# STRUCTURED CONTENT-AWARE DISCOVERY FOR IMPROVING XML DATA CONSISTENCY

*Submitted by*

**THI HONG LOAN VO**

A thesis submitted in total fulfillment
of the requirements for the degree of
Doctor of Philosophy

School of Engineering and Mathematical Sciences
Faculty of Science, Technology and Engineering

La Trobe University
Bundoora, Victoria 3086
Australia

October 2013

# Contents

# List of Tables

**LIST OF TABLES**

# List of Figures

# Lists

# Acknowledgements

I would especially like to thank the following people.

- First of all, I would like to thank Dr. Jinli Cao for her endless support. I sincerely appreciate her contribution of time, guidance, caring help, and advice during the fourth years of my Ph.D. study at La Trobe University. I also thank her for being very patient with my progress.
- I wish to express my gratitude to my second co-supervisor, Associate Professor Wenny Rahayu, for her support and encouragement in relation to my research in general. She provided very helpful comments and ideas on my work. She always supported me whenever I needed her most.
- I owe a very special thank to my good friends, Dr Hong-Quang Nguyen from International University, Vietnam National University, Dr Thi Ngoc Nguyen from National University of Singapore and Dr. Hai Thanh Do for their tremendous support to me. They provided me with insightful ideas, shared valuable tips on how to improve my writing skills and how to present technical materials and gave very helpful comments on my published papers.
- I would like to thank the chair of my research panel Dr Torab Torabi and Dr Fei Lui for participating in my thesis committee and

providing helpful feedback for every stage of my Ph.D. I also thank Ms. Michele Mooney for her careful proof reading of my research papers and the final draft of this thesis.

- I would like to express my gratitude and love to my family for always being there whenever I needed them most, and for supporting me throughout all my thesis years. I would like to thank my parents for their continuous love and support. I would like to thank my husband Phuc Duat Phan, for his constant love, care and encouragement.

# Abstract

With the explosive growth of heterogeneous XML sources, data inconsistencies have become a serious problem, resulting in ineffective business operations and poor decision-making. XML Functional Dependencies (XFDs) are well known as essential semantics to enforce the data integrity of a source. However, existing approaches to XFDs have insufficiently addressed data inconsistencies arising from both semantic and structural inconsistencies inherent in heterogeneous XML data. In this thesis, we address such prevalent inconsistencies by proposing *XDiscover*, *SCAD* and *SC2QA* approaches.

XDiscover is a content-based discovery approach which explores the semantics hidden in data to discover a set of minimal *XML conditional functional dependencies* (XCFDs) from a given source to address semantic inconsistencies. The XCFD notion is extended from XFDs by incorporating conditions into XFD specifications. The experimental results on the synthetic and real datasets and the results from the case studies show that XDiscover can discover more dependencies and the dependencies found convey more meaningful semantics, in terms of capturing data inconsistency, than those of the existing XFDs.

SCAD is a structured and content-aware approach which explores the semantics of data structures and the semantics hidden in the data values to discover a set of *XML conditional structural functional dependencies* (XCSDs) from a given source to address the inconsistencies caused by both

structural and semantic inconsistencies. XCSDs are path and value-based constraints, whereby: (*i*) the paths in XCSD approximately represent groups of similar paths in sources to express constraints on objects with diverse structures; while *(ii)* the values bound to particular elements express constraints with conditional semantics. We conduct experiments and case studies on synthetic datasets which contain structural diversity and constraint variety causing XML data inconsistencies. The experimental results show that SCAD can discover more dependencies and the dependencies found can capture data inconsistencies disregarded by XFDs.

SC2QA utilizes XCSDs to compute customized consistent query answers for queries posted to inconsistent data sources to improve information quality. The query answer is calculated by qualifying queries with appropriate information derived from the interaction between the query and the XCSDs. We conduct experiments on synthetic datasets to evaluate the effectiveness of SC2QA.

# Statement of Authorship

Except where reference is made in the text of this thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis submitted for the award of any other degree or diploma.

No other person's work has been used without due acknowledgment in the main text of the thesis.

This thesis has not been submitted for the award of any degree or diploma in any other tertiary institution.

Thi Hong Loan Vo

*Date*. 16 October 2013

# External Refereed Publications

The results of this thesis have been published in or under reviewed by the following journals and proceedings:

**Vo, L.T.H**., Cao, J., Rahayu, W. and Nguyen, H.-Q. Structured content-aware discovery for improving XML data consistency. *Information Sciences*, 248(1): 168-190, 2013.

**Vo, L.T.H**., Cao, J. and Rahayu, W. Discovering Conditional Functional Dependencies in XML Data. *Australasian Database Conference*, 143-152, 2011.

**Vo, L.T.H**., Cao, J. and Rahayu, W. Structured content-based query answer for improving information quality. *World Wide Web*, under accepted, Jan 2014.

# Chapter 1

# Introduction

The main theme of this thesis is to study XML data consistency. This chapter consists of five sections. Section 1.1 highlights the need to introduce new types of constraints and proposes approaches to discover anomalies in XML data. Requirements to address data inconsistency are also discussed in this section as the motivation for this work. Section 1.2 presents the definitions of the problems which are resolved in this thesis. Section 1.3 briefly introduces our approaches to resolve the identified problems. Section 1.4 summarizes the main contributions of the thesis. The thesis organization is outlined in section 1.5.

## 1.1 Motivation

*Extensible Markup Language* (XML) has emerged as the standard data format for storing business information in organizations [6]. Data in these environments are rapidly changing and highly heterogeneous. This has increasingly led to the critical problem of data inconsistency in XML data because the semantics underlying business information, such as business rules, are enforced insufficiently [58]. XML itself only support for creating

markup languages used as metadata, it does not guarantee how the underlying business information must be structured and expressed in business processes. Data inconsistency appears as violations of constraints defined over a dataset [43, 80] which, in turn, leads to inaccurate data interpretation and analysis [47, 68]. Such problems significantly affect the ability of the system to provide correct information causing inefficient business operations and poor decision making. XML functional dependencies (XFDs) [6, 42, 52, 82, 83] have been proposed to increase the data integrity of the sources. Unfortunately, existing approaches to XFDs are insufficient to completely address the data inconsistency problem to ensure that the data is consistent within each XML source or across multiple XML sources for three main reasons. First, XFDs are defined to represent constraints globally enforced to the entire document [6, 82], whereas XML data are often obtained by integrating data from different sources constrained by local data rules. Thus, they are unable, in some cases, to capture conditional semantics locally expressed in some fragments within an XML document.

Second, the existing XFD notions are incapable of validating data consistencies in sources with diverse structures. This is because checking for data consistency against an XFD requires objects to have perfectly identical structures [82], whereas XML data is organized hierarchically allowing a certain degree of freedom in the structural definition. Two structures describing the same object may not be identical [75, 94, 95]. In such cases, using XFD specifications cannot validate data consistency. Third, existing approaches to XFD discovery focus on structural validation rather than semantic validation [11, 42, 82, 91]. Most existing work on constraint discovery only extracts constraints to solely address data redundancy and normalization [81, 102]. Such approaches cannot identify anomalies to discover a proper set of semantic constraints to support data

inconsistency detection. To the best of our knowledge, there is currently no existing approach which fully addresses the problems of data inconsistency in XML data. Such limitations in prior work are addressed in this thesis.

In the next section, we present certain technical terms relating to data consistency which are necessary to understand the remainder of the thesis.

### 1.1.1 Data consistency

*Consistency* is a data quality dimension capturing the violation of semantic rules defined over a dataset. Integrity constraints are instantiations of such semantic rules which are dependencies typically defined to ensure schema quality [15]. They are properties which must be satisfied by all instances of a database. Data inconsistency describes a source which does not respect one or more constraints defined over a dataset. For example, a condition could be that, in every instance, the customer name (CName) functionally depends on the customer ID (CId), i.e., a customer ID is assigned to, at most, one customer name. This integrity constraint is a functional dependency (FD) denoted as CId → CName, indicating that this dependency should hold for the attributes of the Customer relation. The data in Fig 1.1 is inconsistent with respect to the above FD. This is because the customer ID of "C01" is assigned to two different customer names which violates the above functional dependency.

| CId | CName |
|---|---|
| C01 | Mary |
| C01 | Bob |
| C02 | Clayton |

**Fig 1.1** An simplified inconsistent instance of Customer relation

In XML data, the satisfaction of a source to a set of integrity constraints often cannot be guaranteed, hence, data inconsistency occurs

[43, 80]. Data inconsistency is often caused by *semantic inconsistency* and *structural inconsistency*. Semantic inconsistencies occur when business rules on the same data vary across different fragments [79]. Structural inconsistencies arise when the same real world concept is expressed in different ways, with different choices of elements and structures, that is, the same data is organized differently [75, 95]. In this work, we define integrity constraints for instances calling them constraints. Such constraints are defined based on either the actual data content or data structures to enhance the data consistency within an XML data source. By *data consistency*, we mean that the source syntactically and semantically satisfies a set of constraints.

In the next section, we discuss the essential features about which constraints are required to have so that they can prevent data inconsistencies in XML.

### 1.1.2 Requirements of constraint specifications

Constraints are essential parts of data semantics used to define the criteria that a data source should satisfy. Commonly, the validation of XML data often focuses on the schema level with respect to predefined constraints expressed in the form of schema [5, 6, 11, 82]. However, XML data are often integrated from different data sources, and while there are certain features shared by all data, each fragment might need to maintain certain constraints differently to suit its unique requirements [91]. The existence of various constraints holding on the same object across different fragments causes inconsistencies at the semantic level. In such cases, an additional validation from the content view with respect to different constraints *holding conditionally* on the data is necessary to maintain data consistency.

By holding conditionally, we mean that each constraint holds on a subset of the data specified by an accompanying condition.

In addition to semantic inconsistencies, structural inconsistencies also pose additional challenges to enhance the data consistency. Structural inconsistencies are often caused by the existing various data structures representing the same object. That is, XML data can contain data from different data sources which might contain either nearly, or exactly the same information, but they are represented by different structures. Moreover, even though two objects express similar content, each of them may contain some extra information. In such cases, constraints on XML data should be allowed to hold on similar objects. In summary, in order to ensure the data consistency, constraints not only need to define the data-value bindings to express conditional semantics, but should also be flexible enough to describe the similarity of objects. As far as we are concerned, there is no prior work proposing such constraints to validate data consistency from both structural and content views. We suggest that such constraints should be maintained to preserve the data consistency of applications supported by XML data.

From the requirements of constraint specifications, we now discuss the requirements that discovery approaches should take into account to explore a proper set of constraints to address data inconsistency arising from both semantic and structural inconsistencies in XML data.

### 1.1.3 Requirements of constraint discovery

As XML data becomes more common and its data structures more complex, it is desirable to have algorithms to automatically discover anomalies from XML data sources. Although there is existing work [4, 102] on discovering constraints, there still exist certain limitations and

problems which remain completely unsolved. Existing work cannot explore a proper set of constraints to address data inconsistency. The Apriori algorithm [4] and its variant approaches [13, 61, 71, 84] are well known for discovering association rules, which are associations amongst sets of items; however, such rules contain only constants. By contrast, XML functional dependencies discovered by the work in [102] contain only variables which are solely defined on a structural level. Existing work cannot detect constraints occurring in the data which should be maintained to ensure data consistency. In order to discover such constraints, the discovery process has to convey semantics from both structures and data content. This thesis generalizes the existing techniques relating to association rule [4] and functional dependency discovery [53, 70, 102] to discover the constraints containing either variables or constants. They are constraints defined on a data level. We discuss the features which a system should consider to manage data consistency in XML data in the next section.

### 1.1.4 Consistent data management

The problem of data consistency management in inconsistent data has been widely studied in the database community. Consistent data is formally obtained following two approaches including data repair and consistent query answers [9]. Data repair is to find consistent parts of an inconsistent data source with respect to predefined constraints and minimally differs from the original one [9, 79]. The inconsistent source is often first transformed, by means of deletions or additions, into a consistent one which is then used for calculating query answers [25]. However, repairing data might also result in side effects, for example it could cause incorrect answers to queries and it does not always remove inconsistencies completely. Restoring consistency in an inconsistent data might also be a

computationally complex and non-deterministic process. Moreover, one of the main goals of a database system is to compute answers to queries [47]. This means that finding consistent query answers is more important than repairing data. Hence, it is preferable to leave the data inconsistencies to avoid losing information due to the data repair and instead, manage the potential inconsistencies in answers to queries posted to that source, that is, finding the parts of data which are consistent in query answers. The consistent answer to a query is defined as the common parts of answers to the query on all possible repairs of the data source [43, 45, 76]. XML data is often inconsistent with respect to a set of constraints. Therefore, constraints should be taken into account along with the data source in the process of computing query answers. This thesis addresses the issue of computing consistent answers for queries posted to an inconsistent XML source with respect to a set of constraints.

Focusing on the requirements discussed above, this thesis resolves a number of issues, which can be grouped into three major problems described in the following section. The first two problems involve constraint discovery and the third problem concerns consistent query answers.

## 1.2 Problem definition

The problems of data consistency in relational databases have been extensively studied [27, 31, 36, 38, 39, 40]. This thesis extends this work to XML data. We propose approaches to discover a proper set of constraints used to ensure data consistency in XML data. Constraint discovery can be divided into two problems. The first problem is to deal with a case where a data source conforms to a schema. We only need to discover anomalies caused by semantic inconsistencies. The second problem is a case where a given data source does not follow any schema. The data source is designed

with great flexibility in both data structures and semantics. In such cases, we focus our attention on anomalies arising from both structural and semantic inconsistencies. Two problems can be formulated as follows:

Problem 1*:* "*Given an XML data tree T conforming to a schema S, discover a set of non-redundant XML conditional functional dependencies (XCFDs), where each XCFD is minimal and contains only a single element in the consequence*". The task of constraint discovery only relates to the data content referred to as resolving semantic inconsistencies.

Problem 2*:* "*Given an XML data tree T, discover a set of minimal XML conditional structural functional dependencies (XCSDs), where each XCSD is minimal and contains only a single element in the consequence*". The task of constraint discovery is based on both data content and data structures. The discovery approach handles both data structural and semantic aspects which are referred to as resolving structural and semantic inconsistencies.

In addition, our proposed constraints are applied to compute customized consistent query answers for queries posted to inconsistent XML data. The problem can be formulated as follows:

Problem 3: "*Given an XML data tree T and a set of XCSDs, find a customized consistent answer for query Q posted to tree T"*. The task is to find consistent answer for the query posted to an inconsistent data source with respect to a given set of XCSDs.

The solutions to problems 1, 2 and 3 are in chapter 3, 4, and 5, respectively. We believe that our research is especially relevant nowadays, since a huge amount of data is being exchanged between organizations

using XML data in which it is very difficult to avoid anomalies. In the next section we present an overview of our approaches.

## 1.3 Overview of our approaches

We propose three different approaches, called XDiscover, SCAD and SC2QA to address the three problems defined above, respectively. First, we propose a new XDiscover approach to discover a set of XML conditional functional dependencies (XCFDs) from a given XML data source conforming to a schema. XCFDs are extended from XFDs by incorporating conditions into XFD specifications. The XDiscover is based on semantics hidden in the data to discover constraints. It includes three main functions, named *search lattice generation, candidate identification,* and *validation*.

The search lattice generation is used to generate a search lattice containing all possible combinations of elements in the given schema. The candidate identification is used to identify possible candidates of XCFDs. The identified candidates are then validated by the validation function, to discover satisfied XCFDs. Validation for a satisfied XCFD includes two steps. First, partitions for node-labels associated with each candidate XCFD are calculated based on data values coming with that node-label. Then, the satisfaction of that candidate XCFD is checked, based on the notion of partition refinement [53]. The number of candidate XCFDs and the searching lattice are very large. Therefore, we propose five pruning rules used to remove redundant and trivial candidates from the search lattice in order to improve the performance of XDiscover. The first three rules are used to skip the search for XCFDs that are logically implied by the already found XCFDs. The last two rules are to prune redundant and trivial XCFD candidates. Adoptions of Armstrong's Axioms and closure set [12] are used to prove the correctness of our proposed pruning rules and the

completeness of the set of XCSDs discovered by XDiscover. The experimental results on synthetic and real datasets, and results from case studies show that XDiscover can discover more dependencies and the dependencies found convey more meaningful semantics, in terms of capturing data inconsistency, than those of the existing XFDs.

Second, we observe that it cannot be an assumption that each XML document has a schema defining its structure for two main reasons. First, the flexible nature of XML allows the representation of different kinds of data from different data sources. Second, if a schema exists, each source might follow its own structural definitions through multiple modifications. As a result, the problems of structural inconsistencies cannot be avoided. Therefore, in our second contribution, we propose a structured and content-aware approach, called SCAD, to discover XML conditional structural functional dependencies (XCSDs) from a given data source to address inconsistencies caused by both structural and semantic inconsistencies in XML data. The input to SCAD is an XML data source which does not associate to any schema. XCSDs are path and value-based constraints; the paths in XCSDs approximately represent groups of similar paths in sources to express constraints on objects with diverse structures, and the values bound to particular elements express constraints with conditional semantics. The SCAD approach consists of two phases: *resolving structural inconsistencies* and *resolving semantic inconsistencies*.

In the first phase, a process, called *data summarization*, analyses the data structure to construct a data summary containing only representative data for the discovery process. This aims to avoid returning redundant data rules due to structural inconsistencies. In the second phase, the semantics hidden in the data summary are explored by a process called XCSD Discovery to discover XCSDs. The XCSD discovery algorithm works in the same manner as XDiscover. The main difference is that instead of

discovering constraints from the given data tree as in XDiscover, SCAD discovers non-trivial XCSDs from the constructed data summary. We conducted experiments and case studies on synthetic datasets which contain structural diversity and constraint variation, causing XML data inconsistencies. The experimental results show that SCAD can discover more dependencies than XFD approaches. The dependencies found could capture data inconsistencies disregarded by XFDs.

Third, we show that the answers of queries might be inaccurate when queries are posted to inconsistent XML data. We utilize our proposed XCSDs to compute answers for queries posted to inconsistent source to improve information quality. In particular, we propose an approach called SC2QA, which integrates the semantics of XCSDs into the query process to find consistent data in inconsistent data. The answer is calculated by qualifying a query with appropriate information derived from the interaction between the query and the XCSDs. Especially, the similarity threshold in XCSDs is used to specify the similar objects which are considered to be qualified for queries. Conditions in XCSDs are used to find candidate objects for calculating query answers. The original data is evaluated at each constraint to find the consistent data.

A customized consistent query answer (CCQA) is calculated from true answers in terms of the structural similarity and consistent data with respect to XCSDs. To evaluate SC2QA, experiments were conducted on synthetic datasets containing structural diversity and constraint various causing XML data inconsistencies. The results show SC2QA work more efficiently for constant XCSDs than variable XCSDs (i.e. XFDs). Query answers found by utilizing constant XCSDs are more accurate than that of XFDs. We summarize our main contributions in this thesis in the next section.

## 1.4 Contributions

This thesis addresses the problems of data inconsistency in XML data to improve data consistency. The focus is on discovering constraints from a given XML data source. The key principle used in our approaches is the concept of structure and content awareness. Our approaches have been shown to be superior to other proposed XFD approaches. In addition, we utilize our proposed constraints to compute query answers for queries posted to an inconsistent data source. To summarize, the contributions of this thesis are as follows:

- the introduction of XML conditional functional dependencies (XCFDs);
- the proposal of the XDiscover approach to discover XCFDs to address semantic inconsistencies;
- the introduction of XML conditional structural functional dependencies (XCSDs);
- the proposal of a structural similarity technique to measure the similarity between sub-trees;
- the proposal of the SCAD approach to explore XCSDs to address both semantic and structural inconsistencies;
- proposing the SC2QA approach to compute customized consistent answers for queries posted to inconsistent XML data with respect to a set of XCSDs.

## 1.5 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 reviews prior work on constraints. The topics covered are *(i)* XML database, *(ii)* conditional functional dependency, *(iii)* association rules, *(iv)* different proposals of XML functional

dependencies (XFDs) and *(v)* management of data consistency in inconsistent data sources.

- Chapter 3 presents our proposed XDiscover approach. XDiscover is used to discover XML conditional functional dependency from a given source to address semantic inconsistency in XML data.
- Chapter 4 presents our proposed approach, called SCAD, to discover XML conditional structural functional dependency from a given source. This is to address data inconsistency arising from structural and semantic inconsistencies in XML data.
- Chapter 5 presents our proposed SC2QA approach which is used to compute customized consistent query answers for queries posted to an inconsistent XML source with respect to a set of XCSDs.
- Chapter 6 concludes the thesis and describes our immediate future work.

It is worth mentioning that the results of this thesis appeared in the following publications: the results of Chapter 3 appeared in [85], the results of Chapter 4 appeared in [87] and the results of Chapters 5 appeared in [86].

# Chapter 2

# Related work

This chapter reviews existing work relating to the work in this thesis and is divided into five sections. Section 2.1 presents a brief background on XML databases. Section 2.2 reviews conditional functional dependency which has been extensively studied for improving data consistency in relational databases. Section 2.3 discusses the notion of association rules and its mining algorithms. The association rules are partially related to the specification of our proposed constraints. Section 2.4 discusses different proposals of XML functional dependencies (XFDs) and XFD discovery approaches. Section 2.5 reviews existing approaches to manage data consistency in inconsistent data sources. The final section is a summary of this chapter. Note that additional background specific to each problem is covered in the relevant chapter.

## 2.1 XML database

In this section, we present some background information on XML databases, including definitions of document types and XML data. As in the case of relational databases, a schema is defined to specify the structure of a class of XML documents. There are two predominant proposals to

define the schema: DTD (Document Type Definition) [54] and XML Schema [88]. Even though DTDs are less expressive than XML Schema specifications, in general they are expressive enough for a variety of applications [19]. Therefore, in this thesis, we consider only DTDs. The specification of a DTD is described in the next section.

### 2.1.1 Document Type Definition

A Document type definition (DTD) has a start-tag, which is called the root of the document and is specified by the DOCTYPE declaration. Elements in XML instances are declared by ELEMENT tags. Each element might be followed by one element or an arbitrary number of elements. Fig 2.1 is an example about a DTD for Bookings data, which specifies a nonempty collection of Bookings. <Booking> is an element since <!ELEMENT Booking (Carrier, Trip+, Fare, Tax)> (line 3) appears in the DTD. Each Booking has one Carrier and an arbitrary number of <Trip>,

```
1.  <!DOCTYPE Bookings [
2.  <!ELEMENT Bookings (Booking+)>
3.  <!ELEMENT Booking (Carrier, Trip+, Fare, Tax)>
4.  <!ATTLIST Booking bno CDATA #REQUIRED>
5.  <!ELEMENT Carrier (#PCDATA)>
6.  <!ELEMENT Trip (Departure, Arrival)>
7.  <!ELEMENT Departure (#PCDATA)>
8.  <!ELEMENT Arrival (#PCDATA)>
9.  <!ELEMENT Fare (#PCDATA)>
10. <!ELEMENT Tax (#PCDATA)>
11. ]>
```

**Fig 2.1.** An example of DTD

followed by one <Fare> and one <Tax> element. An ELEMENT declaration also specifies the sub-elements of an element by means of a regular expression. For instance, <!ELEMENT Trip (Departure, Arrival)> (line 6) indicates that the sub-elements of <Trip> have other sub-elements including one <Departure> and one <Arrival> element. #PCDATA is used to indicate elements containing text, such as <!ELEMENT Departure (#PCDATA)> (line7). An ATTLIST declaration is used to specify the attributes of an element, such as <!ATTLIST Booking bno CDATA #REQUIRED> (line 4).

### 2.1.2 XML data

XML documents are widely used to store data [2]. Fig 2.2 is an example of an XML document storing information about Bookings which is an instance of the Booking DTD in Fig 2.1. Each <Booking> element has a Booking number (bno), name of Carrier and information on Trip, Fare, and Tax. Each Trip contains information on Departure and Arrival. The document contains two different types of tags: start-tags, such as <Bookings> and end-tags, such as </Bookings>. These tags must be balanced and are used to delimit elements, for example, <Carrier> Qantas </Carrier>. Every element can contain attributes, other elements, text, or a mixture of them. For instance, **<**Booking bno**="**b1**">**, the <Booking> element contains attribute bno with a value of "b1"; **<**Carrier**>** Qantas **</**Carrier**>** shows that the <Carrier> element contain text of "Qantas"; **<**Trip**> <**Departure**>** BNE **</**Departure**> <**Arrival**>** MEL **</**Arrival**> </**Trip**>** says that the element <Trip**>** contains other elements including Departure and Arrival. An XML DTD or an XML document can be represented as a schema tree or a data tree, respectively.

Fig 2.3 is a representation of the Bookings data tree. In the next section, we discuss conditional functional dependencies (CFDs) which have been extensively studied to improve data consistency in relational databases and highlight the challenges associated with employing such approaches to XML data.

```
<Bookings>
  <Booking bno="b1">
        <Carrier> Qantas </Carrier>
        <Trip>
              <Departure> BNE </Departure>
              <Arrival>MEL</Arrival>
        </Trip>
        <Fare> 200 </Fare>
        <Tax> 40 </Tax>
 </Booking>
<Booking bno="b2">
         <Carrier> Qantas </Carrier>
        <Trip>
              <Departure> PER </Departure>
              <Arrival>MEL</Arrival>
        </Trip>
        <Trip>
              <Departure> MEL </Departure>
              <Arrival>BNE</Arrival>
        </Trip>
        <Fare> 350 </Fare>
        <Tax> 80 </Tax>
</Booking>
</Bookings>
```

**Fig 2.2.** An example of an XML document

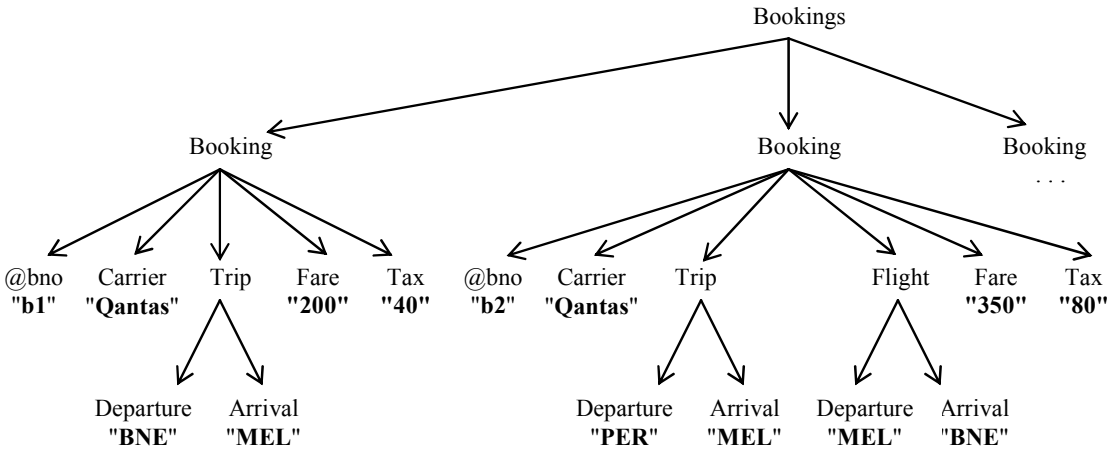


**Fig 2.3.** An example of data tree

## 2.2 Conditional functional dependency

Traditionally, constraints are introduced to improve the quality of schema, such as defining normal forms based on functional dependencies [11]. Recently, constraints have been extensively studied to address the problems of the quality of data, especially data consistency. Conditional Functional Dependencies (CFDs) [20, 31, 36, 38, 40, 41, 100] have been widely used as a technique to detect and correct non-compliant data to improve data consistency while other approaches [27, 39, 48] have been proposed to

| CAR | DEP | ARR | FA | TA |
|---|---|---|---|---|
| Virgin | MEL | SYD | 200 | 50 |
| Virgin | BNE | SYD | 300 | 50 |
| Qantas | MEL | SYD | 300 | 50 |
| Qantas | MEL | BNE | 400 | 100 |
| Qantas | MEL | DRW | 250 | 100 |

**Fig 2.4.** An instance of the Bookings relation

automatically discover CFDs from data instances. A CFD consists of a standard functional dependency (FD) and a pattern tableau specifying the scope of the FD on the data. Given an instance $D$ on a relation schema $R$, a CFD $\partial$ on $R$ is represented as $\partial$: ($X \rightarrow Y$, $T_p$), where $X$ and $Y$ are attribute sets in $R$, $X \rightarrow Y$ is a standard FD, $T_p$ is a pattern tableau of $\partial$ containing all attributes in $X$ and $Y$. For each attribute $A \in (X \cup Y)$, the value of A for $T_p$ is either a value in dom($A$) or a variable value. For example, considering a relation Bookings(CAR, DEP, ARR, FA, TA) specifies the Booking information including Carrier (CAR), Departure (DEP), Arrival (ARR), Fare (FA) and Tax (TA). Fig 2.4 shows an instance of the Bookings relation. Data rules on Bookings can be defined in the forms of CFDs as follows:

$\partial_1$: [ARR= "SYD"] →[TA="50"]

$\partial_2$: [CAR= "Qantas", DEP, ARR] →[TA]

$\partial_1$ states that the functional dependency ARR→TA holds in the context where the value of ARR is "SYD" and the value of TA is "50". $\partial_2$ assumes that the functional dependency DEP, ARR →[TA] only holds in the context where CAR is "Qantas". This is, the TA is identified by DEP and ARR whenever the CAR is "Qantas".

Despite facing similar problems of data inconsistencies with relational counterparts, the existing CFD approaches cannot be applied easily to XML data for several reasons. Firstly, relational databases and XML sources are very diverse in data structure 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. There are several challenges in identifying XML constraints which 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 [101]. Finally, 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 the transformed relation is large. The number of tuples also increases multiplicatively when the XML document contains data with complex data types (e.g. *maxOccurs* in XML Schema). For example, if each Booking contains two Trips (refer to Fig 2.1), then the number of tuples in the transformed relation would double. Therefore, generalizing relational approaches to work on XML data is nontrivial.

## 2.3 Association rules

Association rules describing the co-occurrence of data items in a dataset was first introduced by [4]. Market basket analysis using transaction databases from supermarkets is a well known application of association rules. Each transaction contains items bought by a customer. An association rule represents a relationship between values of elements which has a form of $X \rightarrow Y$ *(s, c)*, $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \varnothing$, where $X$ and $Y$, $I$ are itemsets, $s$ and $c$ are support and confidence, respectively. Support and confidence are used to measure the quality of the rule. Support represents the frequency of $X \cup Y$ in the dataset. Confidence corresponds to the probability of finding $Y$, having found $X$ and is given by $\sup(X \cup Y) / \sup(X)$. For example, assume that "60% of customers who depart at SYD also depart at MEL". This can be expressed in the form of rules, SYD → MEL

(40%, 60%), where 40% is the support of the rule indicating how frequently the customers departure at both SYD and MEL and 60% is the confidence of the rule. Aprior-like algorithms [13, 14, 61, 62, 71, 84, 92, 93] have been introduced to discover data patterns in large datasets to address certain data quality issues, such as data anomaly (e.g. outlier) and to filter out useless data portions. However, association rules are constraints containing only constants which cannot address data inconsistency as required in XML data. In our approaches, the Apriori algorithm is adopted to discover constraints relating to either variables or constants to improve data consistency in XML data.

In the next section, we discuss the different proposals in relation to XML functional dependencies and highlight their limitations in addressing the problems of data inconsistency to further support our motivation.

## 2.4 XML functional dependency

XML offers a rich set of predefined constraints, such as structural, domain and cardinality constraints. However, it lacks the full extensibility to express constraints specifying at an application level in a declarative way [91]. Schema languages, such as DTD [88], W3C XML schema [90] and RelaxNG [30] support type and integrity constraints to specify XML schema. Type constraints only restrict on the element structure of a data source and do not relate to data values. Integrity constraints are not well scoped. For example, primary keys and foreign keys are defined by using ID and IDREF attributes in DTDs. Each ID attribute are unique within the whole document and each element type is specified by at most one ID attribute. DTD cannot express constraints specified in the free text parts. A document validates against DTD also might not conform to the specification.

XML Schema and RELAX NG were created to overcome DTD limitations[57]. Such languages support data types and namespaces which satisfy critical requirements in XML applications. However, such schema languages are not sufficient in situations which have complex constraints. For example, constraints have complicated structural conditions and express relations between values which cannot be captured in a grammar based approach. Thus, with the hierarchical nature of XML, inconsistency in XML data cannot be avoided. To remedy such a problem, XFDs have been introduced in the literature as an integrity enforcement measure to improve XML semantic expressiveness [6, 42, 50, 51, 65, 81, 82, 102]. Although different proposals of XFDs are defined by different terms of expressiveness, in all the proposals presented, data dependencies for XML are formally defined from two perspectives: tree-tuple-based XFD [6, 101, 102] and path-based XFD [42, 82]. They are constraints on the values reached by following either regular expressions or paths in XML trees.

### 2.4.1 Tree-tuple-based functional dependency

The concept of the tree-tuple is similar to the notion of tuple in relational database. Tree-tuple-based functional dependencies (tFDs) [6, 11] are proposed by considering a relational representation of XML data, that is, the XML data is presented as a set of tree-tuples and functional dependencies are defined on it. A tree-tuple is built as follows: for each element, exactly one data node from the data tree is selected to construct the tree-tuple. Fig 2.5 is an example of a Booking tree-tuple constructed by picking data from the Booking node which has bno of "b2" in the Booking data tree in Fig 2.3. While the original Booking contains two

nodes of Trip, the tree-tuple constructed from this Booking only includes one node of Trip at a time.

The tree-tuple representation allows combining node and value equality easily. The former corresponds to the equality between vertices and the latter corresponds to the equality between strings. A tFD over a DTD $D$ is expressed in a form as $X \rightarrow Y$, where $X$ and $Y$ are non-empty subsets of paths in $D$ [6]. For an XML data tree $T \models D$, $t_1$ and $t_2$ are tree-tuples in $T$, if $t_1.X = t_2.X$ and $t_1.X \neq$ null, implies $t_1.Y = t_2.Y$, then $T \models X \rightarrow Y$. For example, in the sub-tree rooted at Booking, a functional dependency such that the Departure and Arrival determines the Tax is expressed by a tFD as follows:

{ Bookings/Booking/Trip/Departure,
Bookings/Booking/Trip/Arrival} → {Bookings/Booking/Tax}.

### 2.4.2 Path-based functional dependency

Paths are an essential component which have been used as one of the basic primitives to define functional dependency in XML data [22, 23]. Given a node $v$ of an XML tree $T$, a path $p$ in $T$ is defined to be the set of all nodes and values reached by following $p$ from $v$ in $T$. Path-based XFDs (pFDs) [42, 60, 82] are functional dependencies defined based on paths. Similar to the tree tuple-based functional dependencies, the notion of pFDs is a generalization of the definition of

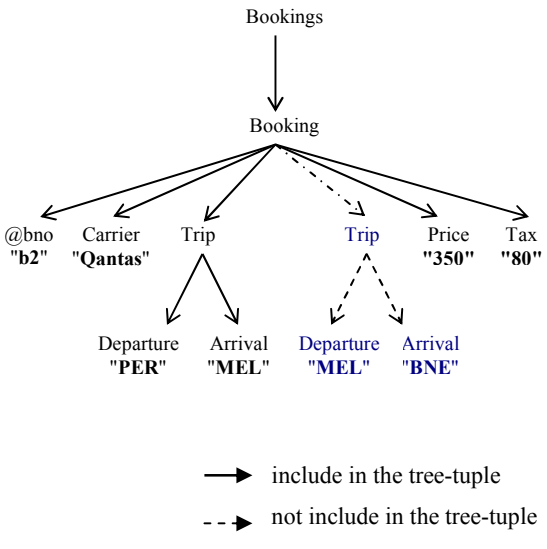


**Fig 2.5.** A tree-tuple illustration

functional dependencies (FDs) in relations. This means there is a correspondence between functional dependencies in relations and in XML data. In particular, an XFD is defined in a form of pFD [82] as $\{p_{x1}, p_{x2},..,p_{xn}\} \rightarrow p_y$, where $p_{xi}$ is a set of paths specifying condition elements and $p_y$ is the path specifying the implication element. For example, the XFD "under the sub-tree rooted at Booking, the Departure and Arrival determine the Tax" is expressed by the pFDs as follows: {//Booking/Departure, //Booking/Arrival}→ //Booking/Tax.

Both tFDs and pFDs have the same expressive power of functional dependencies [64]. The languages for tFDs and pFDs only allow unary functional dependencies holding in the entire document which cannot express the semantics of constraints in XML data in some cases, such as constraints holding conditionally on subset of data, or constraints holding on similar objects. Certain extended proposals of XML functional dependencies have been introduced in existing work to cope with the hierarchical structure of XML data. We review these in the next section.

### 2.4.3 Extended proposals for XML functional dependency

*Sub-graph-based functional dependency:* a sub-graph is a set of paths of XML data. A sub-graph-based functional dependency (gFD) is defined based on the sub-graphs of an data tree [52]. gFDs have pre-image semantics which allow the expression of XFDs involving a set of elements to represent relationships between sub-trees. A gFD has the form $\{v: X \rightarrow Y\}$, where $v$ is a node of data tree $T$, $X$ and $Y$ are $v$-subgraphs. A gFD holds on $T$ iff for any two pre-images $W_i$ and $W_j$ of $T_v$, their projections on $X$ are equal, then their projections on $Y$ are equal, where $T_v$ is a $v$-subgraph of $T$ rooted at $v$. For example, an pFD {//Booking/Departure, //Booking/Arrival}→ //Booking/Tax can be expressed as a gFD:

$\{v_{Booking}: X \rightarrow Y\}$, where $X$ is the $v_{Booking}$ subgraph with leave elements of Departure and Arrival, and $Y$ is the $v_{Booking}$ subgraph with leave element of Tax. Although gFDs allow the expression of the semantics of constraints relating to a set of elements, they are constraints on the entire document which cannot express the semantics of constraints holding conditionally on subsets of data.

*Generalized-tree-tuple-based functional dependency:* the work in [102] introduced another notion of XFDs, called Generalized-tree-tuple-based FD (gtFD) by extending the notion of tree-tuple functional dependencies. A gtFD has a form of $<C_p$, LHS, RHS> which is expressed as $\{P_{l1}, P_{l2},..,P_{ln}\} \rightarrow P_r$ w.r.t $C_p$, where $P_{li}$ $(i=1..n)$ and $P_r$ are paths relating to the path $p$, and $C_p$ is a tuple-class. gtFDs allow capturing constraints with a set of elements. A gtFD holds on an XML data tree $T$ if for any two tree-tuples $t_k$, $t_h$ in $C_p$: *(i)* $\exists i$, $i \in [1..n]$, $t_k.P_{li} = \perp$ or $t_h.P_{li} = \perp$, or *(ii)* $\forall i \in [1..n]$, $t_k.P_{li} =_{pv} t_h.P_{li}$ then $t_k.P_r =_{pv} t_h.P_r$. For example, Fig 2.6 shows a Booking contains two complex nodes of Trip, and each Trip includes Departure and Arrival. The constraint "under the sub-tree rooted at Booking, the value of Tax is identified by Carrier and Trip" can be expressed as follows: gtFD:{Carrier, Trip/Departure, Trip/Arrival}→ {Tax} w.r.t $C_{Booking}$.

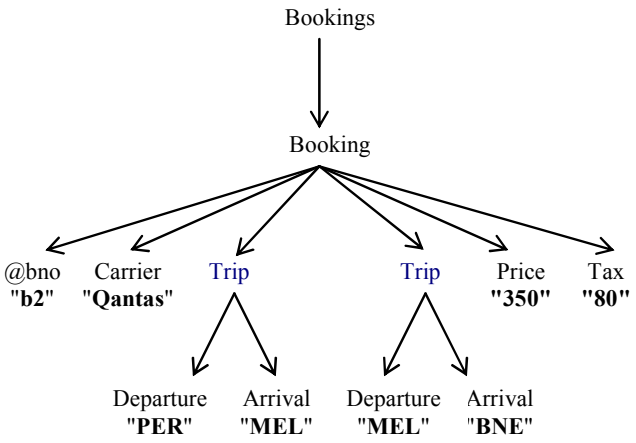


**Fig 2.6.** A sub-tree represents a generalized-tree-tuple-based FD

gtFDs have the same express power as gFDs. Each generalized-tree-tuple used in the gtFD is equal to a *v*-subgraph used in the gFD. For a gtFD

$\{P_{l1}, P_{l2},..,P_{ln}\} \rightarrow P_r$ w.r.t $C_p$, it can be expressed by a gFD $\{v: X \rightarrow Y\}$, where $v$ is the root of the path $p$, $X$ is a $v$-subgraph consists of path $\{P_{l1}, P_{l2},..,P_{ln}\}$ and $Y$ is a $v$-subgraph including path $P_r$. gtFDs can be used to express XFDs involving a set of elements as gFDs. gtFDs consider equality between two XML elements as equality between sub-trees. However, either gFDs or gFDs cannot express constraints holding either on subsets of a data tree or similar sub-trees.

*Local functional dependency (lFD):* to cope with the hierachical structure of XML data, one needs not only the absolute constraints holding on the whole document such as tFDs and pFDs, but also relative constraints holding on subsets of data [21]. Liu et al. [63] introduced the notion of local functional dependencies which are functional dependencies holding on sub-documents. A local functional dependency is defined as $X$ functionally determines $Y$ under a path $p$, denoted by $X \vec{p} Y$, where $X$ and $Y$ are two sets of paths in a DTD $D$ and $p$ is a prefix of every path in $X$ and $Y$. The determinant of the lFD is a path terminated by a label for internal nodes. The scope of lFD is a particular sub-tree and not on the whole tree as in either tFDs or pFDs. For example, Fig 2.7 shows a Bookings data tree including a number of Agents which are distinct by the Agent Id (i.e. @id). For each Agent, the values of bno are distinct. These constraints can be represented as follows:

$lFD_1$: @id $\overrightarrow{Bookings.Agent}$ *Agent*

$lFD_2$: @bno $\overrightarrow{Bookings.Agent.Booking}$ *Booking*

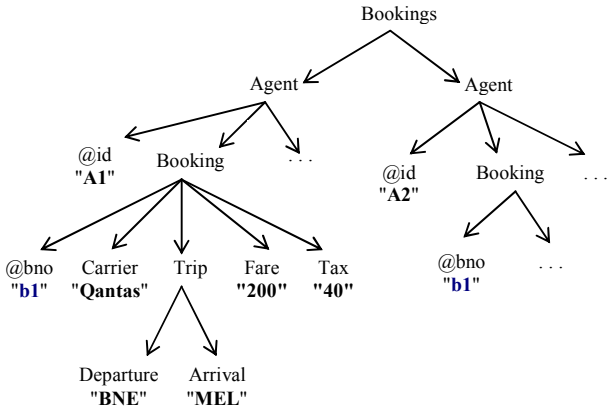


**Fig 2.7.** A sub-tree represents a local functional dependency

Constraint $lFD_2$ cannot be represented by either tFDs or pFDs. For example, it is represented by a pFD as {Bookings/Agent/Booking/ @bno}→ Bookings/ Agent/Booking}. It is clear that $lFD_2$ is violated due to the same bno of "b1" being used to identify more than one Booking.

Despite the lFD notion being more expressive than common XFDs, it is still not sufficient to express the semantics of some applications. That is, lFDs cannot capture the semantics of constraints accurately in situations where constraints hold conditionally on the source. For example, the semantics of a constraint is that 'any Booking with Carrier of "Qantas" having the same Fare should have the same Tax'. This constraint is expressed in the form of lFD as Fare $\overrightarrow{Bookings.Booking}$ Tax which only expresses that the Fare identifies the Tax under the sub-tree specified by the path "Bookings.Booking". Such lFD is impossible to represent the conditional expression Carrier of "Qantas". It is clear that the concept of lFDs is still too strong which cannot express constraints having scope specified by a particular condition.

Although different XFD proposals have different expressiveness terms and their justification is based on their natural occurencies in XML data, existing XFD proposals are insufficient to capture data inconsistency.

The reason is that XFDs cannot express the semantics of constraints related to conditions. Moreover, existing proposals of XFDs [6, 82] treat the equality between two elements as the equality between their identifiers and they do not consider sub-tree comparisons. Such XFDs may work well for redundancy detection and normalization; however, they work improperly in cases where constraints are unknown and required to be extracted from a given source. According to existing approaches, two sub-trees satisfy an XFD if they are equal with respect to the left part of that XFD and they also equal with respect to the right part.

In order to address the limitations in prior work, we first propose a new type of constraint, called XCFD, which is a value-based constraints which allow the expression of constraints with conditions [85]. Then, we introduce XCSDs as path and value-based constraints [87], which are different from XFDs in two aspects. The first difference is that each path *p* in XCSDs represents a group of similar paths to *p*. The second difference is that XCSDs allow binding values to particular elements to express constraints with conditions. XCSDs are constraints with conditional semantics, holding on data with diverse structures which cover both structural and semantic aspects. We introduce an approach based on the similarity of sub-trees to evaluate the satisfaction of a constraint. Our idea is that if two sub-trees are similar with respect to the left part of the constraint, and they are also similar with respect to the right part, then they satisfy the constraint. The similarity of sub-trees is measured by our established measurement, called "sub-tree similarity". Existing work [81, 102] introduced algorithms to discover XFDs. However, such XFD approaches cannot detect proper sets of constraints to address data inconsistency. This thesis proposes new approaches, named XDiscover and SCAD, which generalize existing techniques relating to association rules [4] and functional dependency discovery [53, 70, 102] to discover

constraints containing either variables or constants which can be used to constrain data consistency.

## 2.5 Managing data consistency in inconsistent data sources

In this section, we review existing work commonly used to manage consistent data in inconsistent sources. In particular, we consider data repair and consistent query answer approaches in relational databases and XML data.

*Relational database*: the management of consistent data in inconsistent data has been extensively studied in relational databases [3, 16, 18, 28]. Consistent data is formally obtained following two directions including data repair and consistent query answers. Data repair aims to find consistent parts from inconsistent data which differs from a given inconsistent database in a minimal way [9]. A database $D$ is consistent with respect to a set of integrity constraints *ICs* if $D$ satisfies *ICs*. Otherwise, $D$ is inconsistent with respect to *ICs*. $R$ is a repair of $D$ if $R$ satisfies $IC$ and $\Delta(D, R)= (D\backslash R) \cup (R \setminus D)$ minimal under set inclusion. Computing data consistency with respect to *ICs* can be achieved only through tuple deletions. That is, $R$ is obtained from $D$ by eliminating tuples. $R$ is considered to be a minimal repair of $D$ if $R$ satisfies *ICs* and is maximally contained in $D$, *i.e.* there $R'$ does not exist such that $R'$ satisfies *ICs* and $R \subset R' \subset D$. For example, given inconsistent data $D= \{(a, b, c),(a, c, d), (a, c, e), (b, g, h)\}$, $D$ has two repairs $R_1= \{(a, b, c), (b, g, h)\}$ and $R_2= \{(a, c, d), (a, c, e), (b, g, h)\}$. $\Delta(D, R_1)$ and $\Delta(D, R_2)$ are minimal under set inclusion.

Since a large number of repairs might exist for an inconsistent database, most existing work has only focused on computational

methodologies to retrieve consistent answers for a query posed on an inconsistent database, regardless of its inconsistency [9, 29]. A consistent query answer is defined to be the common part of answers to the query on all possible repairs of the source. Arenas et al. [9] introduce a query rewriting algorithm to compute consistent query answers based on query rewriting. The basic idea is to enforce constraints locally, at the level of data which appears in the query to avoid the explicit computation of data repairs. In particular, the original query $Q$ posted to $D$ is rewritten into a new query $Q'$ such that the answers to $Q'$ in $D$ are the consistent answers to $Q$ from $D$. $Q'$ is constructed by adding conditions from *ICs* to $Q$ to enforce the satisfaction of constraints in *ICs*. However, the work in [9] has a very limited applicability since it applies first order queries and does not include disjunction or quantification, or binary universal integrity constraints. The first order query rewriting technique only works appropriately for a certain types of queries and constraints, which are universal queries and constraints. There does not exist first order rewriting for queries and constraints relating to conjunctive queries with projection and referential constraints; and the problem cannot be solved in a polynomial time. [17].

Chomicki [29] presents a framework for computing consistent query answers based on a graph-theoretic representation of repairs. It considers relational algebra queries without projection and denial constraints. This work handles union queries which can extract indefinite disjunctive information from an inconsistent database. Arenas et al. [8] apply logic programming based on answer sets to retrieve consistent information from an inconsistent database. This work concentrates mainly on logic programs for binary integrity constraints. The work in [7] studies the decidability status of consistent query answering by combining instances, *ICs* and query as input. The notion of consistent query answers are also extended to the case of aggregate queries [10, 46]. Arenas et al. [10] investigate the

problem of consistent answers of aggregate queries in the presence of functional dependencies. The work in [46] provides data violating a set of aggregate constraints. These constraints are defined on numerical attributes (such as Sale Price, Tax, etc.) and are not intrinsically involved in other forms of constraints. Deker [32] introduces a concept, called cause, to specify query answers having integrity in data sources which might violate their integrity constraints. A cause of an answer is a minimal excerpt of the data explaining the reasons why an answer is give to a query. An answer has integrity if one of its causes does not overlap with any cause of integrity violation. Most above cited approaches suppose that tuple insertions and deletions are the basic primitives for repairing inconsistent data. Recently, database repairs and consistent query answering have been considered in the context of conditional functional dependencies [36, 55, 56]. However, due to the different structure of data and the different nature of constraints, existing techniques in relational databases cannot easily be applied to XML data [44].

*XML data:* the notions of repair and consistent query answers have been generalized to the context of XML data. The work in [44, 45, 78] find inconsistent data with respect to a set of XML functional dependencies. The data repair in [45] is found based on replacing node values and introducing functions, indicating the reliability of node information. Tan et al. [78] study the problem of data repair by making the smallest modifications in terms of repair cost. Flesca et al. [43] study the existence of repairs with respect to a set of integrity constraints and a DTD. The existence of repairs using minimal sets of update operations is investigated. The work in [69] considers the problem of data repair with respect to a set of functional dependencies in the merged format of XML data. This work extends the XFDs to be satisfied by comparing sub-trees in a specified

context of the data. Yakout et al. [99] present an approach using machine learning as a guide for repairing data. However, such approaches may correct data improperly, and worse, might result in other inconsistencies when repairing the data. Moreover, the concept of data repair is often used as an auxiliary notion to define consistent query answers and existing approaches do not give any algorithms to compute data consistency.

Second, the problem of finding consistent query answers can be considered as a principled way to manage data inconsistency [9, 74, 76, 103]. The work in [74] studies the problem of computing query answers with respect to a given DTD. This work presents a validity-sensitive method of querying XML data, which extracts more information from invalid data sources than the standard query evaluation. Tan et al. [76] propose an approach to compute consistent query answers from virtually integrated data with respect to a set of constraints. However, they do not take into account constraints which hold conditionally on similar objects, as in our work. Query rewriting techniques have been widely used as powerful methods to calculate query answers [33, 34, 66, 103]. The work in [33, 34, 103] introduces techniques for query rewriting in the represention of constraints. Yu et al. [103] propose a technique incorporating target constraints into query rewiring to calculate query answers through target schemas. However, we found that such work is inapplicable for the scenarios which we consider. To the best of our knowledge, none of the existing work on finding query answers properly combines both structural and data semantics to calculate query answers, as in our approach.

## 2.6 Summary

In this chapter, we first presented background information on XML databases including DTD schema, XML documents and data trees. Second,

we reviewed conditional functional dependency and show that CFD approaches intended to address data inconsistency in relational databases do not work well in XML data. Third, we discussed association rules and pointed out that they cannot properly express semantic constraints in XML data as constraints contain only constants. Fourth, we reviewed several proposals for XML functional dependencies including tree-tuple-based FDs, path-based FDs, sub-graph-based FDs, generalized-tree-tuple-based FDs and XML local functional dependencies. We provided a justification of XFD approaches and pointed out that XFD specifications are constraints containing only variables which, in some cases, cannot target data inconsistency in XML data.

This thesis introduces new notions of constraints based on the idea of conditions in CFDs and a new concept of structural similarity. Such constraints contain either constants or variables which are suitable for capturing the semantics of constraints in heterogeneous XML data sources. Existing XFD approaches cannot detect proper sets of constraints to address data inconsistency since they do not consider constraints with conditions. This thesis presents new approaches which generalize existing techniques of association rule mining and functional dependency discovery to discover constraints containing either variables or constants. Finally, we review existing approaches relating to data repairs and consistent query answers. Computing consistent query answers can be considered as a principled way to manage data consistency. However, none of the existing work on consistent query answers properly calculates answers for queries posted to an inconsistent XML data source caused by both semantic and structural inconsistencies. This thesis proposes a new approach combining both structural and data semantics to calculate customized consistent query answers for queries posted to inconsistent XML data.

# Chapter 3

# Content-based discovery for improving XML data consistency

This chapter introduces a novel approach, called XDiscover, which is a content-based discovery approach to discover XML conditional functional dependencies (XCFDs) from a given data source conforming to a given schema. This is to resolve data inconsistency caused by semantic inconsistency. The XCFD notion is extended from XFDs by incorporating conditions into XFD specifications. The rest of chapter is organized as follows: Section 3.1 presents the introduction to the problem, including our motivation and the summary of our approach; Section 3.2 presents the preliminaries consisting of the notations used in this chapter; Section 3.3 presents our proposed XCFD specification; Section 3.4 describes the detail of XDiscover; Section 3.5 details the experiment results of XDiscover; Ssection 3.6 presents case studies; and Section 3.7 summarises the chapter.

## 3.1 Introduction

The 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 [1, 59, 67]. 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. This leads to constraints on the underlying data from different organizations satisfied by an individual data source which may not be applicable in the federated data. Although XML functional dependency (XFD) is one type of semantic constraint, existing notions of XFD [6, 37, 82] are not sufficient for capturing data inconsistency. This is because XFDs globally express constraints over the whole document; thus, they are unable to capture conditional semantics partially expressed in some fragments of the document.

Fig 3.2 shows an example of a simplified instance of a Flight Bookings data tree *D* constrained by the schema Flight Bookings *S* in Fig 3.1. *D* contains data of Flight Bookings. Each Booking includes information on the Carrier, Trip, Fare and Tax. For each Trip, 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 schema tree *S* and data tree *D* as a key to identify that node in the tree. This notion will be

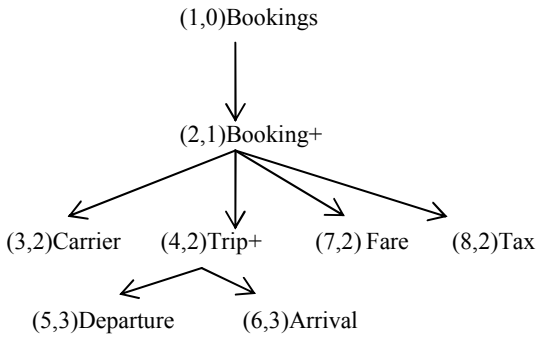


**Fig 3.1** A Flight Bookings schema tree

further described in definition 3.1 (section 3.2). Constraints on *D* have different specifications. We classify them into two types: constraints without condition and constraints with conditions.

*Type 1:* constraints without conditions are constraints containing only variables. They are constraints holding over the whole document and are commonly known as XML functional dependencies (XFDs). For example,

Constraint 1*:* Any Booking with the same Trip including Departure and Arrival should have the same Tax. This is an example of a functional dependency holding for all Bookings in *D*.

*Type 2:* Constraints with conditions include constraints which either contain constants only or both constants and variables. Such constraints hold conditionally on the document. They are not standard XFDs. For example,

Constraint 2a: Any Booking with Carrier of "Tiger Airways" with the same Fare 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 "Tiger Airways" 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 particular fragment containing data from one particular source. Thus, we need to enforce constraints of type 2 to capture data inconsistency.

When constraints with conditional semantics are not enforced explicitly, data inconsistency in some parts of the document cannot be detected. For example, Bookings data in *D* (Fig 3.2) do not satisfy all the above constraints. The Bookings of nodes (12, 1) and (22, 1) contain the same values of Trip including Departure of "DRW" and Arrival of "BNE" and have the same Tax of "30". They satisfy constraint 1 but violate either constraint 2a or constraint 2b. For constraint 2a, if Carrier is "Tiger Airways", the Fare determines the Tax. Node (12, 1) and node (2, 1) have the same Fare of "200" but they contain different values of Tax, which are "30" and "40" respectively, which violates constraint 2a. According to constraint 2b, for a Booking with Carrier of "Virgin" and Departure of "BNE", Tax should be "20" but node (22, 1) contains Tax of "30" which 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. Under such circumstances, deriving a complete set of constraints from a given data instance to constrain the heterogeneous data

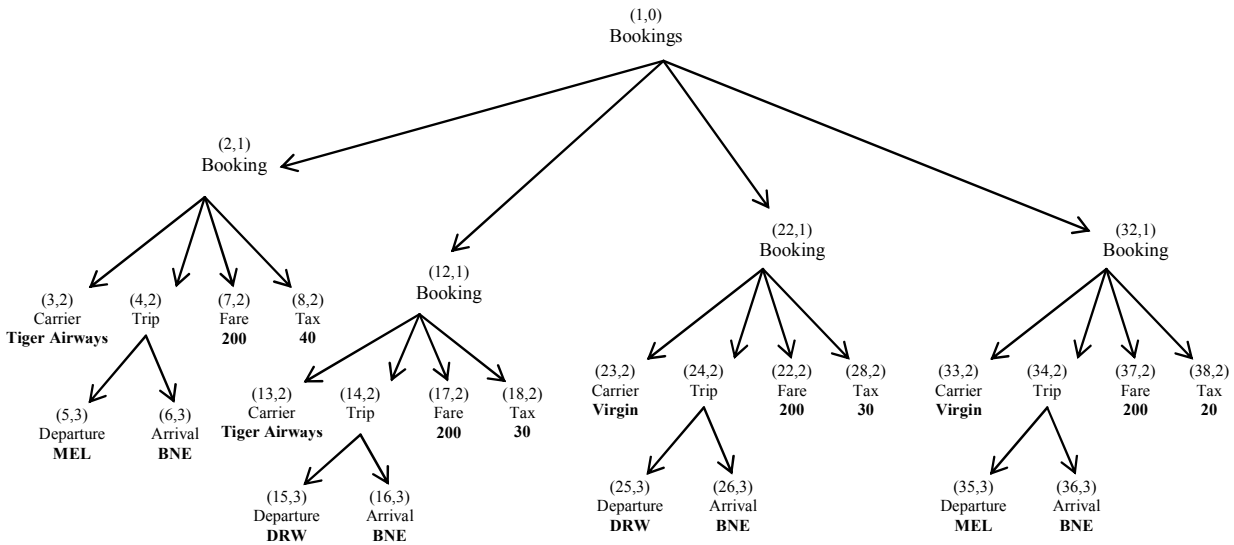


**Fig 3.2.** A simplified Flight Bookings data tree containing semantic inconsistencies

sources is necessary to improve data consistency.

In this chapter, we propose a novel approach, called XDiscover, to discover a set of minimal XML conditional functional dependencies (XCFDs) from a given XML instance to address semantic inconstency. The XCFD notion as constraints of type 2 is extended from XFDs by incorporating conditions into XFD specifications. This overcomes the 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 (CFDs) 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 which can cover more situations of dependencies under some conditions. XDiscover conveys the semantics hidden in data to discover a set of minimal XCFDs from a given instance. A set of our proposed pruning rules is incorporated in the discovery process to reduce the number of XCFD candidates to be checked on the dataset to improve the search performance. Experiments on synthetic and real datasets, and case studies are used to demonstrate the correctness of our approach.

We present preliminary definitions which are necessary for introducing XCFDs in the next section.

## 3.2 Preliminaries

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

We use XPath expression [89] to form a relative path; "." (self): select the context node. ".//": select the descendants of the context node, "[]":qualifier and "*": wildcards. For example, .//Carrier: select Carrier

descendants of the context node; .//Trip/Departure: select all Departure elements which are children of Trip. 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 3.1**. (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 a 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 [90]. 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 of the schema tree.

For example, the schema tree in Fig 3.1 is defined as *S= (E, A, T, root);* where:

$E = E_1 \cup E_2$; $E_1$ = {Booking, Trip}

$E_2$ = {Carrier, Departure, Arrival, Fare, Tax}

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

We assign a *path-ID* to each node in the XML schema tree as shown in Fig 3.1 in a pre-order 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 3.2**. (XML data tree constrained by a schema 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 $\{v_1, v_2, \ldots, v_n\}$ in *V*; *att(v)* maps *v* to a list of attribute nodes $\{v_1', v_2', \ldots v_m'\}$ 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 labeled 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 Fig 3.2, we have $V$ a set of nodes from node (1, 0) through node (38, 2).

*lab*((1,0)Bookings)="Bookings"*; lab*((3,2)Carrier)="Carrier";

*val*((2,1)Booking)= Booking; *val*((3,2)Carrier)="Tiger Airways";

*ele*((2,1 )Booking)={Carrier, Trip, Fare, 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 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 chapter, we assume that the XML data tree is required to conform to the associated XML schema tree. The conformation is defined as follows:

**Definition 3.3**. (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. Fig 3.2 is an example of the Bookings data tree which conforms to the Bookings schema tree in Fig 3.1.

Now we introduce a notion of node-value equality which is an essential feature in our proposed constraints.

**Definition 3.4.** (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 by $v_i =_v v_j$, iff:

- $v_i$ and $v_j$ have the same label, i.e., $lab(v_i) = lab(v_j)$,
- $v_i$ and $v_j$ have the same values:

$$\begin{cases} val(v_i) = val(v_j), \text{ if } v_i \text{ and } v_j \text{ are both simple nodes or attribute nodes.} \\ val(v_{ik})=val(v_{jk}) \text{ for all } k, \text{ where } 1 \le k \le n, \text{ if } v_i \text{ and } v_j \text{ are both complex nodes with } ele(v_i) = [v_{i1}, ..., v_{in}] \text{ and } ele(v_j) =[v_{j1}, ..., v_{jn}] \end{cases}$$

*lab* is a function returning label of a node, *val* is a function returning values of a node. If $v_i$ is a simple node or an attribute node, then $val(v_i)$ is the content of that node, otherwise $val(v_i)=v_i$ and $ele(v_i)$ returns a set of children nodes of $v_i$.

For example, Trip(14, 2) and Trip(24, 2) (in Fig 3.2) are node-value equality with

*lab*((14, 2) Trip)= *lab*((24, 2) Trip)="Trip";

*ele*((14,2) Trip)= {(15,3) Departure, (16,3) Arrival };

*ele*((24,2) Trip)= {(25,3) Departure, (26,3) Arrival };

node(15, 3) Departure $=_v$ node(25, 3) Departure = "DRW" and

node(16, 3) Arrival $=_v$ node(26, 3) Arrival = "BNE".

Based on the above basic concepts, we introduce a new type of constraint in the next section.

## 3.3 XML conditional functional dependency

As our proposed conditional functional dependency notion (XCFD) is defined on the basis of XFDs used by Fan et al. [42], we discuss XFDs before presenting the XCFD definition. In order to avoid returning an unnecessarily large number of constraints, we are interested in exploring *minimal* XCFDs existing in a given data source. Thus, we also include a notion of minimal XCFDs in this section.

**Definition 3.5.** (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:
$\varphi = P_l: (X \rightarrow Y)$; where:

- $P_l$ is a downward context path starting from the root to a considered node with label $l$, called root path. The scope of $\varphi$ is the sub-tree rooted at the node-label $l$;
- $X$ and $Y$ are non-empty sets of nodes under sub-trees rooted at node-label $l$. $X$ and $Y$ are exclusive.
- $X \rightarrow Y$ indicates a relationship between nodes in $X$ and $Y$, such that two sub-trees sharing the same values for $X$ also share the same values for $Y$, that is, the values of nodes in $X$ uniquely identify the values of nodes in $Y$. We refer to $X$ as the *antecedent* and $Y$ as the *consequence.*

**Satisfaction of an XFD:** A data tree $D=(V, lab, ele, att, val, r)$ conforming to $S$, $D|=S$, is said to satisfy $\varphi = P_l: X \rightarrow Y$ denoted $D|= \varphi \hat{} S$ iff for every two sub-trees rooted at $v_i$ and $v_j$ in $D$, if $v_i[X]=_v v_j[X]$ then $v_i[Y]=_v v_j[Y]$;

Let us consider an example, supposing $P_{Booking}$ is the path from the root to the Booking nodes in the Bookings data tree in Fig 3.2.

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

$\varphi = P_{\text{Booking}}$: (./Departure ^./Arrival)→ (./Tax) holds on the whole Bookings data tree.

We now introduce our proposed XCFD. The most important features of XCFDs are path and value-based constraints. The XCFD specification includes two parts: a functional dependency and a Boolean expression. The function dependency in the XCFD is basically defined as in a normal XFD. The only difference is that instead of only representing the relationship between nodes as in XFDs, the functional dependency in an XCFD incorporates with the Boolean expression to specify portions of data on which the functional dependency holds.

**Definition 3.6.** (XML conditional functional dependency - XCFD)
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:

$\psi = P_l: [\mathcal{C}], X \rightarrow Y,$ where:

- $P_l$ is a downward context path starting from the root to a considered node with label $l$, called root path. The scope of $\varphi$ is the sub-tree rooted at the node-label $l$;
- $\mathcal{C}$ is a condition for the XFD $X \rightarrow Y$ holds on $D$. The condition $\mathcal{C}$ has the form: $\mathcal{C} = ex_1 \theta ex_2 \theta \ldots \theta ex_n$; $ex_i$ is an atomic Boolean expression associated to a particular data node. That is, there does not exist any connections in $ex_i$. "$\theta$" is an operator either *AND* (^) or *OR* (∨).

- *X* and *Y* are non-empty sets of nodes under sub-trees rooted at node-label *l*. *X* and *Y* are exclusive.
- $X \rightarrow Y$ indicates a relationship between nodes in *X* and *Y*, such that two sub-trees sharing the same values for *X* also share the same values for *Y*, that is, the values of nodes in *X* uniquely identify the values of nodes in *Y*. We refer to *X* as the *antecedent* and *Y* as the *consequence.*

For example, suppose that $P_{\text{Booking}}$ is the context path from the root to the Booking nodes in the Bookings data tree (Fig 3.2); if there exists an XFD (./Fare)→(./Tax) holding on the Bookings data tree under condition $\mathcal{C}$ = (./Carrier = "Tiger Airways"), then we have an XCFD: $\psi = P_{\text{Booking}}$: (./Carrier= "Tiger Airways", ./Fare)→( ./Tax).

**Satisfaction of an XCFD:** the consistency of an XML data tree with respect to a set of XCFDs is verified by checking for the satisfaction of the data to every XCFD. A data tree *D=(V, lab, ele, att, val, r)* conforming to *S*, $D|=S$, is said to satisfy $\psi = P_l$: [$\mathcal{C}$], ($X \rightarrow Y$) denoted $D|= \psi \wedge S$ iff for every two sub-trees rooted at $n_i$ and $n_j$ in *D*, if $n_i[X]=_v n_j[X]$ then $n_i[Y]=_v n_j[Y]$ under the condition $\mathcal{C}$, where $n_i$ and $n_j$ have the same root node-label *l*.

XDiscover returns minimal XCFDs. The concept of minimal XCSD is defined as follows.

**Definition 3.7.** (Minimal XCFDs)
Given an XML data tree *D= (V, lab, ele, att, val, r)* conforms to the XML schema *S= (E, A, T, root)*, an XCFD $\psi = P_l$: [$\mathcal{C}$], ($X \rightarrow Y$) on *D* is minimal if $\mathcal{C}$ is minimal and $X \rightarrow Y$ is minimal.

- $\mathcal{C}$ is minimal if the number of expressions in $\mathcal{C}$ ($|\mathcal{C}|$) cannot be reduced, i.e., $\forall \mathcal{C}'$, $|\mathcal{C}'| < |\mathcal{C}|$, $P_l$: [$\mathcal{C}'$],( $X \nrightarrow Y$).

- $X \rightarrow Y$ is minimal if none of the nodes in $X$ can be eliminated, which means every element in $X$ is necessary for the functional dependency holding on $D$. In other words, $Y$ cannot be identified by any proper subset of $X$, i.e., $\forall X' \subset X$, $P_l$: [$\mathcal{C}$], ($X' \nrightarrow Y$).

For example, we assume that the XCFD $\psi$ holds on $D$

$\psi$ = $P_{\text{Booking}}$: (./Type="Airline" ^ ./Carrier="Tiger Airways"), (./Departure, ./Arrival → ./Tax)

We have $\mathcal{C}$ = (./Type="Airline" ^ ./Carrier="Tiger Airways")

and $X \rightarrow Y$ = (./Departure , ./Arrival → ./Tax)

We assume that:

- If $\mathcal{C}'$=(./Type="Airline"), $|\mathcal{C}'|$={./Type="Airline"}=1<2={./Type="Airline", ./Carrier= "Tiger Airways"} = $|\mathcal{C}|$

  then $P_{\text{Booking}}$:(./Type="Airline"), (./Departure ^ ./Arrival → ./Tax) does not hold properly on $D$.
- If $X'$=./Departure, $|X'|$={Departure} $\subset$ {Departure, Arrival}=$|X|$, then $P_{\text{Booking}}$: (./Type="Airline"^ ./Carrier="Tiger Airways"), (./Departure →./Tax) does not hold on $D$.

In the next section, we present our proposed approach, XDiscover, for discovering XCFDs from a given XML source associated with a schema.

## 3.4 XDiscover: XML conditional functional dependency discovery

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 $\psi = P_l$: [$\mathcal{C}$], $X \rightarrow Y$; where each XCFD is minimal and contains only a single path in consequence $Y$.

XDiscover aims to discover all non-trivial XCFDs from the data source. Our algorithm works in the same manner as candidate generating and testing approaches [53, 70, 102]. That is, the algorithms traverse the search lattice in a level-wise manner and start finding candidates with small antecedents. The results in the current level are used to generate candidates in the next level. Pruning rules are employed to reduce the search lattice as soon as possible. Supersets of nodes associated with the left-hand side of already discovered XCFDs are pruned from the search lattice. Our approach identifies more pruning rules (section 3.4.4) than the existing approaches. In particular, we include rules to: *(i)* prune equivalent sets relating to already discovered candidates; *(ii)* eliminate trivial candidates; and *(iii)* remove supersets of nodes related to antecedents of already found XCFDs and ignore subsets of nodes associated with conditions of already discovered XCFDs.

The XDiscover algorithm includes three main functions. The first function named *search lattice generation*, generates a search lattice containing all possible combinations of elements in the schema data tree. The second function named *candidate identification* is used to identify possible candidates of XCFDs. The last function is called v*alidation* and is used to validate the identified XCFD candidates to find satisfied XCFDs. The detail of each function is described as follows.

### 3.4.1 Search lattice generation

We adopt the Apriori-Gen algorithm [4] to generate a *search lattice* containing all possible combinations of node-labels. The process starts from nodes with a single label in level $d$= 1. Nodes in level $d$ with $d \geq 2$ are

obtained by merging pairs of node-labels in level (*d-1*). Fig 3.3 is an example of a search lattice of node-labels: A, B and C. Node AC in level 2 is generated from nodes A and C in level 1. The number of occurrences of each node is counted. Nodes with occurrences less than a given threshold $\tau$ are discarded to limit the discovery to only the frequency portions of data.

### 3.4.2 Candidate identification

The link between any two direct nodes in the search lattice is a representation of a possible candidate XCFD. Assume that *W* & *Z* are two nodes directly linked in the search lattice. Each edge*(W, Z)* represents a candidate XCFD: $\psi = P_l:[\mathcal{C}],(X \rightarrow Y)$, where $W = X \cup \mathcal{C}$ and $Z = W \cup \{Y\}$, *X* is a set of variable elements, and $\mathcal{C}$ is a set of conditional elements. For example, for edge*(W, Z)*= edge(AC, ABC) in Fig 3.3, we assume A is the condition, then we have an XCFD $\psi = P_l$: {A}, {C} → {B}.

If the condition A is empty, then $\psi$ becomes a constraint on the whole document as an XFD. This means an XFD is a special case of an XCFD. To check for the availability of a candidate XCFD represented by

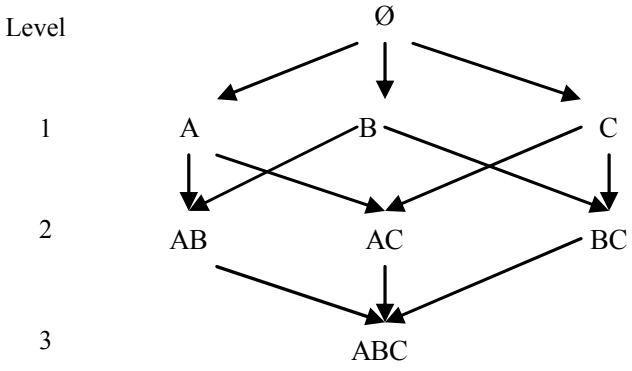


**Fig 3.3**. A set of containment lattice of A, B and C

an edge between $W$ and $Z$, we examine the set of node-labels in $Z$ to see whether it contains one more node-label than $W$. After identifying a candidate XCFD, a validation process is performed to check whether this candidate holds on the data.

### 3.4.3 Validation

Validation for a satisfied XCFD includes two steps. We first calculate partitions for node-labels associated with each candidate XCFD, then we check for the satisfaction of that candidate XCFD, based on the notion of partition refinement [53]. From a general point of view, generating a partition for a node-label classifies a dataset into classes based on data values coming with that node-label. Each class contains all elements with the same value. A partition is defined and calculated as follows:

**Definition 3.8.** (Partition) A partition $\Pi_{W|l}$ of $W$ on $D$ under the sub-tree rooted at node-label $l$ is a set of disjoint equivalence classes $w_i$. Each class $w_i$ in $\Pi_{W|l}$ contains all nodes with the same value. The number of classes in a partition is called the *cardinality* of the partition, denoted by $|\Pi_{W|l}|$. $|w_i|$ is the number of nodes in the class $w_i$.

For example, from schema tree Bookings $S$ in Fig 3.1, we have:
$E$={[(1, 0)Bookings] ,[( 2, 1)Booking],[( 3, 2) Carrier], [(4,2) Trip], [(5, 3) Departure], [(6, 3) Arrival], [(7, 2) Fare], [(8, 2)Tax]}
From the searching lattice, suppose we consider a partition identifier $W$= "Carrier" which corresponds to the node [(3, 2) Carrier] in schema tree $S$. Traversing data tree Bookings $D$ in Fig 3.4 finds all data nodes which have the node name Carrier and *depth* of 2.
The found nodes are grouped into two classes:

$Class_1$= { [(23, 2) Carrier= “Tiger Airways”], [(33, 2) Carrier= “Tiger Airways”] , [(53, 2)Carrier= “Tiger Airways”], [(63, 2) Carrier= “Tiger Airways”]}

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

The partition $\Pi_{\text{Carrier} \mid \text{Booking}}$ to the value of node Carrier with respect to sub-tree rooted at Booking is represented as $\Pi_{\text{Carrier} \mid \text{Booking}} = \{w_1, w_2\}$

$w_1$= {[(22,1) Booking], [(32, 1) Booking], [(52, 1) Booking], [(62, 1) Booking]}

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

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

To simplify the presentation, we omit the node-ID and path-ID associated with each node in the following sections to avoid cluttering. The validation process for a satisfied XCFD is performed follow the following theorem.

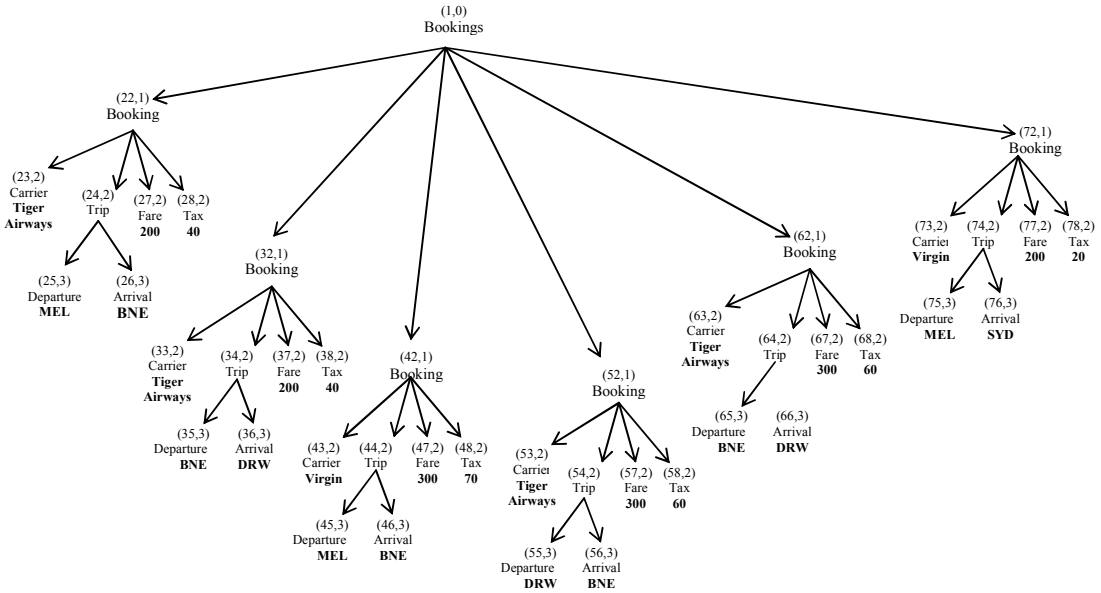


**Fig 3.4.** A simplified Bookings data tree: *each Booking contains only one Trip*

**Theorem 3.1.** Let $W = \{X\} \cup \{\mathcal{C}\}$, $Z = W \cup \{Y\}$ be two sets of nodes in the search lattice, and $\Pi_W$ and $\Pi_Z$ be two partitions of $W$ and $Z$. An XCFD, $\psi = P_l:[\mathcal{C}],(X \rightarrow Y)$ holds on data tree $D$ if either of the below conditions is satisfied:

- There exists at least one equivalent pair $(w_i, z_j)$ between $\Pi_W$ and $\Pi_Z$.

  or

- There exists a class $\mathbf{c}_k$ in $\Pi\mathcal{e}$ that contains all elements of a certain pair $(w_i, z_j)$ in $\Pi_W$ and $\Pi_Z$.

*Proof:* the first condition: according to [102], a functional dependency holds on $D$ if every node in a class $w_i$ of $\Pi_W$ is also in a class $z_j$ of $\Pi_Z$. In our case, the satisfied XCFD does not require every class $w_i$ in $\Pi_W$ to be a class $z_j$ in $\Pi_Z$ because an XCFD can be true on a portion of $D$. This means if there exists at least one equivalent pair $(w_i, z_j)$ between $\Pi_W$ and $\Pi_Z$ then we conclude that $\phi$ holds conditionally on data tree $D$.

The second condition: if there exists a class $\mathbf{c}_k$ in $\Pi\mathcal{e}$ containing exactly all elements in pair $(w_i, z_j)$, this means under condition $\mathbf{c}_k$, all elements in $w_i$, and $z_j$ share the same data rules. Then we conclude that the XCSD: $\psi = P_l: \{\mathbf{c}_k\},(X \rightarrow Y)$ holds on data tree $D$. □

The number of candidate XCFDs and the searching lattice are very large. In order to improve the performance of XDiscover, we introduce five pruning rules used in our approach to remove redundant and trivial candidates from the search lattice.

### 3.4.4 Pruning rules

We start this section by presenting the theoretical foundation including concepts, lemmas and theorems to support our proposed pruning rules.

**Theoretical foundation**: we introduce a concept of *equivalent sets* and four lemmas, which are necessary to justify our proposed pruning rules.

This is to prove that the pruning rules do not eliminate any valid information when nodes are pruned from the search lattice. We employ the following rules which are similar to the well-known Armstrong's Axioms [12] for functional dependencies in the relational database to prove the correctness of the defined lemmas.

Let $X, Y, Z$ be a set of elements of a given XML data $D$. These rules are obtained from adoptions of Armstrong's Axioms [12]. This is, we adapt the notation of exiting rules to conform to the notation of our work.

**Reflexivity** If $Y \subseteq X$, then $P_l: X \rightarrow Y$

**Augmentation** If $P_l: X \rightarrow Y$, then $P_l: XZ \rightarrow YZ$

**Transitivity** If $P_l: X \rightarrow Y$, $P_l: Y \rightarrow Z$, then $P_l: X \rightarrow Z$

The following two inference rules can be derived from above three rules

**Union** If $P_l: X \rightarrow Y$ and $P_l: Y \rightarrow Z$, then $P_l: X \rightarrow YZ$.

**Decomposition** If $P_l: X \rightarrow YZ$, then $P_l: X \rightarrow Y$ and $P_l: X \rightarrow Z$ .

**Definition 3.9.** (Equivalent sets)

Given $W= X$ and $Z=W\cup\{Y\}$, if $\psi = P_l: (X= \text{“}a\text{”}) \rightarrow (Y= \text{“}b\text{”})$ and $\psi' = P_l: (Y= \text{“}b\text{”}) \rightarrow (X= \text{“}a\text{”})$ hold on $D$, where $a, b$ are constants; $X$ and $Y$ contain only a single data node, then $X$ and $Y$ are called *equivalent sets*, denoted $X \leftrightarrow Y$.

**Lemma 3.1.** Given $W= X\cup \mathcal{C}$ and $Z=W\cup\{Y\}$, $X'= X\cup\{A\}$, if $\psi = P_l:[\mathcal{C}]$, $(X \rightarrow Y)$ then $\psi' = P_l:[\mathcal{C}]$, $(X' \rightarrow Y)$.

*Proof:* We have $\psi = P_l: [\mathcal{C}]$, $(X \rightarrow Y)$,

Applying augmentation rule, $P_l: [\mathcal{C}]$, $(X\cup\{A\} \rightarrow Y\cup\{A\})$

Applying decomposition rule, $P_l: [\mathcal{C}]$, $(X\cup\{A\} \rightarrow Y)$ and $P_l: [\mathcal{C}]$, $(X\cup\{A\} \rightarrow \{A\})$

Therefore, $P_l: [\mathcal{C}]$, $(X' \rightarrow Y)$.□

**Lemma 3.2.** Given $W= X \cup \mathcal{C}$ and $Z=W \cup \{Y\}$, if $\psi = P_l$: $[\mathcal{C}]$, $(X \rightarrow Y)$ associated to a class $w_i$ holds on $T$ then $\psi' = P_l$: $[\mathcal{C}']$,$(X \rightarrow Y)$ holds on $D$ where $\mathcal{C}' \subseteq \mathcal{C}$.

*Proof:* If $\psi = P_l$: $[\mathcal{C}]$, $(X \rightarrow Y)$ associated to a class $w_i$ holds on $D$,

Assume that $\mathcal{C} = \mathcal{C}' \cup \mathcal{C}''$,

Applying decomposition rule: $P_l$:$[\mathcal{C}']$,$(X \rightarrow Y)$ and $P_l$: $[\mathcal{C}'']$,$(X \rightarrow Y)$

Therefore, $P_l$: $[\mathcal{C}']$,$(X \rightarrow Y)$ holds on $D$ including elements from class $w_i$. □

**Lemma 3.3.** Given $W= X$ and $Z=W \cup \{Y\}$, if $\psi = P_l$: $(X=$ "$a$"$) \rightarrow (Y=$ "$b$"$)$ holds, and the number of actual occurrences of expression $Y =$ "$b$" in $T$, called $o_b$ , is equal to the size of $|z_b|$ then $X \leftrightarrow Y$.

*Proof:* $\psi = P_l$: $(X=$ "$a$"$) \rightarrow (Y=$ "$b$"$)$ means $|w_a| = |z_b|$ *(1)*

Since

we have $|z_b| = o_b$, $Y=$"$b$" does not occur with any other antecedence *(2)*

From (1) & (2) indicate that $Y=$"$b$" only occurs with the value of $X=$"$a$".

Therefore, $(Y=$ "$b$"$) \rightarrow (X=$ "$a$"$)$ holds. $X \leftrightarrow Y$ is proven. □

**Lemma 3.4.** Let $E$ be a set of distinct nodes in the $D$, the XCFD $\psi = P_l$: $[\mathcal{C}]$,$(X \rightarrow Y)$ is minimal if for all $A \in X$, where $Y \in R(X \setminus \{A\}) \cup R(\mathcal{C})$, $R(X) = \{ Y \in E \mid \forall A \in X: P_l: [\mathcal{C}],(X \setminus \{A, Y\} \nrightarrow Y)\}$.

*Proof*: If $Y \notin R(X \setminus \{A\}) \cup R(\mathcal{C})$ for a given set $X$, then $Y$ has been found in a discovered XCSD where either the antecedent is a proper subset of $X$ or the condition is a proper subset of $\mathcal{C}$. In such cases, $\psi = P_l$:$[\mathcal{C}]$,$(X \rightarrow Y)$ is not minimal. □

**Pruning rules**: we introduce five pruning rules used in our approach to remove redundant and trivial candidates from the search lattice.

Particularly, these rules are used to delete candidates at level *d-1* for generating candidates at level *d*. Pruning rules 3.1-3.4 are justified by Lemmas 3.1-3.4, respectively. Rule 3.5 is relevant to the cardinality threshold. The first three rules are used to skip the search for XCFDs that are logically implied by the already found XCFDs. The last two rules are used to prune redundant and trivial XCFD candidates.

**Pruning rule 3.1**. Pruning supersets of nodes associated with the antecedent of already discovered XCFDs. If $\psi = P_l$: [$\mathcal{C}$], ($X \rightarrow Y$) holds, then candidate $\psi' = P_l$: [$\mathcal{C}$], ($X' \rightarrow Y$) can be deleted where $X'$ is a superset of $X$.

**Pruning rule 3.2**. Pruning subsets of the condition associated with already discovered XCFDs. If $\psi = P_l$:[$\mathcal{C}$], ($X \rightarrow Y$) holds on a sub-tree specified by a class $w_i$, then candidate $\psi' = P_l$: [$\mathcal{C}'$], ($X \rightarrow Y$) related to $w_i$ is ignored, where $\mathcal{C}' \subset \mathcal{C}$.

**Pruning rule 3.3**. Pruning equivalent sets associated with discovered XCFDs. If $\psi = P_l$:($X$= *"a"*)$\rightarrow$($Y$= *"b"*) corresponding to edge($W$, $Z$) holds on data tree $D$, and $X \leftrightarrow Y$ then $Y$ can be deleted.

**Pruning rule 3.4.** Pruning XCFDs which are potentially redundant. If for any $A \in X$, $Y \notin G(X\backslash\{A\}) \cup G(\mathcal{C})$, then skip checking the candidate $\psi = P_l$: [$\mathcal{C}$], ($X \rightarrow Y$).

**Pruning rule 3.5**. Pruning XCFD candidates considered to be trivial. Given a cardinality threshold $\tau$, $\tau >= 2$, we do not consider class $w_i$ containing less than $\tau$ elements i.e. $|w_i| < \tau$. XCFDs associated with such

classes are not interesting. In other words, we only discover XCFDs holding for at least $\tau$ sub-trees.

### 3.4.5 XDiscover Algorithm

We first introduce the concept and the theorem on the Closure set of XCFDs, which is used for completeness of the set of XCFDs discovered by XDiscover. Then, we present the detail of XDiscover. Finally, we also give a theorem (Theorem 3.2) to specify that the set of XCFDs discovered by XDiscover from a given source is greater than or equal to the set of XFDs which hold on that source.

**Definition 3.10.** (Closure set of XCFDs) Let $G$ be a set of XCFDs. The closure of $G$, denoted by $G^+$, is the set of all XCFDs which can be deduced from $G$ using the above Armstrong's Axioms.

**Theorem 3.2.** Let $G$ be the set of XCFDs that are discovered by XDiscover from $D$ and $G^+$ be the closure of $G$. Then, an XCFD $\psi = P_l$: [$\mathcal{C}$], ( $X \rightarrow Y$ ) holds on $T$ iff $\psi \in G^+$.

*Proof:* For a candidate $X$ and $Y$, we first prove that if a constraint XCFD $\psi$ holds on $D$ then the constraint $\psi$ is in $G^+$. After this, we prove that if $\psi$ is in $G^+$ then $\psi$ holds on $D$.

(*i*) Proving if $\psi = P_l$: [$\mathcal{C}$], ( $X \rightarrow Y$ ) holds on $D$ then $\psi \in G^+$

Suppose constraint $\psi$ holds on $D$, $\psi$ may be directly discovered by XDiscover.

- If $\psi$ is discovered by XDiscover, then $\psi \in G$. Therefore, $\psi \in G^+$

- If $\psi$ is not discovered by XDiscover, this means either $X$ is pruned by pruning rule 1 or condition $\mathcal{C}$ is pruned by pruning rule 3.2 or $Y$ is pruned by pruning rule 3.3 and 3.4. Hence, $\psi \in G^+$.

*ii)* Proving if $\psi \in G^+$ then $\psi$ holds on $D$.

Suppose that $\psi = P_l$: [$\mathcal{C}$], ( $X \rightarrow Y$ ) is in $G^+$ but $\psi$ does not hold in $D$

Since $\psi \in G^+$, this means, it can be logically derived from $G$. That is, there exists at least a set of elements $Z$ associated to two constraints in $G$, such that $\psi' = P_l$: [$\mathcal{C}$], ( $X \rightarrow Z$ ) and $\psi'' = P_l$: [$\mathcal{C}$], ( $Z \rightarrow Y$ ) to derive transitively $\psi$. Therefore, $\psi$ is satisfied by $D$. □

**XDiscover algorithm.** Listing 3.1 presents our proposed XDiscover algorithm to find XCFDs from a given data tree $D$. Our algorithm traverses the searching lattice following a breadth-first search manner combining

**Algorithm: 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. $DF \leftarrow \{ \varnothing \}$;
2. Level $d \leftarrow 1$;
3. $PI_d \leftarrow E$;
4. $GP_1 \leftarrow$ generatePartition($D$, $PI_d$);
5. While $|PI_d| \neq \{ \varnothing \}$ do
6. $d$++;
7. $PI_d \leftarrow$ generatePartitionIdentifier($GP_d$);
8. $GP_d \leftarrow$ generatePartition($D$, $PI_d$);
9. $DF \leftarrow DF \cup$ **discoverXCFD**($GP_d$, $GP_{d-1}$);
10. Prune($GP_{d-1}$);
11. Return (*DF*).

**Listing 3.1:** The XDiscover algorithm

with pruning rules. The searching process starts from level 1 ($d$=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 $GP_1$ - Generated Partition (line 4). At level $d$ > 1, node labels are generated from $PI_{d-1}$ and stored in $PI_d$ (line 7) in the form $v_iv_j$; where $v_i \neq v_j$; $v_i$, $v_j \in PI_{d-1}$; $PI_{d-1}$ contains node labels at level ($d$-1); all partitions of $v_iv_j$ nodes at level $d$ are generated and stored in $GP_d$.

**Algorithm: discoverXCFD**

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

*Output:* satisfied XCFDs

1. $DF \leftarrow \{\varnothing\}$;
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. $\Omega_W \leftarrow$ subsumed $w_i$;
6. While $\Omega_W <> \{\varnothing\}$ 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. $DF \leftarrow DF \cup (\mathcal{C}, X \rightarrow Y)$;
11. $\Omega_W \leftarrow \Omega_W \setminus (w_i \in (\mathcal{C}, X \rightarrow Y))$;
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 $\Omega_W$ then
16. $DF \leftarrow DF \cup (\mathcal{C}, X \rightarrow Y)$;
17. $\Omega_W \leftarrow \Omega_W \setminus (w_i \in (\mathcal{C}, X \rightarrow Y))$;
18. Return($DF$).

**Listing 3.2:** The discoverXCFD algorithm

All candidates in the form $c_i, w_i \rightarrow z_j$ are checked; where $v_i = w_i c_i$, $v_j = w_i c_i z_j$ and $z_j \in PI_1 \setminus (w_i \cup c_i)$. The validation for a satisfied XCFD follows the approach described in Section 3.4.3 (line 9: function discoverXCFD in Listing 3.2). The found XCFDs are stored in *DF* - the discovered set of XCFDs. Then the Prune function containing the pruning rules 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.

The function of discoverXCFD depicted in Listing 3.2 searches for XCFDs at each level *d*. If there still exists classes in $\Pi_W$ 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.

The following theorem is to specify that the set of XCFDs discovered by XDiscover from a given source is greater than or equal to the set of XFDs which hold on that source.

**Theorem 3.3.** Let $G$ be the set of XCFDs obtained from $D$ by applying XDiscover and $F$ be a set of possible XFDs hold on $D$, then $|G| \geq |F|$.

*Proof:* we refer to the source instance as $D = (V, lab, ele, att, val, r)$ conforming to a schema $S = (E, A, T, root)$. $G$ is a set of discovered XCFDs. The expression form of XCFD is $\psi = P_l: [\mathcal{C}], (X \rightarrow Y)$.

Let $N$ be a set of elements in $S$, $N = \{e_1, e_2, .., e_n\}$. The domain of $e_i$ is denoted as $dom(e_i)$. $dom(e_i) = \{e_1^i, e_2^i, .., e_k^i\}$, $k > 1$. Assume that $F = \{\varphi_1, \varphi_2, .., \varphi_m\}$ is a set of traditional XFDs on $D$, where $\varphi_i = W_i \rightarrow e_i$, $W_i \subset N$, $e_i \not\subset W_i$, $i = 1..m$.

Suppose that there exist dependencies capturing relationships among data values in $\varphi_i$. This means $\forall\, e_i^t \in dom(e_i),\ \exists\, \psi_i,\ \psi_i = \mathcal{C}_i \rightarrow e_i^t$, where $\forall\, e_c \subset \mathcal{C}_i$, $e_c$ is related to a value in $dom(e_c)$, $\mathcal{C}_i \equiv W_i$, $\psi_i$ is an extension of $\varphi_i$ where each element in either the antecedent or consequence of $\varphi_i$ is a value in its domain. We do not consider an element which has the same value on the whole document. This means the number of distinguished values associated with $e_i$ is greater than 1( $|dom(e_i)| > 1$ ). Therefore, $e_i$ is identified by a set of dependencies $G_i$ extended from $\varphi_i$, instead of only one functional dependency $\varphi_i$. In other words, we have

$$|G_i| > 1 = |\{\varphi_i\}| \qquad (1)$$

Suppose that semantic inconsistencies appear in $D$. This means different dependencies exist to identify the value of the consequence $e_i$ in $\varphi_i$, denoted ($C(\varphi_i)$).

Let $\varphi_i = W_i \rightarrow e_i,\ W_i \subset N,\ e_i \not\subset W_i,\ i=1..m.$

$\forall\, e_i \subset C(\varphi_i),\ \exists\, \psi_i,\ \psi_j$ :

$\psi_i = [\mathcal{C}_i],\ (X_i \rightarrow e_i)$

$\psi_j = [\mathcal{C}_j],\ (X_j \rightarrow e_i)$ ,

where $\psi_i \neq \psi_j$ , $i \neq j$, $\mathcal{C}_i \cup X_i = W_i$, $\mathcal{C}_j \cup X_j = W_i$.

$\forall\, e_c \subset \mathcal{C}_i \cup \mathcal{C}_j$, $e_c$ is related to a value in $dom(e_t)$,

$\forall\, e_v \subset X_i \cup X_j$, $e_v$ is either a value in $dom(e_v)$ or a variable.

We can see that $e_i$ is identified by a set $G'_i$ of conditional dependencies instead of only one functional dependency $\varphi_i$. Hence,

$$|G'_i| >= 2 > |\{\varphi_i\}| \quad (2)$$

Without loss of generality, from (1) & (2), we have $|G| = |\cup_{i=1..m} \{G'_i\}| > |\{\varphi_i\}_{i=1..m}| = |F|$. In other words, the number of discovered XCFDs is much greater than the number of XFDs. Each consequence $e_i$ of a dependency is

identified by a set of XCFDs which include traditional XFDs and its extensions. □

In the following section, we present a summary of the experiments and comparisons between XDiscover and a related approach.

## 3.5 Experimental analysis

We evaluate the performance of our XDiscover using a comprehensive set of experiments on synthetic and real datasets.

### 3.5.1 Synthetic data

*Datasets:* our dataset is on Flight Bookings, which is an extension of the "Flight Bookings" data shown in Fig 3.2. The dataset contains 150 Bookings. All data represents real relationships between elements with inconsistent data rules. Such specifications are needed to verify the existence of constraints holding conditionally on XML data.

*Parameters:* The cardinality threshold $\tau$ determining a minimum number of classes associated with interesting XCFDs was set from 2 to 4 with every step of 1.

*System:* we ran experiments on a PC with an Intel i5, 3.2GHz CPU and 8GB RAM. The implementation was in Java and data was stored in MySQL.

*Comparative evaluation:* to the best of our knowledge, there are no similar techniques for discovering constraints which are equivalent to XCFDs. There is only one algorithm which is close to our work, denoted Yu08, introduced by Yu et al. [102], for discovering XFDs. Such XFDs are considered as XCFDs containing only variables. Both approaches use partitioning techniques with respect to data values to identify dependencies from a given data source. Therefore, we choose Yu08 to draw comparisons with our approach on the number and the semantics of discovered

constraints. Our purpose is to evaluate the correctness of XDiscover in discovering constraints. We ran experiments on the Flight Bookings datasets as described above. The comparisons relate to: (*i)* the number of discovered constraints; and (*ii)* the specifications of the discovered constraints.

| $\tau$ | XDiscover | Yu08 |
|---|---|---|
| | # discovered constraints | # discovered constraints |
| $\tau = 2$ | 23 | 7 |
| $\tau = 3$ | 16 | 7 |
| $\tau = 4$ | 13 | 7 |

**Table 3.1.** XDiscover vs Yu08 on the number of discovered constraints

| XDiscover | Yu08 |
|---|---|
| $P_{\text{Booking}}$: (./Carrier= "Tiger Airways", ./Fare) →./Tax | $P_{\text{Booking}}$: ./Fare →. /Tax |
| $P_{\text{Booking}}$: (./Carrier = "Virgin" ^ ./Trip/Arrival = "BNE") → (./Tax = "20") | $P_{\text{Booking}}$: ./Trip/Departure, ./Trip/Arrival →./Tax |

**Table 3.2.** Samples of constraints discovered by XDiscover vs that of Yu08

The results in Table 3.1 show that while our approach returns from 13 to 23 constraints, Yu08 discovers only 7 constraints. This is because Yu08 does not consider conditional constraints holding on a subset of Flight Bookings as XDiscover does.

Table 3.2 represents the certain number of constraints discovered by XDiscover and Yu08. Yu08 returns inaccurate rules like

$P_{\text{Booking}}$: ./Fare →. /Tax,

$P_{\text{Booking}}$: ./Trip/Departure, ./Trip/Arrival →./Tax

while DisX discovers more specific and accurate dependencies

$P_{Booking}$: (./Carrier = “Virgin” ^ ./Trip/Arrival = “BNE”) → (./Tax = “20”).

In general, the set of constraints discovered by XDiscover is much more numerous than Yu08. This is because XDiscover considers conditional constraints. Yu08 returns inaccurate rules since Yu08 does not consider conditional semantics as XDiscover does. Constraints returned by XDiscover are more specific and accurate than constraints returned by Yu08. The existing algorithm discovers XFDs containing only variables (e.g. ./Fare and ./Tax) and can not detect dependencies which hold partially on documents with conditions. Our approach discovers constraints containing both variables and constants (e.g. ./Carrier= "Virgin" and ./Trip/Arrival="BNE"), or either variables or constants that allow the detection of more interesting semantic constraints than algorithms to discover XFDs.

### 3.5.2 Real life data

Although synthetic dataset can help us analyze the real potential of the approach, experiments on real datasets are necessary to test its practicality. We ran experiments on two available real life datasets including: wikibooks from Wikimedia [96] and the CD dataset as used in [95]. wikibooks consist of about 19 schema elements, the max schema depth being 5. It contains 900 pages (14200 data elements). The CD dataset contains 9763 CDs which is randomly extracted from FreeDB database. It includes 21 schema elements and the max schema depth is 4. The CD dataset contains 298 duplicate objects. We ran XDiscover on these datasets to find the number of checked candidates, the discovery time and the number of discovered constraints in each case. The results summarized in Table 3.3 show that the cardinality threshold influences to the time

consuming and the number constraints discovered by XDiscover. XDiscover works more effectively, in terms of consuming time, when the cardinality threshold is higher. This means XDiscover deals effectively on data sources with constraints holding on a large number of objects.

| Datasets | wikibooks | | | CD Datasets | | |
|---|---|---|---|---|---|---|
| Cardinality threshold $\tau$ | $\tau = 2$ | $\tau = 3$ | $\tau = 4$ | $\tau = 2$ | $\tau = 3$ | $\tau = 4$ |
| #Candidate checked | 221 | 108 | 88 | 427 | 248 | 195 |
| #Discovery time(seconds) | 106 | 95 | 88 | 334 | 258 | 226 |
| #discovered constraints | 15 | 13 | 8 | 61 | 44 | 21 |

**Table 3.3.** Analyzing real life datasets

We present case studies to further demonstrate the effectiveness of the proposed approach in the next section.

## 3.6 Case studies

We use the Flight Booking XML data for our case studies. From schema Bookings *S* in Fig 3.1, we have *E*={Bookings, Booking, Carrier, Trip, Departure, Arrival, Fare, Tax}. The cardinality threshold $\tau$ determines the classes associated with interesting XCFDs. $\tau$ affects the results of XDiscover due to changes in the number of classes which need to be checked. If the value of $\tau$ is too large, then only a small number of equivalent classes is satisfied, which might result in a loss of interesting XCFDs. Therefore, in our case studies, we fix the value of $\tau$ at 2, which means we only consider classes with cardinality equal to or greater than 2. We do not consider constraints holding for only one specific sub-tree, as such constraints are considered trivial.

**Case 3.1.** XCFDs contain only constants.

Suppose data tree *D* in Fig 3.4 conforms to schema Bookings *S* and each Booking only contains one Trip as shown in *D*.

Consider edge(*W, Z*)= (Carrier-Arrival, Carrier-Arrival-Tax). Follow the process described in section 3.4.1 to generate two partitions of Carrier-Arrival and Carrier-Arrival-Tax with respect to the sub-tree rooted at Booking. To simplify the presentation, we omit the node label (e.g. Booking) associated to each node in the classes.

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

$=\{\{(22,1)\}, \{(32,1), (52,1)\}, \{\mathbf{(42,1), (72, 1)}\}, \{(62, 1)\}\}$

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

$=\{\{(22,1)\},\{(32,1)\}, \{\mathbf{(42,1), (72,1)}\}, \{(52,1)\}, \{(62,1)\}\}$

We can see that $w_3$ in $\Pi_{\text{Carrier, Trip/Arrival | Booking}}$ 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" ^ ./Trip/Arrival = "BNE") → (./Tax = "20").

This case demonstrates the XCFD contains only constants. For each XFD, there might exist a number of conditional dependency 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 3.2.** XCFDs contain both variables and constants.

Using the same assumption in case 1, considering edge (*W, Z*)= edge (Fare, Fare-Tax), two partitions of Fare and Fare-Tax with respect to the sub-tree rooted at Booking:

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

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

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

$\Pi_{\text{Carrier, Fare}\,|\,\text{Booking}} = \{\mathbf{w'_1}, w'_2, \mathbf{w'_3}, w'_4\}=\{\{\mathbf{(22,1), (32, 1)}\}, \{(42,1)\},\{\mathbf{(52, 1), (62, 1)}\}, \{(72, 1)\}\}$

$\Pi_{\text{Carrier, Fare, Tax}\,|\,\text{Booking}} = \{ \mathbf{z'_1}, z'_2, \mathbf{z'_3}, z'_4\}=\{\{\mathbf{(22, 1), (32, 1)}\}, \{(42, 1)\}, \{\mathbf{(52, 1), (62, 1)}\}, \{(72, 1)\}\}$

The partition of the condition node Carrier:

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

We have two equivalent pairs $(w'_1, z'_1)$ and $(w'_3, z'_3)$ between $\Pi_{\text{Carrier, Fare}\,|\,\text{Booking}}$ & $\Pi_{\text{Carrier, Fare, Tax}\,|\,\text{Booking}}$ with $|w_1|=2 \geq \tau$ and $|w_3|=2 \geq \tau$. Furthermore, there exists a class $\boldsymbol{c_1}$ in $\Pi_{\text{Carrier|Booking}}$ containing exactly all elements in $w'_1 \cup w'_3$. All elements in class $\mathbf{c}_1$ have the same value for Carrier = "Tiger Airways". This means the nodes in classes $w'_1$ and $w'_3$ share the same condition (./Carrier = "Tiger Airways"). Therefore, an XCFD $\psi = P_{\text{Booking}}$: (./Carrier= "Tiger Airways", ./Fare) →./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 3.3.** Partition identifiers contain a set of complex nodes.

Suppose data tree *D* in Fig 3.5 conforms to schema Bookings *S* in Fig 3.1. Each Booking contains multiple complex nodes Trip. 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 Trip) 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 Trip 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 contain single data nodes.

Consider edge(Trip, Tax) with respect to the sub-tree rooted at Booking. We start generating partitions under the sub-tree rooted at Trip. Following the process described in section 3.4.1, we partition the nodes according to each Trip (including Departure and Arrival) under the sub-tree rooted at Trip:

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

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

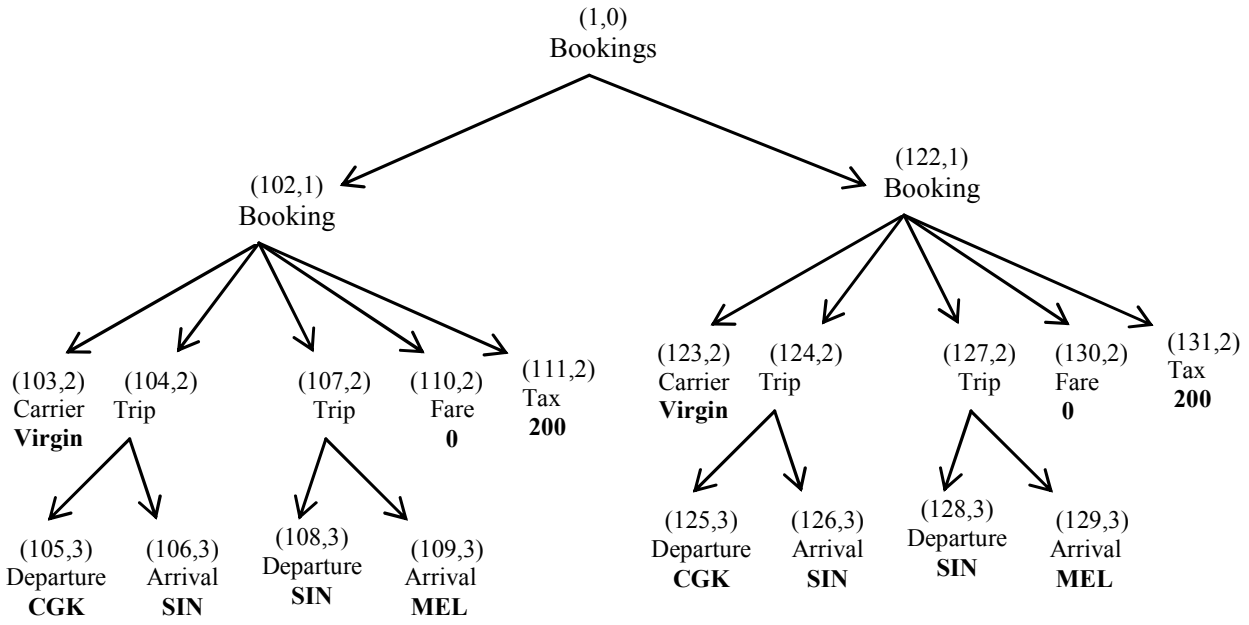


**Fig 3.5.** A simplified Bookings data tree: *each Booking contains a set of complex element Trip*

$\psi_2$=$P_{\text{Booking}}$:(./Carrier="Virgin",{./Trip})→(./Tax); where {./Trip} represents a set of complex data nodes Trip including Departure and Arrival.

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

$\psi'_2 = P_{\text{Booking}}$: (./Carrier = "Virgin", ./Trip)→ (./Tax),

$\psi'_2$ is a special case of $\psi_2$. 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.

## 3.7 Summary

This chapter addressed the issues of data inconsistency caused by semantic inconsistencies. Specifically, we introduced the notion of XML conditional functional dependency which incorporates conditions into dependencies to express constraints with conditional semantics. We proposed the XDiscover algorithm based on semantics hidden in the data to discover a set of possible XCFDs from a given XML data instance. We proposed a set of pruning rules incorporated into the discovery process to improve the performance of XDiscover. Experiments on synthetic and real life datasets, and case studies were used to evaluate XDiscover. In our experiments, we show that XDiscover can discover more situations of dependencies than the XFD approach. XCFDs also have more expressive power, in term of constraining data consistency, than that of XFDs. 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 the manipulation of data. Data inconsistency can be caused by structural inconsistencies inherent in heterogeneous XML data sources. Therefore, our work will be further extended to address such problems in the next chapter.

# Chapter 4

# Structured content-aware discovery for improving XML data consistency

The goal of this thesis is to find principles for improving XML data consistency. The previous chapter introduced a content-based discovery approach to discover XML conditional functional dependencies from a given data source conforming to a given schema. This is to resolve the data inconsistency caused by semantic inconsistencies. Our intention is to extend this approach to deal with data inconsistency caused by either structural or semantic inconsistencies. This chapter introduces a structured and content-based approach to discover anomalies where a data tree does not follow any schema. Our work includes the concept of conditions as in XCFDs and adds a new notion of similarity to work properly in XML data.

## 4.1 Introduction

One of the main features of XML is that it can represent different kinds of data from different data sources. Two predominant proposals exist, namely DTD (Document Type Definition) [49, 54] and XML Schema [90] to

specify the structure of a class of XML documents. However, such proposals have not yet emerged as a standard. In addition, XML documents are flexible which can represent different kinds of data from different data sources. Each source might have its own structural definitions by modifying the original schema [88]. Thus, we cannot assume that each XML document always has a schema defining its structure. In such cases, data inconsistencies often arise from both *structural and semantic inconsistencies* inherent in the heterogeneous XML data sources.

*Structural inconsistencies* arise when the same real world concept is expressed in different ways, with different choices of elements and structures, that is, the same data is organized differently [35, 75, 95]. This is because XML data is integrated from different data sources which might have nearly or exactly the same information but are constructed using different structures. Even though two objects express similar information, each of them may have some extra information with respect to the other. *Semantic inconsistencies* occur when business rules on the same data vary across different fragments [79]. To the best of our knowledge, there is currently no existing approach which fully addresses the problems of data inconsistencies in XML. In the previous chapter, we propose an approach to discover a set of XML conditional functional dependencies (XCFDs) that targets semantic inconsistencies.

This chapter addresses the problem of data inconsistencies caused by both semantic and structural inconsistencies. We assume that XML data are integrated from multiple sources in the context of data integration, in which labeling syntax is standardized and data structures are flexible. We first introduce a novel constraint type, called XML conditional structural functional dependencies (XCSDs), which represent relationships between groups of similar real-world objects under particular conditions. They are

constraints in which functional dependencies are incorporated, not only with conditions as in XCFDs to specify the scope of constraints but also with a similarity threshold. The similarity threshold here is used to specify similar objects on which the XCSD holds. The similarity between objects is measured based on their structural properties using our newly proposed *structural similarity measurement*. Thus, XCSDs are able to validate data consistency on the identified similar, instead of identical, objects in data sources with structural inconsistencies.

In addition, we propose an approach, named SCAD, to discover XCSDs from a given data source. SCAD exploits semantics explicitly observed from data structures and those hidden in the data to detect a minimal set of XCSDs. Structural semantics are derived by our proposed method, called *data summarization*, which constructs a data summary containing only representative data for the discovery process. The rationale behind this is to resolve structural inconsistencies. Semantics hidden in the data are explored in the process of discovering XCSDs. Experiments and case studies on synthetic data were used to evaluate the feasibility of SCAD. The concept of minimal XCSD is the same as that of XCFD (Definition 3.7).

The remainder of this chapter is organized into eight sections. Section 4.2 presents preliminaries. Section 4.3 presents a new measurement, called the structural similarity measurement, which is necessary to introduce the XCSDs described in Section 4.4. Our proposed approach, SCAD, is described in Section 4.5. The complexity analysis of SCAD is presented in Section 4.6. Section 4.7 covers the experiment results. Case studies are presented in Section 4.8. Finally, Section 4.9 concludes the chapter.

## 4.2 Preliminaries

In this section, we give some preliminaries including: (*i)* considering a variety of examples of constraints to further illustrate the anomalies existing in XML data, and discussing the limitations of the existing work in expressing such constraints. This is to emphasize the needs to propose a new type of constraint to capture data inconsistency in XML data; and (*ii)* presenting the definition of a data tree used in this chapter.

### 4.2.1 Constraints

Fig 4.1 is a simplified instance of data tree *T* for Bookings. Each Booking in *T* contains information on Type, Carrier, Departure, Arrival, Fare and Tax. Values of elements are recorded under the element names. We give examples to demonstrate anomalies in XML data. All examples are based on the data tree in Fig 4.1.

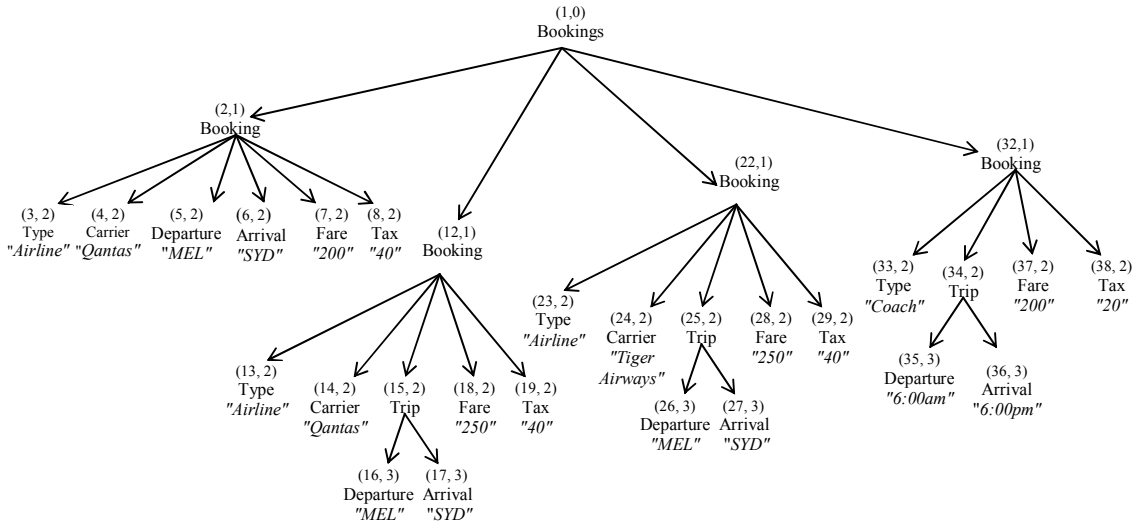


**Fig 4.1**. A simplified Bookings data tree contains structural and semantic inconsistencies

Constraint 1: *For any Booking having the same Fare should have the same Tax.*
Constraint 2a: *For any Booking of "Airline" having Carrier of "Qantas", the Departure and Arrival determines the Tax.*
Constrain 2b: *Any Booking of "Airline" having Carrier of "Tiger Airways", the Fare identifies the Tax.*

Constraint 1 holds for all Bookings in *T*. Such a constraint contains only variables (e.g. Fare and Tax), commonly known as an XFD. Constraints 2a and 2b are only true under given contexts. For instance, constraint 2a holds for Bookings with Type of "Airline" and Carrier of "Qantas". Constraint 2b holds for Bookings with Type of "Airline" and Carrier of "Tiger Airways". These are examples of constraints holding locally on a subset of data. Conditional semantics are common in real data, especially if a data tree contains integrated data from multiple sources, then a constraint may hold only on a portion of the data obtained from one particular source [48]. Constraints 2a and 2b are examples of semantic inconsistencies, that is, for Bookings of "Airline", values of Tax might be determined by different business rules. Tax is determined by Departure and Arrival for Carrier of "Qantas" (e.g. Constraint 2a). Tax is however identified by Fare for Carrier of "Tiger Airways" (e.g. Constraint 2b). We can see that while Bookings of node (2, 1) and node (12, 1) describe the data which have the same semantics, they employ different structures: Departure is a direct child of the former Booking, whereas it is a grandchild of the latter Booking with an extra parent node, Trip. This is an example of structural inconsistencies. Detecting data inconsistencies as violations of XFDs fails due to the existence of such constraints.

We now consider the different expression forms of XFDs under the Path-based approach [82] and the Generalized tree tuple-based approach [102] presented in Table 4.1. It is possible to see that both notions effectively capture the constraints holding on the overall document. For example, Constraint 1 can be expressed in the form of P1 under the Path-based approach and G1 under the Generalized tree tuple-based approach. The semantics of P1 is as follows: "For any two distinct Tax nodes in the data tree, if the Fare nodes with which they are associated have the same value, then the Tax nodes themselves have the same value". The semantics of G1 is, "For any two generalized tree tuples $C_{Booking}$, if they have the same values at the Fare nodes, they will share the same value at the Tax nodes". The semantics of either P1 or G1 are exactly as in the original constraint 1.

| Constraint | Path-based approach [82] | Generalized tree tuple-based approach [102] |
|---|---|---|
| General form | $\{P_{x1},..,P_{xn}\} \rightarrow P_y$, where $P_{xi}$ are the paths specifying antecedent elements, $P_y$: is the path specifying a consequent element. | LHS→ RHS w.r.t $C_p$, where LHS is a set of paths relative to $p$, and RHS is a single path relative to $p$, $C_p$ is a tuple class that is a set of generalized tree tuples. |
| 1 | P1: {Bookings/Booking/Fare} → {Bookings/Booking/Tax} | G1: {./Fare}→ ./Tax w.r.t $C_{\text{Booking}}$ |
| 2a | P2a: {Booings/Booking/Departure, Bookings/Booking/Arrival} → {Bookings/Booking/Tax} | G2a: {./Departure,./Arrival}→ ./Tax w.r.t $C_{\text{Booking}}$ |

**Table 4.1.** Expression forms of XML functional dependencies.

However, neither of the two existing notions can capture a constraint with conditions. For example, the closest forms to which constraint 2a can be expressed under [82] and [102] are P2a and G2a, respectively. The semantics of such expressions is only: "Any two Bookings having the same Departure and Arrival should have the same Tax". Such semantics is different from the semantics of the original Constraint 2a which includes conditions: Booking of "Airline" and Carrier of "Qantas". Moreover, neither existing notions can capture the semantics of constraints holding on similar objects. For example, neither P2a nor G2a can capture the semantic similarity of Booking(2, 1) and Booking(12, 1) (refer to Figure 1). Under such circumstances, these two Bookings are considered inconsistent because Departure and Arrival in Booking(2, 1) and Booking(12, 1) belong to different parents. Departure and Arrival are direct children of the former Booking and are grandchildren of the latter Booking. Our proposed XCSDs address such semantic limitations in expressing the constraints in previous work.

### 4.2.2 XML Data tree

An XML instance is considered as a rooted-unordered-labeled tree. Each element node is followed by a set of element nodes or a set of attribute nodes. An attribute node is considered a simple element node. An element node can be terminated by a text node. An XML data tree is formally defined as follows.

**Definition 4.1.** (XML data tree)
An XML data tree is defined as *T= (V, E, F, root)*, where

- $V$ is a finite set of nodes in $T$, each node $v \in V$ consists of a label $l$ and an *id* that uniquely identify $v$ in $T$. The *id* assigned to each node in the XML data tree, as shown in Figure 1, is in a pre-order traversal. Each *id* is a pair (*order, depth*), where *order* is an increasing integer (e.g. 1, 2, 3...) used as a key to identify a node in the tree; *depth* label is the number of edges traversing from the root to that node in the tree, e.g. 1 assigning for /Bookings/Booking. The depth of the root is 0.
- $E \subseteq V$ x $V$ is the set of edges.
- $F$ is a set of value assignments, each $f(v) = s \in F$ is to assign a string $s$ to each node $v \in V$. If $v$ is a simple node or an attribute node, then $s$ is the content of node $v$, otherwise if $v$ has multiple descendant nodes, then $s$ is a concatenation of all descendants' content.
- *root* is a distinguished node called the root of the data tree.

An XML data tree defined as above possesses the following properties:

For any nodes $v_i, v_j \in V$:

- If there exists an edge$(v_i, v_j) \in E$, then $v_i$ is the parent node of $v_j$, denoted as **parent**$(v_j)$, and $v_j$ is a child node of $v_i$, denoted as **child**$(v_i)$.
- If there exists a set of nodes $\{v_{k1},..,v_{kn}\}$ such that $v_i$ = parent$(v_{k1})$,..,$v_{kn}$ = parent$(v_j)$, then $v_i$ is called an ancestor of $v_j$, denoted as **ancestor**$(v_j)$ and $v_j$ is called a descendant of $v_i$, denoted as **descendant**$(v_i)$.
- If $v_i$ and $v_j$ have the same parent, then $v_i$ and $v_j$ are called **sibling** nodes.
- Given a path $p = \{v_1v_2...v_n\}$, a path expression is denoted as path$(p)$= $/l_1/../l_n$, where $l_k$ is the label of node $v_k$ for all $k \in [1,.., n]$.
- Let $v = (l, id, c)$ be a node of data tree $T$, where $c$ is the content of $v$. If there exists a path $p'$ extending a path $p$ by adding content $c$ into the

path expression of $p$ such that $p' = /l_i/../l_j/c$, then $p'$ is called a **text path**.

- $\{v[X]\}$ is a set of nodes under the subtree rooted at $v$. If $\{v[X]\}$ contains only one node,-it is simply written as $v[X]$.

An XCSD might hold on an object represented by variable structures. In such cases, checking for similar structures is necessary to validate the conformation of the object to that XCSD. To do this, in the next section, we propose a method to measure the structural similarity between two sub-trees.

## 4.3 Structural similarity measurement

The similarity between sub-trees is in general independent of the particular technique adopted to measure the semantics between two XML elements. Any technique which aims to assess whether two elements refer to the same object can be used. Our method follows the idea of structure-only XML similarity [24, 73]. That is, the similarity between sub-trees is evaluated, based on their structural properties, and data values are disregarded. We consider that each sub-tree is a set of paths, and each path starts from the root node and ends at the leaf nodes of the sub-tree. Subsequently, the similarity between two sub-trees is evaluated, based on the similarity of two corresponding sets of paths. The more similar paths the two sub-trees have, the more similar the two sub-trees are.

### 4.3.1 Sub-tree Similarity

Given two sub-trees $R$ and $R'$ rooted at nodes having the same node-label $l$ in $T$. $R$ and $R'$ contain $m$ and $n$ paths, respectively: $R = (p_1,..,p_m)$ and $R' =$

($q_1$,..,$q_n$), where each path starts from the root node of the sub-tree. The similarity between two sub-trees $R$ and $R'$ is denoted by $d_T(R, R')$. Both Cosine [98] and Jaccard [97] functions can be easily adopted to calculate the similarity between two sub-trees. The Cosine function is used to measure the similarity of two non-binary vectors. The Jaccard function is often used to measure the similarity of two objects consisting of asymmetric binary attributes (e.g. 1 and 0). Obviously, the choice of the similarity function is highly dependent on the representation used to describe the two sub-trees. In this work, the similarity between two sub-trees $R$ and $R'$ is evaluated, based on two sets of weights ($w_1$,..,$w_m$) and ($w_1$,..,$w_n$), where $w_i$ and $w'_i$ are the path similarity weights of two paths $p_i$ and $p_j$ in the corresponding sub-trees $R$ and $R'$. The values of $w_i$ and $w'_i$ are real numbers in a range of [0, 1]. Therefore, we used the Cosine similarity formula to compute the similarity between sub-trees.

In our adopted formula, each set of weight can be considered a non-binary vector where each dimension corresponds to a path similarity weight. Consequently, the similarity between two sub-trees is measured based on two non-binary vectors of weights and it is computed as:

$$d_T(R, R') = \frac{\sum_i w_i . w'_i}{\sqrt{\sum_i w_i^2} . \sqrt{\sum_i w_i'^2}},$$

where $w_i$ and $w_i'$ are the path similarity weights of $p_i$ and $q_i$ in the corresponding sub-trees $R$ and $R'$, and the value of $d_T(R, R') \in [0, 1]$ represents that the similarity of two sub-trees changes from a dissimilar to similar status. By defining $d_P(p_i, q_j)$ as the path similarity of two paths $p_i$ and $q_j$, the weight $w_i$ of path $p_i$ in $R$ to $R'$ is calculated as the maximum of all $d_P(p_i, q_j)$, where $1 \leq j \leq n$. The term of *path similarity* $d_P(p_i, q_j)$ is described in the next subsection.

**Algorithm: subtree_Similarity**

*Input*: Two sub-trees $R$ and $R'$ contain paths $(p_1,\ldots, p_m)$ and $(q_1,..,q_n)$ respectively.

*Output*: $d_T(R, R')$

*Process:*

1. //calculating the weight vector of $R$
2. For each path $p_i$ in $R$ do
3. $w_i \leftarrow \max_{j=1..n} \{d_P(p_i, q_j)\}$
4. //calculating the weight vector of $R'$
5. For each path $q_j$ in $R'$ do
6. $w'_j \leftarrow \max_{i=1..m} \{d_P(p_i, q_j)\}$
7. If $m \neq n$ then
8. If $m < n$ then
9. For $k = (m+1)$ to $n$ do $w_k \leftarrow 0$;
10. else if $m > n$ then
11. For $k = (n+1)$ to $m$ do $w'_k \leftarrow 0$;
12. $t \leftarrow \max(m, n)$;
13. $S_1 \leftarrow \sum_{i=1}^{t} w_i \, w'_i$ ;
14. $S_2 \leftarrow \sum_{i=1}^{t} w_i^2$ ; $S_3 \leftarrow \sum_{i=1}^{t} w'^{\,2}_i$
15. $d_T \leftarrow S_1/(S_2^{1/2} \,.\, S_3^{1/2})$;
16. Return($d_T$).

**List 4.1.** The subtree_Similarity algorithm

List 4.1 represents the subtree_Similarity algorithm to calculate the similarity between two sub-trees. The algorithm first calculates the weight $w_i$ of each path $p_i$ in $R$ to $R'$ for all $1 \leq i \leq m$ (line 2-3). Then the weight $w'_j$ of each path $q_j$ in $R'$ to $R$ is calculated for all $1 \leq j \leq n$ (line 5- 6). This means two sets of weights $(w_1,..,w_m)$ and $(w_1,..,w_n)$ are computed. If the cardinalities of the two sets are not equal, then the weights of 0 are added to the smaller set to ensure the two sets have the same cardinality (line 7-

11). The similarity of $R$ and $R'$ is calculated based on these two sets of weights using a Cosine similarity formula (line 13-15). In the following subsection, we describe how to measure the similarity between paths.

### 4.3.2 Path Similarity

Path similarity is used to measure the similarity of two paths, where each path is considered a set of nodes. Consequently, the similarity of two paths is evaluated based on the information from two sets of nodes, which includes *common-nodes*, *gap* and *length difference*. The common-nodes refer to a set of nodes shared by two paths. The number of common-nodes indicates the level of relevance between two paths. The gap denotes that pairs of adjacent nodes in one path appear in the other path in a relative order but there exist a number of intermediate nodes between two nodes of each pair. The numbers of gaps and the lengths of gaps have a significant impact on the similarity between two paths. A longer gap length or a larger number of gaps will result in less similarity between two paths.

Finally, the length difference indicates the difference in the number of nodes in two paths, which in turn, indicates the level of dissimilarity between two paths. We also take into account the node's positions in measuring the similarity between paths. Nodes located at different positions in a path have different influence-scopes to that path. We suppose that a node in a higher level is more important in terms of semantic meaning and hence, it is assigned more weight than a node in a lower level. The weight of a node $v$ having the *depth* of $d$ is calculated as $\mu(v)= (\lambda)^d$, where $\lambda$ is a coefficient factor and $0<\lambda<=1$. The value of $\lambda$ depends on the length of paths.

List 4.2 represents the pathSimilarity algorithm to calculate the similarity of two paths $p= (v_1,..,v_m)$ and $q= (w_1,..,w_n)$, where $v_1$ and $w_1$ have the same node-label *l,* and *m* and *n* are the numbers of nodes in *p* and *q*, respectively. The similarity of two paths *p* and *q*, $d_P(p, q)$, is calculated from three metrics, common-node weight, average-gap weight and length difference reflecting the above factors: common-nodes, gap and length difference (line 1). The common-node weight, $f_c$, is calculated as the weight of nodes with the same node-labels from two paths. The set of nodes with the same node-label between *p* and *q*, called common node-labels, is the intersection of two node-label sets of *p* and *q* (line 3). Assuming that there exist *k* labels in common, the common-node weight can be calculated as:

$$f_c(p,q)= \frac{\sum_{i=1}^{k} \mu(v_i).\mu(w_i)}{\sqrt{\sum_{i=1}^{k} \mu(v_i)^2}.\sqrt{\sum_{i=1}^{k} \mu(w_i)^2}},$$

where $\mu(v_i)$ and $\mu(w_i)$ are the weights of two nodes $v_i$ and $w_i$ in *p* and *q*, respectively. $v_i$ and $w_i$ have the same node-label. The coefficient factor $\lambda = \min(|p|,|q|)/\max(|p|,|q|)$ (line 3). The average-gap weight, $f_a$, is calculated as the average weight of gaps in two paths. The calculation of $f_a$ comprises three steps. First, the algorithm finds the longest gap and the number of gaps between two paths (line 7-9). Second, the gap's weights from one path against the other path and vice versa are calculated. Each gap's weight is calculated based on the total weights of nodes and the number of nodes in the longest gap in that path. The gap's weight of *p* against *q* is calculated by:

$$gw(p, q) = \frac{\sum_{i=1}^{g} \mu(v_i)}{|g|},$$

where *g* is the length of the longest gap of *p* and *q*, and the coefficient factor $\lambda = |g|/|q|$. The same process is applied to calculate the gap's weight

**Algorithm: path_Similarity**

*Input*: two paths $p= (v_1,..,v_m)$ and $q= (w_1,..w_n)$

*Output*: $d_P(p, q)$

Function $d_P(p, q)$

1. $d_P \leftarrow f_c(p, q) - (f_a(p, q) + f_l(p,q))/max(|p|, |q|)$

Return ($d_P$).

Function $f_c$ $(p, q)$ //calculate common node weights of $p_i$ and $q_j$

2. $l^p \leftarrow \{lab(v_1),..,lab(v_m)\}$; $l^q \leftarrow \{lab(w_1),..,lab(w_n)\}$;
3. $comlab \leftarrow l^p \cap l^q$ ; $k \leftarrow |comlab|$;
4. $S_1 \leftarrow \sum_{i=1}^{k} \mu(v_i).\mu(w_i)$ ; $S_2 \leftarrow \sum_{i=1}^{k} \mu(v_i)^2$ ; $S_3 \leftarrow \sum_{i=1}^{k} \mu(w_i)^2$
5. $S \leftarrow S_1/(S_2^{1/2} . S_3^{1/2})$;
6. Return $S$;

Function $f_a(p, q)$ //calculate average gap weight of $p_i$ and $q_j$

7. FindGap($p$ , $q$, $gap1$); FindGap($q$ , $p$, $gap2$);
8. $noG1 \leftarrow |gap1|$; $noG2 \leftarrow |gap2|$;
9. $gap1_{\max} \leftarrow \max_{i=1..noG1}\{\text{gap1}_i\}$; $gap2_{\max} \leftarrow \max_{i=1..noG2}\{\text{gap2}_i\}$
10. $gw_1 \leftarrow \sum_{j=1}^{|gap1_{\max}|} \mu(v_j)/|\text{gap1}_{\max}|$; $gw_2 \leftarrow \sum_{i=1}^{|gap2_{\max}|} \mu(v_i)/|\text{gap2}_{\max}|$;
11. $S \leftarrow (gw_1. noG_1 + gw_2. noG_2)/(noG_1 + noG_2)$;
12. Return $S$;

Function FindGap($p$, $q$, $gap$)

13. For $i$=1 to $m$ do{
14. If $found(v_i , q)$ and $found(v_{i+1} , q)$ then
15. If ($|pos(v_{i+1}, q)- pos(v_i , q)| >1$) then
16. $gap_i \leftarrow subseq(v_i , v_{i+1} , q)$;
17. Else
18. If ($|pos(v_{i+1}, q)- pos(v_i , q)| ==1$) then $gap_i \leftarrow$Null;
19. Else $gap_i \leftarrow p_j$;
20. Return $gap$;

Function $f_l$ $(p, q)$

21. $ld \leftarrow |m- n| /max(m, n)$;

Return ($ld$);

**List 4.2.** The path_Similarity algorithm

of $q$ against $p$ (line 10). Finally, the average of gap's weights is calculated based on two calculated gap's weights and the number of gaps in two paths (line 11). The length difference, $f_l$, is the difference in the number of nodes between two paths (line 21).

For example, given two paths $p$= "Booking/Departure", $q$= "Booking/Trip/Departure", we calculate the similarity score of $p$ and $q$ as follows.

- Calculating the common node weight

  $l^p$ = {Booking, Departure}

  $l^q$ = {Booking, Trip, Departure}

  $comLab(p, q) = l^p \cap l^q$ = {Booking, Departure}

  The *depths* of "Booking" and "Departure" in $p$ and $q$ are {1, 2} and {1, 3}

  The weights in $p$ are $\{2/3, (2/3)^2\}$ and in $q$ are $\{2/3, (2/3)^3\}$.

  $f_c(p, q) = (2/3.\ 2/3 + (2/3)^2.(2/3)^3)/\ (((2/3)^2 + (2/3)^4)^{1/2}.\ ((2/3)^2 + (2/3)^6)^{1/2}) = 0.99$

- Calculating the average gap weight

  Calculating $gw(p, q)$:

  $noG1$ = 1; $gap1_{max}$= "Trip"; $|gap1_{max}|$ =1;

  Assuming that the *depth*("Trip") is 2

  $gw(p, q)$= 0.11

  Calculating $gw(q, p)$

  $noG2$ =2; $gap2_{max}$="Booking/Departure"; $|gap2_{max}|$ =2;

  Assume that *depth*("Booking")=1 and *depth*("Departure")= 2.

  $gw(q, p)$= 1

  The average gap weight $f_a(q, p) = (1/9 * 1 + 1 * 2)/3 = 0.7$

- Calculating the length difference: $f_l(p, q) = 1/3 = 0.33$
- The similarity score of $p$ and $q$: $d_P(p, q) = 0.99-(0.7+0.33)/3 = 0.64$

If the similarity score is larger than a given similarity threshold, then we conclude that the two paths are similar; otherwise, the two paths are not similar. A similarity score equal to 1 indicates that the two paths are the same.

Based on the above definitions, we introduce a new type of constraint, named XML Conditional Structural Functional Dependency (XCSD) in the next section.

## 4.4 XML Conditional Structural Functional Dependency

XML conditional structural functional dependency (XCSD) specifications are defined on the basis of the XFDs used by Fan et al. [42] as in the XCFD definition. The main difference between XCSDs and XCFDs is that XCSD specifications are represented as general forms of constraints composed of a set of dependencies and conditions, which can be used to express both XFDs and XCFDs. In particular, our proposed XCSD specification includes three parts: a functional dependency, a similarity threshold and a Boolean expression.

The function dependency in XCSDs is basically defined as in a normal XFD. The only difference is that instead of representing the relationship between nodes as in XFDs, the functional dependency in an XCSD represents the relationship between groups of nodes. Each group includes nodes with the same label and similar root path. The values of nodes in a certain group are identified by the values of nodes from another group. The similarity threshold in the XCSD is used to set a limit for similar comparisons between paths, instead of equal comparisons as performed on an XFD. The Boolean expression specifies portions of data on which the functional dependency holds.

**Definition 4.2.** (XML conditional structural functional dependency)
Given an XML data tree *T= (V, E, F, root)*, an XML conditional structural functional dependency (XCSD) holding on *T* is defined as:

$\phi = P_l$: [α] [$\mathcal{C}$ ], (*X* → *Y),* where

- α is a similarity threshold indicating that each path $p_i$ in $\phi$ can be replaced by a similar path $p_j$, with the similarity between $p_i$ and $p_j$ being greater than or equal to α, α ∈(0, 1]. The greater value of α, the more similarity between the replaced path $p_j$ and the original path $p_i$ in $\phi$ is required. The default value of α is 1, implying that the replaced paths have to be exactly equivalent to the original path in $\phi$. In such cases, $\phi$ becomes an XCFD [85].
- $\mathcal{C}$ is a condition which is restrictive for the functional dependency $P_l$: *X* → *Y* holding on a subset of *T*. The condition $\mathcal{C}$ has the form: $\mathcal{C}$= $ex_1 \theta ex_2 \theta \ldots \theta ex_n$, where $ex_i$ is an atomic Boolean expression associated to particular elements. "$\theta$" is a logical operator either *AND* (^) or *OR* (∨). $\mathcal{C}$ is optional; if $\mathcal{C}$ is empty then $\phi$ holds for the whole document.
- *X* and *Y* are groups of nodes under sub-trees rooted at node-label *l* and nodes of each group have similar root paths. *X* and *Y* are exclusive.
- *X*→ *Y* indicates a relationship between nodes in *X* and *Y*, such that any two sub-trees sharing the same values for *X* also share the same values for *Y,* that is, the values of nodes in *X* uniquely identify the value of node in *Y*.

For example, there exist two different XFDs relating to Tax. The first XFD is, $P_{\text{Booking}}$:./Departure, ./Arrival→ ./Tax holding for Bookings having Carrier of "Qantas" and the second XFD is, $P_{\text{Booking}}$:(.

/Fair → ./Tax) holding for Bookings having the Carrier of "Tiger Airways". If each XFD holds on groups of similar Bookings with a similarity threshold of 0.5, then we have two corresponding XCSDs.

$\phi_1 = P_{\text{Booking}}$: (0.5) (./Carrier="Qantas″), (./Departure, ./Arrival → ./Tax)

$\phi_2 = P_{\text{Booking}}$: (0.5) (./Carrier="Tiger Airways″), (. /Fair → ./Tax).

Either $\phi_1$ or $\phi_2$ allow identifying the Tax in different Bookings with a similarity threshold of 0.5. $\phi_1$ is only true under the condition of Carrier = "Qantas" and $\phi_2$ is true under the condition of Carrier="Tiger Airways″. Such XCSDs are constraints capturing on sources which have structural and semantic inconsistencies.

**Satisfaction of an XCSD:** The consistency of an XML data tree with respect to a set of XCSDs is verified by checking for the satisfaction of the data to every XCSD. A data tree *T= (V, E, F, root)* is said to satisfy an XCSD $\phi = P_l$: [α] [$\mathcal{C}$], ($X \rightarrow Y$) denoted as $T| = \phi$ if any two sub-trees $R$ and $R'$ rooted at $v_i$ and $v_j$ in $T$ having $d_t(R, R') \geq \alpha$ and if $\{v_i[X]\} =_v \{v_j[X]\}$ then $\{v_i[Y]\} =_v \{v_j[Y]\}$ under the condition $\mathcal{C}$, where $v_i$ and $v_j$ have the same root node-label $l$.

For example, assume that $\phi = P_{\text{Booking}}$: (0.5) (./Carrier="Qantas″), (./Departure, ./Arrival → ./Tax) and the similarity between two sub-trees rooted at nodes (2, 1) and (12, 1) is 0.64, which is greater than the given similar threshold ($\alpha = 0.5$). We are then able to derive that $T| = \phi$.

In the next section, we will present our proposed approach, SCAD, for discovering XCSDs from a given XML source.

## 4.5 SCAD approach: Structured Content-Aware Discovery approach to discover XCSDs

Given an XML data tree *T= (V, E, F, root),* SCAD intends to discover a set of minimal XCSDs in the form $\phi = P_l$: [α][$\mathcal{C}$],($X \rightarrow Y$), where each XCSD is minimal and contains only a single element in the *consequence Y*. Fig 4.2 represents an overview of the SCAD algorithm, consisting of two phases. First, a process called *data summarization* analyzes the data structure to construct a data summary containing only representative data for the discovery process. This is to resolve structural inconsistencies. Second, the semantics hidden in the data are explored by a process called Discovery to discover XCSDs. This is to deal with semantic inconsistencies.

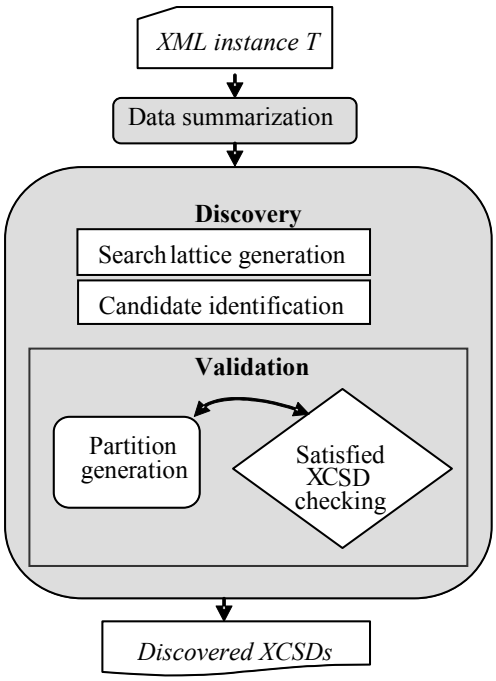


**Fig 4.2.** An overview of the SCAD approach

### 4.5.1 Data summarization: resolving structural inconsistencies

Data summarization is an algorithm constructing a data summary by compressing an XML data tree into a compact form to reduce structural diversity. The path similarity measurement is employed to identify similar paths which can be reduced from a data source. Principally, the algorithm traverses through the data tree following a depth first preorder and parses its structures and content to create a data summary. The summarized data are represented as a list of node-labels, values and node-*ids* where corresponding nodes take place. The summarized data only contains text-paths, each of which is ended by a node containing a value (as described in Section 3). For each node $v_i$ under a sub-tree rooted at node-label $l$, the *id* and *values* of nodes are stored into the list $LV[]|l$. To reduce the structural diversity, all similar root-paths of nodes with the same node-label are stored exactly once by using an equivalent path. That is, if a node $v_i$ can be reached from roots of two different sub-trees by following two similar paths $p$ and $q$, then only the path with a smaller length between $p$ and $q$ is stored in $LV$. Original paths $p$ and $q$ are stored in a list called $OP[]|l$. The data in $LV$ are used for the discovery process. The data stored in the $OP$ are used for tracking original paths. We use the path similarity measurement technique, as described in section 4.2, to calculate the similarity between paths.

In particular, the data summarization algorithm in List 4.3 works as follows. For each node $v_i$, if the root path of $v_i$ is a text path (line 4), then the existing label $l_i$ of node $v_i$ in the $OP$ is checked. If $l_i$ does not exist in $OP$, then a new element in $OP$ with identifier $l_i$ is generated to store the root-path of $v_i$ (line 8); and a new element in the $LV$ with identifier $l_i$ is generated to store the *value* and the *id* of node $v_i$ (line 9). If $l_i$ already exists in the $OP$ at $t$, and the root paths of $v_i$ are not equal but are similar to any

paths stored at $OP[l_i]$ (line 12), then we add the root-path of $v_i$ to $OP[l_i]$ (line 14) and add its *id* and *value* to $LV[l_i]$ (line 15). If there exists an element in *OP* which is equal to $l_i$, then only its *id* and *value* are added to $LV[l_i]$ (line 18).

For example, if we consider the sub-tree rooted at Booking (Fig 4.1), nodes with the label Departure and the path "Booking/Departure"

**Algorithm: data_Summarization**

*Input*: an XML data tree *T=(V, E, F, root)*

*Output*: The summarized document *D=(LV, OP)* for *T*

*Process*:

```
1.  Create empty lists LV[]←{Ø }; OP[]←{Ø };
2.  Traversing the XML data tree in pre-order
3.  For each node v_i do
4.    If not Empty(text(v_i)) then
5.      p_i ← root_context_path(v_i);
6.          l_i ← lab(v_i);
7.          If not exist OP[l_i] then
8.              Generate_New(OP[l_i]); adding p_i to OP[l_i];
9.              Generate_New(LV[l_i]); adding id_val(v_i) to LV[l_i];
10.         else
11.             For each element t in OP[l_i ] do
12.                 If (t<> p_i) then
13.                   If (PathSim(t, p_i)>= α) then
14.                     adding p_i to OP[l_i];
15.                     adding id_val(v_i) to LV[l_i];
16.                     exitFor;
17.                 else
18.                   adding id_val(v_i) to LV[l_i];
19. Return D;
```

**List 4. 3**. The data_Summarization algorithm

occur at node (5,2*)* with a value of "MEL". We first assign *LV*[Departure]|$_{Booking}$={(5,2)*MEL*}, *OP*[Departure]|$_{Booking}$= {"Booking/ Departure"}. The label Departure also appears at nodes (16, 3)MEL, (26, 3)MEL and (35,3)6:00am. The root path of node (16, 3) is "Booking/Trip/Departure" which is different to the stored path "Booking/Departure" in the *OP* list, hence we calculate the similarity between $p_1$="Booking/Departure" and $p_2$="Booking/Trip/Departure", $d_P(p_1, p_2)$= 0.64.

Assuming a threshold for similarity α = 0.5, then two paths $p_1$ and $p_2$ are similar. We continue to add the *id* and the value of node (16, 3) to the list *LV*: *LV*[Departure]|$_{Booking}$= {(5, 2)MEL), (16, 3)MEL)}. Original root path $p_2$ is added to *OP*:

*OP*[Departure]|$_{Booking}$={"Booking/Departure","Booking/Trip/ Departure"}.

Performing the same process for nodes (26,3) and (35,3) then we have LV[Departure]|$_{Booking}$= {(5, 2*)*MEL, (16, 3)MEL, (26,3)MEL, (35,3) 6:00am}.

We use the summarized data as input for the discovery phase. The next section presents the discovery process.

### 4.5.2 XCSD Discovery: resolving semantic inconsistencies

The XCSD discovery algorithm works in the same manner as XDiscover. The main difference is that instead of discovering constraints from the given data tree as in XDiscover, the XCSD discovery algorithm tries to discover non-trivial XCSDs from the data summarization. This is to avoid returning redundant constraints. The discovery of XCSDs comprises three main stages which are performed on the summarized data. The first stage, named *Search lattice generation*, is to generate a search lattice containing

all possible combinations of elements in the summarized data. The second stage is *Candidate identification* which is used to identify possible candidates of XCSDs. The identified candidates are then validated in the last stage, called *Validation*, to discover satisfied XCSDs. The process of each stage is the same as that in XDiscover.

We adopted five pruning rules used in XDiscover to remove redundant and trivial candidates from the search lattice to improve the performance of SCAD. The first three rules are used to skip the search for XCSDs that are logically implied by the already found XCSDs. The last two rules are used to prune redundant and trivial XCSD candidates.

**Pruning rule 4.1**. Pruning supersets of nodes associated with the antecedent of already discovered XCSDs. If $\phi = P_l$: [α][$\mathcal{C}$], ($X \rightarrow Y$) holds, then candidate $\phi' = P_l$: [α][$\mathcal{C}$], ($X' \rightarrow Y$) can be deleted where $X'$ is a superset of $X$.

**Pruning rule 4.2**. Pruning subsets of the condition associated with already discovered XCSDs.

If $\phi = P_l$: [α][$\mathcal{C}$],($X \rightarrow Y$) holds on a sub-tree specified by a class $w_i$, then candidate $\phi' = P_l$: [α][$\mathcal{C}'$],($X \rightarrow Y$) related to $w_i$ is ignored, where $\mathcal{C}' \subset \mathcal{C}$.

**Pruning rule 4.3**. Pruning equivalent sets associated with discovered XCSDs.

If $\phi = P_l$:[α] ($X$= "$a$")$\rightarrow$($Y$= "$b$") corresponding to edge($W$, $Z$) holds on data tree $T$, and $X \leftrightarrow Y$ then $Y$ can be deleted.

**Pruning rule 4.4**. Pruning XCSDs those are potentially redundant.

If for any $A \in X$, $Y \notin G(X\backslash\{A\}) \cup G(\mathcal{C})$ then skip checking the candidate $\phi = P_l$: [α][$\mathcal{C}$], ($X \rightarrow Y$).

**Pruning rule 4.5**. Pruning XCSD candidates considered to be trivial.
Given a cardinality threshold $\tau$, $\tau$ >=1, we do not consider class $w_i$ containing less than $\tau$ elements i.e. $|w_i|<\tau$. XCSDs associated with such classes are not interesting. In other words, we only discover XCSDs holding for at least $\tau$ sub-trees.

According to the above theoretical foundation and ideas, we describe the detail of the SCAD algorithm in the following section.

### 4.5.3 SCAD algorithm

Given a data tree *T,* we are interested in exploring all minimal XCSDs existing in *T*. We adopt the Apriori-Gen algorithm [4] to generate a search lattice containing all possible combinations of node-labels stored in the summarized data LV. For $W= X\cup\mathcal{C}$ & $Z=W\cup\{Y\}$, where *W* and *Z* are nodes in the search lattice, to find all minimal XCSDs of the form $\phi = P_l$: $[\alpha][\mathcal{C}],(X \rightarrow Y)$, we search through the search lattice level by level from nodes of single elements to nodes containing larger sets of elements. For a node *Z*, SCAD tests whether a dependency of the form $Z\backslash\{Y\}\rightarrow\{Y\}$ holds under a specific condition $\mathcal{C}$, where *Y* is a node of single element.

Applying a small to large direction guarantees that only non-redundant XCSDs are considered. We apply pruning rules 1 and 2 to prune supersets of antecedent and the supersets of the condition associated with already discovered XCSDs to guarantee that each discovered XCSD is minimal. That is, we do not consider *Y* in a candidate with antecedent *X'* is a superset of *X*. For every class $w_i$ of $\Pi_W$ that satisfies a minimal XCSD $\phi = P_l$: $[\alpha][\mathcal{C}],(X \rightarrow Y)$, we do not consider $w_i$ in candidate XCSDs $\phi' = P_l$: $[\alpha][\mathcal{C}'],(X \rightarrow Y)$ where $\mathcal{C}' \subset \mathcal{C}$. $w_i$ might be considered in the next candidates with conditions not including $\mathcal{C}$.

We adopted the "compute_dependecies" algorithm in TANE [53] to test for a minimal XCSD. For a potential candidate $Z\backslash\{Y\} \rightarrow \{Y\}$, we need to know whether $Z' \setminus \{Y\} \rightarrow \{Y\}$ holds for some proper subset $Z'$ of $Z$. This information is stored in the set $R(Z')$ of the right-hand side candidates of $Z'$. If $Y$ in $R(Z)$ for a given set $Z$, then $Y$ has not been found to depend on any proper subset of $Z$. It suffices to find minimal XCSDs by testing that $Z\backslash\{Y\} \rightarrow \{Y\}$ holds under a condition $\mathcal{C}$, where $Y \in Z$ and $Y \in R(Z\backslash\{A\})$ for all $A \in Z$.

List 4.4 presents our proposed SCAD algorithm to discover XCSDs from an XML data tree $T$. The summarized data $D$ is extracted from $T$ (line 1). The algorithm traverses the search lattice using the breath-first search manner combining the pruning rules described in Section 6.3.2. The search

**Algorithm: SCAD**

*Input*: An XML data tree $T$, a similar threshold $\alpha$

*Output*: a set of XCSDs

12. $LV \leftarrow$dataSummarization($T$); //List 4.3
13. Init $G \leftarrow \{ \varnothing \}$; $d \leftarrow 1$;
14. $NL_d \leftarrow$nodeLabel($LV$);
15. $GP_d \leftarrow$ generatePartition($d$);
16. While $|NL_d| \neq \{ \varnothing \}$ do
17. increment $d$;
18. $NL_d \leftarrow$ generateNodeLabel($d$);
19. $GP_d \leftarrow$ generatePartition($d$);
20. $G \leftarrow G \cup$ findXCSD($d$);
21. prune($d$);
22. Return ($G$);

**List 4.4.** The SCAD Algorithm

process starts from level 1 ($d$=1). Node-labels at level $d=1$ are a set of node labels from $LV$ which are stored in $NL_d$ in the form $NL_d = \{l_1, l_2, ..., l_n\}$ (line 3). Node-labels at level $d > 1$ are generated by generateNodeLabel in List 4.5 (line 7). Each node label in level $d$ is calculated from node-labels in $NL_{d-1}$ in the form $l_i l_j$ , where $l_i \neq l_j$ , $l_i, l_j \in \mathrm{NL}_{d-1}$. Each node-label might be associated with some candidate *XCSD*s. The generatePartition (List 5) partitions nodes in level $d$ into partitions based on data values. Each candidate XCSDs in the form $c_i, w_i \rightarrow z_j$ is checked for a satisfied XCSD by the sub-function findXCSD in List 4.5 (line 9).

The findXCSD function finds candidate XCSDs at level $d$. A checking process (following the ideas described in 4.2.3) is performed to check for a satisfied XCSD. Pruning rules are employed to prune redundant XCSDs and eliminate redundant nodes from the search lattice for generating candidate XCSDs in the next level (line 10). The searching process is repeated until there are no more nodes in $NL_d$ to be considered (line 5). Any XCSDs found from the findXCSD function are returned to SCAD. The output of SCAD is a set of XCSDs.

**Algorithm: generateNodeLabel**

*Input*: level $d$

*Output:* a list of node $NL_d$

1 $NL_d \leftarrow \{\varnothing\}$; $PB \leftarrow$ prefixBlock($NL_d$);

2 For each prefix block $P$ in $PB$ do

3 For each $\{X_1, X_2\}$ in $P$ do

4 If $(X_1 \neq X_2)$ then $X \leftarrow X_1 \cup X_2$

5 If for all $A$ in $X$, $X \setminus \{A\} \in NL_{d-1}$ then $NL_d \leftarrow NL_d \cup \{X\}$ ;

6 Return $NL_d$

**Algorithm: generatePartition**

*Input:* level $d$

*Output:* a list of generated partitions $GP_d$ at level $d$

1 For each node $W$ of $NL_d$ at level $d$ do

2 If $d$ =1 then $\Pi_W \leftarrow$ classified($LV$, $W$)

3 Else

4 $X \leftarrow$ prefixBlock($W$); $Y \leftarrow W \setminus X$; $\Pi_W \leftarrow \Pi_X \cap \Pi_Y$; $GP_d \leftarrow GP_d \cup \Pi_W$;

5 Return $GP_d$

**Algorithm: findXCSD**

*Input: d*

*Output:* discovered *XCSD*s

1. $G \leftarrow \{\varnothing\}$;
2. For each node $Z \in NL_d$ do
3. $R(X) \leftarrow \cap_{A \in X} R(X \setminus \{A\})$; $R(\mathcal{C}) \leftarrow \cap_{A} \in \mathcal{C} R(\mathcal{C} \setminus \{A\})$;
4. For each node label $W \in NL_{d-1}$ do
5. For each node label $Z \in NL_d$ do
6. If $((Z \setminus W) = \{Y\})$ then
7. For each class $w_i \in \Pi_W$ do
8. For each class $z_j \in \Pi_Z$ do
9. If $w_i = z_j$ and $(|w_i| > \tau)$ then $\Omega_W \leftarrow$ subsumed $w_i$;
10. If $\Omega_W <> \{\varnothing\}$ then
11. While $\Omega_W$ do
12. If $(\mathcal{C}, X \rightarrow Y)$ is valid then $G \leftarrow G \cup (\mathcal{C}, X \rightarrow Y)$
13. Else
14. For each $c_i$ in $\mathcal{C}$ do
15. If $c_i$ contains values only from $\Omega_W$ then $G \leftarrow G \cup (\mathcal{C}, X \rightarrow Y)$;
16. $\Omega_W \leftarrow \Omega_W \setminus (w_i \in (\mathcal{C}, X \rightarrow Y)$;
17. $R(X) \leftarrow R(X) \setminus \{Y\}$; $R(\mathcal{C}) \leftarrow R(\mathcal{C}) \setminus \{w_i\}$;
18. If $(X \leftrightarrow Y)$ then $R(Y) \leftarrow \{\varnothing\}$
19. Return ($G$).

**Algorithm: prune**

*Input: d*

1. For each node $W \in NL_d$ do
2. If $R(W) = \varnothing$ then delete $W$ from $NL_d$

**List 4.5.** Utility functions of SCAD

In the following section, we briefly analyze the complexity of our approach in the worst case and provide further discussion on the practical analysis.

## 4.6 Complexity analysis

The complexities of SCAD mostly depend on the size of the summarized data, which is determined by the number of elements and the degree of similarity amongst the elements in the data source. The time required varies from different datasets. The worst case occurs when the data source does not contain any similar elements or SCAD does not find any constraints. In such a case, the size of the summarized data $|LV|$ is $n$, where $n$ is the number of nodes in the original data tree $T$. Without considering the handling of path similarity, the function dataSumarization makes $n^2$ random accesses to the dataset.

Let $s_{max}$ be the size of the largest level and $S$ be the sum of the sizes of all levels in the search lattice. In the worst case, $S=2^{|LV|}$and $s_{max}= 2^{|LV|}/\sqrt{|LV|}$. During the whole computation, total $S$ partitions are formed, procedure generateNodeLabel makes $S|LV|$ random accesses, the generatePartition makes $S$ random accesses, procedure findXCSD makes $S$|LV| random accesses and procedure prune makes $S$ random accesses. In summary, SCAD has time complexity of $O( n^2 +2 S(|LV|+1) )$. SCAD needs to maintain at most two levels at a time. Hence, the space complexity is bounded by $O(2s_{max})$.

In the worst case analysis, SCAD has exponential time complexity that cannot handle a large number of elements. However, in practice, the size of the summarized data $|LV|$ can be significantly smaller than $n$ as in the worst case due to the similar features in XML data. The more similar

elements are in the original data, the smaller the size of $LV$ is. In addition, by employing the pruning strategies, the size of the largest level $s_{max}$ and the sum of the sizes $S$ can be reduced significantly because the redundant nodes are eliminated from the search lattice.

Suppose that a node $Y$ is eliminated from the search lattice at level $d$, $1<d<n$, then all descendent nodes of $Y$ from level $d+1$ will be deleted from the search lattice by the pruning rules. The number of descendent nodes of $Y$ is $2^{|LV|-d} - 1$. This means the complexity of SCAD reduces by $2^{|LV|-d} - 1$ for every node deleted from the search lattice. The more nodes which are removed from the search lattice, the less time complexity of SCAD. Moreover, in order to avoid discovering trivial XCSDs, the minimum value of the cardinality threshold is often set to at least 2. Thus, the number of checked candidates is reduced considerably. Therefore, the time and space complexity of SCAD are significantly smaller than $O( n^2 +2 S(|LV|+1) ) - 2^{n-d} - 1)$ and $O(2s_{max})$, respectively.

In the following section, we present a summary of the experiments and comparisons between our approach and related approaches.

## 4.7 Experimental analysis

*Datasets:* Synthetic data have been used in our test cases to avoid the noise in real data. The results from synthetic data, in some ways, show the real potential of the approach. Our synthetic dataset is an extension of the "Flight Bookings" data shown in Fig 4.1. The dataset covers common features in XML data, including structural diversity and inconsistent data rules which are needed to verify the existence of constraints holding conditionally on similar objects in XML data. The original dataset contained 150 Bookings (FB1). The DirtyXMLGenerator [72] made

available by Sven Puhlmann was used to generate synthetic datasets. We specified that the percentage of duplicates of an object is 100% to generate a dataset containing similar Bookings. From 150 duplicate Bookings, we specified 20% of data was missing from the original objects so that the dataset contained similar objects with missing data (FB2).

*Parameters:* we set the value of the similarity threshold α from 0.25 to 1 with every step of 0.25. The value of cardinality threshold $\tau$ determining a minimum number of classes associated with interesting XCSDs was set to a default value of 2.

*System:* We ran experiments on a PC with an Intel i5, 3.2GHz CPU and 8GB RAM. The implementation was in Java and data was stored in MySQL.

We first ran experiments to analyze the influence of the similarity threshold on the performance of SCAD. This is to evaluate the effectiveness of our approach in dealing with structural inconsistencies. Then, we ran experiments to make comparisons between SCAD and Yu08 [102] on the numbers and the semantics of discovered constraints. Our purpose is to evaluate the correctness of SCAD in discovering constraints.

**Effectiveness in structural inconsistency***:* we ran experiments on FB1 and FB2 to find the number of checked candidates and the processing times to evaluate the effectiveness of SCAD in dealing with structural diversity. The results are in Fig 4.3 and Fig 4.4. We first analyze the influence of the similarity threshold on the performance of SCAD. Then, we examine the impact of the number of similar objects on the performance of SCAD. The results show that when the similarity

threshold increases from 0.25 to 1 in either FB1 or FB2, the number of checked candidates (Fig 4.3) which lead to the time consumption (Fig 4.4) increase significantly.

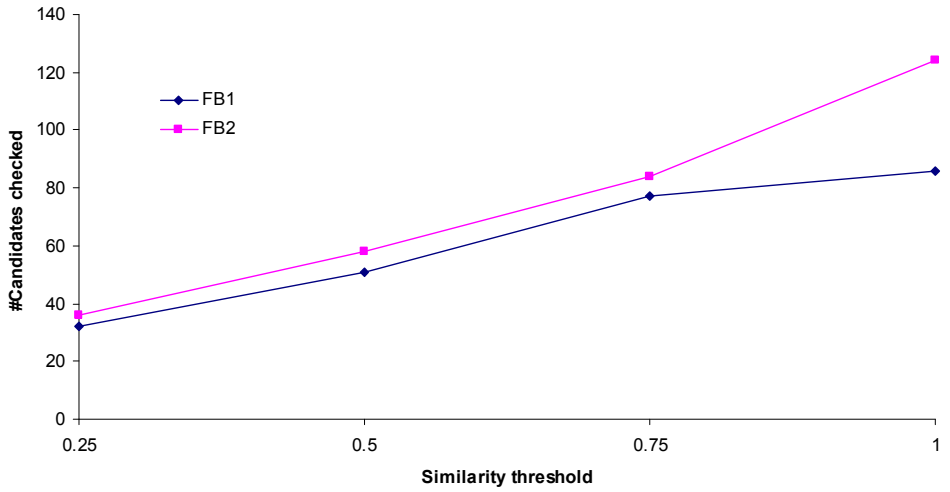


**Fig 4.3**. Numbers of candidates checked vs similarity threshold

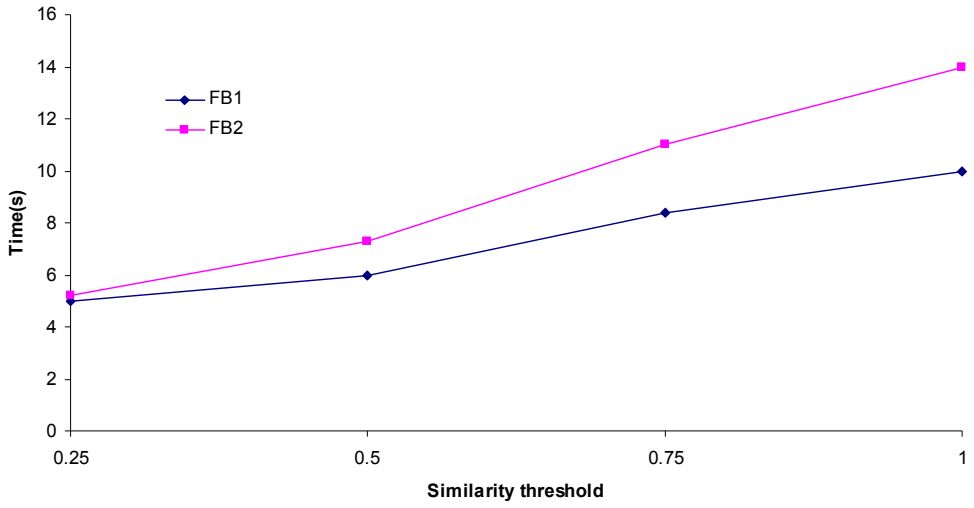


**Fig 4.4.** Time vs similarity threshold

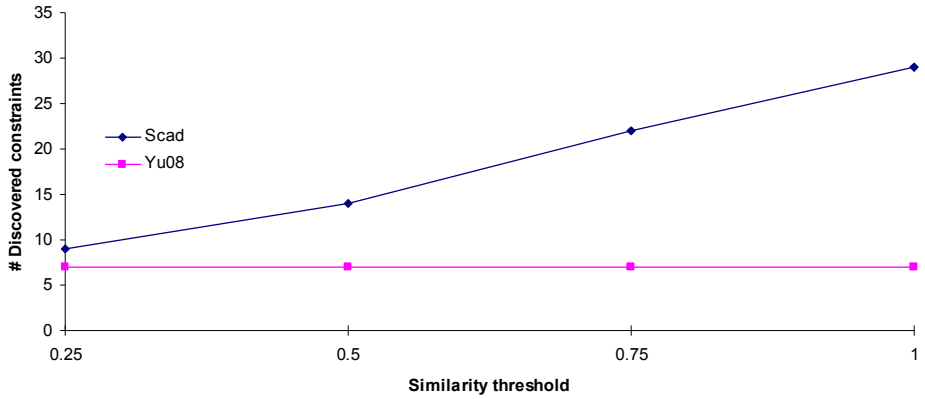


**Fig 4.5**. SCAD vs Yu08

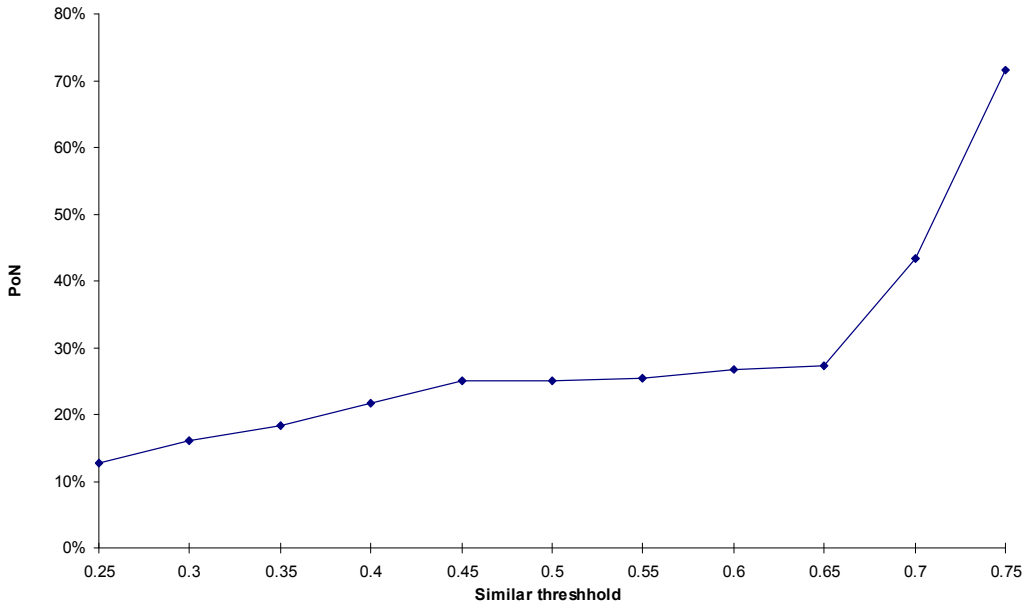


**Fig 4.6**. Range of similarity thresholds

The number of discovered constraints at α of 1 is more than 2.5 times of that at α of 0.25 in either FB1 or FB2. This is because the number of similar elements reduces. The same situation exists for the consumption of time. The processing times increase from 2 to 2.5 times for FB1 and FB2, respectively when α increase from 0.25 to 1.

Moreover, in cases where the similarity threshold α is set to 0.25, while the size of FB2 is as twice that of FB1, the number of checked candidates in two datasets are not much different. When the similarity threshold is set to a higher value, the gap between the numbers of checked candidates between in FB1 and FB2 is considerable. For example, the number of checked candidates in FB2 is more than 1.5 times that in FB1 at α of 1. The same circumstances also happen for the time consumption. The processing times of FB1 and FB2 are nearly the same at α of 0.25; they are significantly different at α of 1 which is nearly 1.5 times. This is because when the similarity threshold increases, the number of elements considered similar in either FB2 or FB1 reduces. This results in the size of summarized data for discovering XCSDs of FB2 being significant larger than that of FB1. Overall, SCAD works more effectively for datasets which contain more similar elements. This means SCAD deals effectively on data sources containing structural inconsistencies.

According to our analysis in Section 4.6, the worst-case time complexity of SCAD is exponential with respect to the number of elements. However, the results from Fig 4.4 show that the processing time is essentially determined by the degree of similarity amongst elements in the data source (i.e. α). SCAD time is proportional to the number of objects in the data summarization that is nearly linear. SCAD saves a significant fraction of the computation compared to the worst-case analysis.

**Comparative Evaluation**: to the best of our knowledge, there are no similar techniques for discovering constraints, which are equivalent to XCSDs. There is only one algorithm which is close to our work, denoted

Yu08, introduced by Yu et al. [102], for discovering XFDs. Such XFDs are considered as XCSDs containing only variables. Thus, we choose Yu08 to draw comparisons with SCAD. We ran experiments on dataset FB1. The value of the similarity threshold α was set from 0.25 to 1 for every step of 0.25. The results in Fig 4.5 show that the number of constraints returned by SCAD is always larger than that of Yu08. This is because SCAD considers conditional constraints holding on a subset of FB1. The number of constraints returned by SCAD also increases significantly when the similarity threshold α increases, whereas the number of constraints discovered Yu08 are stable, because Yu08 does not consider structural similarity between elements as SCAD does.

In cases where the similarity is set to a low value, such as α of 0.25, the number of constraints discovered by SCAD and Yu08 is not much different. The gap between these numbers becomes larger in cases where the similarity threshold is set to a higher value. For example, the number of constraints discovered by SCAD is about 3.5 times larger than that of Yu08 in cases when the similarity threshold is set to 0.5 and about 4 times larger at α of 1.

Since the structural similarity between elements is not considered, constraints returned by Yu08 are redundant.

Yu08 returns redundant constraints like

$P_{\text{Booking}}$: ./Departure, ./Arrival → ./Tax ,

$P_{\text{Booking}}$: ./Trip/Departure, ./Trip/Arrival → ./Tax

while SCAD discovers more specific and accurate dependencies

$P_{\text{Booking}}$:(0.5)(./Type="Airline"^./Carrier="Qantas"
^./Departure="MEL"^./Arrival = "BNE" → ./Tax = "65").

In general, the set of constraints discovered by SCAD is much larger than Yu08. Constraints returned by SCAD are more specific and

accurate than constraints returned by Yu08. A disadvantage of SCAD is that SCAD constructs a data summary containing only representative data for the discovery process to resolve structural inconsistencies. This allows SCAD to work effectively for datasets containing similar elements; however, if there are no similar elements in a data source, the process of data summary is still performed which affects the processing time.

## 4.8 Case studies

We use two case studies to further demonstrate the feasibility of our proposed approach, SCAD, in discovering anomalies from a given XML data. The first case illustrates the effectiveness of SCAD in detecting dependencies containing only constants by binding specific values to elements in XFD specification. The second case aims to demonstrate the capability of SCAD in discovering constraints containing both constants and variables. Our purpose is to point out that SCAD can discover situations of dependencies that the XFD discovery approach cannot detect.

In our approach, the similarity threshold α and cardinality threshold $\tau$ are dataset dependent. The similarity threshold α determines the similarity level of paths for grouping. The cardinality threshold $\tau$ determines the size of classes for checking a candidate XCSD. The settings of these parameters have a great impact on the results of SCAD. If α is too small, then a large number of paths considered to be similar for grouping is returned, which might lead to the issue of important data missing in the summarized data. Consequently, the advantages reduce at a lower similarity threshold, since SCAD might discard some interesting XCSDs. In contrast, if α is too large, the advantages also decrease since

the number of paths identified as similar for grouping is small, leading to the fact that the summarized data might contain duplicate data. This causes the possibility that the set of returned XCSDs might contain redundant and trivial data rules. The execution time also increases. Therefore, the selection of α should be based on a percentage of nodes in the summarized data compared with that in the data source (PoN) so that the summarized data is small enough to take full advantage of the discovery process.

The similarity threshold α is data dependent so its value should be chosen by running experiments on sample datasets. The value of α should be selected from a range of values where such PoNs are stable. This is to ensure that the discovered XCSDs are non-trivial and the execution time is acceptable. In our experiments, the original FB1 dataset is used to find the similarity threshold. We ran the data summarization algorithm (List 4.3) to find the summarized data and calculated the PoN for every value of α, where α varied from 0.25 to 0.75 with every step being 0.05. The results in Figure 6 show that the PoN is stable in the range of similarity thresholds from 0.45 to 0.55. Therefore, we set the value of the similarity threshold to 0.5 as the average of similar thresholds is in such a range for the following case studies.

The cardinality threshold $\tau$ determines classes associated with interesting XCSDs. $\tau$ affects the results of SCAD due to changes in the number of classes which need to be checked. If the value of $\tau$ is too large, then only a small number of equivalent classes is satisfied, which might result in a loss of interesting XCSDs. Therefore, in our case studies, we fix the value of $\tau$ at 2, which means we only consider classes having cardinality equal or greater than 2. We do not consider constraints

holding for only one group of similar object, as such constraints which are considered trivial.

**Case 4.1**. *XML Conditional Structural Dependencies contain only constants.*

We first construct the data summary for the Booking data tree in Fig1 by following the algorithms in List 4.3. A part of the summarized data is as follows:

$LV$[Type]$|_{Booking}$={(3, 2) Airline, (13, 2) Airline, (23, 2) Airline, (33, 2) Coach}

$LV$[Carrier]$|_{Booking}$= {(4,2) Qantas, (14,2) Qantas, (24,2)Tiger Airways, ""}

$LV$[Departure]$|_{Booking}$= {(5, 2)MEL, (16, 3)MEL, (26,3)MEL, (35,3) 6:00am}

$LV$[Arrival]$|_{Booking}$= {(6,2) SYD, (17,3) SYD, (27,3) SYD, (36,3) 6:00pm}

$LV$[Tax]$|_{Booking}$={(8,2) 40, (19, 2) 40, (29, 2) 50, (38, 2) 20}

Then, the search lattice is generated. Assume that we need to find the XCSDs associated with edge($W$, $Z$)= edge(Type-Carrier-Departure-Arrival, Type-Carrier-Departure-Arrival-Tax) with respect to the sub-tree rooted at Booking.

Partitions of Type-Carrier-Departure-Arrival and Type-Carrier-Departure-Arrival-Tax are generated as:

- Partitioning data into classes based on the data value

$\Pi_{Type|Booking}$= {{(3,2),(13,2),(23,2)}Airline, {33,2}Coach}

$\Pi_{Carrier|Booking}$= {{(4,2), (14,2)}Qantas, {(24, 2)}Tiger Airways, {""}}

$\Pi_{Departure|Booking}$= {{(5,2),(16,3),(26,3)}MEL,{35,3}6:00am}

$\Pi_{\text{Arrival}|\text{Booking}}$={{(6,2),(17,3),(27,3)}SYD,{(36,3)}6:00pm}

$\Pi_{\text{Tax}|\text{Booking}}$= {{(8,2), (19,2)}40, {(29,2)}50, {(38,2)}20}

- Converting these classes into the sub-tree rooted at Booking to find their refinements

$\Pi'_{\text{Type}|\text{Booking}}$= {{(2, 1), (12, 1), (22, 1)}, {32, 1}}

$\Pi'_{\text{Carrier}|\text{Booking}}$= {{(2, 1), (12, 1)}, {(22, 1)}, {""}}

$\Pi'_{\text{Departure}|\text{Booking}}$= {{(2, 1), (12, 1)}, {22, 1}, {32, 1}}

$\Pi'_{\text{Arrival}|\text{Booking}}$= {{(2, 1), (12, 1)}, {22, 1}, {32, 1}}

$\Pi'_{\text{Tax}|\text{Booking}}$= {{(2, 1), (12, 1)}, {22, 1}, {32, 1}}

- Calculating partitions of Type-Carrier-Departure-Arrival and Type-Carrier-Departure-Arrival-Tax. Assume that $\tau = 2$ then classes with cardinality less than 2 are discarded in our calculations.

$\Pi_{\text{Type,Carrier,Departure,Arrival}|\text{Booking}}$

$= \Pi'_{\text{Type}|\text{Booking}} \cap \Pi'_{\text{Carrier}|\text{Booking}} \cap \Pi'_{\text{Departure}|\text{Booking}} \cap \Pi'_{\text{Arrival}|\text{Booking}}$

$= \{(2,1), (12, 1)\} = \{w_1\}$

$\Pi_{\text{Type,Carrier,Departure,Arrival,Tax}|\text{Booking}}$

$= \Pi'_{\text{Type}|\text{Booking}} \cap \Pi'_{\text{Carrier}|\text{Booking}} \cap \Pi'_{\text{Departure}|\text{Booking}} \cap \Pi'_{\text{Arrival}|\text{Booking}} \cap \Pi'_{\text{Tax}|\text{Booking}}$

$= \{(2,1), (12, 1)\} = \{z_1\}$

We can see that $w_1$ is equivalent to $z_1$ that is $w_1 = z_1 = \{(2,1), (12, 1)\}$. Nodes in $w_1$ have the same value of Type= "Airline", Carrier= "Qantas", Departure= "MEL" and Arrival= "SYD". Nodes in $z_3$ share the same value of Tax= "40". An XCSD is discovered:

$\phi_1 = P_{\text{Booking}}$:(0.5)(Type="Airline"^./Carrier="Qantas"^./Departure="MEL"^./Arrival= "SYD" → ./Tax = "40").

This case shows that the discovered XCSD contains only constants. The discovered XCSD refines an XFD by binding particular values to elements in the XFD specification. For instance,

$\phi_1$ is a refinement of the XFD

$\varphi_1 = P_{\text{Booking}}$: ./Type, ./Carrier, ./Departure, ./Arrival → ./Tax

There also exists another XCSD refining $\varphi_1$

$\phi'_1 = P_{\text{Booking}}$:(0.5)(./Type="Airline"^./Carrier="Qantas"^./Departure="MEL"^./Arrival= "BNE" → ./Tax = "65")

There might exist a number of XCSDs which refine an XFD. As a result, the number of XCSDs discovered by SCAD is much greater than the number of data rules detected by an XFD discovery approach [102].

**Case 4.2.** *XCSDs contain both variables and constants.*

Fig 4.7 is a representation of a part of the Booking data tree. We use the

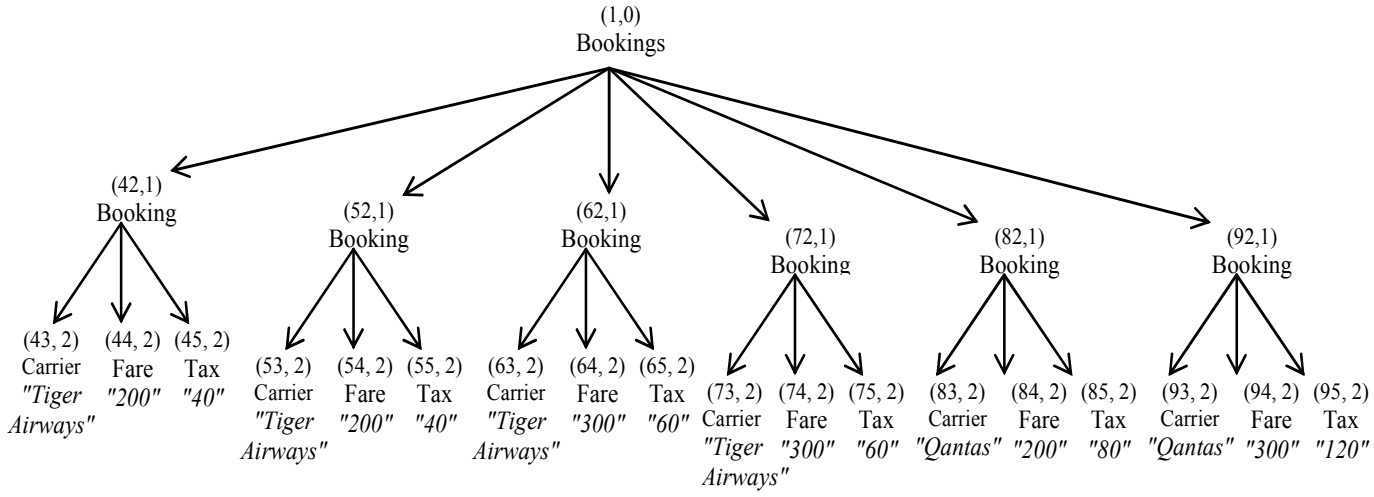


**Fig 4.7**. A simplified Bookings data tree is constrained by constraints containing both variables and constants

same assumptions and follow the same process in Case 4.1 to construct the data summary and the search lattice. Assume that we need to find XCSDs associated with the edge($W$, $Z$)= edge(Fare, Fare-Tax).

- Two partitions of Fare and Fare-Tax are as follows:

  $\Pi_{\text{Fair|Booking}}$ ={{(42,1), (52, 1),(82, 1)}, {(62,1), (72,1), (92, 1)}}

  $\Pi_{\text{Fair,Tax|Booking}}$={{(42,1), (52, 1)},{(62, 1), (72,1)}, {(82,1)}, {(92,1)}}

There does not exist any equivalent pair between two partitions $\Pi_{\text{Fair|Booking}}$ and $\Pi_{\text{Fair,Tax|Booking}}$. In such a case, node-labels from the remaining set of $\{LV[]\}\backslash\{W \cup Z\}$ are added to edge(Fare, Fare-Tax) as conditional data nodes. For example, the node-label of Carrier is added to the edge(Fare, Fare-Tax). We now consider edge($W'$, $Z'$)= edge(Fare-Carrier,Fare-Tax-Carrier).

- Partitions of Fare-Carrier and Fare-Tax-Carrier with respect to the sub-tree rooted at Booking are calculated as:

  $\Pi_{\text{Fair, Carrier|Booking}}$= {{(42, 1), (52, 1)}, {(62, 1), (72, 1)}, {(82, 1)}, {(92, 1)}} = $\{w_1, w_2, w_3, w_4\}$

  $\Pi_{\text{Fair, Tax, Carrier|Booking}}$= {{(42, 1), (52, 1)}, {(62, 1), (72, 1)}, {(82, 1)}, {(92, 1)}} = $\{z_1, z_2, z_3, z_4\}$

- The partition of the condition node Carrier is:

  $\Pi_{\text{Carrier|Booking}}$= {{(42, 1), (52, 1), (62, 1), (72, 1)}, {(82, 1), (92, 1)}} = $\{c_1, c_2\}$

- We have two equivalent pairs $(w_1, z_1)$ and $(w_2, z_2)$ between $\Pi_{\text{Booking, Fair, Carrier|Booking}}$ & $\Pi_{\text{Booking, Fair, Tax, Carrier|Booking}}$ with $|w_1|=2$ and $|w_2|=2 >= \tau$. Furthermore, there exists a class $c_1$ in $\Pi_{\text{Carrier|Booking}}$ containing exactly all elements in $w_1 \cup w_2$:

  $w_1 \cup w_2$ ={(42, 1), (52, 1), (62, 1), (72, 1)}= $c_1$

All elements in class $c_1$ have the same value of Carrier = "Tiger Airways". This means nodes in classes $w_1$ and $w_2$ share the same condition (Carrier = "Tiger Airways"). Therefore, an XCSD $\phi_2$ = $P_{\text{Booking}}$: (0.5) (./Carrier= "Tiger Airways"), (./Fare →./Tax) is discovered.

Case 4.2 illustrates that our proposed approach is able to discover XCSDs which contain both variables and constants. $\phi_2$ cannot be expressed by the existing notion of XFDs. For instance, XFDs [102] only express $\phi_2$ in the form, $P_{\text{Booking}}$: ./Fare →./Tax, which states that the value of an object (./Tax) is determined by the other object (./Fare) for all data. It cannot capture the condition (./Carrier= "Tiger Airways") and the similarity threshold (0.5) to express the exact defined semantics of $\phi_2$.

From both case studies, we can see that our approach is able to discover more situations of dependencies than the XFD discovery approach. There exists a number of XCSDs refining the XFD. Each XCSD refines an XFD by binding particular values to elements in the XFD specification. The existing XFD approach [102] cannot detect the above situations of dependencies due to the existence of conditions in constraints. XFDs only express special cases of XCSDs which have conditions being Null. The results from the tested cases somehow show the real potential of the approach. Hence, we believe that our approach can be generalized to other similar problems where data contain inconsistent representations of the same object and/or inconsistencies in constraining data in different fragments. For example, our approach can discover constraints in the context of data integration where data is combined from heterogeneous sources or in the situation of using XML-

based standards, such as OASIS, xCBL and xBRL to exchange business information.

## 4.9 Summary

In this chapter, we highlighted the need for a new data type constraint called XML conditional structural functional dependency to resolve the XML data inconsistency problem. Existing work has shown some limitations in handling such a problem. We proposed the SCAD approach to discover a proper set of possible XCSDs considered anomalies from a given XML data instance. We evaluated the complexity of our approach in the worst case and in practice. The results obtained from experiments and case studies revealed that SCAD is able to discover more situations of dependencies than XFD discovery approaches. Discovered constraints, which are XCSDs, containing either constants only or both variables and constants, which cannot be formally expressed by XFDs, have more semantic expressive power than existing XFDs. The discovered XCSDs using SCAD may be employed in data-cleaning approaches to detect and correct non-compliant data through which the consistency in data is improved. In the next chapter, we will utilize XCSDs to compute consistent query answers for queries posted to an inconsistent data source to improve information quality.

# Chapter 5

# Structured content-based query answers for improving information quality

This chapter introduces an approach, called SC2QA, which utilizes XML conditional structural functional dependency to compute answers for queries posted to arbitrary XML data to improve information quality. SC2QA integrates the semantics of XCSDs into the query process to handle data inconsistency and find the consistent parts for query answering. This chapter is organized into six sections. Section 5.1 presents an introduction to the problem, including our motivation and the synopsis of our approach. Section 5.2 presents the preliminaries. Section 5.3 describes our proposed SC2QA to compute the query answer. The complexity analysis and correctness of SC2QA are presented in Section 5.4. Section 5.5 demonstrates the experiment evaluations. Section 5.6 summarises the chapter.

## 5.1 Introduction

The Extensive markup language (XML) [88] has been widely adopted as a standard to exchange and integrate data over multiple sources. This allows users to explore large datasets through a declarative query interface, such as XQuery [26] and XPath[89]. However, the results of queries posted to such heterogeneous data sources are often inconsistent due to the anomalies arising from structural and semantic inconsistencies. This significantly affects the ability of the system to provide accurate query answers. The presence of inconsistent data is commonly resolved by repairing data and computing consistent query answers.

Data repair aims is to find consistent parts in an inconsistent data source which minimally differs from the original one [9, 25, 47, 79]. Data repair is then used to calculate consistent query answers. A consistent query answer (CQA) is defined as the common part of answers to the query on all possible repairs of the source [9]. Nevertheless, repairing data might also result in side-effects, for example, it might cause incorrect answers to queries and introduce new inconsistencies. Moreover, finding all possible repairs for inconsistent data to compute a consistent query answer is impossible and impractical since an infinite number of repairs might exist. Hence, we may leave the data inconsistent to avoid losing information due to data repair and only manage potential inconsistencies to compute consistent answers for queries posted to that source.

As XML data is often inconsistent with respect to a set of constraints, constraints are often taken into account during the process of calculating query answers [43, 45, 76]. The work in [74] studies the problem of computing query answers from inconsistent data with respect to a given DTD. Other work [45, 77, 78] focuses on finding CQAs from inconsistent data with respect to a set of functional dependencies. However,

such existing work generally lacks the full extensibility of cases where data inconsistency with respect to constraints holding conditionally in XML data with diverse structures, as in XCSDs.

In this chapter, we propose an approach called SC2QA to compute answers to queries posted to arbitrary XML data with respect to a set of XCSDs. XCSDs are not constraints on database states; they are constraints used to compute answers to queries which are specified locally with the query at hand. That is, SC2AD is flexible for users to specify a set of XCSDs at the query time. The semantics of XCSDs are integrated into the query planner to compute the query answer. The *conditions* in XCSDs are used to specify candidate objects qualified to the query. The *similarity threshold* in XCSDs is used to indicate how similar objects can be considered to be qualified for queries, which allows the retrieval of information from objects with diverse structures. The inconsistencies of the involved objects are repaired locally, following the semantics of each

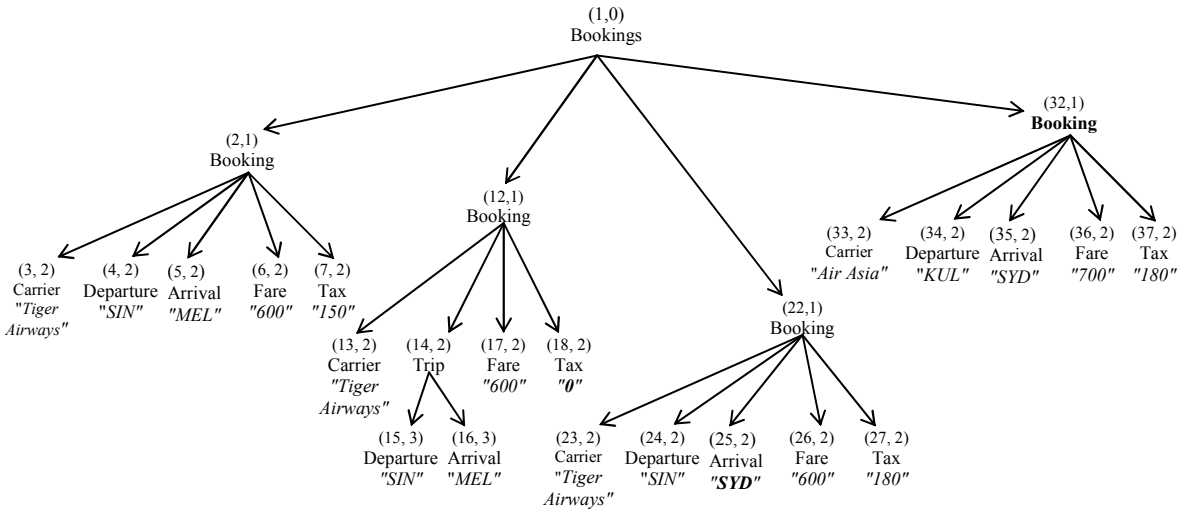


**Fig 5.1.** An inconsistent Flight Booking data tree with respect to XCSDs

related XCSD, to obtain the information which is as consistent as possible. A query answer, called customized consistent query answer (CCQA), is calculated from these data which are considered to be consistent with respect to certain preferred XCSDs. We run experiments on synthetic data to verify the effectiveness and efficiency of SC2QA. We prove that the algorithm is correct in the sense that the result retrieves consistent information.

## 5.2 Preliminary

In this section, we present some preliminary concepts, including XPath notations and examples of querying an inconsistent data source with respect to a set of XCSDs as further motivation to solve our problem.

### 5.2.1 XPath

We use XPath expression [89] to form a relative path; "." (self): select the context node. ".//": select the descendants of the context node, "[]": means qualifier and "*": means wildcards. For example, .//Carrier: select Carrier descendants of the context node Booking; .//Trip/Departure: select all Departure elements which are children of Trip. An XPath is simple if it is free of ".//" and "*". Any XPath $Q$ can be replaced by a set of simple XPath $\{Q_1, Q_2,...,Q_n\}$.

### 5.2.2 Motivation examples

Let us use examples to illustrate the influence of inconsistency caused by structural and semantic inconsistency in XML data to the answers of queries posted to that data. Our discussions are based on a simplified Flight Bookings data tree $T$, as shown in Fig 5.1. Each Booking

contains information on Carrier, Departure, Arrival, Transit, Fare and Tax. The values of elements are recorded under the element names. The data tree *T* is inconsistent with respect to three XCSDs, as shown in Fig 5.2. We suppose that all XCSDs have a similarity threshold of 1, which means that representations of the same object must have the same structure. According to constraint 1, any Booking having the Carrier "Tiger Airways" has the Tax identified by the Fare; however, Booking(2, 1) and Booking(12, 1) contain the same Fare of "600" but the values of Tax are different, which are 150 and 0, respectively. These are inconsistent with respect to constraint 1. Booking(2, 1) and Booking(12, 1) are also inconsistent in their structures. While Departure(4, 2) and Arrival(5, 2) are direct children of Booking(2, 1), the Departure(15, 3) and Arrival(16, 3) are the grandchildren of Booking(12, 1). Considering an XPath query *Q*: /Bookings/Booking posted to *T,* for a consistent query answer as in definition [9], the information of Departure, Arrival and Tax of Booking (2, 1) and Booking (12, 1) are excluded.

Constraint 1: *Any Booking having Carrier of "Tiger Airways", the Tax is identified by the Fare.*

$\phi_1 = P_{Booking}$: [1] [./Carrier ="Tiger Airways" ], (./Fare →./Tax)

Constraint 2: *Any Booking having Carrier of "Tiger Airways" and Departure of "SIN" only arrive at "MEL".*

$\phi_2 = P_{Booking}$: [1] [./Carrier ="Tiger Airways" ], (./Departure="SIN" →./Arrival= "MEL")

Constraint 3: *Any Booking having Carrier of "Air Asia", Departure of "KUL" and Arrival of "SYD", there must exist a Transit of "MEL".*

$\phi_3 = P_{Booking}$: [1] [./Carrier ="*Air Asia* " ], (./Departure="KUL",./Arrival= "SYD") → (./Transit= "MEL")

**Fig 5.2.** XCSDs on the Flight Bookings data tree

The same situation occurs to Booking(22, 1) with respect to constraint 2. That is, for any Booking having the Carrier "Tiger Airways" and Departure of "SIN" only arrives at "MEL"', however; Booking(22, 1) contains Departure of "SIN" but Arrival of "SYD". Considering the query *Q:* /Bookings/Booking posted to *T*, for a consistent query answer, Booking(22, 1) is excluded. For constraint 3, if there exists a Booking having the Carrier "Air Asia", Departure of "KUL" and Arrival of "SYD", then there must exist a Transit node with a value of "MEL". Booking(32, 1) is missing the information of Transit node which results in inaccurate answers to queries relating to Booking(32, 1). This chapter introduces the SC2QA approach which is based on the semantics of XCSDs to compute the query answers by a qualifying query with appropriate information derived from the interaction between the query and the XCSDs. The detail of SC2QA is presented in the next section.

## 5.3 SC2QA: structured content-aware approach for customized consistent query answers

In this section, we first present a theorem about the superiority of XCSDs to XCFDs and XFDs. This is necessary to indicate that our approach only needs to take into account the consistency of data with respect to XCSDs. Then, we present the concepts used in our approach, including a definition of consistent data, a definition of data repair, a notation of node repair and definition of customized consistent query answer. We also mention repairing principles applied to repair inconsistent data which includes repair cost and data repair values. Finally, we present the detail of SC2QA.

**Theorem 5.1.** An XCSD is superior to XML functional dependency (XFD) and XML conditional functional dependency (XCFD).

*Proof:* For an XCSD $\phi = P_l$: [α] [$\mathcal{C}$], ($X \rightarrow Y$), suppose that there exist $\{X_1, X_2,...X_n\}$ similar to $X$, and $\{Y_1, Y_2,...Y_m\}$ is similar to $Y$ with respect to the value of the similarity threshold α. Then $\phi$ can be expressed as a set of XCFDs $P_l$: [$\mathcal{C}$], ($X_i \rightarrow Y_j$), where $i$= 1..$n$ and $j$= 1..$m$. An XCFD can also be expressed as an XCSD with a similarity threshold of 1. Similarly, $\phi$ can be expressed as a set of XFDs in the form of $P_l$: ($X_g \rightarrow Y_h$), where $g$= 1..$n$ and $h$= 1..$m$ and an XFD $P_l$: $X \rightarrow Y$ is a special case of XCSD with a similarity threshold of 1 and the condition $\mathcal{C}$ is empty. □

XCSDs can be used to express the semantics of either XCFDs or XFDs. Therefore, in this work, we only focus on calculating customized consistent query answers for queries posted to an inconsistent XML data with respect to a set of XCSDs. Consistent data is defined as follows:

**Definition 5.1.** (Consistent data)

Given a data tree $T$= (V, E, F, root) and a set of XCSDs Σ, $T$ is consistent with Σ denoted as $T \models \Sigma$, if $T$ satisfies every predefined XCSD in Σ, otherwise $T$ is inconsistent denoted as $T \not\models \Sigma$.

A data tree $T$ is said to satisfy an XCSD $\phi = P_l$: [α] [$\mathcal{C}$], ($X \rightarrow Y$) denoted as $T \models \phi$ if any two sub-trees $R$ and $R'$ rooted at $v_i$ and $v_j$ in $T$ having $d_t(R, R') \geq \alpha$ and if $\{v_i[X]\} =_v \{v_j[X]\}$ then $\{v_i[Y]\} =_v \{v_j[Y]\}$ under the condition $\mathcal{C}$, where $v_i$ and $v_j$ have the same root node-label $l$.

The concept of data repair is used as an auxiliary to describe the definition of a customized consistent query answer in an inconsistent data.

Thus, we define the notion of data repair and the detail of data value repair principles and the repair cost model in the next section.

### 5.3.1 Data Repair

Repair $R$ is found based on the semantics of candidate XCSDs in $\sum$ which relates to query $Q$ to modify the inconsistent data in the original data tree $T$ such that $R$ is consistent with respect to $\sum$ and $R$ is *minimal* different with original data $T$. Minimal here means repair $R$ must be the one that takes as few repair operations as possible to preserve the information from the original data.

**Definition 5.2.** (Data repair)
A repair of $T$ with respect to $\sum$ is a data tree node $R$ such that: *(i)* $R$ conforms to $\sum$; and *(ii)* there does not exist any other repair $R'$ of $T$ such that $R'$ conforms to $\sum$, $cost(T, R') < cost(T, R)$, where $cost(T, R)$ is the repair cost used to transform $T$ to $R$.

In our work, repair $R$ is found by locally repairing every inconsistent data node. A node repair is defined as follows:

**Definition 5.3.** (Node repair)
A repair of a node $v$ with respect to $\phi_i$ is a node $v'$ such that: *(i)* $v'$ conforms to $\phi_i$; *(ii)* there does not exist any other repair $v''$ of $v$ such that $v''$ conforms to $\phi_i$ and *(iii)* $cost(v, v'') < cost(v, v')$, where $cost(v, v')$ is the repair cost used to transform $v$ to $v'$.

*Data value repair principles:* The computation of the repair data is based on the semantics of XCSDs to modify the values of nodes or add missing

data. We do not invent new values as in [43, 45]. Suppose that a node $v$ in $T$ violates an XCSD $\phi = P_l$: [α] [$\mathcal{C}$], ($X \rightarrow Y$), the value of $v$ is repaired based on a set of values occurring in the data tree $T$ or it is a value deduced based on the semantics of XCSDs. The details of data repair computation are as follows:

*i*) Values of node modification:

- If $v$ relates to a constant expression in $\phi$, the value of $v$ is updated by the value of the corresponding constant such that $v$ satisfies $\phi$.
- If the violation of node $v$ relates to a variable expression in $X \cup Y$, this means $v$ violates $\phi$ with another node $v'$, then the value of the violation node is modified with respect to the value $v'$. That is, (*i)* if value of $v$ is null and $v'$ is constant, the value of the $v'$ is set to that constant, *(ii)* if $v$ and $v'$ are not null and they contain different values, such violations have to be resolved by repairing other nodes relating to another expressions in $\phi$.

*ii)* Node insertions: suppose that a sub-tree $T_i$ is inconsistent with respect to an XCSD $\phi = P_l$: [α] [$\mathcal{C}$], ($X \rightarrow Y$) due to missing a node *v, v* is inserted into that sub-tree based on the semantics of XCSD such that $T$ is consistent with respect to the considered XCSD.

**Example 5.1.**

- Booking(22, 1) (in Fig 5.1) violates constraint $\phi_2 = P_{Booking}$: [1] [./Carrier ="Tiger Airways"], (./Departure="SIN" →./Arrival= "MEL") (in Fig 5.2). This is because Booking(22, 1) contains Carrier of "Tiger Airways", Departure of "SIN" but Arrival of "SYD". The Arrival = "SYD" causes violation to the right hand

side of $\phi_2$ which is a constant expression. Therefore, the value of Arrival should be changed to "SYD" as that in $\phi_2$.

- Considering Booking(12,1) and Booking (2,1) in Fig 5.1, and $\phi_1 = P_{Booking}$: [1] [./Carrier ="Tiger Airways" ], (./Fare →./Tax) in Fig 5.2. Booking(12, 1) violates constraint 1 with Booking(2, 1). According to constraint 1: any Booking having the Carrier "Tiger Airways" has Tax identified by the Fare. While both Booking(12, 1) and Booking (2, 1) have the same Carrier of "Tiger Airways" and the Fare of "600", the Tax "0" and "150". Therefore, we modify the value of Tax in Booking(12, 1) to "150".
- Booking(32, 1) in Fig 5.1 violates constraint 3 in Fig 5.2. According to constraint 3: for any Booking the Carrier "Air Asia", Departure of "KUL" and Arrival of "SYD", there must exist a Transit of "MEL". Booking(32, 1) does not satisfy constraint 3 since the information of Transit is missing. Thus, Transit of "MEL" is added into Booking (32, 1).

*Repair Cost:* there are several different ways to resolve a violation. In our approach, we use a cost model to give priority to the repair which is considered to be qualified to a query. The result with a lower transformation cost is considered to be closer to the original data and is preferred over the ones with higher costs. The cost used to repair an inconsistent node is weighted by the total number of operations applied to correct the node so that it satisfies all relevant XCSDs. The cost used to repair an inconsistent data tree is weighted by the total cost applied to all violation nodes such that the data tree satisfies all relevant XCSDs.

Observe that the deletions which may lose information, node modification and node insertion in general can preserve more information

from the original data. Thus, our approach only considers modification and insertion operations. We assume that each operation is assigned a weight in a range between 0 and 1. A cost of the modification operation is the cost of updating a node value. The cost of the insertion operation is the cost to insert a single node which is a descendant of a considered sub-tree. We prefer node updating than node insertion since inserting a node is more complicated than updating a node value. In our approach, we assume that the cost of node insertion is three times the modification cost. The repair cost of a $T$ to a repair $R$ is defined as

$$\cos t(T,R) = \sum_{i=[1..n]} \cos t(v_i, v_i'),$$

where $v_i$ is original node and $v_i'$ is the transformed node.

In addition to choosing a repair with the lowest cost, we also use cost threshold, denoted as γ, to ignore the repairs which are too different to the original. Users choose their preferred repair cost threshold when specifying queries. A cost threshold for each considered node is indicated by the percentage of the repair cost threshold with the number of candidate nodes and the number of XCDSs. Such an evaluation mechanism allows retrieving the desired information which is closer to the original data source.

$$\gamma_{node} = \frac{\gamma}{C_v . C_\phi},$$

where γ is the repair cost threshold of the total data, $C_v$ is the number of candidate nodes relating to the query and $C_\phi$ is the number of candidate XCSDs. In this chapter, we set the value of γ to 1.

**Example 5.2.** Suppose that Booking(42, 1) in Fig 5.3a which is also included in the data tree *T* in Fig 5.2. Booking(42, 1) is inconsistent with respect to constraints 1, 2 & 3 in Fig 5.2. At least two alternative ways exist to correct Booking(42, 1) with different results (Fig 5.3b & 5.3c). Assume that the total number of candidate nodes is 4, the modification cost is 0.01 and the insertion cost is 0.03. The cost repair threshold for each node is $\gamma_{node} = 1/(4 * 3) \approx 0.08$.

The first repair $v_1$ is followed by constraints 1&2. According to constraint 1 'Any Booking having the Carrier "Tiger Airways", the Tax is identified by the Fare'. The Tax and the Fare of Booking(42, 1) are the same with the Tax and the Fare of Booking(2, 1). Thus, the value of the Carrier of Booking(42, 1) is updated by "Tiger Airways" with repair cost of 0.01. According to constraint 2 'Any Booking having Carrier of "Tiger

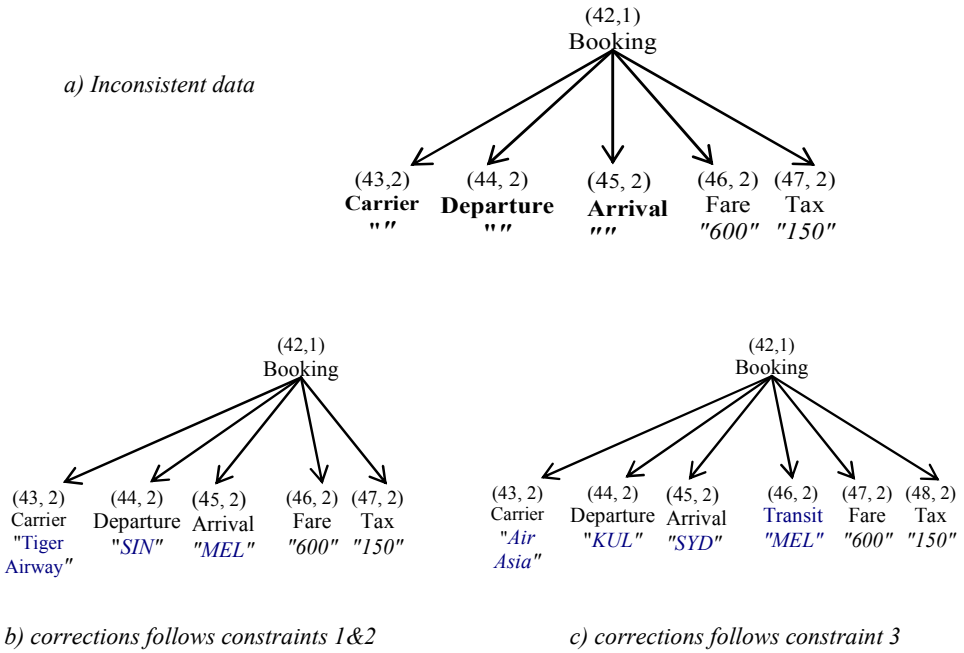


**Fig 5.3.** Repairing consistent data

Airways" and Departure of "SIN", the value of Arrival is "MEL" '. The Carrier of Booking(42, 1) is "Tiger Airways". Hence, the value of Departure of Booking(42, 1) is updated by "SIN" with repair cost of 0.01 and Arrival is modified by "MEL" with repair cost of 0.01.The total cost repair: $cost$(Booking(42, 1), $v_1$)= (0.01 + 0.01 + 0.01)= 0.03 < 0.08 =$\gamma_{node}$.

The second repair $v_2$ is followed constraints 3. According to constraint 3: for 'Any Booking having Carrier of "Air Asia", Departure of "KUL" and Arrival of "SYD", there must exist a Transit of "MEL"'. Following constraint 3, Carrier of Booking(42, 1) is updated by "Air Asia" with a cost of 0.01, Departure is updated by "KUL" with repair cost of 0.01, Arrival is updated by "SYD" with repair cost of 0.01, and insert a node: Transit of "MEL" with a cost of 0.03. The total repair cost $cost$(Booking (42, 1), $v_2$ )= (0.01 + 0.01 + 0.01 + 0.03)= 0.06 < 0.08 =$\gamma_{node}$. The repair costs of the two cases are 0.03 and 0.06, respectively which satisfy the repair cost threshold for a node (i.e. 0.08). However, the former repair is preferred over the latter. Indeed, $v_1$ is closer to original Booking (42, 1) than $v_2$.

**Definition 5.4.** (Customized consistent query answer- CCQA)

Given a data tree $T$, a set of XCSDs $\sum$ over $T$ and a query $Q$ , the customized consistent query answer of the query $Q$ on $T$ with respect to $\sum$, denoted as $Q_c(T, \sum) = \cup_{i=1..k} Q_c(R_{vi}, \sum)$, where $Q_c(R_{vi}, \sum)$ is a consistent query answer of $Q$ on the sub-tree $R_{vi}$, $R_{vi}$ is a repair of sub-tree rooted at $v_i$ w.r.t $\sum$, $k$ is the number of related nodes to $Q$ and $v_i$ is a related nodes to the query $Q$.

A formal approach to calculate the customized consistent query answer will be presented in the next section.

### 5.3.2 Calculating customized consistent query answers

Given an inconsistent XML data tree $T$ and a set of XCSDs $\Sigma$, we aim to compute a customized consistent answer for query $Q$ posted to $T$. Our approach is based on the semantics of XCSDs to find consistent data, preserving the original data source. The original inconsistent data is evaluated at each constraint. The answer is calculated by qualifying query with appropriate information derived from the interaction between the query and the XCSDs. The result of a query is a data tree which is constructed by appropriate projections on the data qualified to $Q$. The similarity parameters in XCSDs may result in a large number of candidate objects qualified to the query, causing difficulties for computing CCQAs. Therefore, we restrict all XCSDs having the same similarity threshold to avoid dealing with a various number of candidate objects.

In particular, SC2QA consists of four processes (List 5.1). First, the function selDC is performed to select all candidates XCSDs $(\phi_1,...,\phi_k)$ relating to the posted query $Q$. This process is based on the comparison between the context paths of XCSDs and paths in $Q$ with respect to the similarity threshold of XCSDs. Second, the function selCanNode is called to select all candidate nodes $(v_1, v_2,..., v_n)$ relating to query $Q$, based on the semantics of the candidate XCSDs, where $v_i$ is an ancestor of target nodes of XCSDs. Third, valXCSD is performed to validate XCSDs, $v_i$ is the domain for validating candidate XCSDs. The process of validating XCSDs is performed locally at each candidate node to retrieve consistent data. For each violation node $v_k$, the list of violation XCSDs, repair values and repair

costs are calculated and stored in $v_k$.list($\phi_{kj}$ , $r_{kj}$, $c_{kj}$), and node $v_k$ is marked as a violation. Finally, the result data is generated in a top-down fashion by combining all consistent data and repaired data. A result is considered valid only if the repair cost of inconsistent data is under a certain cost threshold. That is, if the repair cost of data is above a given cost threshold then the repair cost will be set to 100 to indicate that the result is invalid and the corresponding repair is not considered to be qualified for query $Q$ and query $Q$ is not actually executed. Thus, for each violation node $v_k$, we choose the repair ($\phi_{kj}$ , $r_{kj}$, $c_{kj}$) with the lowest overall repair cost. Commonly, we choose the node repair cost for $v_k$ with the lowest cost.

```
Algorithm SC2QA
Input: Q, Σ, T,  γ
Output: T′
T′← '';
DC← selDC( Q, T, Σ );
CN← selCanNode (Q, T, DC );
repCost← valXCSD( Q, T, DC, Σ, γ )
if repCost ≠ 100
for each candidate node v_i in CN do
        if making v_i then
                v_i′ ← repair(v_i.list( φ_k , r_ik, c_ik));
                T′ ← T′ ∪ v′_i
        else
        T′ ← T′ ∪ v_i;
return T′;
```

**List 5.1.** The SC2QAs algorithm

```
Algorithm selDC //selecting XCSDs relating to Q
Input: Q, T, Σ;
Output: DC  // selected constraints
init  DC ← {Ø};
        for each  ϕ = P_l: [α] [ϱ], (X → Y) in Σ  do //select relevant XCSDs to Q
                candidate←{True};
                for each simple path q_i in Q do
                        if  d_p(q_i,P_l) < α  then  //not similar
                                candidate← {False};
                                exitFor;
                DC← Insert(DC, ϕ ); //insert ϕ into DC in the increasing order
return(DC);

Algorithm selCanNode //selecting candidate target nodes
 Input: Q, T, DC
 Output: CN //set of candidate target nodes
 init CN← {Ø};
  N← satisfiedNodes(Q, T); // N= {n_1, n_2,...,n_k}
  for each XCSD  ϕ = P_l: [α] [ϱ], (X → Y) in DC do
            for each subtree T_i rooted at n_i  do
                if  lab(n_i)= l  and exiting node n_k in T_i  satisfying ϱ  then
                        CN← CN ∪ {n_i};
return(CN);


Algorithm valXCSD
Input: Q, Σ, T,  γ
Output: repCost
DC← selDC( Q, T, Σ );
CN← selCanNode (Q, T, DC );
γ_node ← γ /(CN. |Σ|);
repCost← 0; //repair cost
for each candidate node v_i in CN do
        for each candidate XCSD  ϕ_k  in DC do
                if  ϕ_k  does not hold on sub-tree t_vi rooted at v_i then
                        marking v_i;
                        c← esreco(t_vi,  ϕ_k )
                        if (c < γ_node ) then  //estimating repair cost
                                v_i.list( ϕ_k , r_ik, c);
                                add(repCost, c);
                          else   remove v_i from CN;
If (repCost < γ )  return repCost
else
return  100;
```

**List 5.2**. Utility functions of SC2QA

**Example 5.3.** Finding customized consistent answer for a query *Q*: /Bookings/Booking[Carrier= 'Tiger Airways"], posted to Bookings data tree in Fig 5.1 with respect to three constraints $\phi_4$, $\phi_5$ and $\phi_6$.

$\phi_4$ = $P_l$: [0.6] [./Carrier ="Tiger Airways"] (./Fare →./Tax)

$\phi_5$ = $P_l$: [0.6] [./Carrier ="Tiger Airways" ], (./Departure="SIN" →./Arrival= "MEL")

$\phi_6$ =$P_l$: [0.6] [./Carrier ="*Air Asia* " ], (./Departure="KUL",./Arrival= "SYD") → (./Transit= "MEL")

We follow the process described in section 5.3 to find a CCQA for *Q*. First, we select all candidate XCSDs relating to query *Q* which include $\phi_4$ and $\phi_5$. Second, we select all candidate nodes relating to query *Q* which include *CN*= {Booking(2, 1), Booking(12, 1), Booking(22, 1 )}. Third, we validate XCSDs: for each candidate node in *CN*, we check for the satisfaction of candidate XCSDs and find consistent data for that node. We find Booking(2, 1) is similar to Booking(12, 1). This is because following the sub-trees similarity algorithm described in List 4.1 to calculate the similarity between sub-trees $T_1$ rooted at Booking(12,1) and $T_2$ at Booking(2,1), we have $d_T(T_1, T_2)$= 0.64 > 0.6. Booking (2, 1) satisfies constraints $\phi_4$ and $\phi_5$.

Booking (12, 1) violates $\phi_4$ with Booking(2, 1) since the Tax of two nodes does not satisfy the condition that Tax is identified by the Fare. They contain the same Fare of "600" but the values of Tax are different. While the Tax of Booking(2, 1) is a constant of "150", the Tax of Booking(12, 1) is constant of "0". Thus, we replace the inconsistency of the Tax by updating the Tax in Booking(12, 1) with a value of "150" in

the answer. The Tax(18, 2) node is marked. Booking(22, 1) violates $\phi_4$ and $\phi_5$.

According to $\phi_4$, the Fare identifies the Tax, the Fare of Booking(22, 1) is the same at that of Booking(2, 1) but the Tax is different. Thus, the Tax of Booking(22, 1) is corrected based on the semantics of $\phi_4$. That is, Tax is "150" and Tax(28, 2) is marked. According to $\phi_5$, if Departure of "SIN", then Arrival must be "MEL" but the Arrival of Booking (22, 1) is "SYD" which is replaced by "MEL" in the answer based on $\phi_5$. The result of query $Q$ will include Bookings {Booking (2, 1), Booking (12, 1), Booking (22, 1)} with modified values at marked nodes.

## 5.4 Complexity analysis and correctness

*Complexity analysis*: complexity of the SC2QA algorithm mostly depends on the size of the XML data source, which is determined by the number of elements, the number of XCSDs and the complexity of query $Q$ on $T$. The SC2QA algorithm first performs the selDC to find a set of candidate XCSDs related to the query $Q$. The complexity of selDC depends on the complexity of query $Q$ on $T$ and the size of context paths of XCSDs. In the worst case, we assume that all $n$ nodes in $T$ are satisfied by $Q$. Let $|\Sigma|$ be the total number of XCSDs and $m$ be the maximum size of the context paths of XCSDs $\phi$. Thus, the selDC makes $nm|\Sigma|$ random accesses to the dataset. Then, the function selCanNode is called to select candidate nodes related to the query $Q$ with respect to the conditions of the XCSDs. The selCanNode

depends on the number of XCSDs and the size of the XML data source. The worst case occurs when all $n$ nodes in $T$ related to $Q$ and every XCSD in $\Sigma$ having a condition, without considering the number of expressions in the conditions, the function selCanNode makes $n|\Sigma|$ random accesses to the dataset.

Third, the valXCSD is called to validate candidate XCSDs against every candidate node. In the worse case, we assume that all $|\Sigma|$ XCSDs are in the set of candidate XCSDs, where the set of candidate nodes includes all $n$ nodes in $T$ and each node violates all $|\Sigma|$ candidate XCSDs. The valXCSD makes $n|\Sigma|$ random access to the dataset. Finally, the SC2QA traverses the data tree $T$ on a top-down manner to obtain the query result. Every node in $T$ is visited once which means this step needs $n$ random accesses to the dataset. In summary, SC2QA algorithm has time complexity of $O(n(m|\Sigma| + 2|\Sigma| + 1))$. SC2QA needs to maintain a copy of data source $T$ at a time. Hence, the space complexity is bounded by $O(n)$. However, in practice, the number of related nodes can be significantly smaller than $n$ and the number of XCSDs relating to $Q$ is also smaller than the total number of XCSDs $|\Sigma|$. Therefore, the time complexity can be reduced significantly.

The following theorem states that the SC2QA algorithm must be terminated and returns customized consistent query answers.

**Theorem 5.2.** (Termination) Let $Q$ be a query on a data tree $T$ and $\Sigma$ be a set of XCSDs. The SC2QA always terminates and generates a consistent query answer $Q_c(T, \Sigma)$.

*Proof:* Although in each step of algorithm SC2QA, a violation node with respect to a candidate XCSDs is resolved, it might also introduce new violations. SC2QA proceeds until no more violation nodes exist. However, the set of candidate XCSDs and the number of candidate nodes are limited. Thus, SC2QA always terminates. □

**Theorem 5.3.** (Correctness) Let $Q$ be a query on a data tree $T$ and $\Sigma$ be a set of XCSDs. The query answer obtained by SC2QA is always customized consistent with the given XCSDs.

*Proof:* Suppose that $Qc(T, \Sigma)= \cup_{i=1..k} Qc(R_{vi}, \Sigma)$ is the answer of $Q$. This means each data node in the CCQA satisfies all XCSDs with the lowest repair cost. If there exists an answer $Q_c(R_{vi}, \Sigma)$ for sub-tree $R_{vi}$ rooted at $v_i$ which violates a constraint $\phi_m$, then there exists at least a node $v_j$ in the $R_{vi}$ violates $\phi_m$. In such a case, $Q_c(R_{vi}, \Sigma)$ is not included in the answer $Q_c(T, \Sigma)$ and a repairing with respect to the given XCSDs is impossible. Otherwise, it is a contradiction with the data value repair principles that each $Q_c(R_{vi}, \Sigma)$ is considered valid in the answer set only if it is computed from all consistent data with a repair cost under a certain cost threshold. □

## 5.5 Experimental evaluation

We run experiments on synthetic data to evaluate the calculation efficiency for SC2QA. This is to avoid the noise in real data. Our dataset is an extension of the Flight Bookings data shown in Fig 5.1. The dataset covers common features in XML data, including structural diversity and various data rules. The original dataset contained 100 Bookings. The DirtyXMLGenerator [72] made by Sven Puhlmann was used to generate the synthetic dataset. We specified that the percentage of duplicates of an object is 100% to generate a dataset containing similar Bookings. From 100 duplicate Bookings, we specified 30% of data was missing from the original objects so that the dataset becomes inconsistent due to missing data. We evaluated on 5 constraints, consisting of 3 constant XCSDs and 2 variable XCSDs. We ran experiments on a PC with an Intel i5, 3.2GHz CPU and 8GB RAM. The implementation was in Java and data was stored in MySQL.

*Parameters:* the number of XCSDs and the query influence on the complexity of SC2QA**.** The dataset is fixed, but the number of conditions on the query and the XCSDs change. We consider the effectiveness of cases where: *(i)* the query $Q$ is computed with respect to different types of XCSDs including constant XCSDs and variable XCSDs; and (*ii)* the number of conditions in query $Q$ increases, and the number of XCSDs is stable. Fig 5.4 is a set of XCSDs and Fig 5.5 is a set of queries which are used in experiments. The repair cost threshold $\gamma$ is set to 1. For each query, we recorded the running time.

| | |
|---|---|
| *C1* | $\phi_1 = P_l$: [0.6] [./Carrier ="Tiger Airways"] (./Fare →./Tax) |
| *C2* | $\phi_2 = P_l$: [0.6] [./Carrier ="Air Asia " ], (./Departure,./Arrival) → (./Tax) |
| *C3* | $\phi_3 = P_l$: [0.6] [./Carrier ="Tiger Airways" ], (./Departure="SIN" →./Arrival= "MEL") |
| *C4* | $\phi_4 = P_l$: [0.6] [./Carrier ="Air Asia " ], (./Departure="KUL",./Arrival= "SYD") → (./Transit= "MEL") |
| *C5* | $\phi_5 = P_l$: [0.6] [./Carrier ="Air Asia "], (./Departure= "SIN", ./Arrival="SYD") → (./Tax= "200") |

**Fig 5.4.** Set of XCSDs used in experiments

| | |
|---|---|
| *Q1* | /Bookings/Booking |
| *Q2* | /Bookings/Booking[Carrier= 'Tiger Airways'] |
| *Q3* | /Bookings/Booking[Carrier= 'Tiger Airways' and Departure='SIN' ] |
| *Q4* | /Bookings/Booking[(Carrier= 'Tiger Airway' and Departure= 'SIN' and Fare = '600'] |
| *Q5* | /Bookings/Booking[(Carrier= 'Air Asia' *or* Carrier= 'Tiger Airways'*)* and Departure='SIN'] |

**Fig 5.5.** Set of queries used in experiments

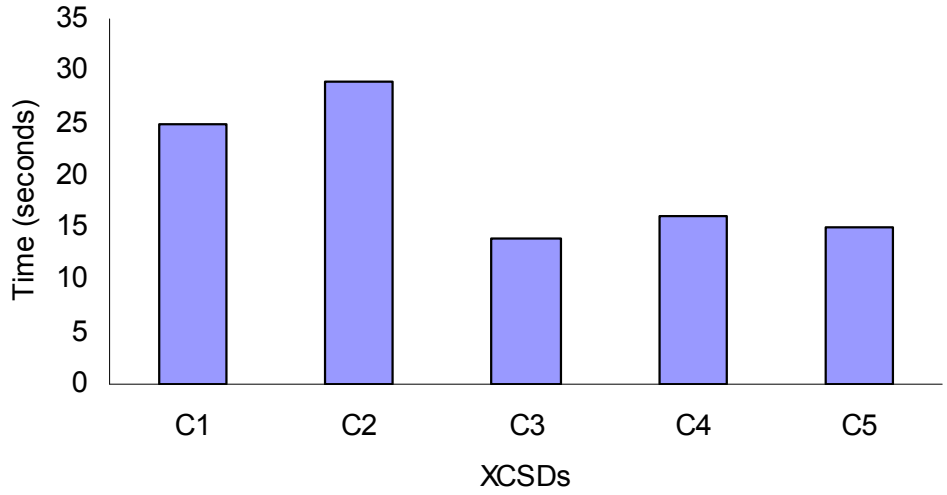


**Fig 5.6** Execution times: constant XCSDs vs variable XCSDs

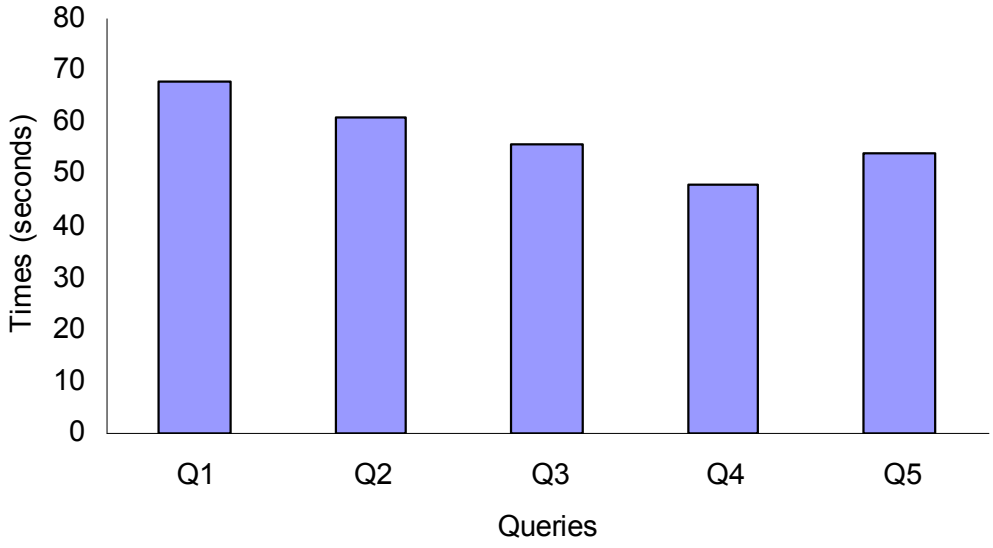


**Fig 5.7** Execution times when varying the number of conditions in queries

The results in Fig 5.6 are the execution times of query *Q1* under various types of XCSDs. The results show that the SC2QA runs more efficiently when utilizing constant XCSDs than variable XCSDs. This is because XCSDs with constant expressions provide more information for validating

and correcting violation data than those with variable expressions. The variable XCSDs hold on a large number of objects, which requires more processing time. For instance, the execution times of query $Q_1$ under $\phi_3$ are around 50% the execution times of either $\phi_1$ or $\phi_2$. The same situation occurs to $\phi_4$ and $\phi_5$.

Fig 5.7 represents the execution times of queries $Q_1$-$Q_5$ where have the number of conditions varies from 0 to 3 and the number of XCSDs remains the same (i.e $\phi_1$- $\phi_5$). Execution time depends on the number of conditions in the queries. In cases where the number of conditions in the queries increases, the execution time is also slightly increased. That is, the times required to analyse the interactions between the query and the XCSDs increase.

## 5.6 Conclusion

This chapter introduced an approach utilizing XCSDs to compute customized consistent query answers for queries posted to an inconsistent data source to improve information quality. Our approach is based on the semantics of XCSDs to find consistent data from involved objects. By identifying every inconsistent node locally with respect to each XCSD, SC2QA is able to collect the information as consistent as possible. Experiments on a synthetic dataset are used to evaluate the effectiveness of SC2QA. The results show that SC2QA works more efficiently for constant XCSDs than variable XCSDs. Constant XCSDs provided more information for validating and correcting violation data than those with variable expressions as XFDs. Thus, we expect that utilizing XCSDs to compute the customized consistent answers to queries are more accurate than that of XFDs.

# Chapter 6

# Conclusion

## 6.1 Thesis summary

This thesis addressed the problems of data inconsistency in XML data. The problem of XML data inconsistency often arises from either semantic or structural inconsistencies inherent from in heterogeneous XML data. Existing XFD approaches have shown several limitations in handling such problems. XFDs are unable to express the semantics of constraints holding conditionally on XML data with diverse structures. Existing XFD discovery approaches cannot explore a proper set of constraints to address inconsistency in XML data. Such limitations are resolved in this thesis.

Chapter 3 introduced the XDiscover approach to address semantic inconsistency. We first introduced the notion of XML conditional functional dependency. XCFDs are constraints which incorporate conditions into XFD specifications to express constraints with conditional semantics. Second, the XDiscover approach was proposed to discover a set of possible XCFDs from a given XML data instance. We conducted experiments on synthetic and real datasets, and examined on case studies to evaluate XDiscover. The obtained results revealed that XDiscover is able to

discover more situations of dependencies than the XFD discovery approach. Furthermore, XCFDs have more semantic expressive power than existing XFDs.

Chapter 4 proposed the SCAD approach to target the problems of data inconsistencies caused by both structural and semantic inconsistencies. First, we highlighted the need for a new data type constraint called XML conditional structural functional dependency (XCSD) to resolve such problems. Second, we proposed the SCAD approach to discover a proper set of possible XCSDs considered anomalies from a given XML data instance. Third, we evaluated the complexity of our approach in the worst case and in practice. Fourth, we ran experiments and case studies on synthetic datasets. The obtained results revealed that SCAD is able to discover more situations of dependencies than the XFD discovery approach. Discovered XCSDs using SCAD also have more semantic expressive power than existing XFDs. SCAD deals effectively with data sources containing structure diversity.

Both XCFDs and XCSDs can be used to enhance data quality management. They can be embedded as an integral part in an enterprise's systems to constrain the data process by suggesting possible rules and identifying non-compliant data to minimize data inconsistency. They also can be used to detect and correct non-compliant data. Chapter 5 utilized XCSDs to compute customized consistent answers for queries posted to an inconsistent data source to improve the quality of information. First, we proposed an approach called SC2QA, which integrated semantics of XCSDs into the query process to compute query answers. Second, we evaluated the complexity of SC2QA in worst case analysis. Third, to evaluate the effectiveness of SC2QA, we conducted experiments on a synthetic dataset which contained structural diversity and constraint variety causing XML data inconsistencies. The results showed that query answers

found by SC2QA work more efficiently for constant XCSDs than variable XCSDs. We proved that customized query answers computed by SC2QA are always consistent with respect to a set of preferred XCSDs.

## 6.2 Future work

There are several possible directions for future work which can use the techniques proposed in this thesis as a foundation. These promising directions are listed as follows:

- This thesis handles inconsistencies at either semantic or structural-level; other inconsistencies might still exist due to element labels. It would be interesting to take a step forward to resolve the problems of data inconsistencies caused by the inconsistencies in the semantics of labels.

- XML data changes very often which may lead to a corresponding change in the semantics of constraints. It is an interesting problem for future research to address the problem of data evolution by extending this work.

- Data inconsistencies also challenges in data integration environment. Inconsistency may arise due to the way in which source data are related with global elements by means of mapping. Data stored at the local source may violate integrity constraints specified at the global level. We would like to extend our discovery techniques to tackle inconsistencies in data integration.

- We would like to extent our SCAD discovery approach to support association rules holding conditionally on data. This extension is

particular interesting since it allows assigning context-dependent to association rules, where each context is represented by appropriate data fragments in which association rule holds.

- We also would like to extent our proposed approaches to support more types of constraints, such as foreign keys, reference integrity and general check constraints.

# Bibliography

[1]. Abiteboul, S., Buneman, P. and Suciu, D. (eds.). Data on the Web: From Relations to Semistructured Data and XML, 2000.

[2]. Abiteboul, S., Buneman, P. and Suciu, D., Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann, 2000.

[3]. Afrati, F.N. and Kolaitis, P.G., Repair checking in inconsistent databases: algorithms and complexity, ICDT '09 Proceedings of the 12th International Conference on Database Theory St. Petersburg, Russia, 2009, pp. 31-41.

[4]. Agrawal, R., Imielinski, T. and Swami, A., Mining association rules between sets of items in large databases, SIGMOD Record (1993), 22 (2), 207-216.

[5]. Ahmad, K., Mamat, A., Ibrahim, H. and Noah, S.A.M., Defining Funtional Dependency for XML, Journal of Information Systems, research & Practices (2008), 1 (1).

[6]. Arenas, M., Normalization Theory for XML, SIGMOD Record (2006), 35 (4), 57-64.

[7]. Arenas, M. and Bertossi, L., On the Decidability of Consistent Query Answering, In proc. Alberto Mendelzon Int. Workshopon Foundations of Data Management, 2010.

[8]. Arenas, M., Bertossi, L. and Chomicki, J., Answer Sets for Consistent Query Answering in Inconsistent Databases, Theory and Practice of Logic Programming (2003), 3 (4), 393-424.

[9]. Arenas, M., Bertossi, L. and Chomicki, J., Consistent query answers in inconsistent databases, PODS '99, Philadelphia, Pennsylvania, USA, 1999, ACM, pp. 68-79.

[10]. Arenas, M., Bertossi, L., Chomicki, J., He, X., Raghavan, V. and Spinrad, J., Scalar aggregation in inconsistent databases, Theoretical Computer Science (2003), 296 (3), 405–434.

[11]. Arenas, M. and Libkin, L., A normal form for XML documents, ACM Transactions on Database Systems (TODS) (2004), 29 (1), 195-232.

[12]. Armstrong, W.W., Nakamura, Y. and Rudnicki, P., Armstrong's Axioms, Journal of Formalized Mathematics (2003), 14.

[13]. Baralis, E., Cagliero, L., Cerquitelli, T. and Garza, P., Generalized association rule mining with constraints, Information Sciences (2012), 194 (1), 68-84.

[14]. Baralis, E., Garza, P., Quintarelli, E. and Tanca, L., Answering XML Queries by Means of Data Summaries, ACM Trans. Inf. Syst. (2007), 25 (3).

[15]. Batini, C. and Scannapieca, M., Data Quality- Concepts, Methodologies and Techniques, Springer Berlin Heidelberg

New York, 2006.

[16]. Bertossi, L., Consistent query answering in databases, SIGMOD Record (2006), 35 (2), 68-76.

[17]. Bertossi, L., Database Repairing and Consistent Query Answering, Synthesis Lectures on Data Management (2011), 3 (5), 1-121.

[18]. Bertossi, L., Database Repairing and Consistent Query Answering. in, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2011.

[19]. Bex, G.J., Neven, F. and Bussche, J.V.d., DTDs versus XML Schema: A Practical Study, Proceedings of the 7th International Workshop on the Web and Databases, Paris, 2004, ACM, pp. 79-84.

[20]. Bohannon, P., Fan, W., Geerts, F., Jia, X. and Kementsietsidis, A., Conditional Functional Dependencies for Data Cleaning, The 23rd International Conference on Database Engineering ICDE 2007, Istanbul, 2007, pp. 746-755.

[21]. Buneman, P., Davidson, S., Fan, W., Hara, C. and Tan, W.-C., Keys for XML, WWW '01, Hong Kong, 2001, ACM, pp. 201-210.

[22]. Buneman, P., Davidson, S., Fan, W., Hara, C. and Tan, W.-C., Reasoning about keys for XML, DBPL '01, 2002, Springer-Verlag, pp. 133--148.

[23]. Buneman, P., Fan, W. and Weinstein, S., Path Constraints in Semistructured Databases, Journal of Computer and System Sciences (2000), 61 (2), 146–193.

[24]. Buttler, D., A Short Survey of Document Structure Similarity Algorithms, Proceedings of the 5th International Conference on Internet Computing, USA, 2004, pp. 3-9.

[25]. Cate, B.T., Fontaine, G. and Kolaitis, P.G., On the data complexity of consistent query answering, Proceedings of the 15th International Conference on Database Theory, Berlin, Germany, 2012, ACM, pp. 22-33.

[26]. Chamberlin, D., XQuery: An XML query language, IBM Syst. J. (2002), 41 (4), 597-615.

[27]. Chiang, F. and J.Miller, R., Discovering Data Quality Rules, Proc. VLDB Endowment (2008), 1 (1), 1166-1177.

[28]. Chomicki, J., Consistent Query Answering: Five Easy Pieces 11th International Conference on Database theory, Springer LNCS, 2007, 1-17.

[29]. Chomicki, J., Marcinkowski, J. and Staworko, S., Computing consistent query answers using conflict hypergraphs, CIKM '04 Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management 2004, ACM Press, pp. 417-426.

[30]. Clark, J. and Makoto, M., RELAX NG Specification, 2001. http://relaxng.org/spec-20011203.html

[31]. Cong, G., Fan, W., Geerts, F., Jia, X. and Ma, S., Improving Data Quality: Consistency and Accuracy, VLDB'07, Vienna, Austria, 2007, VLDB Endowment, pp. 315-326.

[32]. Decker, H., Answers that have integrity, Semantics in Data and Knowledge Bases (2011), 6834, 54-72.

[33]. Deutsch, A., Popa, L. and Tannen, V., Query Reformulation with Constraints, SIGMOD Rec. (2006), 35 (1), 65-73.

[34]. Deutsch, A. and Tannen, V., Reformulation of XML Queries and Constraints, Proceedings of the 9th International Conference on Database Theory, 2002, Springer-Verlag, pp. 225-241.

[35]. El-ghfar, R.M.A., EL-Bastawissy, A. and Elazeem, M.A., DRTX: A Duplicate Resolution Tool for XML Repositories, IJCSNS (2012), 12 (7), 42-49.

[36]. Fan, W., Dependencies Revisited for Improving Data Quality, PODS'08, Vancouver, Canada, 2008, ACM pp. 159-170.

[37]. Fan, W., XML Constraints: Specifications, Analysis, and Application, Database and Expert Systems Applications, 2005, pp. 805- 809.

[38]. Fan, W., Geerts, F. and Jia, X., Semandaq: a data quality system based on conditional functional dependencies, Proc. VLDB Endowment (2008), 1 (2), 1460-1463.

[39]. Fan, W., Geerts, F., Lakshmanan, L.V.S. and Xiong, M., Discovering Conditional Functional Dependencies, ICDE'09, Shanghai 2009, pp. 1231-1234.

[40]. Fan, W., Li, J., Ma, S., Tang, N. and Yu, W., Interaction between record matching and data repairing, SIGMOD '11, Athens, Greece, 2011, ACM pp. 469-480.

[41]. Fan, W., Li, J., Ma, S., Tang, N. and Yu, W., Towards certain fixes with editing rules and master data, The VLDB Journal (2012), 21 (2), 213-238.

[42]. Fan, W. and Simeom, J., Integrity constraints for XML, PODS '00, Dallas, Texas, United States, 2000, ACM pp. 23-34.

[43]. Flesca, S., Furfaro, F., Greco, S. and Zumpano, E., Querying and Repairing Inconsistent XML Data. in WISE 2005, Springer Berlin, Heidelberg, 2005, 175-188.

[44]. Flesca, S., Furfaro, F., Greco, S. and Zumpano, E., Repairing Inconsistent XML Data with Functional Dependencies. in Encyclopedia of Database Technologies and Applications, Idea Group, 2005, 542-547.

[45]. Flesca, S., Furfaro, F., Greco, S. and Zumpano, E., Repairs and Consistent Answers for XML Data with Functional Dependencies. in Database and XML Technologies, Springer Berlin, Heidelberg, 2003, 238-253.

[46]. Flesca, S., Furfaro, F. and Parisi, F., Querying and Repairing Inconsistent Numerical Databases, ACM Trans. Database Syst. (2010), 35 (2), 1-50.

[47]. Giacomo, G.D., Lembo, D., Lenzerini, M. and Rosati, R., Tackling inconsistencies in data integration through source preferences Workshop on Information Quality in Information Systems - QDB, Paris, 2004, pp. 27-34.

[48]. Golab, L., Karloff, H. and Korn, F., On generating Near-Optimal Tableaux, PVLDB (2008).

[49]. Goldfarb, C.F., The SGML Handbook. Oxford University Press, 1991.

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

[51]. Hartmann, S. and Link, S., More Functional Dependencies for XML, LNCS 2798 (2003), 355-369.

[52]. Hartmann, S. and Link, S., More Functional Dependencies for XML. in Advances in Databases and Information Systems, Springer Berlin, Heidelberg, 2003, 355-369.

[53]. Huhtala, Y., Karkkainen, J., Porkka, P. and Toivonen, H., TANE: an Efficient Algorithm for Discovering Functional and Approximate Dependencies, The Computer Journal (1999), 42 (2), 100-111.

[54]. Hunter, D., Rafter, J., Ayers, D. and Vlist, E.V.D., Beginning XML. United Kingdom, 2007.

[55]. Kolahi, S. and Lakshmanan, L.V.S., Exploiting conflict structures in inconsistent databases, ADBIS'10 Proceedings of the 14th East European Conference on Advances in Databases and Information Systems, Novi Sad, Serbia, 2010, Springer-Verlag, pp. 320-335.

[56]. Kolahi, S. and Lakshmanan, L.V.S., On approximating optimum repairs for functional dependency violations, ICDT '09 Proceedings of the 12th International Conference on Database Theory St. Petersburg, Russia, 2009, ACM, pp. 53-62.

[57]. Kosek, J. and Nálevka, P., Relaxed: on the way towards true validation of compound documents, Proceedings of the 15th international conference on World Wide Web Edinburgh, Scotland, 2006, ACM pp. 427-436

[58]. Lampathaki, F., Mouzakitis, S., Gionis, G., Charabalidis, Y. and Askounis, D., Business to bussiness interoperability: A current review of XML data integration standards, Computer Standards & Interfaces (2009), 31 (6), 1045-1055.

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

[60]. Lee, M.-L., Ling, T.W. and Low, W.L., Designing Functional Dependencies for XML, Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology, London, 2002, Springer-Verlag, pp. 124-141.

[61]. Li, X.-Y., Yuan, J.-S. and Kong, Y.-H., Mining Association Rules from XML Data with Index Table, Proceedings of the Sixth International Conference on Machine Learning and Cybernetics, Hong Kong, 2007, pp. 3905 - 3910

[62]. Ling Feng and Dillon, T., Mining Interesting XML-Enabled Association Rules with Templates, LNCS (2005), 3377, 66-88.

[63]. Liu, J., Vincent, M. and Liu, C., Local XML functional dependencies, Proceedings of the 5th ACM international workshop on Web information and data management, New Orleans, Louisiana, USA, 2003, ACM, pp. 23-28.

[64]. Lv, T. and Yan, P., A Survey Study on XML Functional Dependencies, The First International Symposium on Data, Privacy, and E-Commerce, Chengdu, 2007, pp. 143 - 145

[65]. Lv, T. and Yan, P., XML Constraint-tree-based Functional Dependencies, ICEBE, Shanghai 2006, pp. 224-228.

[66]. Manolescu, I., Florescu, D. and Kossmann, D., Answering XML Queries on Heterogeneous Data Sources, Proceedings of the 27th International Conference on Very Large Data Bases, Roma, Italy, 2001, pp. 241-250.

[67]. Moro, M.M., Braganholo, V., Dorneles, C.F., Duarte, D., Galante, R. and Mello, R.S., XML: some papers in a haystack, SIGMOD Rec. (2009), 38 (2), 29-34.

[68]. Müller, H., Problems, methods, and challenges in comprehensive data cleansing. Professoren des Inst. Für Informatik, 2005.

[69]. Ng, W., Repairing Inconsistent Merged XML Data, Database and Expert Systems Applications, 2003.

[70]. Noël Novelli and Cicchetti, R., FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies, International Conference on Database Theory, London, 2001, pp. 189-203.

[71]. Pears, R., Koh, Y.S., Dobbie, G. and Yeap, W., Weighted association rule mining via a graph based connectivity model, Information Sciences (2013), 218 (1), 61-84.

[72]. Puhlmann, S., Naumann, F. and Eis, M., The Dirty XML Generator.

[73]. Rafiei, D., Moise, D.L. and Sun, D., Finding Syntactic Similarities Between XML Documents, Proceedings of the 17th International Conference on Database and Expert Systems Applications, DEXA'06, 2006, pp. 512-516.

[74]. Slawomir Staworko and Chomicki, J., Validity-Sensitive Querying of XML Databases, EDBT Workshops, 2006, pp. 164-177.

[75]. Tagarelli, A., Exploring dictionary-based semantic relatedness in labeled tree data, Information Sciences (2013), 220 (20), 244-268.

[76]. Tan, Z., Liu, C., Wang, W. and Shi, B., Consistent query answers from virtually integrated XML data, Journal of Systems and Software (2010), 83 (12), 2566-2578.

[77]. Tan, Z., Wang, W. and Shi, B., Extending Tree Automata to Obtain Consistent Query Answer from Inconsistent XML Document Proceedings of the First International Multi-Symposium on Computer and Computational Sciences (IMSCCS'06), 2006, pp. 488-495.

[78]. Tan, Z. and Zhang, L., Repairing XML functional dependency violations, Information Sciences (2011), 181 (23), 5304--5320.

[79]. Tan, Z., Zhang, Z., Wang, W. and Shi, B., Computing Repairs for Inconsistent XML Document Using Chase. in Anvances in Data and Web Management, Springer-Verlag 2007, 293-304.

[80]. Tan, Z., Zhang, Z., Wang, W. and Shi, B., Consistent data for inconsistent XML document, Information and Software Technology (2006), 49 (9-10), 497-459.

[81]. Trinh, T., Using Transversals for Discovering XML Functional Dependencies, FoIKS, Pisa, Italy, 2008, Springer-Verlag pp. 199-218.

[82]. Vincent, M.W., Liu, J. and Liu, C., Strong Functional Ddependencies and Their Application to Normal Forms in XML, ACM Transactions on Database Systems (2004), 29 (3), 445-462.

[83]. Vincent, M.W., Liu, J. and Mohania, M., The implication problem for 'closest node' functional dependencies in complete XML documents, J. Comput. Syst. Sci. (2012), 78 (4), 1045-1098.

[84]. Vo, B., Coenen, F. and Le, A.B., A new method for mining Frequent Weighted Itemsets based on WIT-trees, Expert Syst. Appl. (2013), 40 (4), 1256-1264.

[85]. Vo, L.T.H., Cao, J. and Rahayu, W., Discovering Conditional Functional Dependencies in XML Data, Australasian Database Conference, 2011, pp. 143-152.

[86]. Vo, L.T.H., Cao, J. and Rahayu, W., Structured Content-Based Query Answer for Improving Information Quality Submitted to World Wide Web (October, 2013).

[87]. Vo, L.T.H., Cao, J., Rahayu, W. and Nguyen, H.-Q., Structured content-aware discovery for improving XML data consistency, Inform. Sci. (2013), 248 (1), 168-190.

[88]. W3C, eXtensible Markup Language (XML), 2007. http://www.w3.org/xml

[89]. W3C, XML Path Language (XPath), 1999. http://www.w3.org/TR/xpath/

[90]. W3C, XML Schema, 2004. http://www.w3.org/TR/xmlschema-0/

[91]. Wahid, N. and Pardede, E., XML semantic constraint validation for XML updates: A survey, Semantic Technology and Information Retrieval Putrajaya, 2011, IEEE, pp. 57-63.

[92]. Wang, K., He, Y. and Han, J., Mining Frequent Itemsets Using Support Constraints, VLDB '00 Proceedings of the 26th International Conference on Very Large Data Bases Cairo, Egypt, 2000, Morgan Kaufmann Publishers Inc, pp. 43-52.

[93]. Wang, K. and Liu, H., Schema Discovery for Semistructured Data, In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, 1997, pp. 271--274.

[94]. Weis, M. and Naumann, F., Detecting Duplicate Objects in XML Documents, Proceedings of the 2004 international workshop on Information quality in information systems, Paris, France, 2004, ACM, pp. 10-19.

[95]. Weis, M. and Naumann, F., DogmatiX Tracks down Duplicates in XML, Proceedings of the 2005 ACM SIGMOD international conference on Management of data, Baltimore, Maryland, 2005, ACM pp. 431-442.

[96]. Wikimedia, kmwikibooks 2013. http://dumps.wikimedia.org/kmwikibooks

[97]. Wikipedia, Jaccard index. http://en.wikipedia.org/wiki/Jaccard_index

[98]. Wikipedia, Law of cosines. http://en.wikipedia.org/wiki/Law_of_cosines

[99]. Yakout, M., Elmagarmid, A.K., Neville, J. and Ouzzani, M., GDR: a system for guided data repair, SIGMOD, 2010, pp. 1223-1226.

[100]. Yakout, M., Elmagarmid, A.K., Neville, J., Ouzzani, M. and Ilyas, I.F., Guided data repair, Proc. VLDB Endow. (2011), 4 (5), 279-289.

[101]. Yu, C. and Jagadish, H.V., Efficient Discovery of XML Data Redundancies, Proceedings of the 32nd International Conference on Very Large Databases, Seoul, Korea, 2006, VLDB Endowment pp. 103-114.

[102]. Yu, C. and Jagadish, H.V., XML Schema refinement through redundancy detection and normalization, VLDB (2008), 17 (2), 203-223.

[103]. Yu, C. and Popa, L., Constraint-based XML query rewriting for data integration, SIGMOD '04, Paris, France, 2004, pp. 371-382.