Discover Dependencies from Data - A Review

Jixue Liu[1] Jiuyong Li[1] Chengfei Liu[2] Yongfeng Chen[3]

[1] School of Computer and Info. Sci., University of South Australia
{jixue.liu, jiuyong.li}@unisa.edu.au
[2] Faculty of ICT, Swinburne University of Technology
cliu@swin.edu.au
[3] School of Management, Xian University of Architecture and Tech.
chenyf@xauat.edu.cn

November 8, 2010

Abstract

Functional and inclusion dependency discovery is important to knowledge discovery, database semantics analysis, database design, and data quality assessment. Motivated by the importance of dependency discovery, this paper reviews the methods for functional dependency, conditional functional dependency, approximate functional dependency and inclusion dependency discovery in relational databases and a method for discovering XML functional dependencies.

Keyword: integrity constraint, functional dependencies, inclusion dependencies, conditional functional dependencies, XML, knowledge discovery, data quality

1 Introduction

Dependencies play very important roles in database design, data quality management and knowledge representation. Their uses in database design and data quality management are presented in most database textbooks. Dependencies in this case are extracted from the application requirements and are used in the database normalization and are implemented in the designed database to warrant data quality. In contrast, dependencies in knowledge discovery are extracted from the existing data of the database.

The extraction process is called dependency discovery which aims to find all dependencies satisfied by existing data.

Dependency discovery has attracted a lot of research interests from the communities of database design, machine learning and knowledge discovery since early 1980s [36, 18, 25, 5, 16, 21, 27, 40, 41, 3]. Two typical types of dependencies are often involved in the discovery, functional dependencies (FDs) and inclusion dependencies (INDs). FDs represent value consistencies between two sets of attributes while INDs represent value reference relationships between two sets of attributes. In recent years, the discovery of conditional functional dependencies (CFDs) has also seen some work [14, 8].

The aim of dependency discovery is to find important dependencies holding on the data of the database. These discovered dependencies represent domain knowledge and can be used to verify database design and assess data quality.

For example, by checking the data of a medical database which has two columns Disease and Symptom, if pneumonia is a value of Disease and fever is a value of Symptom, and if every pneumonia patient has a fever, then fever is said associated with pneumonia. If such association happens to every pair of Symptom and Disease values, then Disease *functionally* determines Symptom and this is a functional dependency. If this were new knowledge, it would help diagnose the disease more efficiently. In modern health science, finding such associations and dependencies between DNA segments and disease becomes very important to the development of medical science.

Besides knowledge discovery, the dependencies discovered from existing data can be used to verify if the dependencies defined on the database are correct and complete [4, 22] and to check the semantics of data of an existing database [30].

A further use of discovered dependencies is to assess the quality of data. The fundamental role of dependency implementation in a database is to warrant the data quality of the database. Thus by analyzing the discovered dependencies and the missed dependencies that should hold among attributes of data, errors may be identified and inconsistencies among attributes may be located. As a result, the data quality is assessed.

In recent years, the demand for improved data quality in databases has been increasing and a lot of research effort in this area has been given to dependency discovery [41, 3, 40, 14]. However as the research on dependency discovery started at the very beginning of 1980s, some methods like the partition-based methods and the negative cover-based methods have evolved

in many versions in the literature [16, 19]. This imposes the need for a review of all the methods proposed in the past and motivates the work of this paper.

In this paper, we review the methods developed in the literature for discovering FDs, approximate FDs (AFDs), conditional FDs (CFDs), and inclusion dependencies (INDs) in relational and XML databases, and other topics relating to the discovery of the dependencies such as the discovery of multivalued dependencies [12] and roll-up dependencies [7].

The paper is organized as the following, which is also shown in Figure 1. Section 2 reviews the methods used in FD discovery. This section also includes a short description of multivalued dependency, roll-up dependency and key discovery. Section 3 is a review of AFD discovery. Section 4 reviews the discovery of CFDs. In Section 5, we review the definition of INDs and the methods used in discovering INDs. In Section 6, a definition of XML functional dependencies (XFDs) is introduced and a method of discovering XFDs is reviewed. The section 7 concludes the paper.

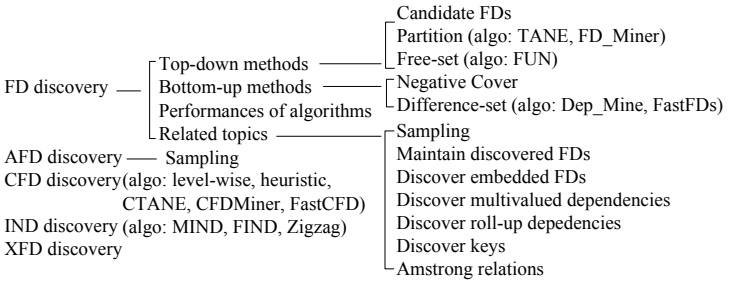

Figure 1: Outline of methods reviewed

2 Discovery of functional dependencies

In this section, we give the definition of FDs and review the methods used in FD discovery.

2.1 Definitions

Let $R = \{A_1, ..., A_m\}$ be a database table schema and r be a set of tuples from $dom(A_1) \times \cdots \times dom(A_m)$ where $dom(A)$ represents the domain of attribute A. The projection of a tuple t of r to a subset $X \subseteq R$ is denoted by $t[X]$. Similarly $r[X]$ represents the projection of r to X. The number of tuples in the projection, called the *cardinality*, is denoted by $|r[X]|$. For

simplicity, we often omit braces when the context is clear. If X and Y are sets and A is an attribute, XA means $X \cup \{A\}$ and XY means $X \cup Y$.

Definition 2.1. A *functional dependency* (FD) is a statement $X \rightarrow Y$ requiring that X functionally determines Y where $X, Y \subseteq R$. The dependency is *satisfied* by a database instance r if for any two tuples $t_1, t_2 \in r$, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. X is called the left-hand side (lhs) or the determinant and Y is called the right-hand side (rhs) or the dependent.

FDs defined in this way are called *exact* FDs because every pair of tuples in r has to satisfy the condition. The term exact FD will be used to distinguish other types of FD definitions shown later.

If relation r satisfies the dependency f, f is said supported by r, or valid or holding on r. Otherwise, f is said violated by r or invalid on r.

Table 1: An example

	ID (I)	**Name (N)**	**Bdate (B)**	**Wage (W)**	**Supvsr (S)**
t_1	e1	Bob	d1	1	e5
t_2	e2	John	d1	1	e1
t_3	e3	John	d1	2	e1
t_4	e4	Peter	d3	2	e2

Example 2.1. Consider Table 1 and FDs $N \rightarrow B$ and $B \rightarrow S$. The FD $N \rightarrow B$, meaning that $Name$ functionally determines birthdate, is satisfied by the table as we can not find a pair of tuples t_i and t_j such that $t_i[N] = t_j[N] \wedge t_i[B] \neq t_j[B]$. In contrast, the FD $B \rightarrow S$ is violated because we can find t_1 and t_2 such that $t_1[B] = t_2[B] = d1$ but $t_1[S] = e5 \neq t_2[S] = e1$.

A FD is *minimal* if removing an attribute from its lhs makes it invalid.

Given a set Σ of FDs and a FD f, f is implied by Σ, denoted by $\Sigma \models f$, if any relation r satisfying Σ also satisfies f. Armstrong gives the following sound and complete axiom of implication for FDs.

(i) If $Y \subseteq X$, then $X \rightarrow Y$.
(ii) If $X \rightarrow Z$, then $XY \rightarrow Z$.
(iii) If $X \rightarrow Z$ and $Z \rightarrow W$, then $X \rightarrow W$.

Given a set Σ of FDs on R and a set X of attributes in R, the closure of X, denoted by X^+, is the set of all attributes that are directly or indirectly dependent on X (including X itself) following Amstrong rules above. For

example, if $U \rightarrow Z$ and $Z \rightarrow W$ and $X = UY$, then $X^+ = UYZW$. For every attribute $A \in X^+$, $X \rightarrow A$ holds.

Given a set Σ of all FDs defined on the database schema R, a *cover* σ of Σ is a subset of Σ such that any FD in Σ is either in σ or is implied by σ, denoted by $\sigma \models \Sigma$. A cover σ is *minimal* if removing any FD from σ causes some FDs in Σ not implied, i.e., causes $\sigma \not\models \Sigma$.

We note that minimal cover and minimal FD are two different concepts. The former means that the set of FDs is minimal while the latter means that the lhs of an individual FD is minimal.

In the context of this paper, we are interested in only the FDs with one attribute on the rhs. We are so because it can be proved, from Armstrong rules, that any relation satisfying $X \rightarrow A_1A_2$ also satisfies $X \rightarrow A_1$ and $X \rightarrow A_2$ or vice versa. We consider only the FDs with $X \cap Y = \phi$, i.e., we are not interested in trivial FDs.

2.2 Methods of FD discovery

In this subsection, we review the methods proposed in the literature on FD discovery. Based on [35], these methods are either top-down or bottom-up. The top-down methods start with generating candidate FDs level-by-level, from short lhs to long lhs, and then check the satisfaction of the candidate FDs for satisfaction against the relation or its partitions. The bottom-up methods, on the other hand, start with comparing tuples to get agree-sets or difference-sets, then generate candidate FDs and check them against the agree-sets or difference-sets for satisfaction. At the end of the subsection, we show some performance comparison results.

2.2.1 Top-down methods

Top-down methods start with candidate FD generation. These methods generate candidate FDs following an attribute lattice, test their satisfaction and then use the satisfied FDs to prune candidate FDs at lower levels of the lattice to reduce the search space. In the subsection, we first present candidate FD generation and pruning. We then present two specific methods: the partition method (algorithms include TANE [16] and FD_Mine [40]) and the free-set method which uses the cardinality of projected relations to test satisfaction (algorithm: FUN [33]).

Candidate FDs and pruning Candidate FDs (canFDs) are FD expressions that are syntactically possible in a relation schema. Their satisfaction

against the relation instance has not been tested.

Given schema $R = \{A_1, \cdots, A_m\}$, canFDs are calculated using all possible attribute combinations of R as lhs. As we are only interested in minimal FDs with single attribute on the rhs, the number of attributes for the lhs of a canFD contains at most $(m-1)$ attributes. For example, the canFDs with zero attributes in their lhs are $\phi \rightarrow A_1$, $\cdots$, $\phi \rightarrow A_m$. The canFDs with one attribute in their lhs can be $A_1 \rightarrow A_2$, $A_1 \rightarrow A_3$, $A_m \rightarrow A_{m-1}$ etc.. The canFDs with two attributes in lhs can be $A_1A_2 \rightarrow A_3$, $A_1A_2 \rightarrow A_4$, $A_{m-1}A_m \rightarrow A_1$ etc.. The canFDs with $(m-1)$ attributes are $A_1 \cdots A_{m-1} \rightarrow A_m$, $\cdots$, $A_2 \cdots A_m \rightarrow A_1$. The lhs can be shown graphically in an attribute lattice [16, 33, 19, 40].

An attribute lattice is a directed graph with the root node (said at Level-0) contains no attribute and represented by ϕ. The children of the root node are Level-1 nodes and each Level-1 node contains one attribute. Totally Level-1 has $\binom{m}{1}$=m nodes [1]. Each node at Level-2 contains a combination of two attributes and thus there are $\binom{m}{2}$ nodes at Level-2. Other levels follow the same rule. Level-m is the final level and contains all attributes. We use n_{ij} to mean the j-th node at Level-i. The same symbol is also used to mean the attributes on the node. A directed edge is drawn between the j-th node at Level-i and the k-th node at Level-$(i+1)$ if $n_{ij} \subset n_{(i+1)k}$. In this way, each edge represents the canFD $n_{ij} \rightarrow (n_{(i+1)k} - n_{ij})$. Figure 2 shows a lattice of $R = \{A, B, C, D\}$ where the edge between the first node from left at Level-2 and the first node from left at Level-3 represent the canFD $AB \rightarrow C$. Node ABC is called the *end* of the canFD and Node AB is called the *parent node* of ABC, and the edge "$AB - ABC$" is called the *parent edge* of ABC.

By Pascal Triangle in mathematics, the total number of nodes in the lattice is

$$\binom{m}{0} + \binom{m}{1} + \binom{m}{2} + \binom{m}{3} + \cdots + \binom{m}{m-1} = 2^m$$

Because $\binom{m}{h} = \binom{m}{m-h}$, the lattice is symmetric to the middle level if m is even or the middle two levels if m is odd. For example in Figure

[1] $\binom{m}{h}$ means the number of combinations of h attributes out of m attributes. When $h = 0$, it equals to 1.

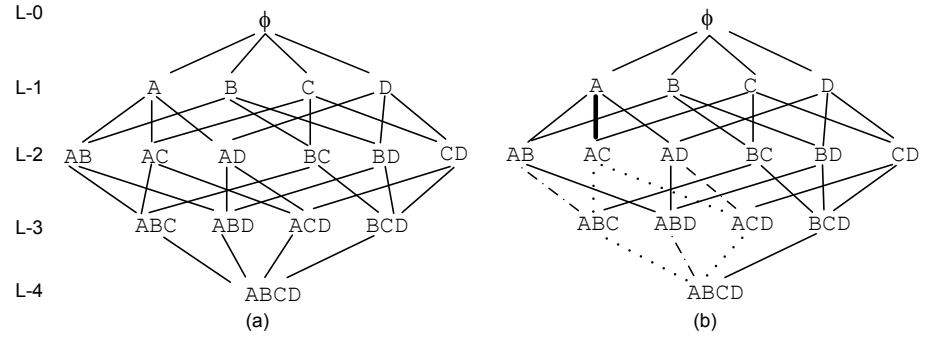

Figure 2: Attribute lattice

2, every Level-1 node has 3 edges to 3 Level-2 nodes and every Level-3 node also has 3 edges to 3 Level-2 nodes. The total number of canFDs (edges) is

$$\left(\begin{array}{c} m \\ 0 \end{array}\right)\times m+\left(\begin{array}{c} m \\ 1 \end{array}\right)\times(m-1)+\left(\begin{array}{c} m \\ 2 \end{array}\right)\times(m-2)+\cdots+\left(\begin{array}{c} m \\ m-1 \end{array}\right)\times 1 \leq \frac{m}{2}2^m$$

where the equality holds when m is odd.

Let $|r|$ represent the number of tuples in the database instance r. By average, each FD involves $\frac{m}{2}$ attributes. Then the complexity of using nested loop to test all canFDs shown in the lattice against r is $O(|r|^2(\frac{m}{2})(\frac{m}{2})2^m)$. This complexity is also the worst complexity of all proposed methods in the literature.

Because the number of canFDs is exponential to the number of attributes, pruning implied FDs from the lattice becomes important to many of the proposed methods. FD pruning is to remove the canFDs (edges) in the lattice implied by the discovered FDs so that we do not check them against r. For example in Figure 2, if $A \rightarrow C$ is supported, then $AB \rightarrow C$ is implied based on Armstrong rules and therefore does not need checking.

The following essential **pruning rules**, which can be proved following Amstrong rules in Section 2.1, are used in the literature [16, 34, 40]. Let Σ be a set of found FDs, X, Y be two subsets of R and A, B be two attributes in R, $X \cap Y = \phi$, $A, B \notin X$ and $A, B \notin Y$ and $A \neq B$.

(1) If $X \rightarrow A \in \Sigma$, $XZ \rightarrow A$ is implied and no checking is needed;

(2) If $X \rightarrow A \in \Sigma$ and if $XAY \rightarrow B$ is a candidate FD, instead of checking $XAY \rightarrow B$, we should check $XY \rightarrow B$. With top-down level-wise checking, $XY \rightarrow B$ must have been checked at the previous level, and thus no checking is needed.

(3) If X is a key, any node containing X is removed.

The pruning rules are of two categories: Rules (1) and (2) prune edges while Rule (3) prunes nodes.

Besides these essential rules, a few other rules are also proposed. FD_Mine [40] uses symmetric FDs (e.g., $X \rightarrow Y$ and $Y \rightarrow X$ are symmetric) to remove attribute set (e.g., Y) from R to reduce the size of the lattice. FUN [34] prunes non-free-sets to reduce the size of the lattice with the closure calculation to cover the lost canFDs. FastFDs [39], a bottom-up method though, uses difference-sets to prune lattice nodes to compute satisfied FDs.

The above essential rules can be unified to the following level-wise algorithm which computes a cover of all FDs holding on r and which is a variation of the algorithms proposed in [16, 34, 40].

The algorithm checks FD satisfaction by following the attribute lattice from left to right and from top to bottom. The reason for the top-down traversal is that the FDs at top levels have less attributes on the lhs. If a FD like $A \rightarrow C$ is discovered, other candidate FDs like $AX \rightarrow C$ can be pruned (Rule (1) above). This is more efficient than the bottom-up traversal of the lattice where if $AX \rightarrow C$ is supported, we still have to check FDs with less attributes such as $A \rightarrow C$ and $X \rightarrow C$.

In Algorithm 1, Line 7 implements pruning rule (3). Line 11 implements pruning rules (1) and (2). In Line 12, $supp(f, r)$ checks canFD f against relation r, if f is supported, the function returns $true$. Otherwise, it returns $false$.

Lemma 2.1. *Algorithm 1 gives minimal FDs.*

Proof. As the algorithm traverses the lattice level-wise, we need to prove that for any $X \rightarrow A$ in Σ (discovered FDs), if no other FDs like $f = XY \rightarrow A$ are added to Σ, the FDs in Σ are minimal. Actually the condition "$\exists\, X \rightarrow A \in \Sigma$ s.t. $X \subset q$ and $XA \subset c$" in Line 11 detects this situation. If the candidate FD is $f = XY \rightarrow A$, then $q = XY$ and $c = XYA$. This makes the condition true, f is pruned and will not add f to Σ. □

We now use an example to show FD pruning with the Algorithm. In Figure 2(b), we assume that $A \rightarrow C$ is in Σ. Then FDs represented by the dotted lines will be pruned by Rule (1) and the FDs represented by the dashed lines will be pruned by Rule (2).

From the example, we see that when FDs with a single attribute on the rhs are discovered, they become very important to pruning large number of edges and to reducing the complexity of the calculation. Such FDs often exist in databases. For example the email address of people, the tax file number of people, the identification number of students, bar code of products, registration number of vehicle and so on.

Algorithm 1: Candidate FD and pruning

Input: A relation schema R and its instance r
Output: A set Σ of FDs discovered from r

```
1  begin
2      let Σ store found FDs and K store found keys;
3      for each level l = 2 to |R| − 1 do
4          for each node p at Level-(l − 1) do
5              generate all child nodes C at Level-l of p;
6              for each node c ∈ C do
7                  if c contains a key then delete c, next loop;
8                  if c is a key, add c to K then ;
9                  for each parent node q at Level-(l − 1) of c do
10                     let f be the FD q → (c − q);
11                     if ∃ X → A ∈ Σ s.t. X ⊂ q and XA ⊂ c then f is
                       implied by X → A and is pruned.;
12                     else if supp(f, r) == true then f is supported, add f
                       to Σ;
13                     else f is not supported by r and is ignored;
```

The worst case performance of the algorithm is exponential as analyzed before. This happens when the lhs of all FDs contain almost all attributes of the relation. In this case, all edges of the lattice needs to be tested against the database.

Lemma 2.2. *Algorithm 1 gives a cover of all FDs supported by r.*

Proof. All candidate FDs are in the lattice. The pruning process (Line 11) deletes implied FDs. Line 13 ignores unsupported FDs. Line 12 adds all other FDs to Σ. No other parts of the algorithm delete FDs. Thus the FDs in Σ form a cover of all supported FDs. □

We note that Algorithm 1 gives a cover, but this cover may not be minimal. The reason is that the calculation does not use the transitivity rule of FD implication and can add implied FDs to Σ. For example, it can add in order $A_1 \to A_2$, $A_1 \to A_3$ and $A_2 \to A_3$ to Σ and here $A_1 \to A_3$ is implied by the other two.

In summary, we presented an algorithm that can prune candidate FDs and nodes of the lattice after a set of FDs and a set of keys are discovered. The next main problem is how a candidate FD can be tested efficiently against the relation r, i.e., how $supp(f, r)$ in Algorithm 1 can be calculated efficiently. The methods will be presented next.

The partition method

The partition semantics of relations is proposed in [10]. The semantics is used in [16, 19, 40] to check the satisfaction of candidate FDs. We call the methods using the partition semantics the *partition methods*. The algorithms implementing the method include TANE [16] and FD_Mine [40]. Both TANE and FD_Mine use the essential pruning rules on Page 7, but FD_Mine uses symmetric FDs too.

We note that, although we use the term 'partition method' to mean a category of algorithms, the concept 'partition' is also used in the algorithms of other categories, such as the free-set method and the bottom-up methods, to optimize performance.

Given a relation r on R and a set X of attributes in R, the *partition* of r by X, denoted by $\mathcal{P}_X$, is a set of non-empty disjoint subsets and each subset contains the identifiers of all tuples in r having the same X value. Each subset of a partition is called an *equivalent class*. A *stripped partition* is a partition where subsets containing only one tuple identifier are removed.

Example 2.2. Consider the relation in Table 1. Then $\mathcal{P}_I = \{\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}\}$, $\mathcal{P}_{NB} = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$, $\mathcal{P}_N = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$ and $\mathcal{P}_W = \{\{t_1, t_2\}, \{t_3, t_4\}\}$. $\{t_1, t_2\}$ is an equivalent class of $\mathcal{P}_W$.

Given two partitions $\mathcal{P}_X$ and $\mathcal{P}_Y$, $\mathcal{P}_X$ is a *refinement* of $\mathcal{P}_Y$, denoted by $\mathcal{P}_X \preceq \mathcal{P}_Y$, if for every subset $u \in \mathcal{P}_X$, there exists a subset $v \in \mathcal{P}_Y$ such that $u \subseteq v$. $\mathcal{P}_X = \mathcal{P}_Y$ iff $\mathcal{P}_X \preceq \mathcal{P}_Y$ and $\mathcal{P}_Y \preceq \mathcal{P}_X$.

Example 2.3. In Example 2.2, $\mathcal{P}_I \preceq \mathcal{P}_{NB} = \mathcal{P}_N$ and $\mathcal{P}_I \preceq \mathcal{P}_W$.

Theorem 2.1. *[19] The FD $X \to A$ holds on r if $\mathcal{P}_X \preceq \mathcal{P}_A$*

Based on Example 2.3, $I \to NB$ (and therefore $I \to N$ and $I \to B$), $NB \to N$, $N \to NB$, and $I \to W$.

The following theorem is an extension of Theorem 2.1. It is very useful for testing FD satisfaction as its attribute combinations, X and XA, correspond to the two nodes connected by an edge in the attribute lattice.

Theorem 2.2. *[16, 19] The FD $X \to A$ holds on r iff one of the follows is true:*
(1) $\mathcal{P}_X = \mathcal{P}_{XA}$ or (2) $|\mathcal{P}_X| = |\mathcal{P}_{XA}|$.

As an example in Table 1, $\mathcal{P}_N = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$, $\mathcal{P}_{NB} = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$. So $N \to B$.

The *product* of two partitions $\mathcal{P}_X$ and $\mathcal{P}_Y$ is defined [10] to be

$$\mathcal{P}_X \times \mathcal{P}_Y = \{p_{ij} \mid p_{ij} = q_i \cap w_j \wedge p_{ij} \neq \phi \wedge q_i \in \mathcal{P}_X \wedge w_j \in \mathcal{P}_Y\}$$

Example 2.4. In Table 1, $\mathcal{P}_N = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$ and $\mathcal{P}_B = \{\{t_1, t_2, t_3\}, \{t_4\}\}$. So $\mathcal{P}_N \times \mathcal{P}_B = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$.

Theorem 2.3. *[10]* $\mathcal{P}_{XY} = \mathcal{P}_X \times \mathcal{P}_Y = \mathcal{P}_Y \times \mathcal{P}_X$.

By comparing $\mathcal{P}_N \times \mathcal{P}_B$ in Examples 2.4 and $\mathcal{P}_{NB}$ in Example 2.2, we see that $\mathcal{P}_N \times \mathcal{P}_B = \mathcal{P}_{NB}$ and the theorem is true.

Now we present how partitions are used with the attribute lattice of R. Firstly the database instance r is scanned to obtain the partition for each attribute of R and the partition is stored on the corresponding Level-1 node of the lattice. We note that the partitions are much smaller in bytes than the actual database. The partitions of multiple attributes are then calculated in the partitions of single attributes and r is not accessed any more. That is, the partition for each node at Level-i is calculated in the partitions of two parent nodes at Level-$(i-1)$. For example in Figure 2, to calculate the partition $\mathcal{P}_{ABC}$ for node 'ABC', we use the partition $\mathcal{P}_{AB}$ for node 'AB' and the partition $\mathcal{P}_{AC}$ for node 'AC'. If $\mathcal{P}_{AB} = \mathcal{P}_{ABC}$, then by Theorem 2.2, $AB \rightarrow C$ is true.

We analyze the complexity of the partition method. Let $m = |R|$. The time complexity for computing the partitions of single attributes is $O(m|r|^2)$ [2]. The time complexity for a partition product is $O(|r|^2)$. So the time complexity for the partition method is $O(m|r|^2 + |r|^2 \frac{m}{2} 2^m)$ where $\frac{m}{2} 2^m$ is the total number of possible candidate FDs. In comparison to the time complexity of nested loop approach on Page 7, the partition method is $\frac{m}{2}$ times faster than the nested loop method. By using hash to compute attribute partitions and a linear algorithm proposed in TANE [16] to compute partition products, this complexity can be reduced to $O(m|r| + |r| \frac{m}{2} 2^m)$.

To store partitions, the partition methods need to allocate space to each node at two levels: Level-$(i-1)$ and Level-(i). Thus the algorithm has extra space cost in comparison to the nested loop approach. The extra cost is formulated as $O(|r|(2^m/sqr(m)))$ where $2^m/sqr(m)$ is the maximal width of the attribute lattice in the number of nodes [16].

[2] If quick-sort is used, the complexity is $O(|r| \times log|r|)$. However, as $|r|$ becomes large, quick-sort runs out of memory.

Free-set

The free-set approach is proposed in [34, 33] and the algorithm implementing the approach is called FUN. FUN uses the cardinality of projection $r[X]$ to test FD satisfaction: $|r[X]| = |r[XA]|$ iff $X \rightarrow A$. We note that $|r[X]|$ is the same as the number of equivalent classes in the partition of X.

A *free set* is a minimal set X of attributes in schema R such that for any subset Y of X, $|r[Y]| < |r[X]|$. Thus, every single attribute is a free set because they do not have a subset. If X is a free set, $A \in (R - X)$, and $|X| < |XA|$ and $|A| < |XA|$, then XA is another free set. The lhs of any minimal FD is necessarily a free set. The free set of relation r, denoted by $Fr(r)$, is a set of all free sets on r. A non-free-set is a set whose cardinality equals to the cardinality of some of its subsets. A superset of a non-free-set is also a non-free-set.

To calculate the FDs supported by r, two more concepts are needed: attribute closure X^+ and quasi-attribute closure $X^\diamond$.

The closure of set X is calculated using cardinality as $X^+ = X + \{A | A \in (R - X) \wedge |r[X]| = |r[XA]|\}$. That is, X^+ contains attribute A on a node at the next level if $X \rightarrow A$.

The quasi-closure of X is $X^\diamond = X + (X - A_1)^+ + \cdots + (X - A_k)^+$ where $X = A_1 \cdots A_k$. In fact $X^\diamond$ contains the attributes on all the parent nodes of X and all the dependent nodes of the parent nodes.

The FDs are constructed using members of $Fr(r)$ and the two closures: $FD = \{X \rightarrow A | X \in Fr(r) \wedge A \in (X^+ - X^\diamond)\}$.

The pruning rule of the free-set method is to prune non-fee-sets X (a node). The method then covers the FDs ending at X by calculating the closure of the parent free-set nodes Y of X with the cardinality of the free-sets and without accessing the partitions. The essential pruning rules on Page 7 are also used in the method.

The algorithm traverses the attribute lattice level-wise. At Level-1, the cardinality of all single attributes are computed. Quasi-closure of each attribute at Level-1 is set to itself. At Level-2, the combinations of two attributes are computed from non-key attributes at Level-1. Then the cardinality for 2-attribute combinations is calculated. If the cardinality of a 2-attribute combination X is the same as the cardinality of its parent P at the previous level: $card(X) = card(P)$, (1) $P^+ = P^+ + X$; (2) $P \rightarrow (X - P)$; and (3) X is a non-free set and does not participate future node generation. After the closure of every attribute set P at the previous level is calculated, the quasi-closure of every attribute set X at the current level is calculated. Then the algorithm moves to node generation at Level-3.

For example, consider the relation in Table 1. Let $X : n$ represent

the attribute set X and its cardinality. Then at Level-1: the nodes are $I:4,\ N:3,\ B:2,\ W:2,\ S:3$. I is a key and does not participate new node generation at the next level. At Level-2: the nodes are $NB:3,\ NW:4,\ NS:3,\ BW:3,\ BS:3,\ WS:4$. WS and NW are keys. From Level-2 cardinalities, we find $card(NB) = card(N)$ and $card(NS) = card(N)$, so $N \rightarrow B$ and $N \rightarrow S$, NB and NS are non-free-sets. $N^+ = NBS$. The closure of Level-1 attributes: $N^+ = NBS$, $B^+ = B$, $W^+ = W$, and $S^+ = NBS$. As an example, the quasi-closure of NB is $NB^\diamond = NBS$.

The complexity of this approach includes the cost of computing free sets and the cost of computing FDs. Let $h = |Fr(r)|$ and $m = |R|$. The complexity of computing $Fr(r)$ is $O(h|r|m/2)$ where by average a free set has $m/2$ attributes. The cost of computing X^+, $X^\diamond$ and FDs is exponential to m in the worst case as the calculation follows the attribute lattice.

2.2.2 Bottom-up methods

Different from the top-down methods above, bottom-up methods compare the tuples of the relation to find agree-sets or difference-sets. These sets are then used to derive FDs satisfied by the relation. The feature of these methods is that they do not check candidate FDs against the relation for satisfaction, but check candidate FDs against the computed agree-sets or difference-sets. The seminal work for this type of methods is [24].

Negative cover

A negative cover [35, 22] is a cover of all FDs violated by the relation [3]. Negative cover is calculated using agree-sets of tuples of the relation [22].

The *agree-set* of two tuples t_1 and t_2, denoted by $ag(t_1, t_2)$, is the maximal set X of attributes such that $t_1[X] = t_2[X]$. The set of all agree-sets on relation r is denoted by $ag(r)$.

Example 2.5. For example in Table 1, $ag(t_1, t_2) = BW$, $ag(t_1, t_3) = B$, $ag(t_1, t_4) = \phi$, $ag(t_2, t_3) = NBS$, $ag(t_2, t_4) = \phi$, $ag(t_3, t_4) = W$. $ag(r)$ is a set containing all these sets as subsets: $ag(r) = \{BW, B, NBS, W\}$.

Agree-sets can be calculated from attribute partitions. Given the partition $\mathcal{P}_A$ of attribute A and two tuples t_1 and t_2, A is in $ag(t_1, t_2)$ if there exists a subset c in $\mathcal{P}_A$ such that t_1 and t_2 are contained in c. For efficiency reasons, stripped partitions are often used in the calculation [22].

[3] The term violated FD is the same as the terms of excluded FD [15] and functional independency [5]

The property of agree sets is that if $ag(t_1, t_2) = X$, then for any $A \in (R - X)$ $(t_1[A] \neq t_2[A])$. In other words, $X \rightarrow A$ is violated by t_1 and t_2. This is the basic principle of the negative cover approach.

The *max-set* of an attribute A, denoted by $max(A)$, contains maximal agree-sets that do not include A:

$$max(A) = \{X | X \in ag(r) \wedge A \notin X \wedge \nexists Y \in ag(r)(X \subset Y)\}$$

Because X does not include A, $X \rightarrow A$ is violated by at least one pair of tuples. Because $\nexists Y \in ag(r)(X \subset Y)$, so X is a maximal set. The reason for selecting maximal set X from $ag(r)$ is that if $XY \rightarrow A$ is violated by a pair of tuples, then $X \rightarrow A$ and $Y \rightarrow A$ are also violated by the same pair. For efficiency reasons, we just need to consider the maximal set, not all agree-sets.

The max-sets of all attributes form a cover of the negative closure [35] which contains all FDs that are violated by the relation.

Example 2.6. From example 2.5, $max(I) = \{BW, NBS\}$, $max(N) = \{BW\}$, $max(B) = \{W\}$, $max(W) = \{NBS\}$, $max(S) = \{BW\}$.

The max-sets are then used to derive FDs supported by r. The FDs with the rhs A, denoted by $FD(A)$, are formulated in two steps [15].

$$FD_1(A) = \{X \rightarrow A | X \in (R - A) \wedge \nexists Y \in max(A)(X \subseteq Y)\}$$

$$FD(A) = \{f | f \in FD_1(A) \wedge \nexists g \in FD_1(A)(lhs(g) \subseteq lhs(f))\}$$

The derivation is based on the observation that for any $Y \in max(A)$, $Y \rightarrow A$ is violated by at least a pair of tuples of r; if some attributes V are added to Y such that YV is not in $max(A)$, then $YV \rightarrow A$ is satisfied. As a result, as X is not a subset of any of such Y, $X \rightarrow A$ must be satisfied. The second formula says that $FD(A)$ contains only minimal FDs.

Guided by the formula, $FD(A)$ can be derived as follows [23]. Let L be the set of all attributes in $max(A)$. We first check every single attribute $B \in L$, if B is not contained in any set of $max(A)$, we add $B \rightarrow A$ to $FD(A)$. We next check combinations of two attributes from L. If a combination, say BC, is not contained in any subset of $max(A)$ and does not contain the lhs of any FD in $FD(A)$, we add $BC \rightarrow A$ to $FD(A)$. This process continues until combinations of three, four, $\cdots$, all attribute combinations of R not containing A are checked. The pruning rule (1) on Page 7 is used to reduce the number of combinations to be checked.

Example 2.7. From example 2.6, $max(I) = \{BW, NBS\}$. As all single attributes other than I are contained in an element of $max(I)$, no single attributes determine I. We next consider the combinations of two attributes: NB, NW, NS, BW, BS, WS. Among these, NW and WS are not contained in any element of $max(I)$. Thus $NW \rightarrow I$ and $WS \rightarrow I$ are added to $FD(I)$. We then consider the combinations of three attributes: NBW, NBS, NWS, BWS. Among these, NBS is in $max(I)$, NBW, NWS and BWS contain the lhs of a FD in $FD(I)$. As a result, there is not FD derived from the combinations of three attributes. Neither from the combination of four. Finally, $FD(I) = \{NW \rightarrow I, WS \rightarrow I\}$. □

The complexity of the negative cover approach is exponential to the number of attributes in R as in the worst case. To compute the agree set $ag(r)$, $m|r|^2$ comparisons are needed where $m = |R|$. Computing $max(A)$, for all A in R, takes $m|ag(r)|^2$ comparisons. Finally deriving FDs from $max(A)$, for all A, takes $m|max(A)|\frac{|L|}{2}2^{|L|}$ where $\frac{|L|}{2}2^{|L|}$ is the number of candidate FDs (see Section 2.2.1) and L contains all attributes in $max(A)$. The total complexity of the approach is $O(m|r|^2+m|ag(r)|^2+m|max(A)|\frac{|L|}{2}2^{|L|})$. The worst case is $L = R$.

Different variations of the negative cover approach are proposed in [12].

Different from using max-set to derive satisfied FDs directly, the work in [22] uses the complement of maximal agree sets to compute satisfied FDs. This approach is reviewed together with the difference-set approach below.

Difference-sets

The term difference-set is same as *necessary-set* [24] and the complement of max-set [22]. The method difference-set employs an opposite thinking from negative cover. The different-set of an attribute A, denoted by $dif(A)$, is a set containing subsets of attributes such that whenever attribute A has different values on two tuples, a subset in $dif(A)$ has different values on the same two tuples too [39].

Once $dif(A)$ is obtained, the lhs of satisfied FDs should contain an attribute from each subset of $dif(A)$.

Although the principle of deriving the lhs of satisfied FDs is simple, the search space of satisfied FD calculation is exponential to the number of all attributes in $dif(A)$. An algorithm called Dep-Miner is proposed in [22] for this purpose. Let R' contain all attributes appearing in any subset of $dif(A)$. Dep-Miner essentially considers all possible combinations of R' level-by-level following the attribute lattice of R'. The lhs of a satisfied FD is a combination in the lattice that intersects with all subsets of $dif(A)$.

The complexity of this level-wise approach would be similar to that of the negative cover.

To reduce the complexity, an algorithm called FastFDs is proposed in [39]. The algorithm constructs a lattice using the elements of a difference-set following a depth-first manner. The construction process ends with a cover of all FDs satisfied. Theoretically the number of the nodes in the constructed lattice is exponential to the number of attributes in the difference-set. The algorithm uses the subsets of the difference-set to reduce the size of the lattice.

We give some details of FastFDs. Each node of the constructed lattice contains a difference-set and an attribute set. The difference-set on the root node is $dif(A)$ and the attribute set is R_p containing all attributes of $dif(A)$. The attribute set is ordered descendingly by the number of subsets in $dif(A)$ that they cover and, if there is a tie, by their alphabetics. A node has $|R_p|$ child nodes and the edge e_i to the i-th child node c_i is labeled by the i-th attribute a_i of R_p. The difference-set D_{ci} of the child node c_i contains all the subsets of $dif(A)$ not containing the label a_i; and the attribute set R_{ci} of c_i contains all attributes of D_{ci} that are on the rhs of a_i in R_p and is ordered based on D_{ci}. A leaf node of the lattice is of two cases: (1) it has an empty difference-set and an empty attribute set. In this case, the labels on the edges leading to the leaf node (EtoLN) form the lhs of a satisfied FD; the minimality of lhs needs to be checked when new lhs is added to the discovered set; (2) it has a non-empty difference-set, but an empty attribute set. This case means that the labels on the EtoLN do not constitute the lhs of a satisfied FD.

As an example, consider Table 1. After redundant sets, equal to the union of some other subsets, are removed, $dif(I) = \{NS, W, NBS\}$. The lattice for the different set is given in Figure 3 where '{ }' contains a difference-set and '[ ]' the attribute set of the difference-set.

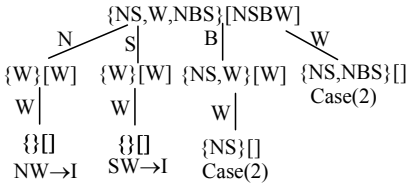

Figure 3: Lattice for a difference-set

2.3 Related topics to FD discovery

In this section, we briefly discuss the topics relating to FD discovery.

Sampling When a relation r is large, the cost of checking a candidate FD against r can be very high. To reduce the time for checking a candidate FD against r, sampling is a method proposed in the literature for this purpose.

Let f be a candidate FD, s be a small sample of relation r, $\delta \in [0, 1]$ a small confidence parameter. The principle of sampling is that if f is satisfied by tuples in s, f is satisfied by r with the confidence $(1 - \delta)$; if f is violated by tuples in s, f is violated by r firmly [20]. Using this principle, candidate FDs not holding on r can be pruned efficiently. As sampling is often used together with other methods, we will review details as we proceed.

Maintenance of discovered FDs [5] In this section, we assume that a relation r is given and all FDs supported by r have been discovered and are stored in Σ. We investigate how Σ is to change when a tuple is inserted or deleted from r.

Intuitively the insertion of a new tuple t may cause some FDs in Σ to be violated but no new FDs will be added. That is, insertion causes valid FDs to become less. In contrast, the deletion of a tuple from r may cause some invalid FDs to become valid, but no FDs in Σ will be removed. That is, deletion may cause new FDs added to Σ.

When a tuple t is inserted to r, the FDs can be maintained in the following way [5]. For each FD $X \to A$ in Σ, let X^+ be the attribute closure [2] computed from Σ. *After* the insertion operation, compute

$$q = \ SELECT\ X^+\ FROM\ r\ WHERE\ X = t[X]$$

If $|q| = 1$, the insertion does not effect Σ. If $|q| > 1$, there must exist B in X^+ that has different values in $q[X^+]$. Find $Z \to B$ in Σ such that $Z \subseteq X^+$ and delete $Z \to B$ from Σ.

When a tuple t is deleted from r, the FDs holding on r shall still hold on $(r - t)$. However new FDs may be added to Σ as the deletion may remove violating tuples of some FDs. Unfortunately there is no simple way to calculate new FDs to be added, but to re-apply FD discovery algorithms to $(r - t)$ [5] because there is no easy way to know if the deleted tuple was the only tuple violating the to-be-added FDs.

Embedded FDs [34] defines embedded FDs (EFDs) to be the FDs satisfied by a projected relation. That is, if $R' \subset R$ and $r' = r[R']$, EFDs are

all FDs satisfied by r and r'. [34] claims that EFDs can not be obtained by deleting the FDs with some attributes in $(R - R')$. Instead, EFDs are calculated using the free-set method reviewed in a previous section. Firstly the embedded free-set $Fr(r')$ is calculated: $Fr(r') = \{X' | X' \in Fr(r) \wedge X' \subset R'\}$. Then the EFDs are calculated by accessing r' to calculate the closure and the quasi-closure of the embedded free-set: $EFDs(r') = \{X' \rightarrow A | X' \in Fr(r') \wedge A \in ((X'^{+}_{r'} - X'^{\diamond}_{r'}) \cap R')\}$.

Embedded FDs have uses in materialized views. If materialized views are projected from some source relations and we know the FDs holding on the source relations, using the above approach, we can calculate the FDs holding on the views more efficiently.

Discover multivalued dependencies A multivalued dependency (MVD) [2] represents the requirement that the existence of some tuples is determined by values of some other tuples. MVDs are important constraints in database normalization.

The work in [12] uses three variations of an induction algorithm to discover MVDs from data. These algorithms are specialized from the same general algorithms from which FD discovery algorithms are generated.

Discover roll-up dependencies The concept of roll-up dependency (RUD) [7] is proposed to be used in data warehouses to determine roll-up granularity along hierarchical dimensions of data. As an example, assume that the relation schema $R(station, Time, Temperature)$ describes thermometer readings of weather stations. The RUD $\{(Time, HOUR), (Station, REGION)\} \rightarrow (Temperature, INTEGER)$ indicates that when the temperature readings, converted to the nearest integer, should be the same for stations in the same region in the same hour.

The work in [7] shows that given a relation r over schema R, a generalization schema H, a support threshold and a confidence threshold, the problem of finding RUDs is exponential to $|R|$ and polynomial to $|r|$. An algorithm is also proposed in the paper.

Discover keys Key discovery is a special case of FD discovery. The following theorem tests if a set of attributes form a key for relation r.

Theorem 2.4.

(1) [34, 19] Let X be a subset of R and r be a relation. X is a key of r iff $|r[X]| = |r|$.

(2) An attribute $A \in R$ is a key iff A is not in any of the agree sets of relation r.

With regard to (1), the main concern of the theorem is cardinality calculation. The cardinality $|r|$ can be obtained from the metadata of r. The cardinality of $|r[X]|$ is calculated in two cases. If X contains one attribute, the $|r[X]|$ can also be obtained from the metadata of r like $|r|$ [17]. If X contains multiple attributes, the partition method presented in a previous section can be used to determine $|r[X]|$ [34]. As the partition method follows the attribute lattice breadth-first, if $|r[X]| = |r|$, X is a minimal key.

The work in [4] shows that given a set of FDs, the problem of deciding whether there is a key of at most k attributes is NP-Complete.

Armstrong relations An Armstrong relation [4] over a set R of attributes and a set Σ of FDs is a relation over R that satisfies every FD in Σ. The importance of the relation is that by populating such a relation, a database designer can verify if a FD in Σ is incorrectly defined and can realize, by reading the example data, if any FD is missed from Σ. Based on [4], the number of tuples of a minimal Armstrong relations is between $\begin{pmatrix} m \\ \lfloor m/2 \rfloor \end{pmatrix}/m^2$ and $\begin{pmatrix} m \\ \lfloor m/2 \rfloor \end{pmatrix}(1 + (c/m^{1/2}))$ where c is a constant and m is the number of attributes in R, and the time complexity of populating an Armstrong relation is exponential to m.

The work in [30] defines an Armstrong database for an existing database and proves the bound for the Armstrong database. The constraints of the Armstrong database is discovered from the existing database.

3 Discovery of approximate FDs

The term *approximate functional dependency* [20] (AFD) is about the approximate satisfaction of a normal FD $f : X \rightarrow Y$. An AFD requires the normal FD to be satisfied by most tuples of relation r. In other words, the AFD f holding on r still allows a very small portion of tuples of r to violate f. Obviously AFDs include exact FDs.

To define the word *approximate* more accurately, violating tuples are used to calculate a *satisfaction error* $g(f, r)$. If $g(f, r)$ is less than or equal to the satisfaction threshold ϵ, f is said approximately satisfied by r or is ϵ-good. Otherwise, f is approximately violated by r or is ϵ-bad.

A number of methods have been proposed to calculate the satisfaction error. These methods are summarized and compared in [13]. In this paper, we review a method proposed in [20] that calculates the satisfaction error using the percentage of the tuples to be deleted to make a relation exactly satisfy the dependency:

$$g_3(X \rightarrow A, r) = 1 - \frac{max\{|s| \mid s \subseteq r, s \models X \rightarrow A\}}{|r|}$$

To check AFDs against r, the methods reviewed previously for checking exact FD satisfaction can be adapted by adding satisfaction error calculation. One such example is the work in [23] which proposes the **negative cover** method (on Page 13). The idea of negative cover is that, for any set $Z \in max(A)$, $Z \rightarrow A$ is violated by relation r. However with AFD discovery, if $Z \rightarrow A$ is not violated by majority of tuples, i.e., if $g_3(Z \rightarrow A, r) \leq threshold$, $Z \rightarrow A$ is an AFD discovered. The paper proposes an SQL query to calculate error.

The **sampling** method proposed in [20] is another approach to AFD discovery. As sampling uses a small portion of tuples to decide if an AFD f holds on the whole relation r, it puts extra conceptual complexity to the problem. Let s be a random sample of relation r. There are two cases if we test f against s. If f is satisfied or approximately satisfied by s, it may be violated by tuples in $r - s$. If f is violated by a small portion of tuples in s, it may be satisfied by r because the tuples in $r - s$ can be all satisfying.

To describe the probabilistic situations between the satisfaction by s and approximate satisfaction by r, a confidence parameter δ is introduced. With this parameter, if f is satisfied by s, we then claim that f is satisfied by r with the probability of $(1 - \delta)$.

Following the same reasoning, a cover of AFDs holding on s becomes a probabilistic cover holding on r.

The size of the random sample affects the accuracy of the cover although it does not fully determine the accuracy. A larger sample may not contain any violating tuples of r, but a smaller sample may contain most violating tuples. Therefore determining the size of the sample becomes very important. [20] proposes the bounds to decide $|s|$ in terms of δ, ϵ, and the size of r.

$$|s| \geq max\left\{\frac{8}{\epsilon}\ ln\frac{2}{\delta},\quad \frac{2}{\epsilon}\left\lceil\frac{log(2/\delta)}{log(4/3)}\right\rceil\left\lceil(2|r|ln2)^{1/2}+1\right\rceil\right\}$$

We comment that there is a difference between using sampling to test exact FDs and to test approximate FDs. In case of exact FDs, sampling

is used to efficiently remove invalid FDs. The principle is that if an FD is violated exactly by a sample s of relation r, it is also violated exactly by r. With this principle, candidate FDs not satisfied by s are efficiently removed. The remaining candidate FDs satisfied by s, denoted by $dep(s)$, need further testing because some FDs in $dep(s)$ may not be satisfied by r. Thus we check each FD $f \in dep(s)$ against r, if f is satisfied by r, we put f in $dep(r)$. In the end, $dep(r)$ contains all dependencies exactly satisfied by r and is an exact cover of all FDs satisfied by r.

In contrast, in the context of approximate FDs, a cover of AFDs discovered from a sample is a probabilistic cover on r.

An alternative name to AFD is soft FD proposed in [17] although [17] studies FDs with single attribute on the lhs only. [17] proposes a sample based approach that uses the system catalog to retrieve the number of distinct values of a column. Let s be a sample of relation r. The principle of this approach is that if s has a reasonable size and $|s[A]| > (1-\epsilon)|s[AB]|$, then the soft FD $A \rightarrow B$ holds on r with the probability of more than $(1-\epsilon)$. This principle has the same origin as $|r[X]| = |r[XA]|$ used the partition method reviewed in a previous section. Attribute correlations are calculated based on a measure called *mean-square contingency* which coincides with χ^2 distribution and from which probabilistic properties of discovered FDs are studied.

[19] uses the error measure of super keys to determine the approximate satisfaction of FDs and shows that $X \rightarrow A$ iff $g_3(X) = g_3(XA)$ where $g_3(Z) = 1 - |r[Z]|/|r|$ is the minimum fraction of rows that need to be removed from r for Z to be a super key.

[16] extends its partition method to compute approximate satisfaction by using the error measure $g_3(X \rightarrow A)$.

4 Discovery of conditional FDs

Conditional FDs (CFDs) are a new type of constraints that extend the traditional functional dependencies for data cleaning purpose [6]. Although CFDs are new to the database community, the work on CFD discovery has started [14, 11]. This section reviews the definition of CFDs and the work on CFD discovery.

Definition 4.1. Given two subsets X and Y of attributes of R, a *conditional FD* (CFD) is a statement $(X \rightarrow Y, S)$ where S is a pattern tableau on XY. A tuple t in a relation r on R satisfies a pattern tuple p in S, denoted by $t[X] \asymp p[X]$, if for every $A \in X$ ($p[A] =$'-' or $p[A] = t[A]$). A CFD is satisfied

by relation r over R iff for any two tuples $t_1, t_2 \in r$ and for each pattern tuple p in S, if $t_1[X] = t_2[X] \asymp p[X]$ then $t_1[Y] = t_2[Y] \asymp p[X]$. $\square$

In the definition, $X \rightarrow Y$ is called the *embedded FD*. This statement is same as the statement of a normal FD. We like to point out that the term embedded FD here is different from the same term on Page 17.

For example consider Table 1. Let $(NB \rightarrow S, S_1)$ be the CFD defining that people born on the date $d1$ with the name $John$ must have the supervisor $e1$ where tableau S_1 is given in Table 2. The tuples matching the lhs of the pattern tuple are t_2 and t_3 of Table 1. Because $t_2[NB] = t_3[NB] = < John, d1 > \asymp p[NB]$, and $t_2[S] = t_3[S] = < e1 > \asymp p[S]$, the CFD is satisfied.

In contrast, we consider the CFD $(NB \rightarrow S, S_2)$ which again has the same statement as in the previous example but the pattern tableau S_2 (Table 2) contains the wildcard '-'. The tuples matching the lhs of the pattern tuple are the first three tuples of Table 1. Because $t_1[S] = < e5 > \not\asymp p[S]$, the CFD is violated.

Table 2: CFD tableaus

S_1

N	**B**	**S**
John	d1	e1

S_2

N	**B**	**S**
-	d1	e1

Now we give the differences between the terms of exact FDs, approximate FDs and conditional FDs. Exact FDs (Definition 2.1) and Approximate FDs (Section 3) are all about the same statement $X \rightarrow Y$. The difference is on the degree of satisfaction. Exact FDs require $X \rightarrow Y$ to be satisfied by all tuples of a relation while AFDs allows a small portion of tuples to violate the FD statement. Conditional FDs use a different statement $(X \rightarrow Y, S)$ and the satisfaction is tested against only the tuples that match the tableau. A CFD is equivalent to an exact FD if the tableau contains a sole pattern tuple with only '-' values. Like FDs, CFDs can also be satisfied approximately [9], meaning that the checking allows a small portion of match tuples to violate the CFD conditions.

4.1 CFD discovery

On the discovery of CFDs, challenges are from two areas. Like in normal FDs, the number of candidate embedded FDs for possible CFDs is exponential. At the same time, the discovery of the optimal tableau for an embedded

FD is NP-Complete [14]. By optimal, it means that the tableau should contain patterns that maximize the number of tuples matching the FD while not allowing any violations.

A **level-wise algorithm** is proposed in [8] to discover CFDs. Candidate FDs are derived from the attribute lattice. The principle used in this algorithm is based on the properties of attribute partitions. All the tuples in an equivalent class of partition $P(Y)$ have the same value on Y. If an equivalent class c in $P(XA)$ equals to an equivalent class in $P(X)$, the tuples of c have the same value on A.

Given a candidate FD $X \rightarrow A$, the lhs X is divided into to subsets Q and W, Q is called the condition set and W is called the variable set. The algorithm assumes the partitions of $P(Q)$, $P(X)$ and $P(XA)$. It then calculates a set U_X to contain all equivalent classes in $P(X)$ that have at leat l tuples (the support) and equal to or are contained in an equivalent class in $P(XA)$. Finally a pattern tuple for the tableau of the CFD is discovered if there exists an equivalent class z in $P(Q)$ such that the tuples of z are contained in U_X. The pattern tuple is $< z[Q], -|- >$ if z is not an equivalent class in $P(XA)$; otherwise, the pattern tuple is $< z[Q], -|z[A] >$.

Consider Table 2 and a candidate FD $NB \rightarrow S$ where $X = NB$ and $A = S$. We let $Q = N$ and $l \geq 2$. Then $P(N) = P(NB) = P(NBS) = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$ and $U_X = \{\{t_2, t_3\}\}$. Consider the second equivalent class $z = \{t_2, t_3\}$ in $P(N)$, as the tuples of z are in U_X and z is in $P(NBS)$, the pattern tuple $< john, - \mid e1 >$ is produced.

A **greedy approximation algorithm** is proposed in [14] to compute a close-to-optimal tableau for a CFD when the embedded FD is given. The closeness of the discovered tableau to the optimal tableau is controlled by two parameters namely the support and the confidence. Given an embedded FD $X \rightarrow Y$ and for each tuple t in a relation r, the algorithm computes candidate patterns by considering all possible combinations of values in $t[X]$ and this results in an exponential number of candidate patterns in $|X|$: $|r[X]|2^{|X|}$. Then for each candidate pattern, the algorithm computes support and confidence. Further it iteratively chooses patterns with highest support over the support threshold to be included in the tableau. The algorithm is claimed to have time complexity of $|r|2^{|X|}$.

While a normal tableau, called *a hold tableau*, contains patterns to be satisfied, a *fail tableau* contains patterns that are violated by some tuples of data [14]. It is interesting that a fail tableau may reveal some interesting events [14]. The algorithm used for discovering hold tableaus is adapted to find fail tableaus [14].

The work in [11] proposes three algorithms called CFDMiner, CTANE

and FastCFD. These algorithms correspond to their relational counterparts FD_Miner, TANE, and FastFD respectively. CFDMiner aims to discover constant CFDs whose patterns have no wildcard, while CTANE and FastCFD discover general CFDs. CFDMiner has the best performance in constant CFD discovery and FastCFD is well scalable to the number of attributes in general CFD discovery.

5 Discovery of inclusion dependencies

Given two relations r_R on schema R and r_S on schema S, an *inclusion dependency* (IND) is a statement $Y \Subset X$ where $X \subseteq R$ and $Y \subseteq S$ and $|X| = |Y|$, requiring that $r_S[Y] \subseteq r_R[X]$. X is called the target and Y is called the reference. An IND can be defined on the same table. In this case $S = R$ and $r_R = r_S$, $X \cap Y = \phi$, $|X| = |Y| \leq \lfloor |R|/2 \rfloor$. We denote the maximum size of X by $maxk$, then $maxk = \lfloor |R|/2 \rfloor$ if $R = S$ or $maxk = min\{|R|, |S|\}$ if $R \neq S$.

Consider Table 1 and IND $S \Subset I$ which requires that supervisor ID values are taken from the ID column . Then $S \Subset I$ is violated because $t_1[S] = e_5$ is a reference not in the target, column I. If $t_1[S]$ were changed to any of e_1, e_2, e_3, e_4, the IND would be satisfied.

Like in FD discovery, generating candidate INDs is critical in IND discovery. With INDs, because $A_1A_2 \Subset A_3A_4$ does not mean $A_1A_2 \Subset A_4A_3$, the order of the attributes matters. Given $R = S = A_1 \cdots A_m$, to choose k ($k <= maxk$) attributes for the reference Y, there are $\begin{pmatrix} m \\ k \end{pmatrix}$ possibilities. Once the reference is chosen, we need to choose k attributes for the target X from the remaining $(m - k)$ attributes of R. There are $\mathfrak{P}(m - k, k)$ possibilities where $\mathfrak{P}(m - k, k)$ is the permutation of k from $m - k$. We note that we did not use permutation for Y because if r satisfies $AB \Subset CD$, then r satisfies $BA \Subset DC$ [27]. As k can vary from 1 to $maxk$, the total number of candidate INDs on R is

$$\begin{pmatrix} m \\ 1 \end{pmatrix} * \mathfrak{P}(m-1, 1) + \cdots + \begin{pmatrix} m \\ maxk \end{pmatrix} * \mathfrak{P}(m - maxk, maxk)$$

The complexity of candidate INDs is higher than that of FDs on the same schema R. With FDs, $\begin{pmatrix} m \\ k \end{pmatrix}$ is the number of nodes at level k of the attribute lattice, while with INDs, $\begin{pmatrix} m \\ k \end{pmatrix}$ is amplified by $\mathfrak{P}(m - k, k)$

which is much larger than 2. [4]

Like in FDs, pruning candidate INDs becomes the most important task in IND discovery. The following pruning rules are proposed in the literature [27, 3, 21]. In the rules, $X_S = A_{i_1} \cdots A_{i_k}$, $Y_S = B_{i_1} \cdots B_{i_k}$, $X_L = A_1 \cdots A_m$, $Y_L = B_1 \cdots B_m$ and $i_1, \cdots, i_k \in [1, \cdots, m]$. That is, X_S and Y_S are smaller attribute sets contained in the larger sets X_L and Y_L respectively with proper attribute positions.

(1) If $X_L \Subset Y_L$ is satisfied by r, $X_S \Subset Y_S$ is satisfied by r.
(2) If $X_S \Subset Y_S$ is violated by r, $X_L \Subset Y_L$ is also violated by r.
(3) If exists i such that $type(A_i) \neq type(B_i)$, then $X_L \not\Subset Y_L$ and $Y_L \not\Subset X_L$.
(4) If $max(A) > max(B)$ or $min(A) < min(B)$, then $A \not\Subset B$.

Rule (1) is used to find valid INDs. It starts with large candidate INDs - INDs with large number of attributes in the reference and the target. If large candidate INDs are not satisfied, it checks candidate INDs with one less attributes. The largest INDs have $maxk$ attributes and the number of largest INDs very large: $\begin{pmatrix} m \\ maxk \end{pmatrix} * \mathfrak{P}(m - maxk, maxk)$. If R contains 9 attributes, then there are 126*240 candidate 4-attribute INDs to be checked. In contrast, Rule (2) is used to find invalid INDs and the checking starts with single attribute (called *unary*) INDs. The attributes of all unary INDs not satisfied by r will not be considered when two-attribute (called *binary*) candidate INDs are generated. Again if R contains 9 attributes, then the number of unary IND to be checked is 9*8. Rule (3) works at the schema level. It checks violation using metadata without having to access data of the table. The combination of Rules (3) and (2) gives efficiency. Rule (4) uses database statistics to find invalid INDs.

In summary, it is more efficient to start IND discovery by checking unary IND satisfaction to find a set of INDs holding on r. Then candidate FDs are generated following the attribute lattice and the pruning rules are applied before the new candidates are checked by accessing the database.

5.1 Unary INDs

The conventional algorithm of checking whether the unary IND $A \Subset B$ is satisfied by r is to compute $r[A]$ and $r[B]$ to get two sets, and then verify the set containment $r[A] \subseteq r[B]$. This can be done in SQL or by programs [3].

[4] Note that the attribute lattice is symmetric in the number of nodes to the middle level.

[27, 28] proposes an algorithm called MIND that acts differently from the conventional approach. It pre-computes an extraction table $xt(d)$ for each domain d of the tables t and s and thus domain types are considered by the algorithm. $xt(d)$ contains two columns V and U. Column V contains distinct values v of d appearing in t and s. Column U contains all attribute names of t and s such that the columns of these attributes contain v. The principle of the algorithm is that, given unary IND $P \Subset Q$, $r \models P \Subset Q$ iff for each domain d, $Q \in \cap\{\pi_U \sigma_{P \in U}(xt(d))\}$, that is, Q is in some tuples containing P.

For example in Table 3(c), $\cap\{\pi_U \sigma_{A \in U}(xt(int))\} = ABD \cap ABD = ABD$, so we have $A \Subset B$ and $A \Subset D$. Similarly $\cap\{\pi_U \sigma_{B \in U}(xt(int))\} = ABD \cap ABD \cap BD = BD$, so we have $B \Subset A$ and $B \Subset D$. There is not other IND on the *int* domain.

Table 3: Example of IND discovery

(a) r

A	**B**	**C**
1	1	a
2	2	a
2	3	b

(b) s

D	**E**
1	a
2	b
3	c
4	d

(c) $xt(int)$

V	**U**
1	ABD
2	ABD
3	BD
4	D

The time complexity of computing $xt(d)$ is $|r| * |R|$. The time complexity for unary IND discovery is $O(|R| * (|xt(int)| + |xt(str)| + |xt(float)|)/3)$ if the domains are *int*, $float$, and *str* only.

The work in [3] compares the efficiencies between two different ways of implementing the discovery of unary attribute INDs. The first way is to use SQL Join, Minus and 'Not In' queries to check the satisfaction of an IND on the SQL server. The other is to use a java program to check the satisfaction on the client computer by respectively scanning the database once for each candidate IND and scanning the database once for all INDs together. The results show that the Join query performed better than other SQL queries, but not better than any of the client program. Among the client programs, the program scanning the database for each candidate IND performed best. The Java programs performed better because the user has better control and order can be used in checking.

5.2 N-ary INDs and cover

To compute a cover of all INDs, algorithm MIND [27, 28] takes a level-wise approach to generate candidate INDs. It is based on the view that $A \Subset B$ and $C \Subset D$ are necessary conditions for $AC \Subset BD$ to be satisfied. That is, if $A \notin B$ or $C \notin D$, then it is impossible that $AC \Subset BD$. In this case, $AC \Subset BD$ is pruned and no testing of its satisfaction is necessary. On the contrary, if $A \Subset B$ and $C \Subset D$, then $AC \Subset BD$ has a chance to be satisfied by the relation and becomes a candidate of size 2. This candidate FD is then tested for satisfaction. After all candidate INDs of size 2 are generated and tested, the algorithm generates and tests candidate INDs of size 3.

The algorithm, called FIND, in [21] also uses the same principle that $A \Subset B$ and $C \Subset D$ are necessary conditions for $AC \Subset BD$ to be satisfied. Unlike [28], FIND generates candidate INDs by converting satisfied INDs of sizes 1 and 2, discovered exhaustively, into a hypergraph and computing cliques of the graph. A 3-clique of the hypergraph corresponds to a candidate ternary IND. The candidate INDs are checked for satisfaction and if they are not satisfied by the relations, they are removed from the graph. A 4-clique is further computed from the 3-cliques to have candidate INDs of size 4.

A recent algorithm called Zigzag proposed in [26] uses the borders of theories to compute the cover of INDs holding on a database. The cover equals to the positive border which contains all INDs not implied by others. The algorithm finds the negative border containing the shortest (in the number of attributes) unsatisfied INDs. It then guesses the positive border using the negative border. The guess enables candidate IND generation to jump to longer (in the number of attributes) candidate INDs directly without having to go through candidate INDs level-by-level. The experiments showed that the performance of the algorithm is better than the MIND algorithm [27].

5.3 Approximate satisfaction

Like in FD discovery, an IND is approximately satisfied if most tuples of the database satisfy the IND. [28] proposes an error measure for approximate satisfaction for the extraction table approach reviewed in second paragraph of subsection 5.1. The error measure is defined by the number of tuples to be deleted to make the database satisfy the IND exactly. If a database satisfies an IND $X \Subset Y$ with the error measure less than a given threshold $\epsilon \in [0, 1]$, the satisfaction is denoted by $r \models_\epsilon X \Subset Y$. With the error measure, the satisfaction checking becomes the calculation of the error. For an unary IND

$A \Subset B$, $r \models_\epsilon A \Subset B$ iff

$$(1 - \Sigma_{v \in V} \frac{|\{t \in xt(d) | v = t[V] \wedge A \in t[U] \wedge B \in t[U]\}}{|\{t \in xt(d) | v = t[V] \wedge A \in t[U]\}}) \leq \epsilon$$

The calculation needs to access $xt(d)$ only once. The cover of all approximate INDs is calculated using a similar pruning rule to that of exact INDs: Let I_1, I_2 be 2 candidate INDs such that I_1 is subsumed by I_2. If $r \not\models_\epsilon I_1$, then $r \not\models_\epsilon I_2$.

The work in [29] conducts a study on how to compute an approximate set of approximately satisfied INDs.

6 Discovery of XML functional dependencies

Functional dependencies in XML (XFDs) are defined differently in several proposals [1, 31, 38]. In this review, we present the XFDs defined in [42], called generalized tree tuple XFDs, as the work done on XFD discovery is based on this definition.

We start with some terms. A set element is an element that can occur multiple times under a parent element in a document. A generalized tree tuple (tree tuple for short) under a pivot node v in a document T is a subtree, denoted by $tt(v)$, such that $tt(v)$ contains all descendant nodes of v, all ancestor nodes a of v, and all descendants d of a such that the path between a and d does not contain a set element, and the edges among all these nodes. In our notation we use $e[i]$ to mean a tree node labeled by e and with the identifier i. Consider Figure 4, where $C, S, L, B, Bn, A, N, F, T, P$ stand for $Company$, $Store$, $Location$, $Book$, $BookNumber$, $Author$, $Name, Affiliation$, $Title$, $Price$ respectively and where S, B and A are set elements. We use A^* to mean A is a set element. Let the pivot node be $B[4]$. Then the tree tuple for the pivot is given in Part (b) where $S[1]$ and $C[0]$ are ancestor nodes of $B[4]$, and $L[3]$ is a descendant of $S[1]$ and the path between $L[3]$ and $S[1]$ does not involve a set element. The path p between the root node of T and v is called the pivot path. $tt(v)$ is called a tuple of p.

Given a pivot path p, a set Q of relative paths to p and a tree tuple $tt(v)$ of p, the projection of $tt(v)$ on Q, denoted by $tt(v)[Q]$, is a subtree of $tt(v)$ by removing the branches of $tt(v)$ that are not labeled by any path in Q. For example let $tt(B[4])$ be the tuple in Figure 4 (b). With the pivot path $p = C/S/B$, the projection of $tt(v)$ to the set of paths $\{Bn, A\}$ is $tt(B[4])[\{Bn, A\}] =$

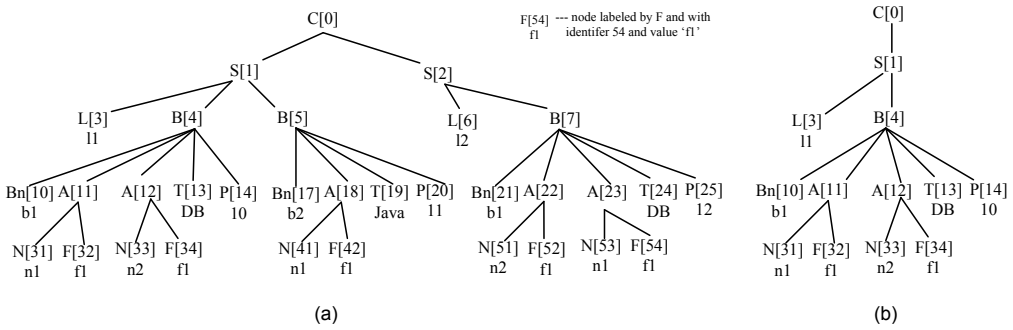

Figure 4: (a) An XML document (b) a tree tuple with pivot B[4] for paths Bn, A^*, T

$(C[0](S[1](B[4](Bn[10]) \quad (A[11](N[31])(F[32])) \quad (A[12](N[33])(F[34])))))$ where the root node is $C[0]$, parallel brackets mean siblings and nested brackets mean children.

Two trees are equal if they have exact same paths, labels, and text values when order between sibling nodes are not considered. All *null* values are distinct and a *null* value is different from any of the non-*null* values in the tree.

An XFD is defined by an expression $(pv, \{p_1, ..., p_m\} \rightarrow p_{m+1})$ where pv is the pivot path, $\{p_1, ..., p_m\}$ (denoted by P) is the lhs, and p_{m+1} is the rhs. The XFD is satisfied if for any two tuples $tt(v_1)$ and $tt(v_2)$ where v_1 and v_2 are pivot nodes of pv, if $tt(v_1)[P] = tt(v_2)[P]$, then $tt(v_1)[p_{m+1}] = tt(v_2)[p_{m+1}]$.

Figure 4 (a) satisfies the XFDs $(C/S/B, Bn \rightarrow A^*)$ and $(C/S/B, \{A^*, T\} \rightarrow Bn)$, $(C/S, B/A^* \rightarrow L)$. It violates $(C/S/B, A^* \rightarrow Bn)$.

XFDs are related to relational FDs and MVDs. XFDs can be mapped from relational FDs via the nest operation if the order of the child nodes under a parent node in XML does not matter [37]. Relational MVDs can be represented in the same way in XML as they are in nested relations because XML and nested relations share many similarities [32, 42]. However MVDs cannot capturer all XML redundancies involving set elements on both sides of a MVD and the notation of XFD is necessary [42].

The problem of discovering XFDs from a given XML document is to firstly find candidate XFDs and then to check the candidate XFDs against the document. The candidate XFDs that are satisfied by the document are included in the result of the discovery.

The approach proposed in [42] converts the input XML document into a set of foreign key connected relational tables. Thus each table contains attributes which are leaf elements (those that do not contain other elements as children) and the parent node identifier. The approach then finds out in each table the XFDs that are satisfied by the attributes (elements) in the table without considering the paths. It further takes the XFDs found in each table into multiple tables.

To translate a document to tables, a table is created for a set-typed or record-typed element and the name of the table is the name of the element. The attributes of the table include the element of the set type or the leaf elements of the record type, plus two attributes, *nid* - the identifer of the node that the tuple is for, and *pid* - the identifier of the parent node. With these rules, the document in Figure 4(a) is translated into the tables in Table 4.

Table 4: Tables for Figure 4(a)

C

nid	**pid**
0	$\perp$

S

nid	**pid**	**L**
1	0	l1
2	0	l2

B

nid	**pid**	**Bn**	**T**	**P**
4	1	b1	DB	10
5	1	b2	Java	11
7	2	b1	DB	12

A

nid	**pid**	**N**	**F**
11	4	n1	f1
12	4	n2	f1
18	5	n1	f1
22	7	n2	f1
23	7	n2	f1

The first step of XFD discovery is to find relational FDs holding on each table by not considering *nid* and *pid*. The partitioning method reviewed in Section 2.2.1 is used for this purpose. A discovered relational FD f corresponds to the XFD x. The pivot path of x is the path reaching the table name and the lhs and the rhs of x are the lhs and the rhs of f respectively. For example in Table A of Table 4, the relational FD discovered is $N \rightarrow F$. The corresponding XFD is $(C/S/B/A, N \rightarrow F)$.

The next step is to consider the relationships among the tables. This starts from the table having no child tables. Two pruning rules are used.

(1) If in a child table, $A \rightarrow B$ is violated by two tuple t_1 and t_2 while $t_1[pid] = t_2[pid]$, then adding any attributes from ancestor tables will not make it satisfied because t_1 and t_2 have the same parent and other attributes at a higher level will not distinguish them. For example, $F \rightarrow A$ is violated by t_{11} and t_{12} in Table A. Adding Bn, or T, or L will not change the satisfaction status because the two tuples share whatever values added. (2) If a child table satisfies $A \rightarrow B$, then adding any attribute from an ancestor table will produce an implied XFD. For example, $(C/S/B/A, N \rightarrow F)$ is satisfied by Table A. Then $(C/S/B/A, \{N, ../Bn\} \rightarrow F)$ is implied.

Thus if an FD $A \rightarrow B$ is violated by a pair of tuples t_i and t_j having different parents in a child table, the algorithm goes to check the parent table. Let t'_i and t'_j be tuples in the parent table referenced by t_i and t_j respectively. The algorithm checks whether any attribute C satisfies $t'_i[C] \neq t'_j[C]$. If yes, we have $\{A, ../C\} \rightarrow B$; otherwise, if t'_i and t'_j have the same parent, $A \rightarrow B$ does not hold; else the search goes toward the parent table of the parent.

For example $(C/S/B, T \rightarrow P)$ is violated by tuples t_4 and t_7 which have different parents t_1 and t_2. In the parent table S, there is an attribute L such that $t_1 \neq t_2$. Therefore $(C/S/B, \{T, ../L\} \rightarrow P)$ is an XFD discovered. We note that here table B has only one pair of violating tuples. Generally, all pairs of tuple needs to be checked.

The complexity of this algorithm is exponential. For each relation r and its schema R, the complexity for computing XFDs from its descendant tables is $O(|r| * |R| * 2^{|R|} + |r| * |R_d| * 2^{|R|+|R_d|})$ where R_d is a set containing attributes of all descendant relations.

7 Conclusion

In this paper, we reviewed the methods for discovering FDs, AFDs, CFDs, and INDs in relational databases and XFDs in XML databases. The dependency discovery problem has an exponential search space to the number of attributes involved in the data. Fortunately, most data contains FDs and INDs with single or a few attributes on the lhs. Some efficient algorithms have been proposed.

With FD discovery, the direction of computation starts with FDs having fewer attributes in lhs. The discovered FDs are then used to prune other candidate FDs in the attribute lattice so that the search space of the computation is reduced. The most commonly proposed and cited method in the literature is the partition method and the negative cover method. The

partition method is also used in XFD discovery.

In discovering AFDs, the sampling method is used to find FDs that are approximately satisfied.

With regard to the IND discovery, the direction of computation starts with small INDs too. The invalid INDs discovered are then used to prune candidate INDs to reduce the complexity of computation.

In the area of CFD discovery, although some algorithms for FD discovery can be adapted for CFD discovery purpose, the discovery of an optimal tableau is NP-Complete and the discover of a good tableau seems not an easy task. More simple but effective and efficient algorithms are still needed.

The work in discovering XML functional dependencies has just started. The only work done on XFD discovery converts XML data into relational data and then applies the partition method to the converted relational data. No work on XML IND discovery has been seen in the literature.

References

[1] Marcelo Arenas and Leonid Libkin. A normal form for xml documents. *ACM Transactions on Database Systems*, 29:195–232, 2004.

[2] Pado Atzgeni and Valeria D. Antonellis. *Relational Database Theory.* The Benjamin/Cummings Publishing Company,Inc, 1993.

[3] Jana Bauckmann, Ulf Leser, and Felix Naumann. Efficiently computing inclusion dependencies for schema discovery. *2nd Intl Workshop on Database Interoperability*, 2006.

[4] Catriel Beeri, Martin Dowd, Ronald Fagin, and Richard Statman. On the structure of armstrong relations for functional dependencies. *Journal of Association for Computing Machinery*, 31(1):30–46, 1984.

[5] Siegfried Bell. Discovery and maintenance of functional dependencies by independencies. *KDD-95*, pages 27–32, 1995.

[6] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. *ICDE*, pages 746–755, 2007.

[7] Toon Calders, Raymond T. Ng, and Jef Wijsen. Searching for dependencies at multiple abstraction levels. *TODS*, 27(3):229–260, 2002.

[8] Fei Chiang and Rene J. Miller. Discovering data quality rules. *VLDB Conference*, 1(1):1166–1177, 2008.

[9] Graham Cormode, Lukasz Golab, Korn Flip, Andrew McGregor, Divesh Srivastava, and Xi Zhang. Estimating the confidence of conditional functional dependencies. *SIGKDD Conference*, pages 469–482, 2009.

[10] Stavros S. Cosmadakis, Paris C. Kanellakis, and Nicolas Spyratos. Partition semantics for relations. *PODS*, pages 261–275, 1985.

[11] Wenfei Fan, Floris Geerts, Laks V. S. Lakshmanan, and Ming Xiong. Discovering conditional functional dependencies. *ICDE*, pages 1231–1234, 2009.

[12] Peter A. Flach and Iztok Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

[13] Chris Giannella and Edward Robertson. On approximation measures for functional dependencies. *Information Systems*, 29(6):483–507, 2004.

[14] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. *VLDB Conference*, pages 376–390, 2008.

[15] G. Gottlob and L. Libkin. Investigations on armstrong relations, dependency inference, and excluded functional dependencies. *Acta Cybernetica*, 9(4):395–402, 1990.

[16] Yka Huhtala, Juha Karkkainen, Pasi Porkka, and Hannu Toivonen. Tane : An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2):100–111, 1999.

[17] Ihab F. Ilyas, Volker Mark, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. *SIGMOD Conference*, 2004.

[18] Martti Kantola, Heikki Mannila, Kari-Jouko Rih, and Harri Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7(7):591–607, 1992.

[19] Ronald S. King and James Oil. Discovery of functional and approximate functional dependencies in relational databases. *Journal of Applied Mathematics and Decision Sciences*, 7(1):49–59, 2003.

[20] Jyrki Kivinen and Heikki Mannila. Approximate dependency inference from relations. *LNCS 646 - Database Theory ICDT '92*, pages 86–98, 1992.

[21] Andreas Koeller and Elke A. Rundensteiner. Heuristic strategies for inclusion dependency discovery. *LNCS 3291 - On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pages 891–908, 2004.

[22] Stephane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. *LNCS 1777 - 7th International Conference on Extending Database Technology (EDBT): Advances in Database Technology*, 1777:350–364, 2000.

[23] Stephane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Functional and approximate dependency mining: database and fca points of view. *Journal of Experimental and Theoretical Artificial Intelligence*, 14(2):93–114, 2002.

[24] Heikki Mannila and Kari-Jouko Rih. Dependency inference. *VLDB*, pages 155–158, 1987.

[25] Heikki Mannila and Kari-Jouko Rih. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40:237–243, 1992.

[26] Fabien De Marchi, Frdric Flouvat, and Jean-Marc Petit. Adaptive strategies for mining the positive border of interesting patterns: application to inclusion dependencies in databases. *Constraint-Based Mining and Inductive Databases - LNCS3848*, pages 81–101, 2006.

[27] Fabien De Marchi, Stephane Lopes, and Jean-Marc Petit. Efficient algorithms for mining inclusion dependencies. *LNCS 2287 - EDBT*, pages 199–214, 2002.

[28] Fabien De marchi, Stephane Lopes, and Jean-Marc Petit. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, 32(1):53–73, 2009.

[29] Fabien De Marchi and Jean-Marc Petit. Approximating a set of approximate inclusion dependencies. *Advances in Soft Computing - Intelligent Information Processing and Web Mining*, 31:633–640, 2005.

[30] Fabien De Marchi and Jean-Marc Petit. Semantic sampling of existing databases through informative armstrong databases. *Information Systems*, 32(3):446–457, 2007.

[31] Wai Yin Mok, Yiu-Kai Ng, and David W. Embley. A normal form for precisely characterizing redundancy in nested relations. *TODS*, 21(1):77–106, 1996.

[32] Wai Yin Mok, Yiu-Kai NG, and David W Embley. A normal form for precisely characterizing redundency in nested relations. *TODS*, 21(1):77–106, 1996.

[33] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. *ICDT*, pages 189–203, 2001.

[34] Noel Novelli and Rosine Cicchetti. Functional and embedded dependency inference: a data mining point of view. *Information Systems*, 26(7):477–506, 2001.

[35] Iztok Savnik and Peter A. Flach. Bottom-up induction of functional dependencies from relations. *AAAI Workshop of KDD*, pages 174–185, 1993.

[36] A.M. Silva and M.A. Melkanoff. A method for helping discover the dependencies of a relation. *Advances in Data Base Theory*, 1:115–133, 1981.

[37] Millist Vincent, Jixue Liu, and Chengfei Liu. Redundancy free mappings from relations to xml. *LNCS 2762 - Fourth International Conference on Web-Age Information Management (WAIM)*, pages 55–67, 2003.

[38] Millist Vincent, Jixue Liu, and Chengfei Liu. Strong functional dependencies and their application to normal forms in xml. *ACM Transactions on Database Systems*, 29(3):445–462, 2004.

[39] Catharine Wyss, Chris Giannella, and Edward Robertson. Fastfds: a heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. *DaWaK*, pages 101–110, 2001.

[40] Hong Yao and Howard J. Hamilton. Mining functional dependencies from data. *Journal of Data Mining and Knowledge Discovery*, 16(2):197–219, 2008.

[41] Cong Yu and H. V. Jagadish. Efficient discovery of xml data redundancies. *VLDB*, pages 103–114, 2006.

[42] Cong YU and H. V. Jagadish. Xml schema refinement through redundancy detection and normalization. *The VLDB Journal*, 17(2):203–223, 2008.

Jixue Liu is a senior lecturer in the University of South Australia. He received his bachelor's and master's degrees in engineering areas in China, and his PhD in computer science from the University of South Australia in 2001. His research interests include view maintenance in data warehouses, XML integrity constraints and design, XML and relational data, constraints, and query translation, XML data integration and transformation, XML integrity constraints transformation and transition, data privacy, and the mining of database constraints.

Jiuyong Li is currently an associate professor at University of South Australia. He received his PhD degree in computer science from Griffith University in Australia. He was a lecturer and senior lecturer at the University of Southern Queensland in Australia. His main research interests are in data mining, privacy preservation and Bioinformatics. He has published more than 50 research papers in various publication venues, including some high impact ones.

Chengfei Liu is a Professor and the head of the Web and Data Engineering research group, Swinburne University of Technology, Australia. He received the BS, MS and PhD degrees from Nanjing University, China in 1983, 1985 and 1988, respectively, all in Computer Science. Prior to joining Swinburne, he taught at the University of South Australia and the University of Technology Sydney, and was a Research Scientist at Cooperative Research Centre for Distributed Systems Technology, Australia. He has published more than 100 peer-reviewed papers in various journals and conference proceedings.

Yongfeng Chen is a professor in Xian University of Architecture and Engineering. He got his bachelor's degree and masters (by research) degree in engineering from Xian University of Architecture and Technology in 1983 and 1988 respectively. His research interests include software engineering, database applications, and the application of computer and operations research in engineering.