Discovery of Constraints from Data for Information System Reverse Engineering

Wie Ming LIM, John Harrison
Centre for Software Maintenance
School of Information Technology
The University of Queensland, QLD 4072, Australia
{wieming, harrison}@it.uq.edu.au

Abstract

The extraction of functional dependencies is a fundamental activity in the database design recovery process which is part of on an overall information systems reverse engineering effort. Existing algorithms for this task are computationally expensive and appear to be infeasible if applied to large legacy database instances, e.g., their performance deteriorated when number of attributes or/and instances is large and they cannot tolerate erroneous data that may occur in deployed commercial systems. The contributions of this paper are as follows. We propose two algorithms for discovering functional dependencies from data. The collective-FD algorithm, which is based on top-down approach, eliminates redundant specialised functional dependencies to be proposed. The attribute-list algorithm, which is based on bottom-up approach, enables more accurate functional dependency hypotheses to be discovered. In anticipating noisy data, we propose an effective method to discover possible data errors and compute partial functional dependencies. The result is an error-tolerant functional dependencies discovery approach that is more applicable to real world databases for design recovery.

1. Introduction

Information Systems are perceive to consist four major components: application programs, databases, screens, and reports. Most of the research on information systems reverse/re engineering emphasise on recovering design information from multiple information sources [2, 4, 6, 7, 17, 18, 19, 22, 25, 28]. These legacy systems share a common characteristic, that is they use, produce, and maintain data, either in the form of flat files, databases, or spread sheets. Studies have shown that data is an invaluable resource for extracting business knowledge [23] as well as systems design information [26]. Research in constraints discovery have also confirmed that data is a reliable source for recovering database constraints [3, 5, 9, 12, 13, 20, 21]. There might be situation where data is the only resource available for design recovery. However, there is no research dedicated to that scenario. Our focus in this part of the information systems reverse engineering research is on extraction of design constraints from data for design recovery purpose.

The discovery of functional dependencies (FDs) is a fundamental activity in the database design recovery process. Many techniques and algorithms for discovering functional dependencies from the data have been proposed [3, 5, 9, 12, 13, 20, 21]. These techniques and algorithms share a common goal: generate minimum number of functional dependency (FD) hypotheses and verify them against the database. These algorithms suffer from a common braw back: the performance of the algorithm deteriorated when the number of attributes or/and the size of the data is large. Furthermore they all assumed that data is error free. This is a dangerous assumption since the information in the real world databases are almost always noisy due to a variety of reasons, e.g., encoding errors, measurement errors, etc. Even though these errors might consist only a small percentage of the overall data in the database, their present however could affect the reliability and accuracy of the FDs discovery result. In this paper, we propose a functional dependencies discovery approach that generates FD hypotheses from a data sample and verifies the FDs against the database extension. The goal of the proposed approach is to further reduce the number of FD hypotheses before verifying them. We propose two algorithms for generating FD hypotheses. The first algorithm is based on *top-down* approach whereas the second algorithm is based on *bottom-up* approach. In anticipating errors in the data, we propose a method for discovering partial FDs by detecting possible data errors in the data. The result is an error-tolerant FDs discovery technique that is more applicable to the real world databases.

This paper is organised into two parts. The first part describes the data sample approach and the second part describes the partial FD discovery approach. In section two we show the related work on functional dependencies discovery techniques. Section three defines the notation used in this paper. Section four gives an overview of the proposed data sample approach. Section five describes the proposed top-down collective FDs specialising algorithm including an illustrated example and algorithm analysis. Section six depicts the bottom-up attribute list algorithm, with an example and algorithm analysis. Section seven describes the *verifying* algorithm. Section eight compares and contrasts the two algorithms. Section nine documents the proposed partial FDs discovery method. The last section contains the summary and future work.

2. Related work on Functional Dependencies Discovery

In the past ten years, many techniques and algorithms have been proposed for discovering functional dependencies from database extensions for various motivations [3, 5, 9, 12, 13, 20, 21]. We do not intend to describe all these techniques in detail. We will however provide a brief summary of the current FD discovery techniques. Generally we can categorise all FDs discovery methods into two broad categories. First category is based on ***bottom-up*** approach where it begins with analysis of the data and compares all tuples, and then proposes or refutes FDs by examine whether the condition "$t_1[A] = t_2[A]$ *and* $t_1[B] = t_2[B]$" is satisfied [5]. If this condition is satisfied, then by the definition of the functional dependency, attribute *A* determines attribute *B*. That is denoted by $A \rightarrow B$. Otherwise attribute B does not depend on *A*. This is a naïve FD discovery approach since all the tuples in the table will need to be compared, which requires $O(n^2)$, and the number of possible combination of the Left-Hand-Side (LHS) of FD is exponential (2^m-1) [12, 20].

The second category is based on ***top-down*** approach where it starts by proposing a set of possible FDs and then verifies them against the database [3, 20, 21]. Due to the exponential number of possible combinations of the LHS of FDs, several methods were proposed to curb the number of possible combinations. A common method is based on the concept of **more-general** and **more-specific** FDs [3, 21]. A dependency $X \rightarrow A$ is defined to be more-general than the dependency $Y \rightarrow A$ if $X \subseteq Y$. Inversely, dependency $Y \rightarrow A$ is defined to be more-specific than dependency $X \rightarrow A$. Based on these concepts, only the most-general FDs (also called *left-reduced* FDs) are proposed and verified. If the verification failed, the FD will be made more-specific by expanding the number of attribute on the LHS of FD. A variation of this method is used by Flach [5], Schlimmer [20], and Bell [3]. Other methods for reducing the number of possible combination include restricting the number of LHS of the FD [20], and assuming that additional information can be obtained from database catalogue [3].

Verification of the hypotheses is generally done in two ways. First is using pair-wise tuples comparison method where all the tuples are compared in pairs. The time complexity for such a comparison is $\mathbf{O(n^2)}$. Savnik [21] uses this method to verify the proposed FD hypotheses. The second method uses sorting to sort all tuples with respect to the LHS of FD, and then compare the number of distinct LHS and Right-Hand-Side (RHS) values. The time complexity of this method is **O(n log n)** [3, 11, 13]. Although more-general and more-specific concepts reduce the number of possible FD, in the worst case, the time complexity for proposing all the possible FDs is still exponential, $\mathbf{O(2^m-1)}$. The cost of verifying a large number of possible FDs with the data is expensive. In order to make the functional dependency discovery process more applicable to the real world databases, there is a need to further reduce the overall time complexity of the FDs discovery process.

3. Definitions and Notations

In general, attributes are denoted by uppercase letters from the beginning of the alphabet such as *A*, *B*, …. Sets of attributes are denoted by uppercase letters from the end of the alphabet such as …, *X*, *Y*, *Z*. Relations are denoted by lower-case letters such as *r*, *s*, *p*,... and tuples are denoted by the lower-case subscripted letters such as $t_1, t_2, ..., t_n$. A relation schema *R*, denoted by $R(\{A_1, ..., A_n\}, F)$, is a set of attributes $\{A_1, ..., A_n\}$, and a set of constraints *F*. Each attribute A_i has a domain D_i, $1 \le i \le n$, which is the set of all possible attribute values for A_i. A tuple on *R* is a mapping t: $R \rightarrow \cup_i D_i$ with $t(A_i) \in D_i$, $1 \le i \le n$. The values of a tuple are denoted as $\langle t[A_1], ..., t[A_n] \rangle$. Let *r* be a relation over a relation schema *R*, which is denoted as $r(R)$, and let *m* be the total number of attributes in *r* and n be the total number of tuples in *r*. The set $r(R)$ is a set of n tuples $r = \{t_1, \ldots, t_n\}$.

Let *X*, *Y* be the subset of *R*. A functional dependency is a relationship among attribute values. The functional dependency (FD) $X \rightarrow Y$ defines the constraints on the relation *r*, that has to be obeyed by each pair of relation tuples. In order for FD $X \rightarrow Y$ to hold in *r*, if the pair of tuples has the same values on the attributes in *X*, then it has the same values on the attributes in *Y* [1]. Formally that is $\forall t_1, t_2 \in r$ (if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$). If *F* is a set of functional dependencies, then $r \models F$ means that all dependencies of *F* hold in *r*. The set of all functional dependencies holding in *r* is denoted by dep(r). For further detail on functional dependencies, readers can refer to [1], [11], and [24].

4. Data Sample Approach

The Data Sample approach generates a set of FD hypotheses from a data sample and then verifies the hypotheses against the database. Mannila [14] has proposed a similar approach which is based on a bottom-up technique that discover FDs from a data sample. In this paper we propose two algorithms, one is based on *top-down* and the other is based on *bottom-up* techniques, to generate FD hypotheses. The detail differences between our approach and Mannila's approach is documented in the later part of this paper. The goal of our approach is to generate a smaller number of FD hypotheses from data sample before verifying them against the entire database. The outline of the approach is as follows:

Step.1 Generate a set of FD hypotheses from a sample data.

Step.2 Verify all the hypotheses against the database.

We propose two algorithms, ***top-down collective FDs specialising algorithm*** (in short, *collective-FD algorithm*) and ***bottom-up attribute-list algorithm*** (in short, *attribute-list algorithm*) for generating FD hypotheses. The detail descriptions of the two proposed algorithms are shown in the next two sections. The FD hypotheses generated from the step 1 are verified using the database extension in step 2. The sort-based FDs verification technique proposed by Maier [11], which has the time complexity of **O(n log n),** can be used to verify the hypotheses. This technique can also be implemented using SQL statements as shown by Bell [3].

5. Top-down Collective FDs Specialising Approach

The Top-down collective FDs specialising approach uses a top-down tree searching method to search for possible FDs and uses the *collective FDs specialising* technique to propose more-specific FDs. Let $R = \{A, B, C, D, E\}$. The search space for the FDs $X \rightarrow Y$, where $X \subseteq R$, can be visualised using a tree structure. The tree structure in figure 1 shows all the possible LHS of FD $X \rightarrow E$. Every node in the lower level represents a specialised FD. The FDs specialisation technique is commonly used to generate specialised FDs when the proposed FD is failed [3, 5, 20].

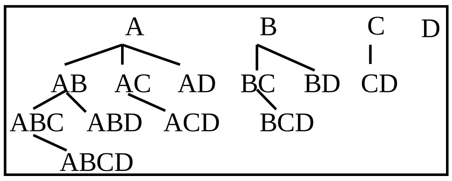

Figure1.A search tree for all possible LHS of FD $X \rightarrow E$

There are two known searching methods for searching specialised FDs. The first method is based on traditional ***depth-first*** search where the children of the invalid node are proposed to be more-specific FDs [5, 20]. Second method is based on ***breath-first*** search where nodes are searched and verified level by level. The children of the invalid node are proposed as the more-specific FDs and being verified at the same level [3]. Using any of the known searching methods to search for more-specific FDs however might generate specialised FDs that need not be verified. In *depth-first* search, more-specific FDs are propose when a FD is determined to be invalid. The proposed more-specific FDs will be verified immediately. For instance, if node *A* is invalid, then all its children nodes (*AB*, *AC*, *AD*) will be proposed as the more-specific FDs for verification. If node *AB* is again invalid, nodes *ABC* and *ABD* would be proposed. If node *C* is verified to be valid in a later stage, the nodes *AC* and *ABC* would in fact be redundant. This could have been prevented if most general attributes were verified first. For instance, if nodes B and C are verified after node A, and if both nodes are verified to be valid, then there is no need to verify nodes AB and AC.

Now consider *breath-first* search. Let *k* represents the different levels of the tree structure from top down. Assuming that all the attributes at level 1 (*k*=1) have failed. Now the search is on the second level where *k*=2. Assuming attribute *AB* failed the test and therefore all its specialised sub nodes (*ABC*, *ABD*) are proposed as the specialised FDs [Bell 95]. However if attributes *AC* and *AD* at level 2 are valid, then the proposed *ABC* and *ABD* attributes would be redundant since we know if $AC \rightarrow E$ and $AD \rightarrow E$, then the supersets of *AC* (e.g., *ABC*) and *AD* (e.g., *ABD*) would also determined attribute E.

5.1 Collective FDs Specialising technique

It was shown in the previous section that the known *depth-first* and *breath-first* search techniques for generating specialised FDs are not optimal with respect to the total number of FDs that need to be verified against the data sample. To further reduce the total number of FDs that need to be verified against the data sample in the FD hypotheses generation process, we propose a FDs specialising technique that avoid proposing specialise FDs that are superset of the valid FDs by delaying the proposal of the specialised FDs towards the end of search at each level. The proposal of the specialised FDs is based on the examination of all the rejected attributes of the same level. Let assume when *k*=2, only BD and CD are valid. Under the known specialisation methods, attribute ABC, ABD, ACD, and BCD would be proposed as the LHS of the specialised FDs (refer to figure 1). To verify these specialised FDs are in fact redundant since we know FDs *BD* → *E* and *CD* → *E* are valid. Using our *collective FDs specialisation* technique, specialised FDs are proposed only at the end of search at each level in order to avoid pre-matured specialisation of FDs. The specialised FDs are propose based on combining the rejected attributes and the attributes will only be combined if they share a common parent and if the combined attributes are not superset of the valid FDs. Those attributes do not have any common parents will be discarded. We named this rule as ***Non Superset Common Parent Combination*** (NSCPC) Rule. Based on the NSCPC rule, the next level of the specialised FDs should only be ABC since AB and BC shared the same parent and it is not a superst of any valid FDs. Node ABD and ACD are discarded since they are the superset of node BD and CD respectively. Node BCD is discarded as well since node BC is a single node . The *Collective FDs Specialising technique* thus prevents redundant nodes ABD and ACD to be proposed as specialised FDs. The Top-Down Collective FDs Specialising Algorithm is shown in figure 2.

```
Algorithm: Top-Down Collective FDs Specialising (R,B)
Input: a relation schema R and the RHS of a FD (B)
Output: the left reduced FDs: X → B that hold in R
1. Let VQUEUE be the verification queue which contains
   LHS of FDs. Let FQUEUE be the queue which
   contains all the LHS of failed FDs. Let FD_
   HYPOTHESES_ LIST be the list of all valid LHS of
   FDs.
   Put all attributes of R exclude B into VQUEUE.
   Set k = 1.
2. For every $A_i$ in VQUEUE
     Verify $A_i \rightarrow B$ with the database;
     If $A_i \rightarrow B$ is not valid Then
       put $A_i$ into FQUEUE;
     Else add $A_i$ into FD_ HYPOTHESES_LIST;
     Delete $A_i$ from VQUEUE;
   EndFor.
   {Applying Non Superset Common Parent Combination
   Rule}
   If FQUEUE contains more than one element Then
     Pairing all attributes in FQUEUE into form $A_uA_v$,
     $u<v$, $uv \in \{1,\ldots, m-1\}$;
   If k = 1 Then put all $A_uA_v$ pairs into VQUEUE;
     Else If $A_u$, $A_v$ share the same parent and $A_uA_v \not\subset E$
     where $E \in$ FD_ HYPOTHESES_LIST, Then put
     $A_uA_v$ into VQUEUE;
   Empty FQUEUE and increment k by 1;
   If VQUEUE is not empty, repeat step 2.
```

Figure 2. Top-Down Collective FDs Specialising Algorithm

Top-Down Collective FDs Specialising Algorithm can be repeatedly executed with different RHS of the FDs to obtain a complete set of valid FDs hold in the data sample.

Example 1. Let *R* = {A, B, C, D, E}and assume B is the key of *r* and other valid FD is AC → E. Execute procedure ***Top-Down Collective FDs Specialising (R, E)***.
VQUEUE = {A, B, C, D}
When *K* = 1
  Since B → E, add it into ***FD_HYPOTHESES_LIST*** and remove *B* from ***VQUEUE***.
  The rest of the attributes are removed from ***VQUEUE*** and put into ***FQUEUE***.
  FQUEUE = {A, C, D}. ***FD_LIST*** = {B → E}.
  Pairing attribute in ***FQUEUE*** into form A_iA_j, $i < j$, $i,j \in \{1,\ldots,m-1\}$ and put them into ***VQUEUE***. ***VQUEUE*** now contains {AC, AD, CD}.
When *K* = 2
  Since AC → E, add it into ***FD_HYPOTHESES_LIST*** and remove AC from ***VQUEUE***.
  FQUEUE = {AD, CD}.***FD_HYPOTHESES_LIST*** = {B → E, AC → E}.
  Combine attributes in ***FQUEUE*** into form A_iA_j, $i < j$, $i,j \in$ {1 to maximum number of attributes in ***FQUEUE***} if A_iA_j share the same parent, and put them into ***VQUEUE***;
  However both attributes AD and CD do not share the same parent. ***VQUEUE*** is empty and therefore stop the procedure.

The result of executing ***Top-Down Collective FDs Specialising*** procedure (with E as the value of **B**) are B → E and AC → E.

5.2 Analysis of the Top-Down Collective FDs Specialising Algorithm

The lower bound of the algorithm is $\boldsymbol{\Omega(m*(2^{m-1}-1)*C*(n \log n))}$ where *m* is the total number of attributes, *n* is the total number of tuples in the data sample, and ***C*** is a constant for the time needed to carry out FDs specialising operation. The $\mathbf{m*(2^{m-1}-1)}$ corresponds to the total number of possible combination of the LHS of the FDs and **(n log n)** corresponds to the time needed for verifying the hypotheses with the sample data.

6. Bottom-Up Disagree Set List Approach

The Bottom-up Disagree-Set-List (DSL) approach analyses the data by comparing all tuples and then constructs disagree attribute lists, which contain attributes sets that have different values in the tuples. Mannila [14] has proposed a similar approach based on the concept of *necessary set*. A series of necessary sets, which contain the minimal left-hand-side of FDs, are produced by Mannila's algorithm after analysing the data. In our approach, the concept of Generalised Disagree Attribute Lists (GDAL), is similar to the concept of necessary set. Our approach is based on the ***comparison matrix concept*** introduced and proposed by Orlowska [15] with modification and improvement. The proposed DSL approach reduces the storage (memory) requirement during computation, compared to the *comparison matrix* approach, by using Generalised Disagree Attribute Lists to store only the most general form of the disagree attributes. Our definitions of Disagree Attribute List and Generalised Disagree Attribute Lists consider attributes that contain identical values in all the tuples.

6.1 Comparison Attribute Lists Construction

The concepts and notations that we use to define the following definitions are similar to the Mannila's paper [14]. A Disagree Set (DS) is defined to be a set of attributes that contains different values in the tuples. DS(t_i, t_j) = {$A \in R \mid t_i[A] \neq t_j[A]$}. A Disagree Set List is defined to be a collection of all Disagree Sets in *r*. DSL(*r*) = {DS(t_i, t_j) | t_i, t_j ∈ r}. A Disagree Attribute List is defined to be a collection of all the disagree sets of a particular attribute *A* in *r*. That is DAL(*r*, *A*) = {DS(t_i, t_j) | t_i, t_j ∈ *r* and *A* ∈ DS(t_i, t_j)}. A Generalised Disagree Attribute List (GDAL) is defined to be a collection of the most general form of the disagree sets of a particular attribute *A* in *r*. That is GDAL(*r*, *A*) = {*X* ∈ DAL(*r*, *A*)\{*A*} | ¯ ∃ Y ∈ DAL(*r*, *A*): Y ⊂ X}.

This definitions are similar but different to the definitions of the ***necessary set*** in Mannila's paper [14]. Two situations have not being considered by Mannila's approach. First is the situation where an attribute *A* might contain identical values in all the tuples. This should be interpreted as the attribute *A* is functionally determined by the remaining attributes. That is $X \rightarrow A$ where $X \subseteq R - A$. Another situation is where all the attributes of any two tuples contain identical values except attribute *A*. This should be interpreted as the attribute *A* is a prime attribute of the relation. That is $A \rightarrow X$ where $X \subseteq R - A$. If any of these conditions occurred, Mannila's approach would produce incorrect result. The example to illustrate this problem is given in [27].

Our approach anticipate these situations and therefore produce accurate result. We have considered these two situations in our definition of DAL(*r*, *A*) and GDAL(*r*, *A*). We allow and anticipate GDAL list to be empty. The GDAL(r, *A*) list is empty if the result of generalising the corresponding DAL(r, *A*) list is empty. This implies that *A* is a key of the relation *r*. If GDAL(*r, A*) becomes empty if the corresponding DAL(r, *A*) list is empty, then a special symbol '+' is inserted into the GDAL(*r, A*) list to indicate all the other attributes of *r* functional determine *A*. The ***Bottom-Up Attribute List Algorithm*** is shown in figure 3.

```
Algorithm: Bottom-Up Attribute List (r)
Input: A relation r
Output: the LHS of the left reduced FDs that hold in r
1. Construct DAL lists
  For all $t_i$, $t_j$ ∈ r
    For all X ∈ DS($t_i$, $t_j$)
      Add DS($t_i$, $t_j$) to DAL(r, X);
2. Construct GDAL lists from DAL lists
  For all the DAL(r, $X_{i=1 \text{ to } m}$)
    If DAL(r, $X_i$) is empty {all the tuples of attribute $X_i$
          contain identical value}
      Then Add '+' to GDAL(r, $X_i$);
    While DAL(r, $X_i$) is not empty
      Delete an element from DAL(r,$X_i$) and assign to A;
      For all B ∈ DAL(r, $X_i$)
        If A f $B_j$ Then remove $B_j$ from DAL(r, $X_i$);
        Else If A ⊖ $B_j$ Then remove $B_j$ from DAL(r, $X_i$)
          and assign $B_j$ to A;
      EndFor.
      Add A into GDAL(r, $X_i$);
    EndWhile.
  EndFor.
3. Drive dep(r) from GDAL lists
  For all the GDAL(r, $X_{i=1 \text{ to } m}$)
    If GDAL(r, $X_i$) contain '+',
      Then all other attributes determine attribute $X_i$;
    If GDAL(r, $X_i$) is empty,
      Then $X_i$ is a Key of the relation r;
    Else Multiply[1] all the elements in GDAL(r, $X_i$);
  EndFor.
```

Figure 3. Bottom-Up Attribute List Algorithm

Example 2. Given a relation *r* and its instances, determine all the valid FDs. We use '*' to represent all the attributes of r.

	A	B	C	D
t1	a1	b1	c1	d1
t2	a2	b1	c1	d1
t3	a3	b2	c2	d1
t4	a4	b3	c3	d1

[1] The technique used in our attribute multiplication process is equivalent to the *transversals hypergraphs* technique described by Mannila [14].

The Disagree Sets that correspond to the relation *r* are: DS = {A, ABC, ABC, ABC, ABC, ABC}. The Disagree Attribute Lists that correspond to different attributes of the relation *r* are: DAL(*r*, *A*) = {A, ABC, ABC, ABC, ABC, ABC}; DAL(*r*, *B*) = {ABC, ABC, ABC, ABC, ABC}; DAL(*r*, *C*) = {ABC, ABC, ABC, ABC, ABC}; DAL(*r*, *D*) = { }. The Generalised Disagree Attribute Lists are: GDAL(*r*, *A*) = { }; GDAL(*r*, *B*) = {AC }; GDAL(*r*, *C*) = {AB}; GDAL(*r*, *D*) = {'+'}. From these GDAL lists, the dependencies set can be computed using the *transversals hypergraphs* technique. The dependencies set that hold in *r* is dep(r) = {A $\rightarrow$ *, C $\rightarrow$ B, B $\rightarrow$ C, * $\rightarrow$ D}.

6.3 Analysis of the Bottom-Up Attribute List Algorithm

To construct Disagree Attribute Lists (DAL) from Disagree Sets (DA) in step 1, all the tuples (*n*) will have to be compared with respect to all the attributes (*m*). That requires $\boldsymbol{m*(n^2 - n)/2}$ comparisons. Under the worst case scenario, all the Disagree Sets might contain all the attributes of *r* (that is all the attributes in *r* are prime). In this case, the DS will have to be stored in all the DAL lists (m). Th lower bound of the step 1 of the algorithm is therefore $\Omega(\boldsymbol{m*m*(n^2 - n)/2}) = \Omega(\boldsymbol{m^2*(n^2 - n)/2})$.

In step 2, GDAL lists are constructed out of the corresponding DAL lists. The worst case scenario of this operation is where all the elements of the DAL lists are comparable and minimal, and therefore all the elements of the DAL lists will become the elements of the corresponding GDAL lists. The maximum number of comparable and minimal elements of the DAL lists is the number of distinct combinations selecting *q* attributes out of the total attribute *m*, that is $\binom{m}{q}*q$, where *q* is the round up of (m/2). If we let D = $\binom{m}{q}*q$, then the lower bound of this operation is therefore $\Omega((\mathbf{D}^2 - \mathbf{D})/2)$ since all the elements of the DAL lists will have to be compared with each other before being added to GDAL lists.

In step 3, all the attributes of the GDAL lists are multiplied to derive dep(r). The time complexity of this operation is O(**D**) since we need to multiple D number of elements from all the GDAL lists. The overall time complexity of the Bottom-up Attribute list algorithm, under the worst case scenario, is therefore $\Omega(\boldsymbol{m^2*(n^2 - n)/2} + (\mathbf{D}^2 - \mathbf{D})/2 + \mathbf{D})$. And it can be simplified to $\Omega(\boldsymbol{m^2*(n^2 - n)/2} + \mathbf{D}*(\mathbf{D} + 1)/2)$.

6.4 Comparing the Collective-FD and Attributes-List Algorithm

Based on the lower bound of the both algorithms it is not surprise to see that the performance of the *top-down collective FDs specialising* algorithm is better than the *bottom-up attribute list algorithm*. In the data sample approach, the performance differences of the both algorithms should not be obvious since they are used on a small percentage of the data to derive the FD hypotheses. The derived FD hypotheses will need to be verified against the database. This process is describes in the next section.

7. Verifying Against The Database

The *top-down collective FDs specialising* and *bottom-up attribute list algorithms* generate FD hypotheses that hold in the data sample. These FD hypotheses will have to be verified against the database. Mannila [14] proposed a similar data sample approach where the *breath-first* search technique is used to verify the FD hypotheses against the database. The proposed *breath-first* search technique is similar to the one described in section 5 which might generate specialised FDs that need not be verified against the database. In our approach, we use the *NSCPC* rule (refer to section 5.1) to eliminate that possibility. The algorithm for verifying (*Verify* algorithm) the derived FD hypotheses based on the *NSCPC* rule is identical to the *top-down collective FDs Specialising* algorithm except the input for the *Verify* algorithm is a set of left reduced FD hypotheses. As for the time complexity of verifying all the hypotheses using the *Verify* algorithm is **O(h*N log N)** where *N* is the total number of tuples in the entire database and *h* is the total number of FD hypotheses that need to be verified.

8. FDs Discovery in Noisy Databases

For many reasons - encoding errors, human errors, measurement errors, etc. - the information in the real world databases are almost always noisy. To detect these errors, we first classify the FD error types and design a technique to discover these errors and compute "almost" FDs from the data. Pawlak [16] has proposed a technique based on rough set theory to compute partial dependency between equivalent relations. The method does not assumed data error and it is not suitable for computing "almost" FDs. The detail analysis of the Pawlak's approach for discovery partial dependency between attributes is described in [27].

8.1 Classification of the FD Error Types

Based on the form of FD $X \rightarrow Y$, we classify and define all the possible FD errors into three types: type X errors, type Y errors, and type XY errors. This classification is based on the fact that errors might occur on either LHS of FD (Type X errors), RHS of FD (Type Y errors), or both LHS & RHS of FD (Type XY errors). Type X errors are errors that occur on the LHS of the FD; type Y errors are errors that occur on the RHS of the FD; type XY errors are errors that occur on both LHS & RHS of the FD. The classification of error types is necessary and important to the discovery of almost FDs from databases.

Definition 1 (Functional Dependencies Error Bound)
Functional Dependencies Error Bound (FD_Err.) is the maximum tolerable error percentage that a FD $X \rightarrow Y$ can fail in the FDs verification process.

Definition 2 (Almost Functional Dependencies)
When the number of failures in verifying a FD $X \rightarrow Y$ is less than or equal the preset ***FD_Err***, that FD is called **Almost Functional Dependency *(Almost FD)***. The set of FDs that almost hold in *r* is denoted by ~dep(*r*).

Definition 3 (FD Discrepancy Set)
Let *r* be a relation over *R*. Let t_i, t_j be any tuple pair in *r* where $1 \leq i, j \leq n$, and n= maximum number of tuple. The ***fd discrepancy set*** is a group of tuples in *r* which contain same *X* values but different *Y* values. That is fdds(t_i, t_j) = $\{X \subseteq r \wedge Y \in r \wedge Y \not\subset X \mid t_i[X] = t_j[X] \wedge t_i[Y] \neq t_j[Y]\}$

Before we define ***Possible Error Tuples***, it is necessary to state that in *almost FDs discovery*, we assume that all the errors consist only a small percentage of the overall database. With this assumption, we can discover tuple errors by detecting majority attribute values in a ***fd discrepancy set***.

Definition 4 (Possible Error Tuples)
In an ***fd discrepancy set***, those tuples that contain the largest number of identical Y values, denoted as Max($t_i[Y] = t_j[Y]$), are considered to be the most likely correct tuples. Whereas the rest of the tuples in the same fd discrepancy set are considered to be ***possible error tuples***. The types of error in these tuples could be either *type X, Y, or XY errors*. The total number of error tuples in an ***fd discrepancy set*** is calculated using formula:

err = | fdds(t_i, t_j) | - Max($t_i[Y] = t_j[Y]$).

The percentage of the overall error tuples with respect to the entire database is calculated using formula:

O_err = 3_{err}/n.

8.2 Almost FDs Discovery

In this section we propose a method for discovering ***type X, Y and XY errors*** by detecting and monitoring ***possible error tuples*** in the ***fd discrepancy sets***. It is important to note that the intention here is not to determine what type of errors have occurred in the database. The intention is to detect possible FD errors. It is not possible to determine the error types by merely examining the data due to unavailability of semantic information. So we are not really concern (at least in this part of the research) with the types of errors in the extension of the database but rather the effect of these errors on the intension of the database.

To compute *almost FDs*, we need to keep track of the number of *possible error tuples* in the *fd discrepancy sets*. We know that all the error types have altered the intension of the database by injecting inequality condition : that is $t_1[X] = t_2[X]$, but $t_1[Y] \neq t_2[Y]$. This makes the FD $X \rightarrow Y$ invalid. To detect such errors, our *Almost FDs Algorithm* first of all sorts the relation *r* with respect to *X* columns and then *Y* columns. After that, it examines and monitors the number of differences in *Y* based on the same *X* values. An error monitoring variable *err* is used to record and accumulate the number of *possible error tuples* in all the *fd discrepancy sets*. Value *err* is computed by finding the differences between the largest number of identical *Y* values and the total number of *Y* values in the *fd discrepancy set*. The overall FDs verification failure percentage (***O_Err***) is computed for each *fd discrepancy set*. The formula for computing *O_Err* is ***O_Err* = err / n**. If *O_Err* exceed *FD_Err*, the algorithm will be terminated and the FD $X \rightarrow Y$ will be treated as an invalid FD. At the end, if the overall FD verification failure percentage (***O_Err)*** does not exceed the preset FD Error Bound (***FD_Err***), that is ***O_Err*** $\leq$ ***FD_Err***, then the FD $X \rightarrow Y$ will be accepted as an ***almost FD***. Otherwise, it will be treated as an invalid FD. The Almost FDs Algorithm is shown in figure 4. The algorithm computes the total number of possible error tuples in the data, calculates the total FD verification failure percentage, and returns the failure percentage when terminate.

```
Algorithm: Almost FDs (r, X → Y)
Input: A relation r and an FD X → Y
Output: true if r satisfies X → Y, otherwise O_Err value
1. Sort the relation r on its X columns and then Y columns
   to bring tuples with equal X-values together.
2. If each set of tuples with equal X-values has equal Y-
   values, Return true;
   Else For each fdds($t_i$, $t_j$),
        err = err + (|fdds($t_i$, $t_j$)| - Max($t_i[Y] = t_j[Y]$));
```

```
    EndFor.
  O_Err. = err / n;
  If O_Err. > FD_Err Then Return (O_Err);
3. Return (O_Err).
```

Figure 4. Almost FDs Algorithm

Example 3. For simplicity reason, we use a small relation with only two attributes and thirty tuples. Given a relation *r* and its instances, use algorithm ***Almost FDs*** to determine whether $A \rightarrow B$ is valid. Assuming the ***FD_Err.*** is 15%. The result of executing step 1 of the algorithm on relation *r* is shown in table 1.

In step 2, two fdds(t_i, t_j) sets were detected in *r*. The first fdds(t_i, t_j) set contains tuple *t1* to *t12*. *err = err* + (12 - 10) where 12 is the total number of *Y* values and 10 is the largest number of identical *Y* value in that set. ***err*** =2 at the end of examining first fdds(t_i, t_j). *O_Err = err/n* = 2/30 = 0.06. This is lesser than *FD_Err* (.15), so the algorithm proceed to examine the second fdds(t_i, t_j) set. Second fdds(t_i, t_j) set contains tuple *t13* to *t20*. *err = err* + (8 - 7) where 8 is the total number of *Y* values in set *X* = 2 and 7 is the largest number of identical *Y* value in that set. Now ***err*** =3 at the end of examining second fdds(t_i, t_j) set. ***O_Err*** = 3/30 = 0.1. This value is still smaller than the *FD_Err*, so the algorithm proceed to step 3. In step 3, *O_Err is returned*. The *O_Err* of FD $A \rightarrow B$ is 10%, which is lower than the preset error bound 15%, therefore $A \rightarrow B$ is accepted as an ***almost FD***.

	A	B		A	B		A	B
t1	1	3	t11	1	4	t21	9	15
t2	1	4	t12	1	5	t22	10	15
t3	1	4	t13	2	6	t23	10	15
t4	1	4	t14	2	8	t24	10	15
t5	1	4	t15	2	8	t25	10	15
t6	1	4	t16	2	8	t26	10	15
t7	1	4	t17	2	8	t27	10	15
t8	1	4	t18	2	8	t28	11	16
t9	1	4	t19	2	8	t29	11	16
t10	1	4	t20	2	8	t30	11	16

Table 1. Sorted table with respect to attribute A and B

8.3 Analysis of the Almost FDs Algorithm

The lower bound for sorting relation *r* has been proven to be **O(n log n)** [10]. After sorting the relation, the table must be scanned to detect *fd discrepancy set* and to compute *err*. In the worst case, this operation requires *n* comparisons since *Y* values are sorted and grouped by *X* values. The total time complexity for the Almost FDs algorithm, in the worst case scenario, is therefore **Ω(n + n log n)**. In most cases, for invalid FDs verification, the algorithm does not have to scan the entire table since *O_Err* will exceed *FD_Err* and that will terminate the algorithm.

8.4 Almost FDs Discovery in Top-down Collective FDs Specialising Approach

Implementing Almost FDs Discovery in the top-down algorithm is rather straight forward. To detect and calculate errors in the sample data, we need to replace the conventional *FDs verification algorithm* with the ***Almost FDs Discovery Algorithm***. However the **n** variable in the formula for computing the overall failure percentage (***O_Err = err / n***) is correspond to the entire database rather the sample size. This is due to the fact that the errors detected in the sample data are the errors of the entire database even though they are discovered in the sample data. The almost FDs that are within the tolerable error bound are proposed to be FD hypotheses. In the step 2 of the data sample approach, the ***Almost FDs Algorithm*** is used to verify the hypotheses. The FDs that have FD verification failure percentage (***O_Err)*** lower or equal to the tolerable FD Error Bound (***FD_Err***) will be accepted as *almost FDs*.

8.5 Almost FDs Discovery in Bottom-up Attribute List Approach

Implementing Almost FDs Discovery in the bottom-up algorithm is not as straight forward as in *top-down* approach since sorting is not used in computing FDs set. We know that the *bottom-up* approach computes valid FDs by identifying attributes that contain different values in the tuples. In this process, we can also search for attributes that contain identical values in the tuples and construct a series of negative sets which contains invalid FDs. These invalid FDs represent the Functional Independencies (FIs) of *r*. The concept of FI has been introduced by Janas [8] to mirror functional dependencies. A definition of the Functional Independency is shown in definition 5.

Definition 5 (Functional Independency)
A relation *r* satisfies ***Functional Independency*** (**FI**) $X \not\rightarrow Y$ ($r \models X \not\rightarrow Y$), if there exist tuples t_i, t_j of *r* with $t_i[X] = t_j[X]$ and $t_i[Y] \neq t_j[Y]$.

Theorem 1. Let r be a relation over a relation schema *R*, and X & $Y \subseteq r$. If *r* satisfies FI $X \not\rightarrow Y$, then *r* must also satisfies all $x \not\rightarrow Y$ where *x* is a subset of *X*. That is If ($r \models X \not\rightarrow Y$), then $\forall x \subseteq X$ ($r \models x \not\rightarrow Y$).

Proof. Let $AB \not\rightarrow C$ where $A,B,C \in r$. The theorem 1 would be FALSE if either *A* functional determines *C* ($A \rightarrow$

C) or *B* functional determines *C* ($B \rightarrow C$). If either $A \rightarrow C$ or $B \rightarrow C$ is true, then *AB* would have functional determined *C* ($AB \rightarrow C$). This contradicts $AB \not\rightarrow C$.

In theorem 1, the FI $X \not\rightarrow Y$ is actually a ***more specific*** form of FIs $x \not\rightarrow$ Y. This leads us to the next definition.

Definition 6 (More Specific Functional Independency) A functional independency FI $X' \not\rightarrow Y$ is more specific than FI $X \not\rightarrow Y$ iff $X \subset X'$.

We now need to define a list (negative list) that will hold all the FIs of *r* based on Disagree Set (DS) and Agree Set (AS). The definition of An Agree Set is as follows. An Agree Set (AS) is defined to be a set of attributes that contain identical values in the tuples. That is $\mathrm{AS}(t_i, t_j) = \{A \in R \mid t_i[A] = t_j[A]\}$. A negative list is defined to be a list of attributes set that represent FIs of a particular attribute *A* in *r*. That is $\mathrm{neg}(r, A) = \{\mathrm{AS}(t_i, t_j) \backslash \{A\} \mid t_i, t_j \in r \text{ and } A \in \mathrm{DS}(t_i, t_j)\}$.

To minimise the number of invalid FDs that need to be verified, we would want only the ***most specific*** form of the FIs (refer to theorem 1). A Most Specific negative list is defined to be a list of ***most specific*** form of the FIs of a particular attribute *A* in *r*. That is $\mathrm{MSneg}(r, A) = \{X \in \mathrm{neg}(r, A) \mid \neg \exists\, Y \in \mathrm{neg}(r, A)\text{: } Y \supset X\}$.

We extend the ***bottom-up attribute list algorithm*** to incorporate the construction of MSneg lists and computation of the almost FDs (refer to figure 5).

Algorithm: ***Bottom-Up Attribute List with almost FDs discovery*** (*r*)

```
Input: A relation $r$
Output: dep($r$) and ~dep($r$)
1. Construct GDAL & MSneg lists
   For all $t_i, t_j \in r$
     If $t_i[A] \neq t_j[A]$
       For all $X \in \mathrm{DS}(t_i, t_j)$
         Add DS($t_i, t_j$) to DAL($r, X$);
       EndFor.
     Else For all $Y \in \mathrm{AS}(t_i, t_j)$
         Add AS($t_i, t_j$) to neg($r, X$);
       EndFor.
   EndFor.
2. Construct GDAL lists{Same as step 2 of the Bottom-Up Attribute List algorithm}
3. Construct MSneg lists from neg lists
   For all the neg($r, X_{i=1 \text{ to } m}$)
     While neg($r, X_i$) is not empty
       Remove an element from neg($r, X_i$) and assign to $A$;
       For all $B \in$ neg($r, X_i$)
         If $A$ f $B_j$ Then remove $B_j$ from neg($r, X_i$);
           Else If A ⊖ $B_j$ Then remove $B_j$ from neg($r, X_i$)
             and assign $B_j$ to $A$;
       EndFor.
       Add $A$ into MSneg($r, X_i$);
     EndWhile.
4. Derive dep($r$) from GDAL lists {Same as step 3 of the Attribute List algorithm}
5. Compute ~dep($r$) from MSneg lists
   For all $Y \in r$
     Pop an element from MSneg($r, Y$) and assign to $X$
     While X is not empty
       Verify $X \rightarrow Y$ using Almost FDs Discovery Algorithm;
       If NOT fail
         Put $X \rightarrow Y$ into an Almost FD list;
         If $X \rightarrow Y$ can be generalised
           Insert the next level generalised elements into MSneg($r, Y$);
       Pop an element from MSneg($r, Y$) and assign to $X$.
     EndFor.
```

Figure 5. Bottom-Up Attribute List with Almost FDs Discovery Algorithm

In the step 5 of the algorithm, ~dep(*r*) is computed based on verifying the invalid FDs contain in MSneg(*r*, *Y*) lists. Each of the elements in the MSneg(*r*, *Y*) list is verified using the *almost FDs algorithm*. If the invalid FD $X \rightarrow Y$ did not satisfy the pre-specified FD Error Bound percentage, it will be treated as a FI and will be removed from the MSneg(*r*, *Y*) list. From theorem 1 we know that all its *more general* FIs would also be true. If the error percentage of the invalid FD $X \rightarrow Y$ satisfies the FD Error Bound percentage, the FD will be put into an *Almost FD list*. If the FD $X \rightarrow Y$ is not in the *most general* form, that is *X* is not a single attribute, all its next level of *more general* FDs will be generated and put into the MSneg(*r*, *Y*) list for verification. This process will continue until all the MSneg(*r*, *Y*) lists are empty. The last task is to remove any *more specific* FDs from the *Almost FD list*. The result is a set of Almost FDs that hold in *r*, ~dep(*r*). Reader can refer to [27] for an illustrated example, and the detail algorithm analysis.

9. Concluding Remarks and Future Work

In this paper, we have proposed a data sample approach for discovering functional dependencies from data that further reduces the total number of FD hypotheses that need to be verified. The *collective-FD algorithm* reduces the number of specialised FDs from being generated, whereas the *attribute-list algorithm* enables accurate FD hypotheses to be discovered. The main contribution of this paper is in the proposal of the *partial FDs* discovery method. We have incorporated error detection into the proposed data sample approach. The result is an *error-tolerant* FD discovery approach that is more applicable to large real world databases. We are in the process of carrying out an empirical study on the proposed approach using real world databases and we are currently working on parallelising the FD discovery approach by exploring parallel discovery and computation of FDs and FD hypotheses from data.

Acknowledgment: We will like to express our utmost gratitude to *Prof. Maria Orlowska* for her comments on the material of this paper.

References

[1] Paolo Atzeni, Valeria De Antonellis, ***Relational Database Theory***, The Benjamin/ Cummings Publishing, 1993.

[2] Martin Andersson. ***Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering***. ER'94.

[3] Siegfried Bell, Peter Brockhausen, ***Discovery of Data Dependencies in Relational Databases***, University of Dortmund, Computer Science Department, LS-8 Report 14, Apr. 1995.

[4] Roger H.L. Chiang, Terrence M. Barron, Veda C. Storey. ***Reverse engineering of relational databases: Extraction of an EER model from a relational database***. Data & Knowledge Engineering 12 (1994) 107-142.

[5] Peter A. Flach, ***Inductive characterisation of database relations***, Methodologies for Intelligent Systems, 5, pp. 371-378, 1990.

[6] J.L. Hainaut, M. Chandelon, C. Tonneau, M. Joris. ***Contribution to a Theory of Database Reverse Engineering***. Proc. of the IEEE Working Conf. On Reverse Engineering, Baltimore, May 1993, IEEE CSP, 1993.

[7] J.L. Hainaut, V. Englebert, J. Henrard, J.M. Hick, D. Roland**.** ***Requirements for Information System Reverse Engineering Support***. Proc. of the 2nd Working Conference on Reverse Enginering, Toronto, Ontario, Canada, July 1995.

[8] J.M. Janas, ***On functional independencies***. Foundations of Software Technology and Theoretical Computer Science. Springer, Lecture Notes in Computer Science 338, 1988. pp487-508.

[9] Martti Kantola, Heikki Mannila, Kari-Jouko Raiha, Harri Siirtola, ***Discovering Functional and Inclusion Dependencies in Relational Databases***, International Journal of Intelligent Systems, Vol. 7, 591-607 (1992).

[10] Donald E. Knuth, ***The Art of Computer Programming***, Addison-Wesley, 1973.

[11] David Maier, ***The Theory of Relational Databases***, Computer Science Press, 1983.

[12] Heikki Mannila, Kari-Jouko Raiha, ***Dependency Inference***, Proc. of the 13th VLDB Conference, 1987.

[13] Heikki Mannila, ***On the complexity of inferring functional dependencies***, Discrete Applied Mathematics, 40 (1992) 237-243.

[14] Heikki Mannila, Kari-Jouko Raiha, ***Algorithms for inferring functional dependencies from relations***, Data & Knowledge Engineering 12 (1994) 83-99.

[15] Maria Orlowska, ***Dependencies from Data - Computational Considerations***, IS Research Colloquium, 21 May 1996, Computer Science Department, University of Queensland.

[16] Zdzislaw Pawlak, ***Rough Sets Theoretical Aspects of Reasoning about Data***, Kluwer Academic Publishers, 1991.

[17] J.M. Petit, F. Toumani, J.F. Boulicaut, J. Kouloumdjian. ***Using Queries to Improve Database Reverse Engineering***. ER'94.

[18] William J. Premerlani, Michael R. Blaha. ***An Approach for Reverse Engineering of Relational Databases***. Communication of ACM, May 94/Vol.37, No.5.

[19] William J. Premerlani, Michael R. Blaha. ***An Approach for Reverse Engineering of Relational Databases***. Communication of ACM, May 94/Vol.37, No.5.

[20] Jeffrey C. Schlilmmer, ***Using Learning Dependencies to Automatically Construct Sufficient and Sensible Editing Views***, Knowledge Discovery in Databases Workshop 1993.

[21] Iztok Savnikm, Peter A. Flach, ***Bottom-up Induction of Functional Dependencies from Relations***, Knowledge Discovery in Databases Workshop 1993.

[22] Oreste Signore, Mario Loffredo, Mauro Gregori, Marco Cima. ***Reconstruction of ER Schema from Database Applications: a Cognitive Approach***. ER '94.

[23] Gregory Piatetsky-Shapiro, Knowledge Discovery in Databases, Proc. Of AAAI-93 Workshop on Knowledge Discovery in Databases.

[24] Bernhard Thalheim, ***Dependencies in Relational Databases***, B. G. Teubner Verlagsgesellschaft mbH, Stuttgart Leipzig 1991.

[25] Mark W.W. Vermeer, Peter M.G. Apers. ***Reverse engineering of relational database applications***. OOER'95.

[26] Peter H. Aiken. ***Data Reverse Engineering***. McGraw-Hill, 1996.

[27] Wie Ming Lim, John Harrison. ***Functional Dependencies Discovery***. Technical Report No. TR-408, 1997, School of Information Technology, The University of Queensland.

[28] John V. Harrison, Anthony Berglas, Ian D. Peake. ***Legacy 4GL Application Migration Via Knowledge-Based Software Engineering Technology: A Case Study***. To be published in the proceeding of the Australian Software Engineering Conference, ASWEC'97, Sydney, 28 Sep. - 3 Oct. 97.