THE NATIONAL UNIVERSITY
of SINGAPORE

School *of* Computing
Lower Kent Ridge Road, Singapore 119260

TRD6/03

Inferring and Applying Functional Dependencies in Schematic Discrepant Transformations

Tok Wang LING and Qi HE

June 2003

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Inferring and Applying Functional Dependencies in Schematic Discrepant Transformations

Tok Wang Ling, Qi He

Dept. of Computer Science, School of Computing
National University of Singapore
{lingtw, heqi}@comp.nus.edu.sg

Abstract. In relational model, schematic discrepancy occurs when the same information is modeled differently as attribute values, attribute names, relation names or database names in different schemas. Originally raised in schema integration, people have identified many applications of it recently. This paper focuses on the inference and use of functional dependency (FD) constraints in the transformations among schematic discrepant schemas. We first study restructuring operators which are used to implement schematic discrepant transformations. Specifically, we study the reconstructibility and commutativity of restructuring operators, which can be used to simplify a transformation. Then to infer FDs in transformed relations, we propose restricted FDs to represent integrity constraints in original relations. We also study the properties on how restricted FDs change when applying each kind of restructuring operators. Then we give an algorithm to infer FDs in schematic discrepant transformations. Our algorithm can compute all FDs in transformed relations which can be inferred from restricted FDs in original relations. At last, we identify the use of FDs in 3 scenarios: (1) use FDs to judge the correctness of SchemaSQL views; (2) use FDs to normalize a transformed relation; (3) use FD constraints to verify the integrity of data from different sources.

1 Introduction

Schema integration [2, 3, 8, 16] is the activity to integrate the schemas of existing or proposed databases into a global, unified schema. The application of schema integration includes view integration, i.e. to produce a global schema of a proposed DB by integrating user views in DB design, and DB integration, i.e. to produce a global schema of a collection of databases. Recently the integration of heterogeneous multiple databases [15, 16] become a hot topic as it meets business requirements [18]. When integrating database schemas, people need to resolve several kinds of semantic incompatibilities, such as naming conflicts (i.e. the same name is used for two concepts, or the same concept is described by different names), structural conflicts (i.e. different structures are used to model the same concept, also called schema mismatch), constraint conflicts (different choice of integrity constraints) and so on. Once the conflicts in schema level are resolved, the next step is data integration [17], which involves reconciliation of data at the instance level. The tasks of data integration include object matching (i.e. to eliminate duplicates), inference of missing values, error correction and so on.

In schema integration, one kind of semantic incompatibility, i.e., *schematic discrepancy*, hasn't been well studied in existing works. In relational model, *schematic discrepancy* occurs when the same information is modeled as attribute values, DB names, relation names or attribute names in different schemas.

Example 1.1: In Figure 1.1, the four databases use different schemas to record the same information: prices of products supplied by some suppliers in different months. In *DB1*, all the information (i.e., products, suppliers, months and prices) are modeled as values, while in *DB2* and *DB4*, the months *Jan,…,Dec* which are attribute values in *DB1* become attribute names (whose values are prices of products). And in *DB3* and *DB4*, the supplier names *s1, s2,…* which are also attribute values in *DB1* become relation names. Note in *DB3* and *DB4*, universal relation assumption does not hold, i.e., in these databases, same attribute names may represent different meanings. For example, in *DB3*, the attribute price in the relations *s1* and *s2* represent the prices of products supplied by *s1* and *s2* respectively. And therefore people can not simply merge the two relations to one whose schema is the same as *s1* or *s2*.□

DB1:
Supply

product	supplier	month	price
p1	s1	Jan	100
p1	s1	Feb	105
p1	s2	Jan	99
p1	s2	Feb	107
…	…	…	…

DB2:
Supply

product	supplier	Jan	Feb	…	Dec
p1	s1	100	105	…	110
p1	s2	99	107	…	103
…	…	…	…	…	…

DB3:

s1

product	month	price
p1	Jan	100
p1	Feb	105
…	…	…

s2

product	month	price
p1	Jan	99
p1	Feb	107
…	…	…

…

DB4:

s1

product	Jan	Feb	...	Dec
p1	100	105	…	110
…	…	…	…	…

s2

product	Jan	Feb	...	Dec
p1	99	107	…	103
…	…	…	…	…

…

Figure 1.1: Schematic discrepancy: supplier names and months are modeled differently in different databases

When people interoperate schematic discrepant data in relational databases, generally they need to transform between relations in which data are modeled as attribute names, relation names, DB names and data values respectively. In this paper, we call such transformations "*schematic discrepant transformation*" (or just "transformation"). For example, the transformation between any pair of DBs in Figure 1.1 is such a transformation.

Since schematic discrepancy is common in real life applications [12], people have studied how to transform, how to query schematic discrepant data [4, 12], how to use schematic discrepancy [5, 13] a lot recently. But existing works focus on the use of schematic discrepant transformations, little work has been done on the transformations themselves, such as the reconstructibility of restructuring operators, which determines the correctness of works using schematic discrepant transformations. Another interesting work is to study how integrity con-

straints (ICs) evolve during transformations; and how to use ICs in works handling schematic discrepancy. [6] identified the roles of ICs in database interoperation, and [7, 9] studied the derivation of ICs in integrated schema. But none of them consider schematic discrepancy in schema integration.

Organization and Contributions: Inferring and using FDs in schematic discrepant transformations is the main objective of this paper. In Section 2, we first survey some previous works in schematic discrepancy. The main contribution of this paper is in Section 3-6:

In Section 3, we study the schematic discrepant transformations in detail. Generally, schematic discrepant transformations can be implemented using restructuring operators: *unfold, fold, split, unite*, *db_split* and *db_unite* (Section 3.1, 3.2). In Section 3.3, we give the reconstructibility and commutativity of those operators, which is useful to simplify a transformation.

To infer FDs in schematic discrepant transformations, we need to consider a broad class of ICs which may be changed to FDs in transformed relations. To this end, we propose restricted FD (Section 4), which is an extension to regular FD.

Then in Section 5, we study how to infer FDs in schematic discrepant transformations. We first study the properties of restructuring operators on how restricted FDs changing when applying each kind of operators (Section 5.1). Using these properties, we give an algorithm to infer FDs in a transformation which is implemented as a sequence of restructuring operators (Section 5.2). We prove that our algorithm can compute all the FDs in the transformed relations which can be inferred from restricted FDs in original relations.

At last, in Section 6, we study how to use FDs in a transformed relation. Specifically, we identify the use of FDs in 3 scenarios: (1) Use FDs as a condition to check the correctness of SchemaSQL views [4]. To this end, we define the correctness of a SchemaSQL view, and give a method to automatically check the correctness of views; (2) Use FDs to normalize a transformed relation; (3) Use FD constraints to verify the integrity of data from different sources.

Section 7 concludes the whole paper.

Notations: In this paper, schema labels (database names, relation names and attribute names) and query statements are represented using *italic*. The notation *DB::R* is used to represent a relation *R* in a database *DB*, conforming SchemaSQL conventions. And *dom(A)* is used to represent the domain of the attribute *A* in a relation.

2 Research Works in Schematic Discrepancy

In this section, we survey some previous works in schematic discrepancy. We first introduce a multi-database language, SchemaSQL, which is used widely to solve schematic discrepancy problem in different places. Then we introduce some works using schematic discrepant transformations.

2.1 SchemaSQL

In this subsection, we survey Lakshmanan et al's work, SchemaSQL language [4,5]. SchemaSQL is an extension to SQL for enabling multi-database interoperability. It treats data and schema labels in a uniform manner, i.e., variables can range data and schema labels, which facilitates the interoperability among schematic discrepant databases. Furthermore, it can be used to define restructuring views which are schematic discrepant from original relations.

In [4], Lakshmanan et al proposed five types of variables for SchemaSQL: (1) The expression "**à** *D"* declares *D* to be a database name variable ranging over all database names in a

federation. (2) The expression "*db → R*" declares *R* to be a relation name variable ranging over all relation names in *db*. (3) The expression "*db::rel → A*", declares *A* to be an attribute name variable ranging over attribute names in the relation *rel* of database *db*. (4) The expression "*db::rel T",* declares *T* to be a tuple variable ranging over all tuples in the relation *rel* in the database *db*. (5) The expression "*db::rel.attr V*", declares *V* to be a domain variable ranging over the set of values of attribute *attr* in *db::rel*. Note that the declarations of variables are nested. That is, the declaration of a variable itself may include some variables.

Example 2.1: in Figure 1.1, we can transform *DB4* to *DB1* using the following view definition:

```
create view  DB1::Supply(product, supplier, month, price)
select                T.product,  S,  M, T.M
from          DB4 → S,  DB4::S → M,  DB4::S T
where         M<>product
```

There're 4 variables in the view definition, i.e., relation name variable *S* (ranging over relation names in *DB4*), attribute name variable *M* (ranging over attribute names in the relation *DB4::S*, and *M* can not be *product*), tuple variable *T,* and domain variables *T.product* and *T.M*.□

We find some SchemaSQL views are problematic. In Section 6.1, we will see how to use FDs to check the correctness of SchemaSQL views.

2.2 Research Works Using Schematic Discrepant Transformations

Originally raised in database integration, schematic discrepant transformations are used in a broad area in database world. The follows list some of the works.

- *Querying using dynamic views:* In [13], Miller identified three scenarios in which schematic discrepancy may occur. That is database integration, data publication on the web and physical data independence. To solve these problems, Miller proposed to use SchemaSQL views [4], which is an extension to the idea of answering queries using views ([1]).
- *Storage and querying of e-commerce data*: In [11], Agrawal et al argued that new generation of e-commerce applications require data schemas that are constantly evolving and sparsely populated. They believed that a vertical representation of objects is much better on storage and querying performance than conventional horizontal row representation. On the other hand, to facilitate writing queries, they create a horizontal view of the vertical table, and transform queries on this view to the vertical table. The following figure shows an example of horizontal and vertical schemas. The attribute names *A1, A2* and *A3* in horizontal schema are modeled as *Key* values in vertical schema.

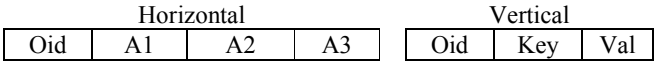


Figure 2.1: Horizontal and vertical representations

- *Data cube:* In data warehouse, typically data are multi-dimensional (with several dimensional attributes and one measure attribute), which is modeled as a data cube conceptually. For example, in Figure 1.1, the data has 3 dimension attributes (*product, supplier* and *month*) and one measure attribute (*price*). Furthermore, *DB1::Supply* can be seen as a logical schema to keep the data in a relational DB, since the values of dimension attributes are modeled as data values in that schema (i.e. a fact table). Users usually require generating report tables which can only have 2 dimensions. The relations in DB4 are examples of report tables. That

is, for each value of the dimension attribute *supplier*, generate a table, such that the tuples in the table have the same *supplier* value.

❙ *Physical database design:* In physical database design, to achieve high performance, people may horizontally partition a table based on values of some column. E.g., in Figure 1.1, Suppose *DB1::Supply* is the original table. We can partition the table based on supplier values, i.e. one table for each supplier. And the attribute *supplier* can be extracted out from each partitioned table since the values of the attribute in each partitioned table are the same. Consequently, we obtain *DB3* as the result.

3 Schematic Discrepant Transformation

In this section, we study schematic discrepant transformation itself. First, we give the definitions and properties of restructuring operators which are used to implement schematic discrepant transformations (Section 3.1). Then we study schematic discrepant transformations based those operators (Section 3.2).

3.1 Restructuring Operators

In this subsection, we first define the restructuring operators which are used to implement a schematic discrepant transformation, then we give properties (reconstructibility and commutativity) of those restructuring operators.

3.1.1 Definitions of Restructuring Operators

In this subsection, we give the definitions of restructuring operators (i.e., *unfold, fold, split, unite, db_split* and *db_unite*). The former four operators are applied in one database, while the latter two are applied on a higher, multidatabase level. The names of the former four operators are the same as those in [5], but the definitions are different for *unfold* and *fold* operators. [5] did not distinguish *null value* and *no value* (represented as "-") in the definitions, which will cause some problems in transformation. The definitions of *split* and *unite* are the same as those in [5]. *db_split* and *db_unite* are defined by us.

***unfold*(*R,B,C*):** Let R be a relation with the schema $R(A_1,...,A_n,B,C)$. *unfold(R,B,C)* transforms R to a relation $S(A_1,...,A_n,b_1,...,b_m)$, where $\{b_1,...,b_m\}$ is the set of distinct values appearing in column B of R. The content of S is defined as:

$$S=\{(a_1,...,a_n,c_1,...,c_m) \mid (c_i \neq \text{"-"} \,\&\, (a_1,...,a_n,b_i,c_i) \in R) \text{ or } (c_i=\text{"-"} \,\&\, (\neg\exists \text{ a tuple } t \in R: t[A_1,...,A_n,B] = (a_1,...,a_n,b_i)), 1 \le i \le m\}.$$

Example 3.1: Figure 3.1 explains the difference between *null value* and *no value* ("-"). In the figure, *R2=unfold(R1,supplier,price)*. In *R2*, the no-value symbol "-" is used as not exist a tuple $t \in R$, such that t[*product,supplier*]=(*p2,s2*).

In [5], they did not distinguish *no value* and *null value*, and therefore no matter the 2nd tuple of *R1* exists or not, *R1* will be *unfold* to a same relation. □

R1

product	supplier	price
p1	s1	100
p1	s2	null
p2	s1	200

R2

Product	s1	s2
p1	100	null
p2	200	-

Figure 3.1: Distinguish *null value* and *no value* ("-")

***fold(R,B,C)*:** Let R be a relation with the schema $R(A_1,...,A_n,b_1,...,b_m)$. Suppose $b_1,...,b_m$ are values from $dom(B)$, and all entries appearing in columns $b_1,...,b_m$ of R are from $dom(C)\cup\{\text{-}\}$ (the no-value symbol), for some attribute names $B,C\notin\{A_1,...,A_n\}$. $fold(R,B,C)$ transforms R to a relation $S(A_1,...,A_n,B,C)$, defined as:

$$S=\{(a_1,...,a_n,b_i,c_i) \mid \exists\, t\in R\text{: } t[A_1,...,A_n]=(a_1,...,a_n) \;\&\; t[b_i]=c_i \;\&\; c_i\neq\text{“-”}\}.$$

***split(R,B)*:** Let R be a relation with the schema $R(A_1,...,A_n, B)$. $split(R,B)$ transforms R to a set of relations $b_i(A_1,...,A_n)$, for each b_i appearing in column B of R. The content of b_i is defined as:

$$b_i=\{t[A1,...,An] \mid t\in R \;\&\; t[B]=b_i\}.$$

***unite(R_B,B)*:** Let $R_B=\{b_1,...,b_m\}$ be a set of relations in a given database, such that each relation name b_i (i=1,2,…,m) is an element of the domain of some fixed attribute B, and all the relations have a common schema $b_i(A_1,...,A_n)$. $unite(R_B,B)$ transforms $b_1,...,b_m$ to a relation $S(B,A_1,...,A_n)$, defined as:

$$S=\{t \mid \exists\, t'\in b_i\text{: } t[A1,...,An]=t'[A1,...,An] \;\&\; t[B]=b_i\}.$$

In the following, we give the definitions of *db_unite* and *db_split*, which are quite similar to *unite* and *split* resp. The difference is that *db_unite* is applied on a set of relations from different databases and transform the database names to attribute values in the transformed relation, while *unite* is applied on a set of relations from one database and transform relation names to attribute values in the transformed relation. *db_split* does the converse of *db_unite.*

***db_split(R,B)*:** Let R be a relation with the schema $R(A_1,...,A_n,B)$. $db\_split(R,B)$ transforms R to a set of relations $b_i::S(A_1,...,A_n)$, for each b_i appearing in column B of R. The content of $b_i::S$ is defined as:

$$b_i::S=\{t[A_1,...,A_n] \mid t\in R \;\&\; t[B]=b_i\}.$$

db_unite(R_B,B): Let $R_B=\{b_1::R,...,b_m::R\}$ be a set of relations from different databases, such that each database name b_i (i=1,2,…,m) is an element of the domain of some fixed attribute B, and all the databases have a relation with the same schema $R(A_1,...,A_n)$. $db\_unite(B)$ transforms $b_1::R,...,b_m::R$ to a relation $S(B,A_1,...,A_n)$, defined as:

$$S=\{t \mid \exists\, t'\in b_i::R,\; t[A_1,...,A_n]=t'[A_1,...,A_n] \;\&\; t[B]=b_i\}.$$

Example 3.2: Figure 3.2 shows an example using those operators to transform database schemas. Databases *DB1, DB2* and *DB3* are from Figure 1.1. Databases *s1,...,sn* (i.e. supplier names are modeled as database names) have a common relation *Supply*. Suppose in the relation *DB2::Supply*, the FD $product,supplier\rightarrow Jan,...,Dec$ holds (the reason why we need this FD will be explained later in Theorem 3.4, Section 3.1.2). □

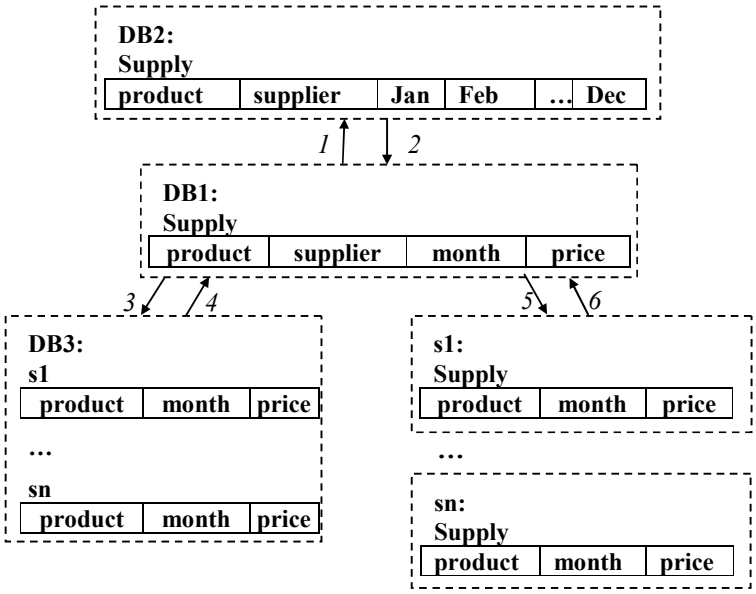


Figure 3.2: Restructuring operators: (1) *unfold(DB1::Supply,month,price)*, (2) *fold(DB2::Supply,month,price)*, (3) *split(DB1::Supply,supplier)*, (4) *unite({DB3::s1,...,DB3::sn},supplier)*, (5) *db_split(DB1::Supply,supplier)*, (6) *db_unite({s1::Supply,..., sn::Supply},supplier)*

3.1.2 Reconstructibility and Commutativity of Restructuring Operators

In the following, we give two kinds of properties, reconstructibility and commutativity, of restructuring operators. These properties can be used to simplify a transformation implemented using a sequence of restructuring operators.

We first see the reconstructibility. That is, relations transformed by applying *unite* (*fold*, *db_unite*) can be recovered to original relations by applying *split* (*unfold*, *db_split*) on the transformed relation, and vice versa.

Theorem 3.1 (*Reconstructibility of split*): Let R be a relation with the schema $R(A_1,...,A_n,B)$. Then:

$$unite(split(R,B),B)=R.$$

Theorem 3.2 (*Reconstructibility of unite*): Let $R_B=\{b_1,...,b_m\}$ be a set of relations in a given database, such that each relation name b_i (i=1,2,…,m) is an element of the domain of some fixed attribute B. Suppose also that they all have a common schema $b_i(A_1,...,A_n)$. Then:

$$split(unite(R_B,B),B) = R_B.$$

The reconstructibility of *db_split* and *db_unite* are similar to *split* and *unite*, which are omitted by us.

Theorem 3.3 (*Reconstructibility of unfold*): Let R be a relation with the schema $R(A_1,...,A_n,B,C)$. Then:

$$fold(unfold(R,B,C),B,C)= R.$$

Theorem 3.4 (*Reconstructibility of fold*): Let R be a relation with the schema $R(A_1,...,A_n,b_1,...,b_m)$. Suppose $b_1,...,b_m$ are values from $dom(B)$, and all data values appearing the columns $b_1,...,b_m$ of R are from $dom(C) \cup \{-\}$ (the no-value symbol), for some attribute names $B,C \notin \{A_1,...,A_n\}$.

If in R, the FD $A_1,...,A_n \rightarrow b_1,...,b_m$ holds, then $unfold(fold(R,B,C),B,C) = R$.

We omit the formal proofs of the above 4 theorems, while explain the reconstructibility of *fold* a little. Generally, the $fold(R,B,C)$ operator is a many to one mapping from original relations to transformed relations, which causes the transformation not reconstructible. But if $A_1,...,A_n \rightarrow b_1,...,b_m$ holds in R, we can ensure the mapping is one to one. The following example explains this.

Example 3.3 (Figure 3.2): Suppose in relations *R1* and *R2*, the attribute names *b1,b2* are values of a fixed attribute *B*, and the values of *b1,b2* are values of a fixed attribute *C*. Suppose the FD $A \rightarrow b1,b2$ does not hold in *R1* and *R2*.

We can transform *R1* or *R2* to the same relation *S* by applying $fold(Ri,B,C)$ *(i=1* or *2)*. That is, the mapping from the original relations to the transformed relations is many to one, which makes the recovering impossible. □

R1

A	b1	b2
a1	c1	c2
a1	c3	c4

R2

A	b1	b2
a1	c1	c4
a1	c3	c2

S

A	B	C
a1	b1	c1
a1	b2	c2
a1	b1	c3
a1	b2	c4

Figure 3.2: Un-recoverable *fold* operator

In practice, reconstructibility of operators is important to the implementation of lossless transformations, as we will see in Section 3.2.2.

Next, let's see the other kind of property, commutativity, of restructuring operators. That is, operators can commute in a transformation. In the following, we only give the commutativity of *fold* and *unite* operators. The commutativity of the other pairs of restructuring operators are similar.

Theorem 3.5 (*Commutativity* of *fold and unite*): Let $b_1,...,b_m$ be a set of relations in a given database, such that each relation label b_i (i=1,2,…,m) is an element of the domain of some fixed attribute B_1. Suppose also that they all have a common schema $b_i(A_1,...,A_n,u_1,...,u_m)$. For each b_i, suppose $u_1,...,u_m$ are values from $dom(B_2)$, and all data values appearing the columns $u_1,...,u_m$ are from $dom(C) \cup \{-\}$ (the no-value symbol), for some attribute names $B_1,B_2,C \notin \{A_1,...,A_n\}$. Then:

$$unite(\{fold(b_i,B_2,C) \mid i=1,...,m\},B_1) = fold(unite(\{b_1,...b_m\},B_1),B_2,C).$$

Example 3.4: Suppose in Figure 1.1, we transform *DB4* to *DB1*. This can be implemented using any one of the following transformations:

DB1::Supply=unite({fold(DB4::si,month,price) | i=1,...,n}, supplier), or

DB1::Supply=fold(unite({DB4::s1,...,DB4::sn},supplier),month,price).□

3.2 Schematic Discrepant Transformation

In this sub-section, we study schematic discrepant transformations based on restructuring operators. We first introduce how to implement a schematic discrepant transformation using a sequence of restructuring operators. Then we define lossless and non-redundant transformations.

3.2.1 Implement Schematic Discrepant Transformations Using Restructuring Operators

In this paper, we study schematic discrepant transformations implemented using a sequence of restructuring operators (i.e. *unfold, fold, split, unite, db_split* and *db_unite*). Generally, we can represent a transformation as follows:

$$R0 \xrightarrow{T1} R1 \xrightarrow{T2} \ldots \xrightarrow{Tn} Rn$$

In the above representation, each Ri represents a relation or a set of relations from one or a set of databases. $R0$ and Rn are the original and transformed relations resp; $R1,\ldots,Rn\text{-}1$ are intermediate relations. Each Ti represents one or a set of restructuring operators which transform $Ri\text{-}1$ to Ri. We call the sequence $T=<T1,\ldots,Tn>$ a *schematic discrepant transformation* (or just *transformation*).

Example 3.5: in Figure 1.1, we want to transform the relations in *DB4* to the relation in *DB2*. The transformation can be implemented as follows:

$$\text{DB4} \xrightarrow{fold(si,month,price)} \text{DB3} \xrightarrow{unite(\{s1,\ldots,sn\},supplier)} \text{DB1} \xrightarrow{unfold(Supply,month,price)} \text{DB2}$$

That is, firstly transform the relations in *DB4* to the relations in *DB3* by applying *fold(si,month,price)* on each relation *si* (*i=1,…,n)* in *DB4*; then transform the relations in *DB3* to the relation in *DB1* by applying *unite({s1,…,sn},supplier)* on the relation set {*s1,…,sn*} of *DB3*; at last transform the relation in *DB1* to the one in *DB2* by applying *unfold(Supply,month,price)* on the relation *Supply* in *DB1*.

Note there're other sequences of operators which can transform *DB4* to *DB2*. For example, perform *unite({s1,…,sn},supplier)* on the relations of *DB4* directly, and get the target relation *DB2::Supply*. But in works using schematic discrepant transformation, it is possible to generate a redundant transformation as shown above. We'll give such an example in Section 6.1. And in Section 3.2.2, we propose to simplify a transformation to a non-redundant (shortest) one.□

Generally, using restructuring operators, we can implement schematic discrepant transformations as follows. Given original relation (or relation set) *R* and target relation (or relation set) *S*, a transformation is implemented in 2 phases:

(1) using *fold*, *unite* and/or *db_unite* to transform *R* to a relation, say *R'*, such that schema labels in *R* become attribute values in *R'*;

(2) then using *unfold, split* and/or *db_split* to transform *R'* to *S*, such that attribute values in *R'* become schema labels in *S*.

In this way, we can implement any complex transformations. That is, if some information is modeled differently in original relations and transformed relations, which cause the schematic discrepancy, we first transform the information in original relations to attribute values in Phase 1, then transform the attribute values to needed positions (attribute names, relation names or database names) in Phase 2. Note not all transformations need both the 2 phases. In

Example 3.5, the preceding *fold* and *unite* operators consist Phase 1, and the following *unfold* operator consists Phase 2.

3.2.2 Lossless and Non-redundant Transformation

Generally, one is mostly interested in semantics-preserving transformations, i.e. transformations such that both original and transformed relations represent exactly the same real world facts, though with a different syntax. A relation (or relation set) can be losslessly converted into another relation (or relation set), and conversely, hence the name of *lossless transformation.*

Definition 3.1 (*Lossless transformation*): Given a schematic discrepant transformation T, let S be the schema of the original relation (or relation set) of T. If there exist an inverse transformation T' of T, such that for any instance r of S,

$$T'(T(r))=r$$

then T is called a *lossless transformation*.

Theorem 3.6: Given a schematic discrepant transformation T consisting of a sequence of restructuring operators, if each *fold* operator in T satisfies the reconstructibility, then T is a lossless transformation.

A lossless transformation may be redundant because unnecessary operators are used. And using reconstructibility and commutativity of restructuring operators, we can simplify a redundant transformation.

Definition 3.2 (*Non-redundant transformation*): Given two transformations T and T' performed on a same relation (or relation set) R, we define a partial order $\leq$ as follows: $T' \leq T$ provided $\forall t' \in T' \exists t \in T$: t' and t are of the same kind of operators, and have the same attribute names as parameters. A transformation T is non-redundant provided $\forall T' \leq T$ & $T'(R)=T(R) \Rightarrow T \leq T'$.

Lemma 3.1: A lossless transformation T is a non-redundant transformation iff in T, the operators *db_unite*(R_B,*B*) and *db_split*(*R*,*B*) are not used at the same time, neither are *unite*(R_B,*B*) and *split*(*R*,*B*), and neither are *fold*(*R*,*B*,*C*) and *unfold*(*S*,*B*,*C*), where R_B is a set of relation names, R and S are relation names, B and C are attribute names.

Intuitively, in a non-redundant transformation which is lossless, we never apply *db_unite* on some attribute B, and apply *db_split* on the same attribute later on, so do *unite* and *split*, and so do *fold* and *unfold.* The transformation in Example 3.3 is not a non-redundant transformation, since *fold(DB4::si,month, price)* and *unfold(DB1::Supply, month, price)* have the common attributes (*month* and *price*) as parameters.

Generally, given a lossless transformation implemented using a sequence of restructuring operators which satisfy reconstructibility and commutativity, we can simplify the transformation by removing redundant operators. That is, for example, if both *unite*(R_B,*B*) and *split*(*R*,*B*) are used in the transformation, which have the common attribute B as the parameter, we can first bring them together by swapping them with other operators (commutativity), then remove them both (reconstructibility). The cases for other operators are similar.

Theorem 3.7: A lossless transformation can be simplified to a non-redundant one.

Example 3.6: Given the transformation in Example 3.5, according to Theorem 3.5, we can commute the set of *fold(DB4::si,month,price)* operators with *unite({s1,…,sn},supplier)* and get the following transformation.

$$DB4 \xrightarrow{unite(\{s1,...,sn\},\ supplier)} DB2 \xrightarrow{fold(Supply,\ month,price)} DB1 \xrightarrow{unfold(Supply,\ month,price)} DB2$$

Suppose in Figure 1.1, the FD $product,supplier \rightarrow Jan,Feb,...,Dec$ holds in *DB2::Supply*. Then both the *fold* and *unfold* operators can be removed according to the reconstructibility of *fold*. Consequently only the *unite* operator is needed in this transformation. □

4 Restricted FDs

Integrity Constraints (ICs), including FDs, are identified and specified based on the semantics of the real-world enterprise being modeled. But an IC may be expressed differently in different relations modeling a same real-world enterprise. Specifically, in schematic discrepant transformations, some FDs in original relations may not be expressed as FDs in transformed relations. And some ICs which are not FDs in original relations become FDs in transformed relations.

Example 4.1: Suppose in Figure 1.1, the databases model a same real world application, but with different schemas. Suppose a constraint says, "*the prices of products supplied by supplier s1 are unique in the same month.*" This constraint is expressed as the FD $product,month \rightarrow price$ in the relation *s1* of *DB3*, or as $product \rightarrow Jan,Feb,...,Dec$ in the relation *s1* of *DB4*. But can not be expressed as any FD in *DB1* and *DB2*, as the other suppliers may not use the same constraint. □

Our goal is to obtain FDs in transformed relations given ICs in original relations. Our solution consists of two parts: (1) Expressing ICs of original relations in a standard form. This facilitates automatically inferring FDs for transformed relations; and (2) Given a transformation implemented using a sequence of restructuring operators, inferring FDs holding in the transformed relations from ICs in original relations. For part 1 (this section), we propose restricted FDs which is an extension to regular FDs to express ICs. For part 2 (Section 5), we study through the restructuring operators.

In the following, we explain the concept of restricted FD. Intuitively, the restriction information may be modeled as attribute values, attribute names, relation names or database names. In the following, we first explain the former three cases one by one using example 1.1 (the case restriction information is modeled as database names is similar to the case of relation names). Then we give the general definition of restricted FDs which concludes all the cases.

Example 4.2: Suppose in *DB1* (Figure 1.1), an IC says, (**C1**) "*in the first quarter, the price of a product is determined by the product and supplier.*" So the IC is only valid when months are *Jan, Feb* are *Mar.* We represent this IC as follows.

$$DB1::Supply(product,supplier,month_{\sigma=\{Jan,Feb,Mar\}} \rightarrow price).$$

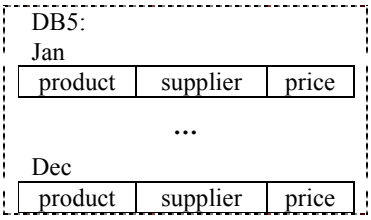
The meaning can be interpreted as follows. Given any pair of tuples *t1, t2* from *DB1::Supply*, If *t1.product=t2.product*, *t1.supplier=t2.supplier*, and both *t1.month* and *t2.month* are from {*Jan,Feb,Mar*}, then *t1.price=t2.price.* □

Example 4.3: Suppose in *DB2* (Figure 1.1), the same IC, C1, holds. Suppose in *DB2::Supply*, the attribute names *Jan, Feb,...,Dec* are values of attribute *month*, and the values of attributes *Jan, Feb,...,Dec* are values of attribute *price*. Note *month* and *price* do not appear in the schema of *DB2::Supply*. We can represent C1 in *DB2* as follows. Note though the IC is the same, the restriction information (months) is modeled as attribute names now.

$$DB2::Supply(product,supplier \rightarrow price(month_{\sigma=\{Jan,Feb,Mar\}}))$$

The meaning of the IC is interpreted as follows. Given any pair of tuples *t1, t2* in *DB2::Supply* (*t1, t2* may refer to the same tuple), if *t1.product=t2.product* and *t1.supplier=t2.supplier*, then *t1.M1=t2.M2* for any *M1,M2* ∈ {*Jan,Feb,Mar*} such that *t1.M1* ≠ "-" and *t2.M2* ≠ "-", i.e., the values of the attributes *Jan, Feb, Mar* in *t1* and *t2* are all the same except the no-values "-". □

Example 4.4: Suppose in Example 1.1, we have another database *DB5*:



The relation names *Jan,...,Dec* are from the domain of attribute *month*. Suppose the same IC, C1, holds in *DB5*. But the restriction information (months) is modeled as relation names in this case. We can represent C1 as follows.

$$DB5::month_{\sigma=\{Jan,Feb,Mar\}}(product,supplier \rightarrow price).$$

The meaning is interpreted as follows. Let $R=DB5::Jan \cup DB5::Feb \cup DB5::Mar$. Then the FD $product,supplier \rightarrow price$ holds in R. □

In the following we give the general definition of restricted FD, considering all the cases restriction information can be modeled.

Definition 4.1 (*Restricted FD*): Generally a restricted FD is represented as:

$$DB::R(X \rightarrow Y).$$

DB represents one or a set of database names. It can be one of the following forms:

(1) a fixed database name, or

(2) $B_{\sigma=S}$, representing a set of database names S, such that *B* is an attribute name, $S \subseteq dom(B)$, and each element of S is modeled as a database name.

R represents one or a set of relation names. It can be one of the following forms:

(1) a fixed relation name, or

(2) $B_{\sigma=S}$, representing a set of relation names S, such that *B* is an attribute name, $S \subseteq dom(B)$, and each element of S is modeled as a relation name.

Let $DB_1,...,DB_m$ be the databases denoted by *DB*, i.e. if *DB* is a fixed database name, *m=1*, and $DB_1=DB$; otherwise (i.e., *DB* has a form $B_{\sigma=S}$), $\{DB_1,...,DB_m\}=S$. Similarly, let $R_1,...,R_n$ be relations denoted by *R*. Then each DB_i ($i=1,...,m$) contains the set of relations $\{R_1,...,R_n\}$, and all the relations $DB_i::R_j$ ($i=1,...,m, j=1,...,n$) have the same schema.

X is a set. An element of *X* can be one of the following forms:

(1) *A*, an attribute name of relation $DB_i::R_j$, or

(2) $A_{\sigma=S}$, where *A* is an attribute name of relation $DB_i::R_j$, and $S \subseteq dom(A)$.

Y is a set. An element of Y can be one of the following forms:

(1) A, an attribute name of relation $DB_i::R_j$, or

(2) $C(B_{\sigma=S})$, where B, C are attribute names not appearing in $DB_i::R_j$, $S \subseteq dom(B)$, and each element of S is modeled as an attribute name of relation $DB_i::R_j$, the values of which are from $dom(C)$.

Let R' be the union of all the relations $DB_i::R_j$, i.e. $R' = \mathop{\mathbf{U}}_{i=1,\ldots,m;\ j=1,\ldots,n} DB_i::R_j$. For any two tuples $t1$, $t2$ of R', let Z be an element of X or Y. We define the meaning of sentence "$t1[Z]=t2[Z]$" as follows.

Case1 ($Z = A$): $t1[A] = t2[A]$;

Case2 ($Z = A_{\sigma=S}$): $t1[A] \in S$ & $t2[A] \in S$;

Case3 ($Z = C(B_{\sigma=S})$): $t1[M1]=t2[M2]$, for any $M1$, $M2 \in S$ such that $t1[M1] \neq$ "-" and $t2[M2] \neq$ "-".

Let $X=\{X_1,\ldots,X_l\}$, $Y=\{Y_1,\ldots,Y_k\}$, we call the restricted FD $DB::R(X \rightarrow Y)$ holds if the following holds for any pair of tuples $t1$, $t2$ of R':

If $t1[X_1]=t2[X_1]$ and … and $t1[X_l]=t2[X_l]$, then $t1[Y_1]=t2[Y_1]$ and … and $t1[Y_k]=t2[Y_k]$.

This completes the definition of *restricted FD*.

We call "$DB::R$" the *context* of the restricted FD $DB::R(X \rightarrow Y)$. If the context is clear, i.e. $DB::R$ refers to an only relation, we may omit the context and represent the restricted FD as $X \rightarrow Y$ directly. Furthermore, if X and Y only include attributes names, $X \rightarrow Y$ is just a regular FD holding in the relation $DB::R$.

At last, before ending this section, we give an interesting property of restricted FD which will be used in the rest of the paper.

Theorem 4.1: The set of restricted FDs $DB::R(X,Z_{\sigma=\{z\}} \rightarrow Y)$ hold for each z appearing in column Z of the relations denoted by $DB::R$ iff $DB::R(X,Z \rightarrow Y)$ holds.

Example 4.5: Suppose in Figure 1.1, in *DB1,* the restricted FDs $DB1::Supply(product,month,supplier_{\sigma=\{si\}} \rightarrow price)$ hold for each *si* appearing in column *supplier* of the relation *DB1::Supply*, i.e., for each supplier, the prices are determined by products and months. According to Theorem 4.1, these restricted FDs are equivalent to the FD $supplier,product,month \rightarrow price$ in *DB1::Supply*. □

5 Inferring FDs in Schematic Discrepant Transformations

In this section, we study how to infer FDs in schematic discrepant transformations. The purpose is to obtain FDs in transformed relations from restricted FDs in original relations. Section 5.1 gives properties of restructuring operators on how restricted FDs change when applying each kind of restructuring operators. Section 5.2 gives an algorithm to use those properties given in Section 5.1 to infer FDs in schematic discrepant transformations. We also prove the properties given in Section 5.1 are sufficient to infer all the FDs in transformed relations from restricted FDs in original relations.

5.1 Changing of Restricted FDs When Applying Restructuring Operators

In the following, we give the properties on how restricted FDs change when applying each kind of restructuring operators. We give the properties of *split/unite*, *db_split/db_unite* and *unfold/fold* in a pairwise way.

Theorem 5.1 (*Property of split/unite*): Given a relation with schema $DB1::R(A_1,...,A_n,B)$, let $DB2::b_i(A_1,...,A_n)$ (i=1,2,…m) be the schemas of relations obtained by transforming R using *split(DB1::R,B)*, i.e., the distinct values of B in R, $\{b_1,...,b_m\}$, become relation names of the transformed relations. We have the following property:

$$DB2::B_{\sigma=S}(X \rightarrow Y) \text{ holds iff } DB1::R(X,B_{\sigma=S} \rightarrow Y) \text{ holds}$$

where $S \subseteq dom(B)$; B does not appear in X and Y.

This property also holds for *unite* operator. That is, given a set of relations $R_B=\{DB2::b_i(A_1,...,A_n) \mid 1 \le i \le m\}$, such that each b_i (i=1,2,…,m) is an element of the domain of some fixed attribute B, let $DB1::R(A_1,...,A_n,B)$ be the relation transformed by applying $unite(R_B,B)$. Then the above property holds. □

Example 5.1: Suppose in Figure 1.1, we transform *DB3* to *DB1*. Suppose in *DB3*, the FD $DB3::supplier_{\sigma=\{s1\}}(product,month \rightarrow price)$ holds. Then according to Theorem 5.1, in *DB1*, $DB1::Supply(product,month,supplier_{\sigma=\{s1\}} \rightarrow price)$ holds.□

Then we give the property of *db_split/db_unite* which is similar to *split/unite*.

Theorem 5.2 (*Property of db_split/db_unite*): Given a relation with schema $DB1::R(A_1,...,A_n,B)$, let $b_i::R(A_1,...,A_n)$ (i=1,2,…m) be the schemas of relations obtained by transforming *DB::R* using *db_split(DB1::R,B)*, i.e., the distinct values of B in *DB1::R*, $\{b_1,...,b_m\}$, become database names of the transformed relations. We have the following property:

$$B_{\sigma=S}::R(X \rightarrow Y) \text{ holds iff } DB1::R(X,B_{\sigma=S} \rightarrow Y) \text{ holds,}$$

where $S \subseteq dom(B)$; B does not appear in X and Y.

This property also holds for *db_unite* operator. That is, given a set of relations $R_B=\{b_i::R(A_1,...,A_n) \mid 1 \le i \le k\}$, such that each b_i (i=1,2,…,m) is an element of the domain of some fixed attribute B, let $DB1::R(A_1,...,A_n,B)$ be the relation transformed by applying $db\_unite(R_B,B)$. Then the above property holds. □

We omit the example of Theorem 5.2 as it is quite similar to Theorem 5.1.

Theorem 5.3 (*Properties* of *unfold/fold*): Given a set of relations with schema $DB1_i::R_j(A_1,...,A_n,B,C)$, ($i=1,...,k$; $j=1,...,l$), let $DB2_i::R_j(A_1,...,A_n,b_1,...,b_m)$ be the schema of the relation obtained by transforming $DB1_i::R_j$ using $unfold(DB1_i::R_j,B,C)$ operator for each pair of (i, j) values, i.e., the values of B in $DB1_i::R_j$, $\{b_1,...,b_m\}$, become attribute names in $DB2_i::R_j$, and the values of C in $DB1_i::R_j$ become values of $b_1,...,b_m$ in $DB2_i::R_j$. We have the following properties.

(1) $DB2::R(X \rightarrow C(B_{\sigma=S}))$ holds iff $DB1::R(X,B_{\sigma=S} \rightarrow C)$ holds;
(2) $DB2::R(X \rightarrow Y)$ holds iff $DB1::R(X \rightarrow Y)$ holds;
(3) $DB2::R(X,b_{i\,\sigma=V} \rightarrow Y)$ holds iff $DB1::R(X,B_{\sigma=\{bi\}},C_{\sigma=V} \rightarrow Y)$ holds,

where *DB2::R* and *DB1::R* represent a subset of relations of $\{DB2_i::R_j \mid 1 \le i \le k,\ 1 \le j \le l\}$ and $\{DB1_i::R_j \mid 1 \le i \le k,\ 1 \le j \le l\}$ resp.; $S \subseteq dom(B)$; $V \subseteq dom(C)$; b_i ($i \in \{1,...,m\}$) does not appear in X and Y.

The above 3 properties also hold for *fold* operator. That is, given a set of relations $DB2_i::R_j(A_1,...,A_n,b_1,...,b_m)$, $(i=1,...,k; j=1,\ldots,l)$, such that $b_1,...,b_m$ are values from $dom(B)$, and all entries appearing in columns $b_1,...,b_m$ are from $dom(C) \cup \{-\}$ (the no-value symbol), let $DB1_i::R_j(A_1,...,A_n,B,C)$ be the transformed relation by applying $fold(DB2_i::R_j,B,C)$ for each pair of (i, j) values. Then the above properties hold. □

Example 5.2: Suppose in Figure 1.1, we transform *DB4* to *DB3* by applying $fold(DB4::si,month,price)$ $(i=1,...,n)$ on each relation *si* in *DB4*. Suppose in *DB4*, the restricted FD $DB4::supplier_{\sigma=\{s1,...,sn\}}(product \rightarrow Jan,...,Dec)$ holds, i.e., $DB4::supplier_{\sigma=\{s1,...,sn\}}(product \rightarrow price(month_{\sigma=\{mi\}}))$ holds for each $mi \in \{Jan,...,Dec\}$. According to Theorem 5.3 property 1, we have $DB3::supplier_{\sigma=\{s1,...,sn\}}(product,month_{\sigma=\{mi\}} \rightarrow price)$ holds for each $mi \in \{Jan,...,Dec\}$. That is, the restricted FD $DB3::supplier_{\sigma=\{s1,...,sn\}}(product,month \rightarrow price)$ holds in *DB3* (Theorem 4.1). □

5.2 Inferring FDs in Schematic Discrepant Transformations

Using the properties on how restricted FDs change when applying each kind of restructuring operator, we can infer FDs for complex transformations which are implemented using a sequence of restructuring operators (see Section 3.2). In the following, we first give an algorithm to describe this process. Then we prove that our algorithm computes all the FDs in transformed relations which can be inferred from restricted FDs in original relations.

Algorithm 5.1 (Infer FDs in schematic discrepant transformation)

INPUT: (1) a set of relations, R_0, from one or a set of databases; (2) a set of restricted FDs, F_0, which hold in the relations of R_0; (3) a sequence of restructuring operators, $<T_1,...,T_n>$, which is applied on the relations of R_0.

OUTPUT: a set of FDs, F_n, which hold in the transformed relations, R_n, after applying the sequence of operators.

For $i:=0$ to $n-1$ **do**

Let R_{i+1} be the relations transformed from R_i after applying T_{i+1};

For each restricted FD in F_i **do** /*F_i is the set of restricted FDs holding in the relations of R_i*/

use the properties of T_i (i.e. Theorem 5.1, 5.2 or 5.3) to infer the corresponding restricted FD in F_{i+1}. □

In the rest this section, we'll prove that Algorithm 5.1 computes all the valid FDs holding in transformed relations which can be derived from the restricted FDs in original relations.

The point is some restricted FDs will be lost when applying restructuring operators. This is caused by the nature of schematic discrepant transformations.

Example 5.3: Suppose in Figure 1.1, we transform from *DB1* to *DB3*. Suppose in *DB1*, the FD *product* → *supplier* holds. The transformation is implemented by applying *split(DB1::Supply,supplier)*. Note the given FD can not be changed to any FD in the transformed relations, as the values of *supplier* become relation names in the transformed relations. □

As our goal is to infer FDs in transformed relations from restricted FDs in original relations, if the restricted FDs lost after applying some operator will never be changed to FDs in the target transformed relations, we can ignore them (the FD mentioned in Example 5.3 is

such an example). Recall Theorem 5.1-5.3 give the properties on how restricted FDs change when applying restructuring operators. In each property, we consider the changing of one class of restricted FDs that have a certain form. The question is: Do those properties consider all kinds of restricted FDs which may be changed to FDs finally? The answer is yes. In the following, we first prove this point for *simple transformations* (Definition 5.1) in Lemma 5.2, then for any transformations in Theorem 5.4.

Definition 5.1 (*Simple transformation*): We call a transformation T a *simple transformation* if in T, the operators *db_unite*(R_B,B) and *db_split*(R,B) are not used at the same time, neither are *unite*(R_B,B) and *split*(R,B), and neither are *fold*(R,B,C) and *unfold*(S,B,C), where R_B is a set of relation names, R and S are relation names, B and C are attribute names.

Generally, a simple transformation is always a non-redundant one, but a non-redundant transformation may not be a simple one. But as to lossless transformation, the definition of simple transformation is equivalent to that of non-redundant transformation (Lemma 3.1).

Lemma 5.1: A transformation T can not be simplified to a simple one iff T includes both operators *fold*(R,B,C) and *unfold*(S,B,C), and *fold*(R,B,C) does not satisfies the recoverability, where R and S are relation names, B and C are attribute names.

Lemma 5.2: Given a simple transformation T consisting of a sequence of restructuring operators, let t be any restructuring operator in T which is applied on one or a set of relations R_t. For any restricted FD f holding in R_t, if f can be changed to a FD in the target transformed relations of T, then f (or an equivalent of f) has a form considered in Theorem 5.1 (if t is *split* or *unite*), Theorem 5.2 (if t is *db_split* or *db_unite*), or Theorem 5.3 (if t is *unfold* or *fold*).

Proof of Lemma 5.2: For lack of space, we only prove the lemma when t is a *split* operator. For operator *split*(R_t,B), suppose the original relation R_t has the schema $R_t(A_1,...,A_n,B)$, and $b_1,...,b_m$ are values appearing in column B of R_t. We consider the following two forms of restricted FD f holding in R: (1) $X,B_{\sigma=S} \rightarrow Y$ or (2) $X \rightarrow B$, B does not appear in X and Y. That is, B appears either in the left hand side or right hand side of f. Note any restricted FD in R either has one of the 2 forms, or has equivalent having one of the 2 forms. For example, $X,B \rightarrow Y$ is equivalent to the restricted FDs $X,B_{\sigma=\{bi\}} \rightarrow Y$ for each b_i appearing in column B of R_t (Theorem 4.1), which conforms to form 1.

If f has form 1, Theorem 5.1 gives the property on the changing of f. If f has form (2), such restricted FDs can only be expressed in a relation in which $b_1,...,b_m$ are attribute values. On the other hand, as T is a simple transformation, *unite*(B) will not be used in T because *split*(R_t,B) is used, and therefore, $b_1,...,b_m$ have no chance to become attribute values again after applying *split*(R_t,B). So f will not be changed to a FD in the target transformed relations. □

Theorem 5.4: Given a transformation T consisting of a sequence of restructuring operators, let R and S be the original and transformed relations (or relation sets) of T resp. Algorithm 5.1 computes all the FDs in S which can be inferred from restricted FDs holding in R.

Proof of Theorem5.4: We consider 2 cases. If T has an equivalent simple transformation T', Lemma 5.2 ensures that after each step applying an operator of T', the restricted FDs lost will never become FDs in S. Finally, we get all the FDs in S which can be inferred from restricted FDs in R.

On the other hand, if *T* has not an equivalent simple transformation, then *T* must include two operators *fold(R1,B,C)* and *unfold(R2,B,C)*, and *fold(R1,B,C)* does not satisfy reconstructibility. Then the problem is: if a restricted FD *f* holding in *R1* is lost after applying *fold(R1,B,C)*, is it possible *f* will become some restricted FD later on? The answer is impossible even though *unfold(R2,B,C)* is applied later on. We omit the formal proof here (which need to discuss all possible forms *f* can have), but point out that as the *fold* operator does not satisfy reconstructibility, the relation transformed by the later *unfold* operator is totally different from the original one, which makes the restricted FD invalid. □

6 Apply FDs to Works in Schematic Discrepancy

In Section 5, we have proposed a method to infer FDs in schematic discrepant transformations. In this section, we study how to use those inferred FDs. Specifically, we identify three scenarios to apply FDs to works using schematic discrepant transformations, each of which consists a subsection.

6.1 Check Correctness of SchemaSQL Views

Recently, SchemaSQL views have been used to solve a broad range of problems [4, 13]. But we find SchemaSQL views may be problematic even for query purpose. In the following, we first give a real world example to see the problem of SchemaSQL views. Then we define the "correctness" for SchemaSQL views in general. At last, we give a method to automatically checking the correctness of a SchemaSQL view by inferring FDs holding in the view.

6.1.1 A Problematic Example

Example 6.1: In Figure 6.1, *Agency1* and *Agency2* are 2 databases of travel agencies. The relation *Tour* in each database records the tour information, in which each tuple records a tour number (used to identify tours in an agency) and the travel route of the tour. Note travel routes of different agencies may include different maximum number of visiting countries.

People may integrate the information from *Agency1* and *Agency2*, convenient for the customers to select a favorite travel route for example. The view definition *V* is proposed for this purpose. The expected view *Tour_view* (also shown in Figure 6.1) integrates tour information from the 2 agency databases. Note the view does not include the attribute *tour#,* since its values are numbers used to identify a tour in an agency, which is not so interesting to customers.

The benefit to define the integrated view *Tour_view* using SchemaSQL is as follows. Consider an open environment (e.g. on the web) in which agencies may join or exit from the integration freely. And the schemas of original relations may vary because of the insertion or deletion of some travel plans in a relation. Such a scenario requires the integrated view depends on the original database schemas dynamically. That is, though the number and schemas of original databases tend to change often, the view definition needn't be changed. SchemaSQL can meet this requirement since it uses variables to bind the schema labels. In this example, *V* includes a database-name variable *D* and attribute name variable *C* that are used for this purpose.

Agency1:
Tour

tour#	country1	country2
001	Singapore	China
002	China	
…	…	…

Agency2:
Tour

tour#	country1	country2	country3
001	Singapore	Malaysia	Thailand
002	China	Singapore	
…	…	…	…

V:
create view Tour_view (agency, C)
select D, T.C
from **à** *D, D::Tour T, D::Tour* **à** *C*
where C <> tour#

Tour_view

agency	country1	courtry2	courtry3
Agency1	Singapore	China	
Agency1	China		
Agency2	Singapore	Malaysia	Thailand
Agency2	China	Singapore	
…	…	…	…

Figure 6.1: Agency example with problematic view

Semantics of the view definition *V*: In the following, we interpret *V* following the SchemaSQL semantics [4], to see what problem will arise. The notations and terms used here are introduced in Section 2.1.

In *V*, the database-name variable *D* ranges over the names of databases to be integrated, i.e., {*Agency1, Agency2*}. And attribute name variable *C* ranges over attribute names of the *Tour* relation given a database indicated by *D*. The semantics of *V* is shown as follows.

(1) The valid instantiations of the variables appearing in *V* are shown in Figure 6.2. Each column name in Figure 6.2 represents a variable of *V*. The values of tuple variable *T*, t_i, represent the *i*-th tuple in corresponding relation.
(2) Determine the schema of output view. As there is a variable *C* in the "create view" clause, *V* creates a view whose schema depends on the instantiation values of *C*. As *C* ranges over {*country1, country2, country3*}, the output view has the schema as Figure 6.3.
(3) Allocate the instantiations of last step to the output view schema. Each tuple in instantiation table becomes one tuple in the allocated table. The result is shown in Figure 6.3.
(4) Merge the tuples in the allocated table (Note this step is problematic). According to the merge method given in [4], two tuples are merge-able if for common attribute, either the values of the 2 tuples are the same, or at least one value is null. E.g., in Figure 6.3, the 2nd tuple can be merged with the 1st or 3rd tuple. Consequently, one of the possible results is as Figure 6.4.

D	T	C	T.C
Agency1	t1	country1	Singapore
Agency1	t1	country2	China
Agency1	t2	country1	China
…	…	…	…

Figure 6.2: Valid instantiations

agency	country1	country2	country3
Agency1	Singapore	null	null
Agency1	null	China	null
Agency1	China	null	null
…	…	…	…

Figure 6.3: Allocated table

agency	country1	country2	country3
Agency1	Singapore	null	null
Agency1	China	China	null
…	…	…	…

Figure 6.4: Problematic view relation

In the result view (Figure 6.4), we found that the view contains some tuples (e.g. the 2 tuples shown in the view) that do not exist at all. This will cause wrong results when querying through the view. □

6.1.2 Correctness of SchemaSQL Views

We extend and use information capacity [14] to define the correctness of a SchemaSQL view. We consider views for query purpose, not for update purpose.

Definition 6.1 ***(Correctness of SchemaSQL view)*:** Given a SchemaSQL view definition V, let $S1$ be the set of relations (or relation sets) on which V is defined, $S2$ be the set of relations (or relation sets) generated by V. If the view definition $V: S1 \rightarrow S2$ is a many to one mapping, we call V is correct.

The intuition of this definition is as follows. Though a SQL view defines a mapping from instances of original relations to instances of the view relation, a SchemaSQL view defines a mapping from original relations to view relations which include schemas and instances both. And a SchemaSQL query queries data values and schema information both. Formally, for a query Q on a view $S \in$ S2, the following holds:

$$Q(S) = Q(V(R)) = Q \circ V(R)$$

That is, the query Q on S is mapped to the unique query $Q \circ V$ on original relation $R \in$ S1 if V is a many to one mapping.

The following theorem gives a condition to check the correctness of a SchemaSQL view.

Theorem 6.1: A schemaSQL view is correct if it satisfies the following condition: if the definition of the view schema has a form "$DB::R(A_1,...,A_n,C)$", such that $A_1,...A_n$ are constants each of which represent a fixed attribute, and C is a variable representing a set of attributes Z, the following FD holds: $A_1,...A_n \rightarrow Z$.

Proof of Theorem 6.1: In the following, we prove the theorem briefly. The semantics of a SchemaSQL view V can be divided to 4 steps: finding all valid instantiations of variables in V, determining the view schema, allocation of valid instantiations, and merging the tuples in allocated table. The first 3 steps can be done in a deterministic way, each of which represents a many to one mapping. For the last step of merging, Theorem 6.1 requires the FD $A_1,...A_n \rightarrow Z$ holds in the view. That is, given the values of $A_1,...,A_n$, there's only one value for each attribute in Z. So the set of tuples in the allocated table which have the same values on $A_1,...,A_n$ are merged to one tuple in the view relation. And this is deterministic. So the view relation is unique. □

Note Theorem 6.1 implies that if the definition of a view schema hasn't a form "$DB::R(A_1,...,A_n,C)$", the view is always correct without any additional conditions.

6.1.3 Automatically Checking Correctness of SchemaSQL Views

An interesting problem is to develop a method to check the correctness of a SchemaSQL view automatically. According to Theorem 6.1, in order to judge the correctness of a SchemaSQl view, we need to infer FDs in the view. To achieve this, we propose to extend Algorithm 5.1 by incorporating relational algebra in schematic discrepant transformations.

In [5], Lakshmanan et al proposed to implement SchemaSQL queries/views in a single database using an extended relation algebra which consists of classical relational algebra and the 4 restructuring operators – *unfold*, *fold*, *split* and *unite*. That is, firstly transform relations involved in the query to a relation using *fold* and *unite* operators, such that schema labels in original relations become attribute values in the transformed relation; then apply a Cartesian product together with necessary selection and projection; finally, apply *split* and *unfold* operators to conform to the output view schema. This method can be extended to implement SchemaSQL queries/views in a multidatabase environment by adding *db_unite* and *db_split.*

We propose to infer FDs for a SchemaSQL view through inferring restricted FDs for each operator (classical relational algebra and restructuring operators) applied in the implementation of the view. Theorem 5.1 - 5.3 have given properties on how restricted FDs change when applying those restructuring operators. In the following, we study how restricted FDs change in classical relational algebra, i.e. Cartesian product, selection and projection.

Cartesian product makes no difference to restricted FDs. That is, the set of restricted FDs holding in the transformed relation is the union of the sets of the restricted FDs in original relations involved in the Cartesian product.

For projection, given a restricted FD *f* holding in the original relation, if all the attributes appearing in *f* are included in the projection list, *f* remains unchanged in the relation after projection; otherwise *f* is lost.

Selection makes no difference to regular FDs, but may change restricted FDs as follows. Given a restricted FD $X,Z_1{}_{\sigma=V1},\ldots,Z_n{}_{\sigma=Vn} \rightarrow Y$ holding in the original relation, if the Boolean combination ($Z_1 \in$ V1 and…and $Z_n \in$ Vn) satisfies the selection condition (i.e. the selection condition implies the Boolean combination), then the given restricted FD becomes $X \rightarrow Y$ which holds in the relation after selection. The following example explains this.

Example 6.2: In Figure 1.1, suppose in relation *DB1::Supply*, the restricted FD $prod,month,supplier_{\sigma=\{s1\}} \rightarrow price$ holds. Then in the relation $\sigma_{supplier=s1}DB1::Supply$, the FD $prod,month \rightarrow price$ holds.□

Another problem is: are the properties given in Theorem 5.1-5.3 still sufficient to infer restricted FDs in a transformation which consists of not only restructuring operators, but also Cartesian product, selection and projection? The answer is yes. The proof of this point is similar to the proof of Theorem 5.4. Recall in the proof of Theorem 5.4, we first simplify a schematic discrepant transformation to a simple transformation, then prove the theorem based on the simple transformation (Lemma 5.2). The situation is similar here, but to simplify a transformation which includes both restructuring operators and relational algebra, we need to commute restructuring operators with Cartesian product, selection and projection resp [20].

Example 6.3: In the following, we explain the process of checking the correctness of the SchemaSQL view in the travel agency example (Example 6.1), which is actually a process of inferring FDs holding in the view relation. Suppose in each *Tour* relation of Figure 6.1, *tour#* is a key.

Using restructuring operators and relational algebra, we can implement the view in Figure 6.1 as follows.

(1) Transform the *Tour* relations from different agency databases to one relation *Tour1(agency,tour#,country#,country)*. This is done by first applying

fold(Ai::Tour,country#,country) for each *Ai*∈ {*Agency1,Agency2*}, and get the transformed relations *R*. Then apply *db_unite(R ,agency)*. As a result, in the transformed relation *Tour1*, database names, *Agency1* and *Agency2,* become values of the attribute *agency*; attribute names, *country1, country2* and *country3*, become values of *country#*; and the country names, *Singapore, China,...,* become values of *country*.

(2) Project out *tour#* in *Tour1* which is not used in the view definition, and get a relation *Tour2(agency,country#,country)*.

(3) Applying *unfold(Tour2,country#,country)*, we get the target view relation.

Then we infer FDs holding in the view relation. As *tour#* is a key in the *Tour* relation of each agency database, according to the properties of *fold* (Theorem 5.3 property 1) and *db_unite* (Theorem 5.2), in the transformed relation *Tour1*, the FD *agency,tour#,country#* → *country* hold. Then in *Tour2*, as the attribute *tour#* is projected out, the FD is lost. Consequently, no FDs hold in the target view relation. So the correctness condition of Theorem 6.1 is not satisfied, which means the view is possibly wrong.

On the contrary, if the view definition in Figure 6.1 includes attribute *tour#* in the view schema, then the implementation of the view can be simplified to *db_unite({Agency1::Tour*, *Agency2::Tour}*, *Agency)*.[1] According to the property of *db_unite* and Theorem 4.1, the FD *agency,tour#* → *country1*,*country2,country3* holds in the view relation. According to Theorem 6.1, the view is correct. □

6.2 Use FDs to Normalize a Transformed Relation

As mentioned at the beginning of the paper, schema integration is the activity to integrate component database schemas to a unified, non-redundant one. If schematic discrepancy exists among component databases, it should be reconciled by schematic discrepant transformations. People may suppose the component database schemas are in good normal form (BCNF or 3NF) if they are well designed. But how about the schemas after schematic discrepant transformation? Actually, we find in the presence of schematic discrepancy among component database schemas, the transformed and integrated schemas may be not in 2nd normal form because schematic discrepant transformations may introduce redundancy. Note redundancy not only wastes space, but also causes update, insertion and deletion anomalies if the integrated database is materialized.

People have developed decomposition and synthesizing methods to convert relations to BCNF or 3NF given FDs holding in those relations ([19]). And using Algorithm 5.1, we can obtain FDs in the integrated database from restricted FDs in component databases, which are used to normalize the relations in the integrated database then. The following examples illustrate this.

Example 6.4: We want to integrate two relations from two bookstore databases, i.e., *BS1::book(isbn,title,first_author,price)* and *BS2::book(isbn,title,first_author,price).* Suppose the database names are from the domain of attribute *store*. In each relation, *isbn* is the key. Suppose the following restricted FDs hold: $store_{\sigma=\{BS1,BS2\}}$*::book*(*isbn* → *title,first_author*) and $store_{\sigma=\{BSi\}}$*::book*(*isbn* → *price*) for each *BSi*∈ {*BS1,BS2*}. Note a book's title and first author are independent of bookstores, but price is dependent on bookstores.

Though the two relations have the same schemas, we can not integrate them to a relation with the same schema, as value inconsistency will occur on the *price* attribute. A better solution is to integrate the two relations to one with the schema *BS::book(store,isbn,title,first_author,price)* by applying *db_unite({BS1::book,BS2::book}*, *store)*. We can infer FDs holding in *BS::book* from those given restricted FDs in original

[1] As the schemas of *Agency1::Tour* and *Agency2::Tour* are not exactly the same, the implementation of *db_unite* may be a little tricky.

relations according to the properties of *db_unite*, i.e., *isbn* → *title,first_author* and *store,isbn* → *price*. Then we found that the integrated relation is not in 2nd normal form. Consequently, there will be many redundancies if *BS::book* is a materialized relation. E.g. for any book sold in both stores, the *title, first_author* of the book is repeated twice. Using the normalization theory, the integrated relation can be normalized by decomposing it into two relations: *BS::book_price(store,isbn,price)* and *BS::book_info(isbn,title,first_author)*. □

6.3 Formulate Global Constraints for Integrated Schema

Generally, global constraints holding in the integrated database can be used to verify the integrity of data from different sources, to optimize queries on the integrated schema, or to validate update transactions at the global level [9]. Formulating global constraints for integrated database has been studied in [7, 9]. But they did not consider schematic discrepancy in schema integration. Our method (Algorithm 5.1) provides a way to formulate FDs for integrated database in the presence of schematic discrepancy. The following example shows the use of global FD constraints to verify the integrity of data from different databases.

Example 6.5: Let's see the integrated and normalized relation *BS::book_info(isbn,title,first_author)* in Example 6.4. As mentioned, we have formulated the FD constraint holding in the relation, i.e., *isbn* → *title,first_author* which should be satisfied by instances of the integrated relation. This constraint can be used to verify the data integrity when integrating tuples from the two source databases *BS1* and *BS2*. For example, suppose we find two tuples from the two bookstore databases resp. have a same *isbn* value but with different *title* (or *first_author*) value. This violates the FD constraint holding in *BS::book_info*, which means error occurs and should be solved by the integrator. □

7 Conclusion

In this paper, we focused on inferring and applying FD constraints in schematic discrepant transformations. The results of this paper can be applied to a broad range of works using schematic discrepant transformations. We first studied transformations based on restructuring operators (i.e., *fold, unfold, unite, split*, *db_unite* and *db_split*). We gave the reconstructibility and commutativity of restructuring operators which can be used to simplify a transformation.

Then we studied how to infer FDs holding in transformed relations from ICs holding in original relations. We proposed restricted FDs to represent ICs in original relations. And gave properties on how restricted FDs change when applying each type of restructuring operator. Using these properties, we gave an algorithm to infer FDs in a transformation which is implemented using a sequence of restructuring operators. We proved that our algorithm can compute all the FDs in the transformed relations which can be inferred from restricted FDs in original relations.

At last, we identified 3 scenarios to apply FDs to works using schematic discrepant transformations: (1) For SchemaSQL views, we defined the correctness of SchemaSQL views, and gave a theorem in which FDs are used to check the correctness of a SchemaSQL view. We also proposed a method to check the correctness of views automatically. (2) Use FDs to find redundancy in a transformed relation, and normalize it. This is useful if the transformed relation is materialized, as schematic discrepant transformations may introduce redundancy in the transformed relation. Note that redundancy will lead to update, insertion and deletion anomalies in relations, and therefore should be removed. (3) Use FD constraints to verify the integrity of data from different sources.

Reference

[1] Alon Y.Levy, Answering queries using views: a survey, technical report, CS Dept., Washington Univ., 2000.

[2] C.Batini, M.Lenzerini, S.B.Navathe, A comparative analysis of methodologies for schema integration, CN computing surveys, 1986.

[3] Joseph Albert. Theoretical Foundations of Schema Restructuring in Heterogeneous Multidatabase Systems. CIKM 2000.

[4] Lakshmanan, L. V. S., Sadri, F., Subramanian, S. N. SchemaSQL—an extension to SQL for multidatabase interoperability, TODS 2001

[5] Lakshmanan, L. V. S., Sadri, F., Subramanian, S. N. On efficiently implementing schemaSQL on SQL database system, VLDB conference 1999

[6] Mark W.W.Vermeer, P.M.G.Apers. The role of integrity constraints in database interoperation. VLDB'96

[7] Mong Li Lee, Tok Wang Ling, Resolving constraint conflicts in the integration of ER schemas, ER 97

[8] Mong-Li Lee, Tok Wang Ling: Resolving Structural Conflicts in the Integration of Entity Relationship Schemas. OOER1995.

[9] M.P.Reddy, B.E.Prasad, Amar Gupta, Formulating global integrity constraints during derivation of global schema, Data & Knowledge Engineering, 1995

[10] O. Tsatalos et al, The versatile tool for physical data independence. VLDB journal, 1996

[11] Rakesh Agrawal, Amit Somani, Yirong Xu, Storage and querying of e-commerce data, VLDB conf, 2001.

[12] Ravi Krishnamurthy, Witold Litwen, William Kent, Language features for interoperability of databases with schematic discrepancies, SIGMOD 1991

[13] R J Miller, Using schematically heterogeneous structures, SIGMOD 1998.

[14] R.J.Miller, Y.E.Ioannidis, r.Ramakrishnan, The use of information capacity in schema integration and translation, VLDB conf, 1993.

[15] Sheth, A.P. and Gala, S.K., Federated database systems for managing distributed, heterogenous, and autonomous databases, ACM Computing Surveys, 1990.

[16] T.W.Ling, M.L.Lee. Issues in an Entity-relationship based federated database system. CODAS, 1996.

[17] Wai Lup Low, Mong Li Lee, Tok Wang Ling, A knowledge-based approach for duplicate elimination in data cleaning. Information Systems, 2001.

[18] What's next for the database: www.intelligententerprise.com/020509/508feat1_2.shtml

[19] Raghu Ramakrishnan, Johannes Gehrke, Database Management Systems, 2nd Ed, pp438-444, McGraw-Hill, 1999.

[20] Keir B. Davis, Fereidoon Sadri, Optimization of SchemaSQL Queries, IDEAS 2001.