street_name	varchar(60)			
s_street_type	char(15)			
s_suite_number	char(10)			
s_city	varchar(60)			
s_county	varchar(30)			
s_state	char(2)			
s_zip	char(10)			
s_country	varchar(20)			
s_gmt_offset	decimal(5,2)			
s_tax_percentage	decimal(5,2)			

Table A10: Warehouse column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
w_warehouse_sk	identifier	N	Y	
w_warehouse_id	char(16)	N		
w_warehouse_name	varchar(20)			
w_warehouse_sq_ft	integer			
w_street_number	char(10)			
w_street_name	varchar(60)			
w_street_type	char(15)			
w_suite_number	char(10)			
w_city	varchar(60)			
w_county	varchar(30)			
w_state	char(2)			
w_zip	char(10)			
w_country	varchar(20)			
w_gmt_offset	decimal(5,2)			

Table A11: Ship_mode column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
sm_ship_mode_sk	identifier	N	Y	
sm_ship_mode_id	char(16)	N		
sm_type	char(30)			
sm_code	char(10)			
sm_carrier	char(20)			
sm_contract	char(20)			

Table A12: Web_site column definitions.

Column	Datatype	NULLs	Primary Key	Foreign Key
web_site_sk	identifier	N	Y	
web_site_id (B)	char(16)	N		
web_rec_start_date	date			
web_rec_end_date	date			
web_name	varchar(50)			
web_open_date_sk	identifier			d_date_sk
web_close_date_sk	identifier			d_date_sk
web_class	varchar(50)			
web_manager	varchar(40)			
web_mkt_id	integer			
web_mkt_class	varchar(50)			
web_mkt_desc	varchar(100)			
web_market_manager	varchar(40)			
web_company_id	integer			
web_company_name	char(50)			
web_street_number	char(10)			
web_street_name	varchar(60)			
web_street_type	char(15)			
web_suite_number	char(10)			
web_city	varchar(60)			
web_county	varchar(30)			
web_state	char(2)			
web_zip	char(10)			
web_country	varchar(20)			
web_gmt_offset	decimal(5,2)			
web_tax_percentage	decimal(5,2)			