Discovering Conditional Functional Dependencies in XML Data

Loan T.H Vo, Jinli Cao, Wenny Rahayu
Department of Computer Science Engineering
La Trobe University
Melbourne, Australia
t7vo@students.latrobe.edu.au,{J.Cao, W.Rahayu}@latrobe.edu.au

Abstract

XML data inconsistency has become a serious problem since XML was widely adopted as a standard for data representation on the web. XML-based standards such as OASIS, xCBL and xBRL have been used to report and exchange business and financial information. Such standards focus on technical rather than semantic aspects. XML Functional Dependencies (XFDs) have been introduced to improve XML semantic expressiveness. However, existing approaches to XFD discovery that have been proposed mainly for enhancing schema design are not capable of dealing with data inconsistency. They cannot find a proper set of semantic constraints from the data, and thus are insufficient for capturing data inconsistency. In this paper we propose an approach, called XDiscover, to discover a set of minimal XML Conditional Functional Dependencies (XCFDs) from a given XML instance to improve data consistency. The XCFD notion is extended from XFDs by incorporating conditions into XFD specifications. XCFDs can be used to constrain data process and also to detect and correct non-compliant data. XDiscover incorporates pruning rules into discovering process to improve searching performance. We present several case studies to demonstrate the effectiveness of our approach.

Keywords: Discovering data rules, data inconsistencies, data quality, XML data.

1 Introduction

Extensible Markup Language (XML) has become a standard for representing data on the web. XML-based standards, such as OASIS, xCBL and xBRL have been introduced for reporting and exchanging business and financial information (Lampathaki et al. 2008). However, such standards only provide schema document frameworks for preparing reports and exchanging data. Most XML-based standards do not address the semantics of underlying business information. Constraints on the underlying data from different organizations are often violated.

In other words, these constraints satisfied by an individual data source may not be applicable in the federated data. Under such circumstances, deriving a complete set of constraints from a given data instance to constrain the heterogenous data sources is necessary for improving data consistency.

Although XML functional dependency (XFD) is one type of semantic constraint, existing notions of XFD (Vincent et al. 2004, Fan 2005, Arenas 2006) are not

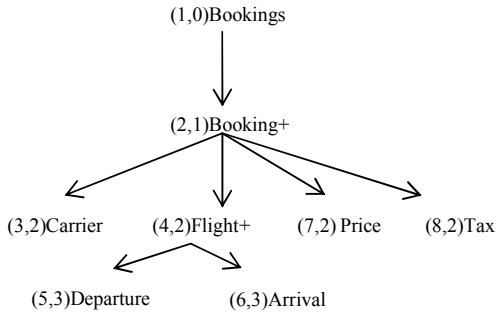

Figure 1: An example of *Booking* Schema Tree

sufficient in capturing data inconsistency. This is because XFDs globally express data constraints over the whole document; and thus, they are unable to capture conditional semantics partially expressed in some fragments of the document.

Figure 2 shows an example of a simplified instance of a Bookings data tree *D* constrained by the schema Bookings *S* in Figure 1. *D* contains data of Flight Bookings. Each Booking includes information on the Carrier, Flight, Price and Tax. For each Flight, information on Departure and Arrival are maintained. Values of elements are recorded under the node names (in bold). We assign a pair (*order, depth*) to each node in the schema tree *S* and the data tree *D* as a key to identify that node in the tree. This notion will be further described in Definition 1 (Section 3). Constraints on D have different specifications. We classify them into two types.

***Type 1:* Constraints without conditions**

Constraints without conditions contain only variables. They are data constraints hold over the whole document and are commonly known as Functional Dependencies (FD). For example,

Constraint 1: Any Booking with the same Flight (including Departure and Arrival) should have the same Tax.

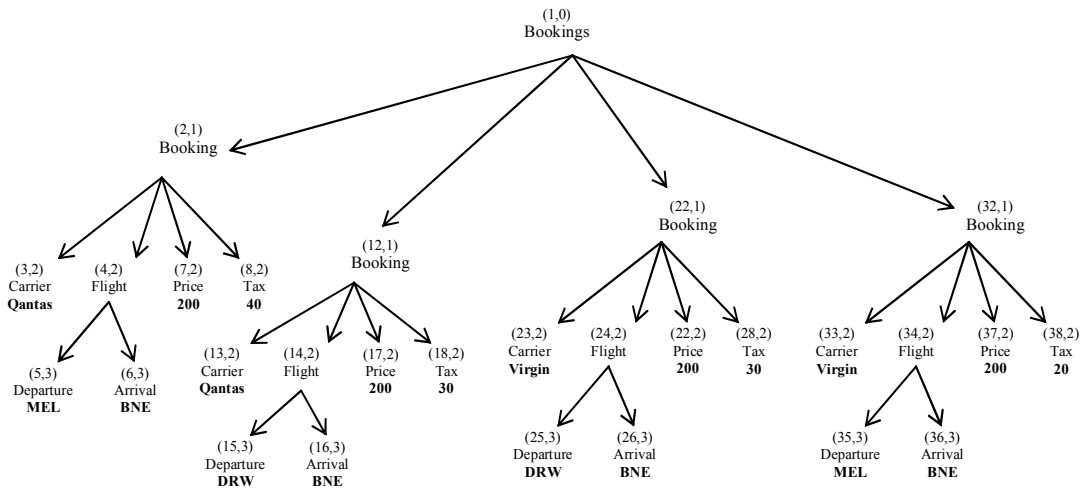

Figure 2: A simplified example of Booking **data tree**

Constraint 1 is an example of a FD holding for all Bookings in *D*.

Type 2: **Constraints with conditions**

This type includes constraints which either contain constants only or both constants and variables. Such constraints hold conditionally on the document. They are not standard FDs. For example,

Constraint 2a: Any Booking with Carrier of *Qantas* having the same Price should have the same Tax.

Constraint 2b: Any Booking with Carrier of *Virgin* and Arrival of *BNE* has a Tax of *20*.

Constraints 2a and 2b are supposed to hold for Bookings with Carrier of *Qantas* or for Bookings with Carrier of *Virgin* and Arrival of *BNE* respectively. They refine Constraint 1 by binding particular values to elements in the constraints e.g. *Qantas* or *Virgin, BNE* and *20* for Carrier, Arrival and Tax respectively. Constraints of type 2 are very common in real data, especially for data from multiple sources that use XML-based standards. Each constraint holds only on a subset of a document containing data from one particular source.

To capture data inconsistency, we need to enforce constraints of type 2. This is because when constraints with conditional semantics are not enforced explicitly, the data inconsistency in some parts of document cannot be detected. By *data inconsistency*, we mean XML data violates certain constraints. For example, Bookings data in *D* (Figure 2) do not satisfy all above constraints. The Bookings of nodes (12, 1) and (22,1) contain the same values of Flight including Departure (*DRW*) and Arrival (*BNE*) and have the same value of Tax (*30*). They satisfy constraint 1 but violate either constraint 2a or constraint 2b. For constraint 2a, if Carrier is *Qantas*, the Price determines the Tax. Node (12, 1) and node (2, 1) have the same Price (*200*) but they contain different values of Tax (*30* and *40* respectively) that violate constraint 2a. According to constraint 2b, if a Booking with Carrier of *Virgin* and Departure of *BNE,* Tax should be *20* but node (22, 1) contains the Tax of 30 that violates constraint 2b. We can see that if constraint 2a and 2b are not enforced, the inconsistency of node (12, 1) and node (22, 1) cannot be identified.

In this paper, we propose a novel approach to improve XML data consistency. Our contributions are twofold. Our first contribution is to introduce a new notion of XCFDs as constraints of type 2 which revise XFDs by incorporating conditions into XFD specifications. This overcomes limitations of the previous work in two aspects: (*i*) XCFDs can express constraints in the hierarchical structure in XML data, as opposed to Conditional Functional Dependencies in relational databases; (*ii*) XCFDs are more powerful than XFDs in term of capturing data inconsistency. This is because XCFDs allow binding specific constants to particular elements. They cover more situations of dependencies under some conditions. Additionally, we propose an approach, named XDiscover, to search for a set of minimal XCFDs from a given instance. XDiscover also incorporates a set of pruning rules in discovery process. Our purpose is to reduce the searching lattice and the number of XCFD candidates to be checked on the dataset to improve searching performance.

Discovered XCFDs can be embedded as an integral part in an enterprise's systems to constrain the data process to minimize the data inconsistency. Our approach also can be used within the process of data quality management to detect and correct non-compliant data.

The rest of paper is organized as follows. In section 2, we review related work and discuss the limitations of existing approaches including CFDs in relational databases and XFDs. Section 3 presents preliminary definitions which are necessary for introducing XCFDs in section 4. The detailed approach is described in section 5. The case studies and comparative evaluation are presented in section 6, and conclusions of our work are discussed in section 7.

2 Related Work

The problem of data inconsistency has been extensively studied for relational databases. The notion of Conditional Functional Dependencies (CFDs) (Bohannon et al. 2007) has been widely used as a technique to

detect and correct data inconsistency. Many approaches (Chiang and J.Miller 2008, Fan et al. 2009, Golab et al. 2008) have been proposed to automatically discover CFDs from data instances. Despite facing similar problems with relational counterparts, the existing approaches of CFDs cannot be applied directly into XML data for several reasons.

Firstly, there are significant differences in the data structures and the nature of constraints. For relational databases, each object is defined by a single row. Discovering CFDs from data stored in tables has a clearly defined structure. By contrast, XML data has a hierarchical structure and constraints often involve elements from multiple hierarchical levels. Identifying XML data constraints faces challenges that are not encountered in discovering CFDs.

Secondly, different notions of equality are used for constraints. Whereas relational equality simply is the equality of values, the equality of two objects in XML has to be compared according to both structure and data (Yu and Jagadish 2006). Finally, relational CFD discovery algorithms cannot scale well when the XML data structure is complex. This is because applying these algorithms to XML data requires an XML document be transformed into a single relational table. When the structure of schema is complex, the number of attributes in transformed relation is large. The number of tuples also increases multiplicatively when the XML document contains data with complex data type (e.g. *maxOccurs* in XML Schema). For example, if each Booking contains two Flights, then the number of tuples in transformed relation would double.

Most existing work in XML data (Grahne and Zhu 2002, Hartmann and Link 2003, Yu and Jagadish 2008, Lv and Yan 2006, Trinh 2008) focuses on introducing XFDs to improve schema design. Such approaches are insufficient to capture data inconsistency. This is because XFDs are sematic constraints on the whole document whereas XML data is often obtained by integrating data from different sources with conditional semantics. Discovery algorithms to XFDs cannot find a proper set of constraints. Therefore, it is necessary to introduce new notions of XML data constraints with conditional semantics.

Some previous work (Flesca et al. 2003, Tan et al. 2007, Flesca et al. 2005) has addressed the XML data inconsistency problems. They assume that the set of constraints is known and they only focus on finding consistent parts from inconsistent XML documents w.r.t predefined constraints. In fact, discovering data constraints from data instances manually is a tedious process which requires extensive searching and is time consuming. As XML data becomes more common and more complex in the data structure, it is necessary to develop a formal approach that automatically discovers sematic constraints. To the best of our knowledge, no existing work addresses such problems yet.

XFDs are formally defined in two perspectives (Vincent et al. 2004, Arenas 2006, Hartmann and Link 2003) which are path-based and tree tuple-based approaches. These approaches cannot express the semantics of constraints with a set of complex elements as in our cases. Only the notion of Generalized tree tuple-based XFDs (Yu and Jagadish 2008) is closest to ours. However, this approach adopts tuple-like semantics which requires XML data to be stored in a set of tables and cannot express constraints on the XML data tree directly. No existing XFD definitions can be used to fully extend to our XCFD notion.

To facilitate our approach, we first introduce necessary preliminaries in the next section.

3 Preliminaries

In this section, we present the background and some definitions such as XML schema tree, data tree, data–schema conformation and node-value equality.

We use XPath expression to form a relative path; "." (self): select the context node. ".//": select the descendants of the context node. For example, .//Carrier: select Carrier descendants of the context node Booking; .//Flight/Departure: select all Departure elements which are children of Flight.

In this paper, we consider an XML schema or an instance as rooted-unordered-labelled trees, referred to as a schema tree or a data tree, respectively. Each element node is followed by a set of element nodes or a set of attribute nodes. For the instance, the element node can be terminated by a text node.

We give formal definitions for an XML schema tree and an XML data tree as follows:

Definition 1. (XML schema tree)

An XML schema tree is defined as $S= (E, A, T, root)$, where:

- $E = E_1 \cup E_2$ is a finite set of element nodes in S in which each node is associated with frequency label of ?, +, *, 1; For every node e_j in E, the number of nodes from an instance mapped to e_j is at most one if node e_j has frequency label *?*; exactly one if e_j has a frequency either label 1 or no label at all; at least one if node e_j has frequency label +; and unlimited occurrences if e_j has a frequency label *. E_1 is a set of complex nodes; E_2 is the set of simple nodes.
- A is a finite set of attribute nodes; attribute nodes only appear as leaf nodes.
- T is a finite set of node types; for each node $e \in E_1 \cup E_2 \cup A$ is associated with a data type $t \in T$; t can be a simple data type (e.g. string, int, float) or a complex data type (e.g., the data type represents for the *maxOccurs*, "*choice*" and "*all*" model groups) in XML Schema Language (W3C 2004). An element node is called a simple element node if it is defined with a simple data type. Otherwise it is called a complex node. An attribute node is considered as a simple element node.
- *root* is the root the schema tree.

For example, the schema tree in Figure 1 is defined as *S= (E, A, T, root);* where:

$E = E_1 \cup E_2$; E_1 = feature

E_2 = {Carrier, Departure, Arrival, Price, Tax}

A= {∅}; *root*= Bookings; T = {String, int, Booking, Flight}; Booking and Flight are complex data types.

We assign a *path-ID* to each node in the XML schema tree as shown in Figure 1 in a preorder traversal. Each path-ID is a pair (*order, depth*); where *order* is an increasing integer (e.g. 1, 2, 3...) which is used as a key to identify the path from the root to a particular node and *depth* label is the number of edges traversing from the root to the node in the schema tree. The depth of the root is 0; e.g. assigning 0 for /Bookings; 1 for /Bookings/Booking

Definition 2. (XML data tree)

An XML data tree constrained by an XML schema tree *S= (E, A, T, root)* is defined as *D= (V, lab, ele, att, val, r)*, where:

- *V* is a set of nodes in *D*; each $v \in V$ consists of a label *e* and a *node-ID* that uniquely identify node *v* in *D*.
- *lab* is a labelling function which maps the set *V* to the set $E \cup A$. Each $v \in V$, *v* is called an element node if $lab(v) \in E$; *v* is called an attribute node if $lab(v) \in A$.
- *ele* is a partial function from *V* to a sequence of *V* nodes; for each complex element node $v \in V$, the function *ele(v)* maps *v* to a list of element nodes $\{v_1, v_2, ..., v_n\}$ in *V*; *att(v)* maps *v* to a list of attribute nodes $\{v_1', v_2', ...v_m'\}$ in *V* with distinct labels.
- *val* is a function that assigns values to simple element nodes and attribute nodes. Each node $v \in V$; *val(v)* is the content of attribute if $lab(v) \in A$ or the content of simple node if $lab(v) \in E1$; $val(v) = v$ if $lab(v) \in E2$.
- $r \in V$, $lab(r) = root$ that is the unique root node and is labelled with complex data types.

The *node-ID* in the XML data tree is assigned the same ordering as the path-ID in the XML schema tree. Each *node-ID(order, depth)* contains values uniquely identifying its position in the data tree.

For example, from Figure 2, we have *V* a set of nodes from node (1, 0) through node (38, 2).

lab(node(1,0))=Bookings;
lab(node(3,2))="Carrier"; *val*(node(2,1))= Booking;
val(node(3,2))="Qantas"; *ele*(node(2,1))={Carrier, Flight, Price, Tax}.

From Definition 2, we have the following properties:

i) If $v_2 \in ele(v_1)$ then v_2 is called a child node of v_1.

ii) $\{v[P]\}$ is a set of direct nodes that can be reached following path *P* from *v*, where *P* is the path from the root to the node *v*. The path *P* can be a single node, e.g. *root*[*root*] = {all direct children nodes of *root*}. If there is only one node in $\{v[P]\}$, we write *v*[*P*].

In this paper, we assume that the XML data tree is required to conform to the associated XML schema tree. The conformation is defined as follows:

Definition 3. (XML data –Schema tree conformation)

An XML data tree *D= (V, lab, ele, att, val, r)* is said to conform to a schema tree *S= (E, A, T, root)* denoted as $D \models S$ if and only if (iff):

- $lab(r) = root$.
- Every node $v \in V$, $lab(v) \in E \cup A$. There is a homomorphism from *V* to $E \cup A$ such that for every pair of mapping nodes (v_i, e_j), the node name and the data type are preserved. Figure 2 is an example of the Bookings data tree which conforms to the Bookings schema tree in Figure 1.

Now we introduce a notion of node-value equality which is an essential feature in the definition of XFDs. Two nodes are called node-value equality if two corresponding sub-trees rooted at the two nodes are identical.

Definition 4. (Node-value equality) Two nodes v_i and v_j in an XML data tree *D= (V, lab, ele, att, val, r)* are node-value equality, denoted $v_i =_v v_j$ iff:

- v_i and v_j has the same label $lab(v_i) = lab(v_j)$
- if v_i and v_j are both simple element nodes or attribute nodes, then $val(v_i) = val(v_j)$
- if v_i and v_j are both complex element nodes and $ele(v_i) = [v_{i1}, ..., v_{in}]$ then $ele(v_j) = [v_{j1}, ..., v_{jn}]$ and $v_{ik} =_v v_{jk}$ for all *k*; $1 \le k \le n$ and vice versa.

For example, node(14, 2) and node(24, 2) (in Figure 2) are node-value equality. Because we have:

lab(node(14,2))=*lab*(node(24,2))="Flight";
ele(node(14,2))={node(15,3),node(16,3)};
ele(node(24,2))={node(25,3),node(26,3)};
node(15, 3) $=_v$ node(25, 3)= "*DRW*" and
node(16, 3) $=_v$ node(26, 3) = "*BNE*".

The definitions above are basic concepts used in our proposed notions in the next section.

4 XML Conditional Functional Dependency

In this section, we present the new notion of XML Conditional Functional Dependency (XCFD). The most important features of XCFDs are path and value-based constraints.

An XCFD is defined as an extension of XFD. Therefore we need to describe a precise XFD definition before introducing XCFDs.

Definition 5. (XML Functional Dependency)

Given an XML data tree *D= (V, lab, ele, att, val, r)* conforming to an XML schema tree *S= (E, A, T, root)*, an XML Functional Dependency over *D* is defined as:

$\varphi = P_v: \{X\} \rightarrow \{Y\}$; where:

- P_v is a downward context path starting from the root to a considered node *v*. The scope of φ is the sub-tree rooted at node *v[P]*;
- *X, Y* are non-empty set of paths rooted at *v[P]*.
- $\{X\} \rightarrow \{Y\}$ indicates a relationship between the nodes in *X* and *Y*, such that for two sub-trees that share the same values for *X* must share the same values for *Y*. That is, the values of nodes following the *X* path uniquely identify the values of the nodes following the *Y* path. We refer to *X* as the *antecedent* and *Y* as the *consequence*.

A document *D=(V, lab, ele, att, val, r)* conforming to *S*, $D \models S$, is said to satisfy $\varphi = P_v: \{X\} \rightarrow \{Y\}$ denoted $D \models \varphi \wedge S$ iff any two nodes n_i and n_j in *D*, if $\{n_i[X]\} =_v \{n_j[X]\}$ then $\{n_i[Y]\} =_v \{n_j[Y]\}$.

Let us consider an example, where we suppose $P_{Booking}$ is the context path from the root to the *Booking* nodes in the *Booking* data tree (in Figure 2).

X= (.//Flight/Departure, .//Flight/Arrival) and *Y* = (.//Tax) then we have an XFD:

$\varphi = P_{\text{Booking}}$: (.//Flight/Departure,.//Flight/Arrival)→ (.//Tax).

After defining the XFD notion, we propose a novel notion of XCFDs by revising the XFD definition.

Definition 6. (XML Conditional Functional Dependency) Given an XML data tree *D=(V, lab, ele, att, val, r)* conforming to a schema tree *S =(E, A, T, root)*; an XML Conditional Functional Dependency holding on *D* is defined as:

$\psi = P_v$: {$\mathcal{C}$}, {X} → {Y}, where: $\mathcal{C}$ is a condition for the XFD {X}→ {Y} holds on a subset of *D*. The condition $\mathcal{C}$ has the form: $\mathcal{C} = ex_1 \theta ex_2 \theta \ldots \theta ex_n$; ex_i is a Boolean expression associated to a particular data node. "θ" is an operator either *AND* (^) or *OR* ($\vee$).

For example, suppose we assume that P_{Booking} is the context path from the root to the Booking nodes in the Bookings data tree (Figure 2); if there exists an XFD (.//Price)→(.//Tax) holding on the Bookings data tree under condition $\mathcal{C}$ = (.//Carrier = "Qantas"), then we have an XCFD:

$\psi = P_{\text{Booking}}$: (.//Carrier= "Qantas", .//Price)→(.//Tax).

XDiscover only returns minimal XCFDs. Before proposing the XDiscover algorithm, we define the minimal XCFDs as follows.

Definition 7. (Minimal XCFDs)
Assume that an XML data tree *D= (V, lab, ele, att, val, r)* conforms to the XML schema *S= (E, A, T, root)*. An XCFD $\psi = P_v$: {$\mathcal{C}$}, {X} → {Y} *on D* is minimal if $\forall \mathcal{C}' \subset \mathcal{C}$, $\psi' = P_v$: {$\mathcal{C}'$}, {X} → {Y} does not hold on *D*.

Based on aforementioned definitions, we formally define the problem of discovering XCFDs as follows:

Problem Statement: Given an XML data tree *D= (V, lab, ele, att, val, r)* conforming to a schema S= *(E, A, T, root)*; the goal of XDiscover is to discover a set of non-redundant XCFDs in the form $\psi = P_v$: {$\mathcal{C}$}, {X}→ {Y}; where each XCFD is minimal and contains only a single path in the *consequence Y*.

The approach of discovering XCFDs, called XDiscover, includes a number of functionalities. We describe the XDiscover algorithm in detail in the next section.

5 XDiscover: Discovering XCFDs

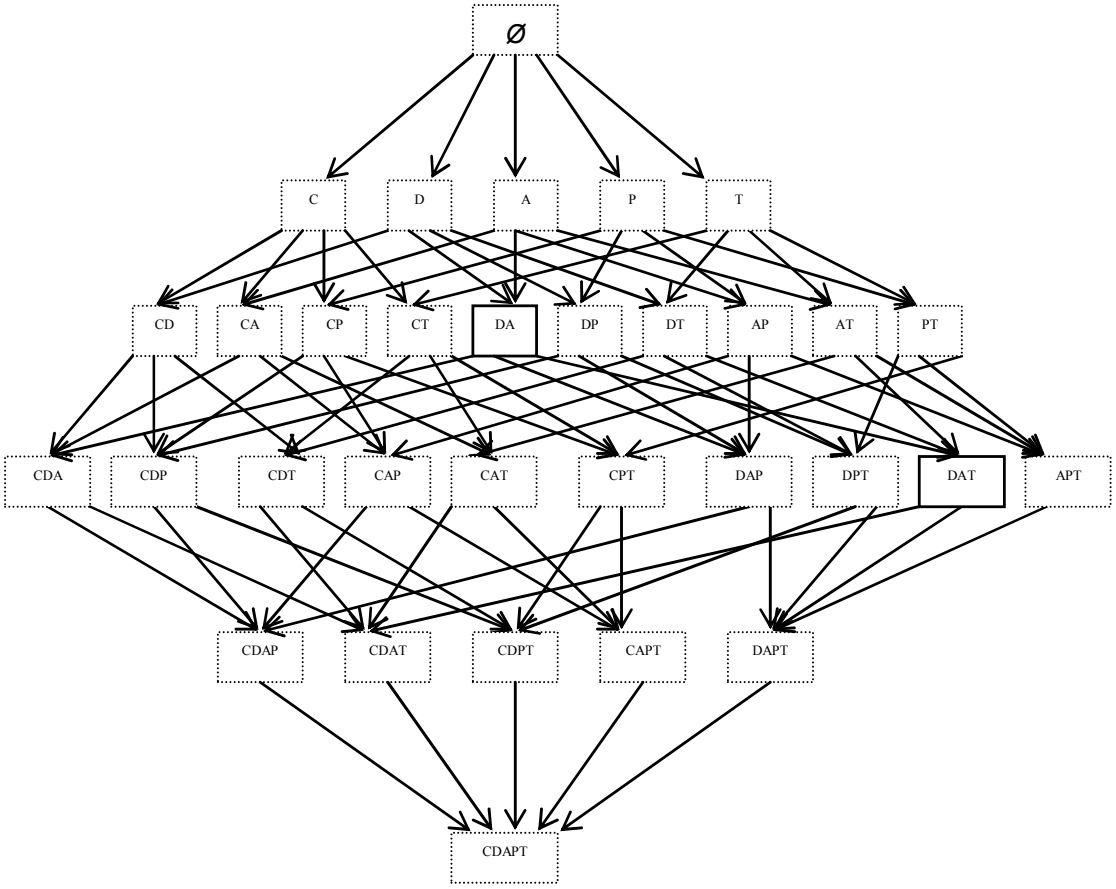

Figure 3: A set of containment lattice of Carrier **(C),** Departure **(D),** Arrival **(A),** Price **(P) and** Tax **(T)**

The XDiscover algorithm includes four main functions to discover XCFDs. The first function is to generate partition identifiers and identify *candidate* XCFDs. The second function is to generate partitions of *partition identifiers* associated to each candidate XCFD. Such generated partitions are used in the third function to validate for a satisfied XCFD. The last function performs a set of pruning rules to search for minimal XCFDs and remove redundant candidate XCFDs in the next level in the searching lattice.

5.1 Candidate XCFD Identification

To search for all XCFDs, the first function generates a *searching lattice* which contains all possible combinations of node labels. The process starts from nodes with a single label (level l= 1). Node labels in level l with l >=2 will be obtained from the node labels in level (*l-1*). Figure 3 is an example of a searching lattice of node labels: C, D, A, P and T (represent for Carrier, Departure, Arrival, Price, and Tax respectively). The node label (CP*)* in level 2 is generated from nodes C and P in level 1.

Assume that W & Z are two nodes directly linked in the searching lattice; and $Z=W\cup\{Y\}$. Each edge*(W, Z)* represents a candidate XCFD $\psi = \{ P_v: \{\mathcal{C}\}, \{X\} \rightarrow \{Y\}\}$; where $W= \{X\}\cup \{\mathcal{C}\}$; X is a set of variable nodes; $\mathcal{C}$ is a set of conditional nodes. For example, for edge*(W, Z)*= (CP, CPT) in Figure 3, we assume the conditional data node is C, then we have $\psi = P_{\text{Booking}}$: C, P $\rightarrow$ T. If condition $\mathcal{C}$ is empty then ψ become a data constraint on the whole document as an XFD. That means an XFD is a special case of an XCFD.

Node labels (e.g, W and Z) in the searching lattice associated with a candidate XCFD are called *partition identifiers*.

Once the searching lattice has been established, *partitions* of partition identifiers in each candidate XCFD are generated. The results of partition generation are the input of *discovering XCFDs* function.

5.2 Partition Generation

The partition generation for each partition identifier classifies a considered node in the data tree into classes based on the node's values in the partition identifier. Each class contains all elements which have the same values on the partition identifier. Partitions play a vital role in validating a satisfied XCFD on the data tree. In the following, we present a formal definition of partition.

Definition 8. (Partition) A partition $\Pi_{W|v}$ of W on D under the sub-tree rooted at v is a set of disjoint equivalence classes w_i. Each class w_i in $\Pi_{W|v}$ contains all nodes with label v having the same values on partition identifier W.

The number of classes in a partition is called *cardinality* of the partition, denoted $|\Pi_{W|v}|$; $|w_i|$ is the number of nodes in class w_i.

For example, from schema tree Bookings S in Figure 1, we have: E={[(1, 0)Bookings] ,[(2, 1)Booking],[(3, 2) Carrier], [(4,2) Flight], [(5, 3) Departure], [(6, 3) Arrival], [(7, 2)Price], [(8, 2)Tax]}

From the searching lattice (Figure 3), suppose we consider a partition identifier W= "Carrier" which corresponds to the node [(3, 2) Carrier] in the schema tree S. Traversing data tree Bookings D in Figure 4 to find all data nodes which have the node name as Carrier and *depth* of 2.

The found nodes are grouped into two classes:

$Class_1$= { [(23, 2) Carrier= "Qantas"], [(33, 2) Carrier= "Qantas"] , [(53, 2)Carrier= "Qantas"], [(63, 2) Carrier= "Qantas"]}

$Class_2$ ={[(43, 2) Carrier= "Virgin"], [(73, 2) Carrier= "Virgin"]}

The partition $\Pi_{\text{Carrier}\,|\,\text{Booking}}$ to the value of node Carrier w.r.t sub-tree rooted at Booking is represented as $\Pi_{\text{Carrier}\,|\,\text{Booking}} = \{w_1, w_2\}$

w_1= {[(22,1) Booking], [(32, 1) Booking], [(52, 1)

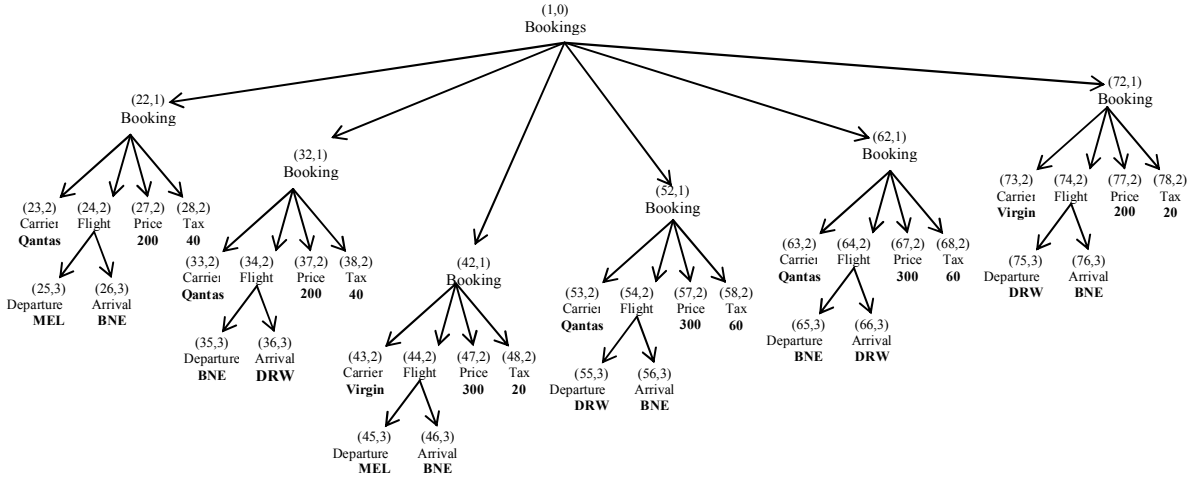

Figure 4: A simplified instance of Bookings **data tree:** *each Booking contains only one Flight*

Booking], [(62, 1) Booking]}

w_2 ={[(42, 1) Booking], [(72, 1)Booking]}

$|\Pi_{\text{Carrier}\,|\,\text{Booking}}| = 2$; $|w_1| = 4$; $|w_2| = 2$.

To simplify the presentation, we omit the node-ID and path-ID associated with each node in following sections to avoid cluttering.

After finding partitions of partition identifiers in a candidate XCFD, a *validation function* is performed to check whether or not this candidate holds on the data tree. The detail of this function is in the next subsection.

5.3 XCFD Validation

We still use the same assumptions of W & Z in Section 5.1. Validating for a satisfied XCFDs is based on the concept of partition refinement defined by Huhtala et.al (Huhtala et al. 1999):

Definition 9. (Partition refinement): A partition Π_W refines Π_Z if any nodes in a class w_i of Π_W are also in a class z_j of Π_Z.

A functional dependency holds on document D if Π_W refines Π_Z (Yu and Jagadish 2008). Because an XCFD $\psi = P_v$: $\{\mathcal{C}_j, \{X\} \rightarrow \{Y\}$ holds only on the subset of data tree D, the satisfied XCFD does not require every class w_i in Π_W be the subset of a class z_j in Π_Z. This means if there exists at least one equivalence pair (w_i, z_j) between Π_W and Π_Z then ψ holds on document D.

For general cases, let Ω_W be a set of all classes in Π_W, where each w_i in Π_W has a corresponding equivalent class z_j in Π_Z. If there exists a class c_i in Π_e which contains exactly all elements of Ω_W, the class c_i will be the condition for an XCFD: $\psi = P_v$: $\{c_i\}, \{X\} \rightarrow \{Y\}$ holds on data tree D.

The number of candidate XCFDs and the searching lattice are very large. Our algorithm uses a set of pruning rule, introduced in the next section, to reduce the number of XCFDs to be checked on the dataset.

5.4 Pruning rules

We introduce rules which are used to (*i*) skip searching for those XCFDs that are logically implied by the already found XCFDs and *(ii)* prune redundant XCFD candidates from the searching lattice.

Rule 1: Let E be a set of nodes in the schema S, and ψ be an XCFD= P_v: $\{\mathcal{C}_j, \{X\} \rightarrow \{Y\}$. For each $\{X\}$ we only check XCFDs which have *consequence* $\{Y\}$ in the remaining set of elements in E; $Y \in E \setminus (\{X\} \cup \{\mathcal{C}_j\})$.

For example, if E={ D, A, P, T}, X={D}, and $\mathcal{C}$= {A} then only XCFDs contain $Y \in$ {P, T}:

$\psi_1 = P_v$: {A},{D}$\rightarrow$ {P};

$\psi_2 = P_v$: {A}, {D}$\rightarrow$ {T}; to be checked on the data.

Rule 2: We Prune supersets of nodes associated with the already discovered XCFDs. Given partition identifiers $W = X \cup \mathcal{C}$ & $Z = W \cup \{Y\}$, and there exists a discovered XCFD $\psi = P_v$: $\{\mathcal{C}_j, \{X\} \rightarrow \{Y\}$ corresponding to edge (W, Z) then we remove the supersets of nodes in the edge(W, Z) from the searching space. The reason is that if ψ holds on data tree D then any extra label adding to the *antecedence* of ψ (e.g. XA) will produce implied XCFD in the form of $\psi' = P_v$: $\{\mathcal{C}_j, \{X\text{A}\} \rightarrow \{Y\}$; ψ' is redundant and is not minimal.

Rule 3: If $\psi = P_v$: $\{\mathcal{C}_j, \{X\} \rightarrow \{Y\}$ is a minimal XCFD then we do not consider $\psi' = P_v$: $\{\mathcal{C}'_j, \{X\} \rightarrow \{Y\}$ where $\mathcal{C}' \supset \mathcal{C}$. This guarantees that the discovered XCFD is minimal. For example, if class w_1 = {22, 32, 52, 62} satisfies an XCFD $\psi = P_{\text{Booking}}$: (.//Carrier = "Qantas", .//Price) $\rightarrow$ (.//Tax); $\mathcal{C}$= (.//Carrier = "Qantas") is the condition then we do not consider class w_1 in subsequence candidates ψ' with condition $\mathcal{C}'$ = (.//Carrier = "Qantas"^.//Arrival = "*BNE*")

Rule 4: If $\psi = P_v$: (X= "*a*")$\rightarrow$(Y= "*b*") corresponding to edge(W, Z) holds on data tree D, this means $|w_a| = |z_b|$ where a, b are constants; X and Y contain only a single data node. If the number of actual occurrences of expression Y = "*b*" in D is equal to the size $|z_b|$ of class z_b then (Y= "*b*")$\rightarrow$(X= "*a*") holds on data tree D. X and Y are called *equivalent sets*, denoted $X \leftrightarrow Y$. In this case we do not check candidates with *antecedence* as Y= "*b*" on the dataset because we can infer from the discovered XCFD ψ. Therefore, if X and Y are equivalent sets, no further XCFDs with *antecedent* containing Y need to be considered.

Rule 5: Given a threshold τ, we only consider nodes associated with class w_i such that $|w_i| > \tau$, e.g. for τ =1, we do not consider classes containing only one element. The constraint associated with such classes is trivial.

In the following, we present the algorithm of XDiscover.

5.5 Algorithm XDiscover

Listing 1 presents our proposed XDiscover algorithm to find XCFDs from a given data tree D. Our algorithm traverses the searching lattice following a breath-first search manner combining with pruning rules described in section 5.4.

The searching process starts from level 1 (l=1); all nodes from E are stored in Partition Identifier PI_1= {v_1, v_2,... v_n} (line 3). Each node in E is a partition identifier with a single label associated with some candidate XCFDs. Partitions of Partition identifiers are generated and stored in GP_l - Generated Partition *(*line 4*)*. At level l > 1, node labels are generated from PI_{l-1} and stored in PI_l (line 7) in the form v_iv_j ; where $v_i \neq v_j$; v_i , $v_j \in PI_{l-1}$; PI_{l-1} contains node labels at level (l-1); all partitions of v_iv_j nodes at level l are generated and stored in GP_l . All candidates in the form c_i,$w_i \rightarrow z_j$ are checked; where v_i= w_ic_i, v_j=$w_ic_iz_j$ and $z_j \in PI_1 \setminus (w_i \cup c_i)$. The validation for a satisfied XCFD follows the approach described in Section 5.3 (line 9: function DiscoverXCFD in Listing 2). The found XCFDs are stored in Discovered set of XCFDs *DF*. Then the Prune function including the pruning rules (in Section 5.4) is performed to prune redundant nodes

and edges from the searching lattice for the next level (line 10). The searching process is repeated until no more partition identifiers are considered (line 5). The output of XDiscover is a set of minimal XCFDs.

Function: XDiscover

Input: XML data tree $D=(V, lab, ele, att, val, r)$ schema tree $S=(E, A, T, root)$

Output: a minimal set of XCFDs

1. DF:= { ∅ };
2. Level l:=1;
3. PI_l:= E;
4. GP_1 = GeneratePartition(D, PI_l);
5. While $|PI_l| \neq$ { ∅ } do
6. l++;
7. PI_l := GeneratePartitionIdentifier(GP_l);
8. GP_l := GeneratePartition(D, PI_l);
9. DF:= DF ∪ **DiscoverXCFD**(GP_l, GP_{l-1});
10. Prune(GP_{l-1});
11. Return (DF).

Listing 1: The XDiscover Function

The function of **DiscoverXCFD** depicted in Listing 2 searches for XCFDs at each level l. If there still exists classes in Π_W which do not belong to any discovered XCFD then we continue to consider such classes with additional condition nodes. DiscoverXCFD calls the **GenerateAdditionPartition** function to calculate partitions with additional condition nodes. The DiscoverXCFD returns XCFDs to XDiscover.

Function: DiscoverXCFD

Input: GP_l, GP_{l-1} // partitions at level l and l-1

Output: satisfied XCFDs

1. DF:= {∅};
2. For each partition of $W \in GP_{l-1}$ do
3. For each partition of $Z \in GP_l$ do
4. If $Z = (W \cup \{Y\})$ then
5. Ω_W:= subsumed w_i;
6. While Ω_W <> {∅} do
7. For each class $w_i \in \Pi_W$ do
8. For each class $z_i \in \Pi_Z$ do
9. If (($|w_i| > \tau$) and ($|w_i| == |z_i|$)) then
10. DF:= DF ∪ $(\mathcal{C}, X \rightarrow Y)$;
11. Ω_W:= $\Omega_W \setminus (w_i \in (\mathcal{C}, X \rightarrow Y))$;
12. If not found XCFD then
13. GenerateAdditionPartition;
14. For each c_i in $\mathcal{C}$ do
15. If c_i contains values only from Ω_W then
16. DF:= DF ∪ $(\mathcal{C}, X \rightarrow Y)$;
17. Ω_W:= $\Omega_W \setminus (w_i \in (\mathcal{C}, X \rightarrow Y))$;
18. Return(DF).

Listing 2: The DiscoverXCFD Function

6 Case studies and comparative evaluation

To evaluate and demonstrate the effectiveness of the proposed approach, we use the Flight Booking XML data for our case studies.

From schema Bookings S in Figure 1, we have E={Bookings, Booking, Carrier, Flight, Departure, Arrival, Price, Tax}. We assume $\tau = 1$, which means we only consider XCFDs holding for more than one data node.

Case 1: **XCFDs contain only constants**.

Suppose the data tree D in Figure 4 conforms to schema Bookings S and each Booking only contains one Flight as shown in D.

Consider edge(W, Z)= edge(CA, CAT) = (Carrier-Arrival, Carrier-Arrival-Tax) in Figure 3. Following the process described in section 5.2 to generate two partitions of Carrier-Arrival and Carrier-Arrival-Tax w.r.t sub-tree rooted at Booking. To simplify the presentation, we omit the node label (e.g. Booking) associated to each node in classes.

$\Pi_{\text{Carrier, Flight/Arrival | Booking}} = \{ w_1, w_2, \mathbf{w_3}, w_4\}$

={{(22,1)}, {(32,1), (52,1)}, {**(42,1), (72, 1)**}, {(62, 1)}}

$\Pi_{\text{Carrier, Flight/Arrival, Tax | Booking}} = \{ z_1, z_2, \mathbf{z_3}, z_4, z_5\}$

={{(22,1)}, {(32,1)}, {**(42,1), (72,1)**}, {(52,1)}, {(62,1)}}

We can see that w_3 in $\Pi_{\text{Carrier, Flight/Arrival | Booking}}$ is *equivalent* to z_3 in $\Pi_{\text{Carrier, Flight/Arrival, Tax | Booking}}$. That is, $w_3 = z_3$ ={(42,1), (72,1)}. Nodes in w_3 have the same value of Carrier= *"Virgin"* and Arrival= "BNE". Nodes in z_3 share the same value of Tax = "*20*". An XCFD is discovered:

$\psi_1 = P_{\text{Booking}}$: (.//Carrier = "*Virgin*" ^ .//Flight/Arrival = "*BNE*") → (.//Tax = "*20*").

This case demonstrates the XCFD contains only constants. For each XFD, there might exist a number of conditional dependencies XCFDs which refine this XFD by binding particular values to elements in its specification. Such constraints cannot be expressed by using the XFD notion.

Case 2: **XCFDs contain both variables and constants.**

Using the same assumption in case 1, considering edge (W, Z)= edge(P, PT) = (Price, Price-Tax) in Figure 3, two partitions of Price and Price-Tax w.r.t the sub-tree rooted at Booking:

$\Pi_{\text{Price | Booking}} = \{w_1, w_2\}$={{(22, 1), (32, 1), (72, 1)}, {(42, 1), (52, 1), (62, 1)}}

$\Pi_{\text{Price, Tax | Booking}} = \{ z_1, z_2, z_3, z_4\}$={{(22,1), (32,1)}, {(42,1)}, {(52, 1), (62, 1)}, {(72, 1)}}

There does not exist any equivalent pair between two partitions $\Pi_{\text{Price | Booking}}$ and $\Pi_{\text{Price,Tax | Booking}}$. We need to add more data nodes from the remaining set of E\{$W \cup Z$}. For example, the node of .//Carrier can be added to edge(P, PT) as a conditional data node. We now consider edge(W', Z')= edge(CP, CPT) = (Carrier-Price, Carrier-Price-Tax). Partitions of Carrier-Price and Carrier-Price-Tax w.r.t sub-tree rooted at Booking are as follows:

$\Pi_{\text{Carrier, Price | Booking}}$ = {***w'1, w'2, w'3, w'4***}={{**(22,1), (32, 1)**}, {(42,1)},{**(52, 1), (62, 1)**}, {(72, 1)}}

$\Pi_{\text{Carrier, Price, Tax | Booking}}$ = { ***z'1, z'2, z'3, z'4***}={{**(22, 1), (32, 1)**}, {(42, 1)}, {**(52, 1), (62, 1)**}, {(72, 1)}}

The partition of the condition node .//Carrier:

$\Pi_{\text{Carrier | Booking}}$ = { c_1, c_2, c_3}={{**(22, 1), (32, 1), (52, 1), (62, 1)**}, {(42, 1)},{(72, 1)}}

We have two equivalent pairs *(w'1, z'1)* and *(w'3, z'3)* between $\Pi_{\text{Carrier, Price | Booking}}$ & $\Pi_{\text{Carrier, Price, Tax | Booking}}$ with $|w_1|=2$ and $|w_3|=2 > \tau$. Furthermore, there exists a class c_1 in $\Pi_{\text{Carrier|Booking}}$ containing exactly all elements in $\Omega_{w'}= w'_1 \cup w'_3$. All elements in class c_1 have the same value at Carrier = "*Qantas*". This means nodes in classes w'_1 and w'_3 share the same condition (.//Carrier = "*Qantas*"). Therefore, an XCFD ψ = P_{Booking}: (.//Carrier= "*Qantas*", .//Price) →.//Tax is discovered.

Case 2 illustrates techniques to find an XCFD with extra data nodes which are referred to as the condition of the XCFD. Such XCFDs contain both variables and constants.

Case 3: **Partition identifiers contain a set of complex nodes.**

Suppose data tree *D* in Figure 5 conforms to schema Bookings *S* in Figure 1. Each Booking contains multiple complex nodes Flight.

For partition *identifiers* containing a set of complex data nodes, the calculating partitions are processed in a bottom-up fashion. We first consider the sub-tree rooted at the bottom level in the data tree (e.g Flight) to calculate partitions. Then we convert all classes in each generated partition into the corresponding parent of this complex node (i.e., the parent of Flight is Booking) to find the refinement. We repeat converting the found partition to obtain its refinement until reaching the sub-tree rooted at the considered nodes (i.e., Booking). The validation for a satisfied XCFD is similar to the cases which deal with the partition identifier which contains single data nodes.

Consider edge(*W, Z*)= (Flight, Tax) w.r.t. sub-tree rooted at Booking, we start generating partitions under the sub-tree rooted at Flight. Following the process described in section 5.2, we partition the nodes according to each Flight (including Departure and Arrival) under the sub-tree rooted at Flight:

$\Pi_{\text{Flight/Departure, Flight/Arrival | Flight}}$ = {{(104, 2), (124, 2)}, {(107, 2), (127, 2)}}

Then converting these classes into the Booking sub-tree, we have a refinement: $\Pi_{\text{Flight | Booking}}$ = {{(102, 1), (122, 1)}}. Validating for a satisfied XCFD is done similarly to case which partition identifiers contain only single data nodes. The discovered XCFD is represented in the form:

ψ_2=P_{Booking}:(.//Carrier="*Virgin*",{.//Flight})→(.//Tax); where {.//Flight} represents a set of complex data nodes Flight (including Departure and Arrival).

In case there is only one Flight node in the constraint, the XCFD can be represented as:

ψ'_2 = P_{Booking}: (.//Carrier = "*Virgin*", .//Flight)→ (.//Tax),

ψ'_2 is a special case of ψ_2. Generally, a partition identifier containing simple nodes is a special case of the partition identifier containing complex nodes. Therefore, we apply the same process to deal with the partition identifiers which contain complex nodes for both cases.

Comparative Evaluation: There does not exist any approach to discover Conditional Functional Dependencies in XML data. The only algorithm for discovering XFDs (Yu and Jagadish 2006) is close to ours. However, the existing algorithm discovers XFDs containing only variables and can not detect for dependencies which hold partially on documents with conditions. Our approach discovers constraints containing both variables and constants, or either variables or constants that allows the detection of more interesting semantic constraints than algorithms to discover XFDs.

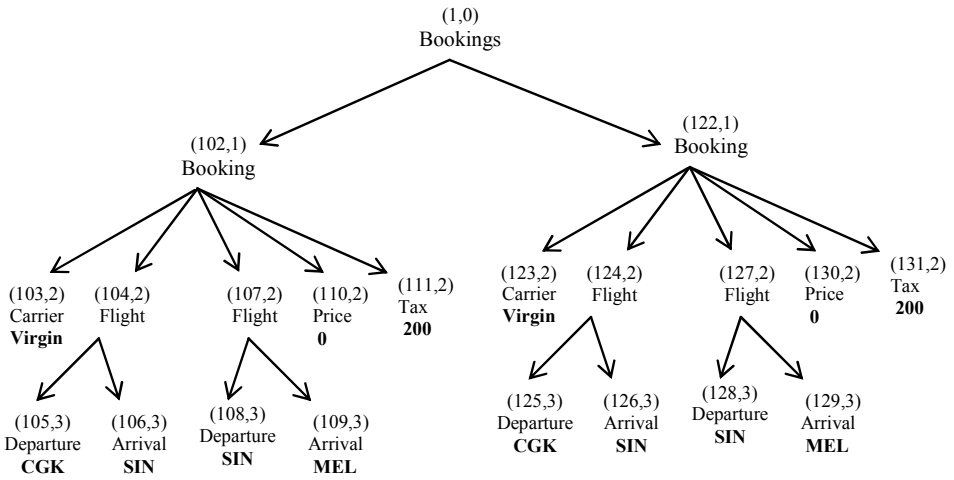

Figure 5: An example of Bookings **data tree:** *each* Booking *contains a set of complex element* Flight.

7 Conclusion

This paper addressed the issues of XML data inconsistency. We pointed out the limitations of existing work in handling such problems, and introduced the notion of XML Conditional Functional Dependency which incorporates conditions into data constraints. We proposed the XDiscover algorithm to detect a set of possible XCFDs on a given XML data instance. Our approach employs the set of pruning rules to reduce the searching space and the number of XCFDs to be checked on the dataset. Our approach can be used to enhance data quality management by suggesting possible rules and identifying non-compliant data. Discovered XCFDs also can also be embedded into an enterprise's systems as an integral part to support manipulating data. XML data changes very often which may lead to a corresponding change in the semantics of data constraints. Therefore, our work can be further extended to address the problem of data evolution.

References

Arenas, M. (2006): Normalization Theory for XML. *SIGMOD Record,* 35**,** 57-64.

Bohannon, P., Fan, W., Geerts, F., Jia, X. and Kementsietsidis, A. (2007): Conditional Functional Dependencies for Data Cleaning. *ICDE.*

Chiang, F. and J.Miller, R. (2008): Discovering Data Quality Rules. *VLDB.*

Fan, W. (2005): XML Constraints: Specifications, Analysis, and Application. *Database and Expert Systems Applications***,** 805- 809.

Fan, W., Geerts, F., Lakshmanan, L. V. S. and Xiong, M. (2009): Discovering Conditional Functional Dependencies. *IEEE.*

Flesca, S., Furfaro, F., Greco, S. and Zumpano, E. (2003):Repairs and Consistent Answers for XML Data with Functional Dependencies. *Xsym.*

Flesca, S., Furfaro, F., Greco, S. and Zumpano, E. (2005):Querying and Repairing Inconsistent XML Data. *WISE*

Golab, L., Karloff, H. and Korn, F. (2008): On generating Near-Optimal Tableaux. *PVLDB*.

Grahne, G. and Zhu, J. (2002): Discovering Approximate keys in XML data. *CIKM'02***,** 453-460.

Hartmann, S. and Link, S. (2003): More Functional Dependencies for XML. *LNCS 2798***,** 355-369.

Huhtala, Y., Karkkainen, J., Porkka, P. and Toivonen, H. (1999): TANE: an Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The computer journal,* 42**,** 101-111.

Lampathaki, F., Mouzakitis, S., Gionis, G., Charalabidis, Y. and Askounis, D. (2008): Bussiness to Bussiness interoperability: A current review of XML data integration standards. *Computer Standards & Interfaces***,** 1045-1055.

Lv, T. and Yan, P. (2006): XML Constraint-tree-based Functional Dependencies. *ICEBE.*

Tan, Z., Zhang, Z., Wang, W. and Shi, B. (2007): Consistent data for inconsistent XML document. *Information and Software Technology.*

Trinh, T. (2008): Using Transversals for Discovering XML Functional Dependencies. *LNCS 4932.*

Vincent, M. W., Liu, J. and Liu, C. (2004): Strong Functional Dependencies and Their Application to Normal Forms in XML. ACM Transactions on Database Systems.

W3C (2004): XML Schema. http://www.w3.org/TR/xmlschema-0/. Accessed 20 Oct 2010

Yu, C. and Jagadish, H. V. (2006): Efficient Discovery of XML Data Redundancies. VLDB.

Yu, C. and Jagadish, H. V. (2008): XML Schema refinement through redundancy detection and normalization. VLDB