Fast Detection of Functional Dependencies in XML Data

Hang Shi[1], Toshiyuki Amagasa[2], and Hiroyuki Kitagawa[2]

[1] Department of Computer Science,
Graduate School of Systems and Information Engineering
[2] Center for Computational Sciences,
University of Tsukuba,
1–1–1 Tennodai, Tsukuba 305–8573, Japan
eric_shi@kde.cs.tsukuba.ac.jp, {amagasa,kitagawa}@cs.tsukuba.ac.jp

Abstract. In this paper we discuss a scheme for efficiently detecting functional dependency in XML data (XFD). The ability to detect XFD in XML data is useful in many real-life applications, such as XML schema design, relational schema design based on XML data, and redundancy detection in XML data. However, detection of XFD is an expensive task, and an efficient algorithm is essential in order to deal with large XML data collection. For this reason, we propose an efficient way to detect XFD in XML data. We assume that XML data being processed are represented as hierarchically organized relational tables. Given such data, we attempt to detect XFDs existing within and among the tables. Our basic idea is to adopt the PipeSort algorithm, which has been successfully used in OLAP, to detect XFDs within a table. We modify the basic PipeSort algorithm by incorporating a pruning mechanism by taking the features of XFDs into account, thereby making the whole process even faster. Having obtained a set of XFDs existing in tables, we attempt to detect XFDs existing among tables. In this process, we also make use of the features of XFDs for pruning. We show the feasibility of our scheme by some experiments.

1 Introduction

A functional dependency (FD) [1] is a kind of constraints between two sets of attributes in a relation from a database; a set of attributes, called determinant attributes, functionally determine another (dependent) attribute, if and only if each value in the determinant attributes is associated with precisely one value in the determinant attribute. FD plays a crucial role in defining the normal forms (NFs) in relational databases, by which we can capture and/or control the redundancy existing in a relational database, thereby making it possible to keep the consistency and integrity of the database.

In an analogy to the FD in relational database, functional dependencies in XML data (XFD) have been proposed [2] owing to the recent rapid diffusion of XML [3]. The objective of XFD is to describe dependency relations among XML

M.L. Lee et al. (Eds.): XSym 2010, LNCS 6309, pp. 113–127, 2010.

elements, thereby defining the concept of normal forms in XML data (XNFs). It is worth to mention that, unlike relational databases, XML data can be flexible and hierarchical. For this reason, defining XFD (and XNF as well) is not a trivial task, and there have been many proposals depending on the way how data items (in XML data) and the keys are defined, consequently. Specifically, XFDs that have been proposed so far can roughly be categorized into the following two approaches: path-based approaches [4] and tree-tuple based approaches [2,5,6]. In the former, target elements are identified in terms of path expressions, whereas, in the latter approach, tree tuples are extracted for the subsequent functional dependency description.

Those having said, in real-life applications, XFDs may not be specified in many cases. However, there is a demand for detecting XFDs from a given set of XML data for several reasons: 1) one may wish to detect redundancies existing in the XML data for saving storage space and/or improving updatability; 2) one may want to enhance the existing XML schema, written in DTD, W3C XML Schema, or RELAX NG, by incorporating XFDs; and 3) one may want to refer to XFDs extracted from an existing XML data collection as a hint to design a new XML schema. When doing so, efficiency on the detection process is another important concern due to the growing volume of XML data, that is, give a large XML data, fast detection of XFDs is quite important.

However, most of the existing studies on XFDs focus on the definition of functional dependencies in XML data, and the efficiency in the detection is out of their scopes. Yu and Jagadish [5,6] proposed the generalized tree tuple based XML functional dependency and algorithms for detecting XML data redundancies in XML data. In their approach, XML data are converted into a set of hierarchically linked relational tables. Given such tables, XFDs are detected if the grouping over the determinant attributes and the grouping over determinant and dependent attributes result in the same group. This fact implicates that the more the attributes and/or the relations exist, the more we need to perform grouping operations, which leads to the poor performance against a large XML data.

To cope with this problem, we adopt the PipeSort algorithm [7], which has been successfully used in OLAP [8], to detect XFDs within a table. We modify the basic PipeSort algorithm by incorporating a pruning mechanism by taking the features of XFD into account, thereby making the whole process even faster. Having obtained a set of XFDs existing in tables, we attempt to detect XFDs existing among tables. In this process, we also make use of the features of XFD for pruning. We show the feasibility of our scheme by some experiments.

The rest of this paper is organized as follows. Section 2 introduces the definition of XFD and the detection algorithm proposed by Yu and Jagadish [5,6]. Section 3 introduces the proposed scheme, followed by experimental evaluation in Section 4. Section 5 briefly reviews the related work. And Section 6 concludes this paper.

2 Preliminaries

In this section, we briefly review the definitions of XFDs, XML data redundancies, and detection algorithm by Yu and Jagadish [5,6]. The reason why we are based on this approach is that it is powerful enough to capture XFDs proposed by other researches.

2.1 Generalized Tree Tuple Based XML Functional Dependency

At first, we formally define schema, XML data, path expression, and equality of XML data fragments before defining XML functional dependency (XFD). Most of the definitions are based on [5,6], but are simplified for brevity.

Definition 1 (Schema). *A schema is defined to be* $S = \langle E, T, r \rangle$*, where:*

- E *is a finite set of element labels.*
- T *is a finite set of element types. Each* $e \in E$ *is associated with a* $\tau \in T$ *(written as* $(e : \tau)$*) of the form:*

$$\tau ::= \mathsf{str} | \mathsf{int} | \mathsf{float} | \mathsf{SetOf}\tau | \mathsf{Rcd}[e_1 : \tau_1, \ldots, e_n : \tau_n] | \mathsf{Choice}[e_1 : \tau_1, \ldots, e_n : \tau_n]$$

- $r \in E$ *is called the element type of the root. We assume that* r *does not appear in* T *for any* $\tau \in E$*.*

This definition corresponds to the core part of W3C XML Schema [9], i.e., str, int, and float are simple types, and SetOf, Rcd and Choice are complex types. Specifically, SetOf corresponds to a complex type whose `maxOccurs` value is greater than 1. Rcd and Choice correspond to `sequence` and `choice` complex types, respectively. Figure 1 shows a schema used as a running example.

A schema element e_k is identified by a path expression, $path(e_k) = /e_1/e_2/\ldots/e_k$, where $e_1 = r$ and e_i is a label in $\mathcal{V}$. A schema element e_k is a set of elements, it is called *repeatable*.

An XML data instance is then modeled as a rooted labeled tree.

```
university: Rcd
  department: SetOf Rcd
    name: str
    course: SetOf Rcd
      cid: str
      cname: str
      student: SetOf Rcd
        sid: str
        sname: str
        gender: str
        score: str
        addr: SetOf Rcd
```

Fig. 1. An example of schema

Definition 2 (XML tree). *An XML tree is defined to be* $T = \langle N, P, V, r \rangle$*, where:*

- N *is a set of labeled date nodes.*
- $P \subset N \times N$ *is a set of parent-child edges. For each* $n \in N$ *except for* r*, there exists an exactly one* $n' \in N$ *such that* (n', n) *and* $n \neq n'$*.* n' *is called* n*'s parent.*

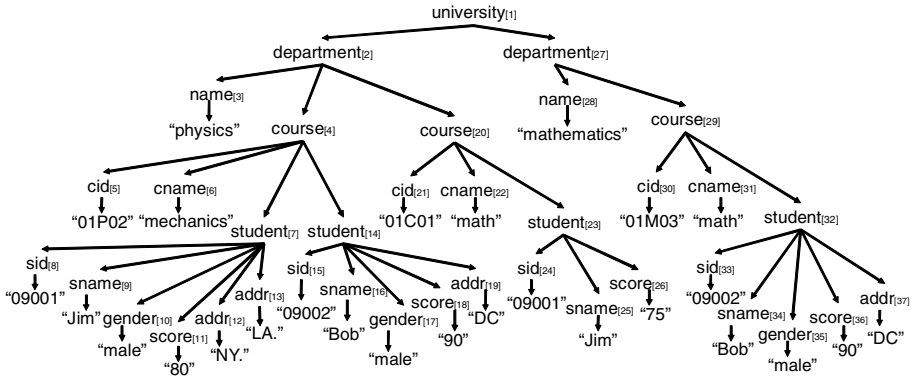

Fig. 2. An example of XML data

- *V is a set of value assignments. For each leaf node $v \in N$, there is exactly one $(v, s) \in V$ that assigns v to the simple value s.*
- *$r \in N$ is the distinguished root node.*

In an analogy to schema elements, data nodes can be addressed using a path expression. A data element is said to be *repeatable*, if its corresponding schema element is repeatable. Figure 2 shows an example of XML data that conforms to the schema in Figure 1. Note that each node is annotated by its node ID, which is the pre-order number of the node.

In Definition 3, we discuss the equality between two XML nodes.

Definition 3 (XML data equality). *Given two nodes n_1 in $T_1 = \langle N_1, P_1, V_1, r_1, \rangle$ and n_2 in $T_2 = \langle N_2, P_2, V_2, r_2, \rangle$, they are* node-value equal*, denoted as $n_1 =_{nv} n_2$, iff:*

- *n_1 and n_2 exist and have the same label.*
- *There is a one-to-one mapping between the child nodes of n_1 and n_2, such that $n_1' =_{nv} n_2'$, where n_1' and n_2' are child nodes of n_1 and n_2, respectively.*
- *$(n_1, s) \in V_1$ iff $(n_2, s) \in V_2$, where s is a simple value.*

This definition intuitively states that two data nodes are node-value equal iff the subtrees rooted at the two nodes are identical without considering the order among sibling nodes. For example, `student` nodes 14 and 32 in Figure 2 are node-value equal.

2.2 XML Functional Dependency (XFD)

We are now ready to define XML functional dependency (XFD).

Definition 4 (Generalized tree tuple). *A generalized tree tuple (GTT) of data tree $T = \langle N, P, V, n_r \rangle$, with regard to a particular data node n_p (pivot node), is a tree $t^T_{n_p} = \langle N^t, P^t, V^t, n_r \rangle$, where:*

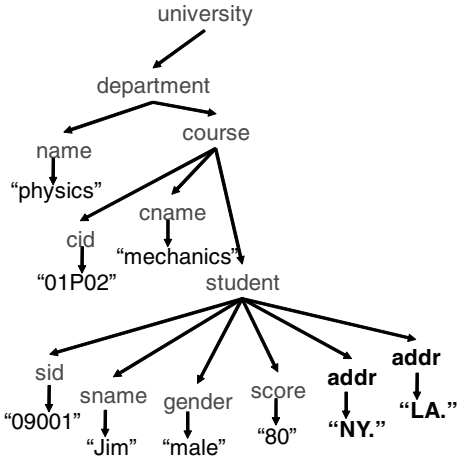

Fig. 3. An example of generalized tree tuple with student as the pivot node

- $N^t \subseteq N$ *such that* $n_p \in N^t$*;*
- $P^t \subseteq P$*;*
- $V^t \subseteq V$*;*
- $n \in N^t$ *iff 1)* n *is a descendant or an ancestor of* n_p*, or 2)* n *is a non-repeatable direct descendant of an ancestor of* n_p*;*
- $(n_1, n_2) \in P^t$ *iff* $(n_1, n_2) \in P$ *and both* n_1 *and* n_2 *are in* N^t*; and*
- $(n, s) \in V^t$ *iff* $(n, s) \in V$ *and* $n \in N^t$*.*

Intuitively, a GTT is a data tree constructed around a pivot data node (n_p), that is, the separation is done only at subtree rooted above the pivot node. So, both the entire subtree rooted at n_p and all ancestors of n_p are kept. Figure 3 illustrates an example of generalized tree tuple with student as the pivot node. Note that both addr nodes of the student are preserved in GTT. A *tuple class* C_p is the set of all GTTs, whose pivot nodes share the same path p (called *pivot path*).

Definition 5 (XML functional dependency (XFD)). *An XFD is a triple* $\langle C_p, LHS, RHS \rangle$*, written as* $LHS \rightarrow RHS$ *w.r.t.* C_p*, where* C_p *is a tuple class,* LHS *is a set of paths* $(P_{l_i}, i = \{1, 2, \ldots, n\})$ *relative to* p*, and* RHS *is a path* (P_r) *relative to* p*. An XFD holds on a data tree* T*, iff any two GTTs* $t_1, t_2 \in C_p$ *satisfy the following conditions:*

- $\exists i \in \{1, 2, \ldots, n\}, t_1.P_{l_i} = \perp$ *or* $t_2.P_{l_i} = \perp$*, or*
- *If* $\forall i \in \{1, 2, \ldots, n\}, t_1.P_{l_i} =_{pv} t_2.P_{l_i}$*, then* $t_1.P_r \neq \perp$*,* $t_2.P_r \neq \perp$*, and* $t_1.P_r =_{pv} t_2.P_r$*.*

where $t.P$ *denotes the set of node(s) identified by following path* P *from the pivot node of* t*, and* $\perp$ *denotes a null value.*

An XFD states that, for any two GTTs t1, t2 in C_p, if they share the same LHS, they have the same RHS.

Examples of XFD. The followings are examples of XML data redundancies, that can be found in the previous example, and how they are captured in terms of XFD.

1. Any two students with the same `sid`, e.g., nodes 14 and 32, must have the same `sname`.
$$\{sid\} \rightarrow sname \text{ w.r.t. } C_{student}$$
2. Any two students with the same `sid` must have the same set of addresses.
$$\{sid\} \rightarrow addr \text{ w.r.t. } C_{student}$$
3. For a student, his/her `score` is determined by the *course* that he/she is enrolled.
$$\{cid, sid\} \rightarrow score \text{ w.r.t. } C_{course}$$

2.3 Detecting XML Data Redundancies

Having defined XFD, the next problem is how to detect XML data redundancies.

XML data representation. First, we need an efficient representation of XML data. In [5,6], they represent XML as a set of hierarchically-linked relational tables. Figure 4 depicts a storage example that corresponds to Figure 2. We are not able to explain the detail due to the space limitation, but the main idea is to represent *essential tuple classes*, which are tuple classes with pivot paths that correspond to repeatable schema elements, as a set of relational tables, and represent their hierarchical relationships using foreign key references. In the example, `department`, `course`, `student`, and `addr` elements are represented as relational tables, because they are repeatable (see Figure 1)[1]. All non-repeatable nodes are represented as relational attributes, and parent-child relationships are represented as foreign key references (`parent` attribute). A GTT can be retrieved by joining those tables using the keys.

Note that, although we are based on the relational XML storage in this paper, the proposed scheme is not necessarily limited to the storage structure; it can easily be applied to other storage schemes, such as native XML storage, if we extract necessary information from the stored XML data.

XFD detection strategy. Because GTTs are stored in separated relational tables, the process to detect XFD becomes complicated, i.e., we need to find both *intra-relation* and *inter-relation* functional dependencies. Specifically, at each relation, we perform the following process:

1. Detect intra-relation FDs.
2. Detect inter-relation FDs involving descendant relations.
3. Generate candidate inter-relation FDs for subsequent detection at the parent level.

[1] Note that `university` is the exception due to being the root.

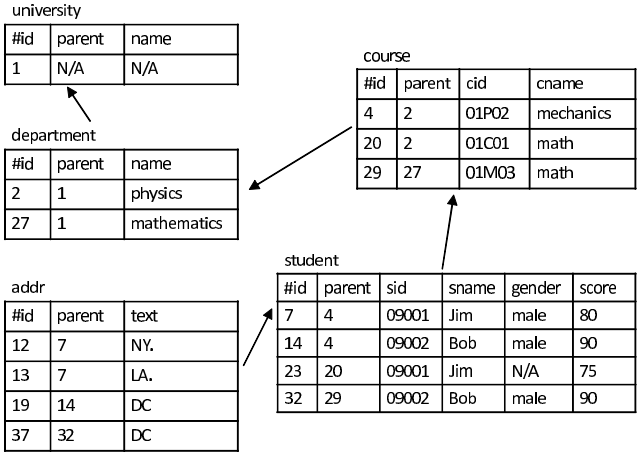

Fig. 4. A relational representation of XML

Intra-relation FD detection. The algorithm for detecting intra-relation FD is based on well-known partition-based algorithms, e.g, TANE [10]. An*attribute partition* of a set of attributes $X(\Pi_X)$ is a set of partition groups, where each group contains all tuples sharing the same X value. For example, in $R_{student}$, $\Pi_{sid,sname} = \{\{7, 23\}, \{14, 32\}\}$. An important fact here is that an XFD ($LHS \rightarrow RHS$ w.r.t. C_p) holds iff $\Pi_{LHS \cup RHS} = \Pi_{LHS}$ in R_p. In fact, we can easily observe that $\Pi_{sid,sname} = \Pi_{sid}$. So, we conclude that an XFD, $\{sid\} \rightarrow sname$ w.r.t. $C_student$ holds.

Utilizing this feature, we can detect intra-relation FD in a bottom-up manner (Figure 5). I, N, G, S stands for SID, SName, Gender and Score, respectively. It illustrates a lattice structure over the student table shown in Figure 4. Each node represents an attribute set, and each edge (X, Y) corresponds to an XFD. Let $Y = X \cup \{A\}$, then edge (X, Y) corresponds to an XFD $X \rightarrow A$ w.r.t. C_p . Starting from the smallest attribute sets, the algorithm finds intra-relation FDs by following the lattice. It is important to mention that, having detected an XFD, we may be able to prune away some edges without checking partitions. In the above example, from (I, IN) (thick edge), we can infer such XFDs that can be derived by adding the same set of attributes to both LHS and RHS, i.e., (IG, ING), (IS, INS), and $(IGS, INGS)$ (dotted edges) are pruned away, and there is no need to check them, consequently.

Inter-relation FD detection. Having detected intra-relation FDs, the next task is to detect inter-relation XFDs by using the results of intra-relation FDs. The key concept is *candidate inter-relation FD generation*. Let P' be a parent relation of P, an XFD is a candidate inter-relation FD if it satisfies the following properties: 1) for $LHS \cup X \rightarrow RHS$ w.r.t. C_P to hold for any attribute set X in relation P', $LHS \cup \{./parent\} \rightarrow RHS$ w.r.t. C_P must hold; and 2) for

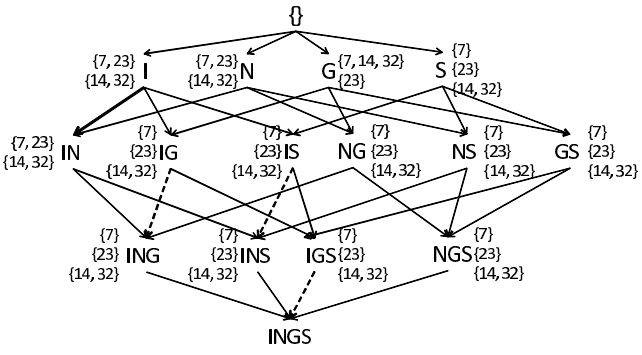

Fig. 5. Intra-relation FD detection ($R_{student}$)

$LHS \cup X \rightarrow RHS$ *w.r.t.* C_P to be a non-trivial XFD for any attribute set X in relation P', $LHS \rightarrow RHS$ w.r.t. C_P must *not* hold.

Taking R_{course} for example, by adding `parent` attribute, we can check non-trivial candidates of inter-relation FD by $\Pi_{parent,sid} = \Pi_{parent,sid,score}$. As a result, two non-trivial candidates of inter-relation FDs can be found:

$$\{cid, sid\} \rightarrow score \text{ w.r.t. } C_{course}$$
$$\{cname, sid\} \rightarrow score \text{ w.r.t. } C_{course}$$

2.4 Problems

Although the above approach successfully detects XFDs, it contains several problems. First, for intra-relation FD detection, it requires a large number of grouping operations according to the size of attribute lattice, which leads to the poor performance against a large XML data. Second, for inter-relation FD detection, more aggressive pruning of XFDs can be considered by taking into account the axioms of FDs.

3 The Proposed Scheme

As defined above, the procedure to detect XFDs is consisting of two subtasks, namely, intra- and inter-relation FD discovery. Actually, each subtask potentially contains unnecessary grouping or comparison operations. So, we attempt to eliminate those inefficiencies by introducing an optimized algorithm taking the feature of functional dependency.

3.1 PipeSort for Intra-relation FD Detection

Our idea is to exploit PipeSort [7] for intra-relation FD detection. PipeSort was originally proposed for optimizing the computation of multiple GROUP-BYs in

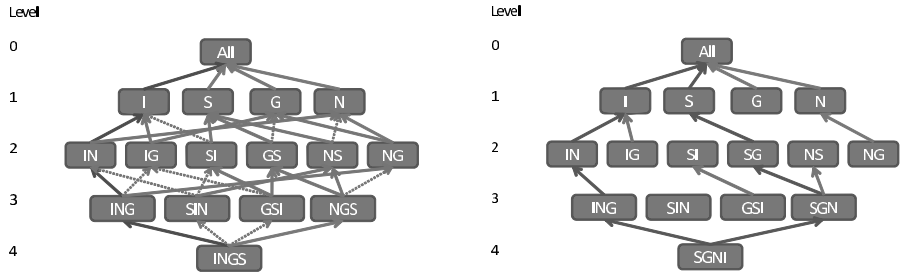

Fig. 6. PipeSort applied to $R_{student}$ (before (left) and after pruning (right))

OLAP systems. The computation of attribute partitions is essentially the same with GROUP BY. So, we try to minimize the number of sort operations.

The PipeSort algorithm is based on sort operation and incorporates the cache result, amortize scan, and share sorts to reduce the disk I/O cost. Suppose that there is a relational table with four attributes, A, B, C, and D, and sort this table using the all attributes ($ABCD$) as the sort key in this order. As a consequence, not only the sorted relation $ABCD$, but also ABC, AB, and A can be obtained by just disregarding attributes with lower-priorities. This is called a *pipeline*. An important notice here is it does not require any additional sort operation.

When applying PipeSort to intra-relation FD detection, we firstly construct an attribute lattice. Unlike the original intra-relation FD detection algorithm in which the process proceeds in a bottom-up manner, our approach proceeds in a top-down fashion. We decide the priority of all attributes, and sort the relation according to the priority. We then try to detect intra-relation FDs within the pipeline.

Notice that, for R_p, as soon as we know $X \rightarrow Y$ w.r.t. C_p does *not* hold, we also know that any intra-relation FDs having subsets of X as its LHS and Y as its RHS can not hold, respectively. Meanwhile, if we get $X \rightarrow Y$ w.r.t. C_p, which means X determines Y. We can presume that any superset of X can also determine Y. In this way, we can perform pruning on the attribute lattice.

Let us take a look at an example in Figure 6. First, we sort the relation according to `sid`, `sname`, `gender`, and `score` in this order (written as $INGS$), and form the first pipeline (Figure 6 (left)). We then get some intermediate results as follows: $ING \rightarrow S$[2] does not hold (written as $ING \nrightarrow S$), but $IN \rightarrow G$ and $I \rightarrow N$ hold. From $ING \nrightarrow S$, we can infer that any subset of ING can not determine S, so immediately we know the following results:

$$IN \nrightarrow S,\ IG \nrightarrow S,\ NG \nrightarrow S,\ I \nrightarrow S,\ N \nrightarrow S,\ G \nrightarrow S$$

In addition, we know that any superset of IN can determine G, and any super set of I can determine N:

$$IN \rightarrow G,\ INS \rightarrow G,\ I \rightarrow N,\ IG \rightarrow N,\ IS \rightarrow N,\ IGS \rightarrow N$$

[2] This is a shorthand of $\{sid, sname, gender\} \rightarrow score$.

Having inferred those additional results, the corresponding edges in the attribute lattice are removed (dotted edges in Figure 6 (left)).

In the next step, we choose another pipeline so that its length is as long as possible. Consequently, we form $SGNI$, and do the same process. In this way. based on the intermediate results, we can prune some unnecessary edges. The process is kept doing till we handle all the pipelines. Finally, we get the results in Figure 6 (right), in that there are only 14 edges left.

The following is the pseudo-code for detecting intra-relation FDs by using PipeSort.

Algorithm. DiscoverFdWithPipeSort
Input: R_p with attributes $a_1, \ldots, a_n$
$FDs \leftarrow \emptyset$
Generate the attribute lattice (LAT) containing all potential FDs over $a_1, \ldots, a_n$.
repeat
  Generate the longest pipeline (*pipe*) from LAT.
  Detect FDs (fds) from *pipe* using attribute partitions.
  Generate inferred FDs ($ifds$) from fds.
  Generate non-FDs ($nfds$) from fds and *pipe*.
  $FDs \leftarrow FDs \cup fds \cup ifds$
  Remove edges corresponding to $fds \cup ifds \cup nfds$ from LAT.
until $LAT \neq \emptyset$
return FDs

3.2 Further Optimization

When comparing attribute partitions, in a naive implementation, we need multiple scans over the table. Suppose that we try to process the pipeline, $ABCD$, ABC, AB, and A. We first compare $ABCD$ and ABC, then compare ABC and AB, and so on. In fact, this process can be done in a single scan over the $ABCD$ partition by checking multiple combinations of attributes. This has a significant impact on the process, and we will demonstrate it in the experiments.

3.3 Enhancing Inter-relation FD Discovery

XFDs exist not only in a relation but also across relations. Our objective here is to make the process of inter-relation FDs detection more efficient. The potential target for this stage is the attributes that are not determined by intra-relation FDs. Such attributes may be determined by combining attributes outside of the relation. In the previous example, we found that `score` is not determined by any attributes inside the relation. We also found that the following FDs hold:

$$\{sid\} \rightarrow sname \text{ w.r.t. } C_{student}$$
$$\{sid\} \rightarrow gender \text{ w.r.t. } C_{student}$$
$$\{cid\} \rightarrow cname \text{ w.r.t. } C_{course}$$

In principle, we need to check all possible combinations as follows:

(1) $\{sname, cid\} \rightarrow score$, $\{sname, cname\} \rightarrow score$, $\{sname, cid, cname\} \rightarrow score$
(2) $\{gender, cid\} \rightarrow score$, $\{gender, cname\} \rightarrow score$, $\{gender, cid, cname\} \rightarrow score$
(3) $\{cname, sid\} \rightarrow score$, $\{cname, sname\} \rightarrow score$, $\{cname, gender\} \rightarrow score$, $\{cname, sid, sname\} \rightarrow score$, $\{cname, sid, gender\} \rightarrow score$, $\{cname, sname, gender\} \rightarrow score$, $\{cname, sid, sname, gender\} \rightarrow score$
(4) $\{sid, cid\} \rightarrow score$
(5) $\{sname, gender, cname\} \rightarrow score$

Now, we know that $sid \rightarrow sname$. This indicates that if `sid` with the parent attribute cannot determine `score`, then `sname` combined with this parent attribute cannot determine `score` either. So, we consider `sid` as a candidate for determining `score`, and we do not need to check any combination of `sname` with parent attribute (1). As a consequence, we only need to check the combinations of `sid` with attributes in its parent relation.

The same discussion applies to `gender` (2). In R_{course}, $\{cid\} \rightarrow cname$ holds. As for (3), combinations of `cname` with attributes in $R_{student}$ need not to be checked. So cid is considered as another candidate for determining score.

In summary, instead of checking all the combinations, first we only need to check the combination of the candidates. Particularly in this case, we check combination of `sid` in $R_{student}$ and `cid` in R_{course} (4). In order to cover all the situations, we also check the combination of `sname`, `gender`, and `cname` (5) which turned out to be much fewer than exhaustive combinations. Finally we have

$$\{sid, cid\} \rightarrow score \text{ w.r.t. } C_{course}$$

4 Experimental Evaluation

We implemented the proposed scheme on top of Microsoft SQL Server 2008 using C#. We converted the XML file into hierarchically linked tables and stored them on disk. All experiments were conducted on a PC with a 2.0GHz P4 CPU and 2GB RAM, running Windows XP (SP2) and .NET3.5. For timing measurements, each experiment was run three times and the average reading was recorded.

In order to show the efficiency, we implement our approach by applying PipeSort in two ways: one is naive implementation and the other one is using the technique of single scan.

4.1 Real-Life Datasets

To show the effectiveness of our proposed algorithms, we discover XFDs on two available real-life data sources. One is the Mondial [11] geography dataset and the other is DBLP [12] bibliography dataset. To examine its practicality and to verify the existence of data redundancies in real world datasets, we conducted some experiments to perform comparative analysis on the proposed algorithm.

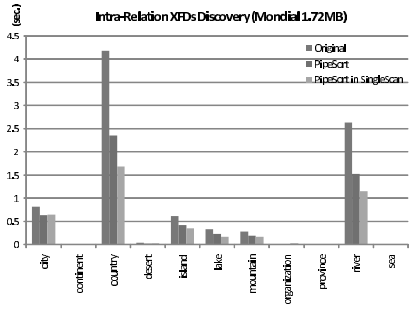

Fig. 7. Intra-Relation XFDs Discovery on Mondial

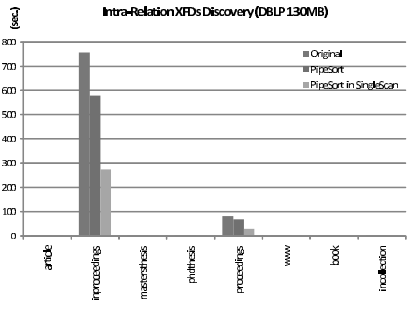

Fig. 8. Intra-Relation XFDs Discovery on DBLP

4.2 Experimental Results

Figure 7 to Figure 10 show the experimental results of the time cost when discovering XFDs in intra-relation and inter-relation.

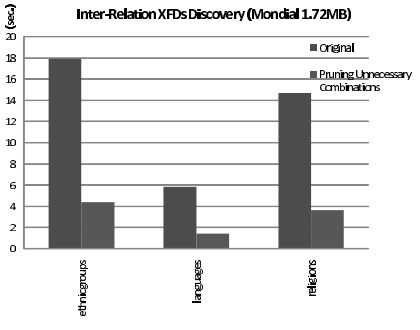

Fig. 9. Inter-Relation XFDs Discovery on Mondial

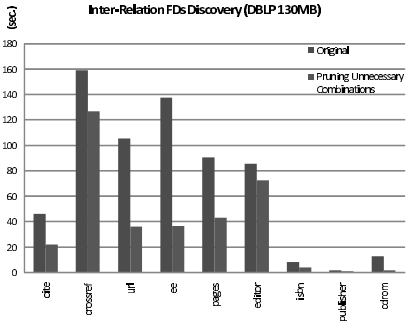

Fig. 10. Inter-Relation XFDs Discovery on DBLP

In Figure 7 and Figure 8 the X axis stands for the table name while Y axis is the time cost. We can see clearly that by applying the PipeSort method, the time cost of finding XFDs is much less than the original algorithm. Meanwhile, the degree of loops can be reduced largely in one single-scan so that we can make the discovery faster. Especially when the table size is really large or contains many attributes, like the relations "country", "river" and "inproceedings", then the PipeSort is much faster according to Figure 7 and Figure 8.

Figure 9 and Figure 10 show the time cost of inter-relation XFDs discovery. The X axis is the attribute cannot be determined during the intra-relation while Y axis also stands for the time cost. Since we prune some unnecessary combinations by using the properties of functional dependency. The discovery has been processed fast. The performance is shown in Figure 9 and Figure 10.

5 Related Work

XML data is usually described as a "schema-less" labeled tree, which means that there is no pre-designed schema. The problem with this is that it is so flexible that it only presents syntax without any constraints. To address this problem, a number of constraint specifications have been proposed recently including the notion of XML Keys [13,14], XML Functional Dependencis [15,16,17,18,19,20,21], and XML Normal Forms [22,2,23,24], among which undoubtedly XFD is the foundation of XML constraints and also plays the important role in defining normal forms.

XFDs, which have been proposed so far, can roughly be categorized into the following two main approaches: path-based approach [16,4,19,20,21] and tree-tuple based approaches [2,5]. In the former, the authors define the XFDs by specifying the target element of the constraint. But, in reality, it is often difficult. In the latter approach, instead of specifying target elements, the authors use tree tuples, each of which is a XML sub-tree by picking up each schema element and projecting away all the other nodes.

However, this approach can not handle set and multi-valued dependencies. In order to cope with this problem, C. Yu and H. V. Jagadish proposed generalized tree tuple based XFDs definition. The approach they developed can not only be used for the previous definitions of XFDs, but also capable of detecting XFDs involve set elements while maintaining clear semantics [5].

One of the major drawbacks of [5] is its efficiency in finding XFDs. In the first place, it needs to make many partitions of each table, followed by comparisons among those partitions, Obviously, this process contains some unnecessary combinations and there is not an effective way to deal with the comparisons.

In [25] the authors present a scalable and efficient algorithm to discover the composite keys in large database. The main idea is based on a cube operation to find all non-keys. This is similar to our work because we all apply some cube operations to detect constraints existing in data collection. In our PipeSort algorithm, group-by is the foundational operation, while in [25] slicing and group-by are proposed as the primary method. Also, in both approaches some necessary pruning has been done to speed up the discovery process. However, generally speaking, rather than detecting the non-FD, which means that functional dependencies do not hold (like discovery of non-keys). We think that it is important and interesting to discover the relationship of one attribute set can determine another (functional dependency).

In the paper, so naive implementation of this process is quite time-consuming. To speed up this process, we apply PipeSort method [7], which was originally developed for OLAP computation [8,26]. This process incorporates the cache-result, amortize-scan, and share-sorts to reduce the disk I/O cost, and based on the intermediate results we can prune some unnecessary combinations and this can make our system efficiently.

6 Conclusions

XML data redundancies are greatly existing and have a richer semantic. And functional dependency is playing an important role in avoiding data redundancies. Many definitions are given on XML functional dependency, but there is not an efficient way to discover XML functional dependencies(XFDs).

In our paper, based on the notion given in [5], we propose an efficient way to discover XFDs. We apply the PipeSort method to speed up the XFDs discovery process, and also based on the XFD properties many combinations can be pruned.

In order to see the effectiveness of our proposal, we did the implementation and experimentation using the data source from the real world. Based on the experimental results, we can verify that our proposal is useful and effective. As a part of our future work, we plan to build the system without transforming XML document into relational databases.

Acknowledgments

This study has been partly supported by the Grant-in-Aid for Scientific Research in Priority Areas (#21013004) and Grant-in-Aid for Young Scientists (B) (#21700093).

References

1. Codd, E.F.: Further normalization of the data base relational model. IBM Research Report, San Jose, California RJ909 (1971)
2. Arenas, M., Libkin, L.: A normal form for XML documents. In: Proc. PODS 2002, pp. 85–96 (2002)
3. W3C: Extensible Markup Language (XML) 1.0, 5th edn., Recommendation (November 2008), `http://www.w3.org/TR/xml/`
4. Vincent, M.W., Liu, J., Liu, C.: Strong functional dependencies and their application to normal forms in XML. ACM Trans. Database Syst. 29(3), 445–462 (2004)
5. Yu, C., Jagadish, H.V.: Efficient discovery of XML data redundancies. In: Proc. VLDB 2006, pp. 103–114 (2006)
6. Yu, C., Jagadish, H.V.: XML schema refinement through redundancy detection and normalization. VLDB J. 17(2), 203–223 (2008)
7. Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J.F., Ramakrishnan, R., Sarawagi, S.: On the computation of multidimensional aggregates. In: Proc. VLDB 1996, pp. 506–521 (1996)
8. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. Data Min. Knowl. Discov. 1(1), 29–53 (1997)
9. W3C: XML Schema Part 2: Datatypes, 2nd edn., Recommendation (October 2004), `http://www.w3.org/TR/xmlschema-2/`
10. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: An efficient algorithm for discovering functional and approximate dependencies. Comput. J. 42(2), 100–111 (1999)

11. May, W.: Information Extraction and Integration with Florid: The Mondial Case Study, http://www.dbis.informatik.uni-goettingen.de/lopix/lopixmondial.html
12. Ley, M.: DBLP Bibliography, http://www.informatik.uni-trier.de/~ley/db/
13. Grahne, G., Zhu, J.: Discovering approximate keys in XML data. In: Proc. CIKM 2002, pp. 453–460 (2002)
14. Hartmann, S., Link, S.: Unlocking keys for XML trees. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. 4353, pp. 104–118. Springer, Heidelberg (2006)
15. Lee, M.L., Ling, T.W., Low, W.L.: Designing functional dependencies for XML. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 124–141. Springer, Heidelberg (2002)
16. Liu, J., Vincent, M.W., Liu, C.: Functional dependencies, from relational to XML. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 531–538. Springer, Heidelberg (2004)
17. Hartmann, S., Link, S.: More functional dependencies for XML. In: Kalinichenko, L.A., Manthey, R., Thalheim, B., Wloka, U. (eds.) ADBIS 2003. LNCS, vol. 2798, pp. 355–369. Springer, Heidelberg (2003)
18. Lv, T., Yan, P.: A survey study on XML functional dependencies. In: Proc. ISDPE 2007, pp. 143–145 (2007)
19. Fassetti, F., Fazzinga, B.: Approximate functional dependencies for XML data. In: Ioannidis, Y., Novikov, B., Rachev, B. (eds.) ADBIS 2007. LNCS, vol. 4690, pp. 86–95. Springer, Heidelberg (2007)
20. Shahriar, M.S., Liu, J.: On defining functional dependency for XML. In: Proc. IEEE ICSC 2009, pp. 595–600 (2009)
21. Zhao, X., Xin, J., Zhang, E.: XML functional dependency and schema normalization. In: Proc. HIS 2009, pp. 307–312 (2009)
22. Mok, W.Y., Ng, Y.-K., Embley, D.W.: A normal form for precisely characterizing redundancy in nested relations. ACM Trans. Database Syst. 21(1), 77–106 (1996)
23. Arenas, M., Libkin, L.: An information-theoretic approach to normal forms for relational and XML data. In: Proc. PODS 2003, pp. 15–26 (2003)
24. Pankowski, T., Pilka, T.: Transformation of XML data into XML normal form. Informatica 33(4), 417–430 (2009)
25. Sismanis, Y., Brown, P., Haas, P.J., Reinwald, B.: GORDIAN: Efficient and scalable discovery of composite keys. In: Proc. VLDB 2006, pp. 691–702 (2006)
26. Sarawagi, S., Agrawal, R., Gupta, A.: Research report on computing the data cube. Technical report, IBM Almaden Research Center