Consistent query answers from virtually integrated XML data

Zijing Tan\textsuperscript{a,}\textsuperscript{*}, Chengfei Liu\textsuperscript{b}, Wei Wang\textsuperscript{a}, Baile Shi\textsuperscript{a}

\textsuperscript{a}School of Computer Science, Fudan University, Shanghai, China
\textsuperscript{b}Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia

\begin{abstract}
When data sources are virtually integrated, there is no common and centralized method to maintain global consistency, so inconsistencies with regard to global integrity constraints are very likely to occur. In this paper, we consider the problem of defining and computing consistent query answers when queries are posed to virtual XML data integration systems, which are specified following the local-as-view approach. We propose a powerful XML constraint model to define global constraints, which can express keys and functional dependencies, and which also extends the newly introduced conditional functional dependencies to XML. We provide an approach to defining XML views, which supports not only edge-path mappings but also data-value bindings to express the join operator. We give formal definitions of repair and consistent query answers with the XML data integration settings. Given a query on the global system, we present a two-step method to compute consistent query answers. First, the given query is transformed using the global constraints, such that to run the transformed query on the original global system will generate exactly the consistent query answers. Because the global instance is not materialized, the query on the global instance is then rewritten in the form of queries on the underlying data sources by reversing rules in view definitions. We illustrate that the XPath query transformations can be implemented in XQuery. Finally, we implement prototypes of our method and evaluate our algorithms in the experiments.
\end{abstract}

© 2010 Elsevier Inc. All rights reserved.
underlying data sources in our setting not only need to define the data-value bindings, but also should be flexible enough to define the document restructuring. Moreover, the mapping is usually incomplete. That is, the source instances cannot provide all the required data for the global instance. We introduce skolem functions in this process to provide the necessary default values. With an incomplete mapping, the number of global instances can be large, or even infinite, so we consider virtual XML data integration, where the global XML data instance is not materialized.

We concentrate on the LAV approach to define mappings. This setting is more flexible than the GAV setting for adding new data sources into a global system. Preexisting data sources in the system need not be considered when a new source is added. In consequence, inconsistencies are more likely to occur. Furthermore, the LAV approach is more challenging than the GAV approach when queries on the global system need to be processed. A GAV approach considers the global schema as a view of the data sources, so queries are usually easy to answer by “unfolding” the mapping from global schema. On the other hand, the LAV defines the mapping in a reversed manner, which is certainly more difficult for global query answering.

In our settings, certain integrity constraints are defined at the global level. However, we cannot guarantee that such global integrity constraints hold, because they are not maintained at the global level, especially in virtual integration. In fact, even if every integrated data locally satisfies the same integrity constraints, the constraints may be globally violated. The global constraints may fail to be maintained at all, and inconsistent data cannot be eliminated because of the autonomy of different data sources.

A more natural scenario is the one where integrity constraints are considered when queries are posed to the system. In this case, we only get data from the global system that is consistent w.r.t. global constraints. This is the basic intuition of consistent query answers. However, it is not easy to give an appropriate definition of consistent query answers in the XML data integration environment. In fact, this concept has not yet been well-studied for a single XML document, as reported by Chomicki (2007) and Fan (2008). In the LAV approach, given a set of data sources, there may be several or even infinite potential global instances that produce the data sources. Furthermore, with virtual data integration, we do not have a real global instance materialized at all. The data integration environment is quite different when consistent query answering must be considered.

Next we introduce our running example, to illustrate the basic settings of consistent query answering from XML data integration.

**Example 1.** In Fig. 1, we give three source schemas $S_0$, $S_1$ and $S_2$, and a global schema $S$. The schemas describe the account and customer information in a bank, and the meanings of most of the labels are self-explanatory. $S_0$ divide the accounts into two categories: deposit accounts ($DAccount$) with balance $\geq 0$, and loan accounts ($LAccount$) with balance $< 0$. $S_1$ denotes only the customer information. $S_2$ identifies an account with its holder in a composite unit $AC$. We also provide a global schema to integrate data from the three data sources. In the global schema $S$, accounts and customers are listed separately. If necessary, customers and their accounts are joined together with elements labeled with $K$ and $K$ in a key to foreign-key style. Each source schema is defined as a view of the global schema in the LAV approach. Intuitively, the division of accounts in $S_0$ can be implemented by queries on the $Account$ node in the global schema $S$ with filters (predicates). We need type mappings to rename labels between schemas. For example, the $Client$ node in $S_1$ should be mapped to the $Customer$ node in $S$. We also need edge-path mappings in the view definition of $S_2$ to cope with different hierarchies in $S$, e.g., the $Customer$ node. We leave the formal discussion of the view definitions to Section 6.

The available three source instances are also given in Fig. 1 as well, with the node values under the node labels in bold. We also provide one possible global instance, for illustration only. The mappings from source to global are incomplete, because data sources cannot provide all the necessary data for the global instance. Variables are introduced to node values as placeholders for the global instance. The variable “Y” occurs twice in the global instance, to join the account with its holder, because both of them are from a single $AC$ subtree in $I_2$. We will discuss how this is implemented in Section 7.

It is important to stress that the global instance is given for illustration only, it is never materialized in our framework. In the LAV approach, the number of potential global instances that produce the given source data can be large or even infinite, so it is usually impractical for materialization.

In a bank, a customer can have many accounts, whereas an account can have only one related customer. The zip code determines the city in the address information. We have three integrity constraints that are assumed to hold at the global level, whose formal definitions will be given in Section 3.

1. For $Account$, the $No$ is a key.
2. For $Customer$, the combination of $K$ and $Name$ is a key.
3. For $Address$, the $Zip$ determines the $City$.

For an XPath query \( /Global\!/Account/No \), the answers of this query on the global instance $I$ are the two $No$ nodes. The answers satisfy global constraints, so they are also the consistent query answers.

Now consider another query \( /Customer \) posed on the global instance, which is to find all the customers. The answers will be the two customer subtrees, with the name and address information. Here the nodes $K$ with variable values are removed to make the answers ground.

For consistent query answers, the address subtrees will be excluded. As for the zip “100000”, two different cities are given, which violates the global constraint 3. Here we regard the subtree rooted at $Address$ as a single unit, which is the target node of constraint 3. We can modify either the zip value or the city value to make the global instance consistent, but there is no single deterministic way of rectifying inconsistencies. As the consistent query answer is defined based on the intersection of all possible ways to restore consistency, the whole $Address$ subtrees are excluded from consistent query answers. We will give the formal definition of consistent query answers in Section 4.

We give one possible global instance to help readers understand the former example. In real applications, we need effective algorithms to compute consistent query answers for a global system without materialization. In this paper, we provide a method to compute consistent query answers based on rewriting of constraints and view rules. The idea is that the virtually integrated XML data is left as is, and queries are transformed when they arrive; as such, the transformed query on the original, possibly inconsistent XML data will yield exactly the consistent query answers. There are some problems that must be addressed. First, the global constraints need to be incorporated into the queries to validate the consistency of the answers. Second, because the data sources are defined as views of the global schema, the rules in the views need to be reversed for queries on the global system. Finally, we need efficient implementation of the query transformations.

We need a proper setting for the discussion of consistent query answering for XML data integration. There are two important foundations that must be laid before we can address the core problem. One concerns how to define global constraints, and the other concerns how to give the mappings between global schema and data sources. With varying settings for these two factors, the consis-
tent query answering problems can be quite different. In this paper, we give a flexible view definition approach to define the mappings between the global schema and the data sources and a powerful constraint model to define global constraints. In our settings, we assume that the global constraints and mappings between source and global schemas are predefined. The problem of discovering constraints and creating mappings are beyond the scope of this paper. They are discussed in some prior works (Popa et al., 2002; Fuxman et al., 2006; Yu and Jagadish, 2006; Fan et al., 2009) with different constraint models and mapping languages, which are complementary to the work in this paper.

The main contributions of this paper are as follows:

1. We provide a general constraint model for XML, which can express the most commonly discussed constraints, including functional dependencies and keys. It also naturally extends the recently discussed relational conditional functional dependencies (Bohannon et al., 2007; Fan et al., 2008) to XML.

2. We propose an approach to defining XML views that supports edge-path mappings and data-value bindings. The given XML views are flexible enough to express hierarchy restructuring, and to incorporate bindings of semantically related values, which can give the XPath with join expressions commonly found in applications.

3. We give a formal definition of consistent query answers for an XML data integration system in the LAV approach. The consistency of a global system is first defined on the basis of the introduction of minimal legal global instances. Then we come to the definition of a repair of a global system, which is an instance satisfying global constraints that minimally differs from a minimal legal global instance. Finally, the consistent query answer for a given query Q is defined to be the common part of the answers to Q on all possible repairs of the global system.

4. We provide effective algorithms to compute consistent query answers based on query rewriting. First, we rewrite queries on the global system using global constraints for consistent query answers. Second, we give a method to reverse rules in the definition of views to compute global query answers on the underlying data sources. Finally, for a given XPath query, we illustrate that the transformations can be implemented in XQuery.

5. We implement prototypes of our method and evaluate our framework and algorithms in the experiments.

1.1. Related works

In this subsection, we give an overview of related works in the literature and compare our approach with them.

Consistent query answering. There are many research studies about how to get consistent information from inconsistent relational databases, following two basic ideas which are formally introduced in Arenas et al. (1999). Repairing a database is to find another consistent database that minimally differs from the original one. Consistent query answering for a given query is to find the answer that is common in every possible repairs of the original database. The repairing and consistent query answering problems have been studied for first-order and scalar aggregation queries, considered using logic-based formalisms, and discussed for different types of constraints, including keys, functional dependencies,
inclusion dependencies, and denial constraints. The complexity of the consistent query answering problem is determined by the repair model, the constraint language and the query language involved. Thus, there are many different versions of this problem, with different data complexity bounds in various settings. Two recent surveys (Chomicki, 2007; Fan, 2008) provide an overview of this field.

In this paper, we discuss the consistent query answering problem in the XML data integration environment. This is a very challenging problem, as even the basic notions of repair and consistent query answers need to be redefined. The data model, the query language and the constraint language for XML are far more complicated than the relational ones. For example, to cope with the hierarchical structure of XML data, we need not only absolute constraints that hold on the entire document, but also relative ones that hold only on certain sub-documents. There are also intricate interactions between XML constraints and DTDs (Fan and Libkin, 2002). The query languages for XML, e.g., XPath and XQuery, are also totally different.

Consistent query answering with data integration. Most of the works on consistent query answering discuss only single databases. Certainly, data quality can be even worse with distributed data. There are limited attempts to tackle this issue in the relational data integration environment. Bertossi et al. (2002) discuss the consistent query answering problem in the data integration setting, by using first-order logic to express the data model, the query language, the constraint language, and the mappings between the global system and the data sources. Lembo et al. (2002) present a semantics for data integration in the presence of incomplete information sources and inconsistency of data with respect to constraints over the global schema, and they define a method to process queries on the global schema when the constraints are keys and foreign keys. Bertossi et al. (2002) and Lembo et al. (2002) both rely on Datalog programs to compute query answers.

As mentioned before, the problem we seek to address is certainly more difficult, with a complicated data model, query language, constraint language, and mapping language to cope with the semi-structured model of XML data. We give practical solutions by rewriting XML queries on the global schema to the queries on the underlying data sources, and we also provide experimental evaluations.

XML constraints. Based on the DTD and XML Schemas, several papers have addressed the topic of how to improve the semantic expressiveness of XML. The most commonly discussed constraints for XML are keys and functional dependencies. In Buneman et al. (2001), Buneman et al. (2002), and Hartmann and Link (2007), absolute and relative keys are discussed. Arenas and Libkin (2004) and Vincent et al. (2004) define functional dependencies (FDs) for XML. Different types of constraints, e.g., path constraints and inclusion constraints, are given in Buneman et al. (1999) and Fan and Simeon (2000).

In this paper, we give a more general constraint model for XML. It can express keys and functional dependencies for XML, and it also has the similar feature of incorporating bindings of semantically related values to capture the consistency of data like that of conditional functional dependencies (Bohannon et al., 2007; Fan et al., 2008).

XML data inconsistency. There are some discussions of XML data inconsistencies in the literature. Staworko and Chomicki (2006) consider the problem of querying XML documents that are not valid with respect to given DTDs. They present a validity-sensitive method of querying XML documents, which extracts more information from invalid XML documents than the standard query evaluation. Their work is different from ours as they consider DTD validity, not constraint satisfaction. Ng (2003) discusses the problem of resolving the inconsistency of merged XML documents with respect to a set of functional dependencies in the most concise merged format. The setting in Ng (2003) differs from this work; actually it considers how to put conflicting information from merged documents into a new document with a concise merged format, by introducing new element node types. Flesca et al. (2003) consider inconsistent XML data with respect to a set of functional dependencies. The repairs are based on replacing node values and introducing a function stating whether the node information is reliable. The definition of functional dependencies in Flesca et al. (2003) comes from Arenas and Libkin (2004). Flesca et al. (2005) investigate the existence of repairs with respect to a set of integrity constraints and a DTD. For all the cases where the existence of a repair is decidable, the complexity of providing consistent answers to a query is characterized. However, Flesca et al. (2005) do not provide any algorithms to compute consistent query answers. It is clear that the former work differ from our goal to define and compute consistent query answers from integrated XML data. We discuss a more powerful constraint model, give formal definitions, and provide practical solutions and experimental evaluations.

XML query rewriting. We use query rewriting to compute consistent query answers, so we also give a brief review of techniques in this area. In Fan et al. (2004), Fan et al. (2007) and Fan and Bohannon (2008), view definition and query rewriting are also discussed. In Fan et al. (2004) and Fan et al. (2007), a view is defined on the stored data, and queries are given on the view, so the queries are answered by a GAV approach. In Fan and Bohannon (2008), the source schema is embedded into the target schema, that is, an edge in the source schema is mapped to a path in the target schema. Our mapping language supports the edge-path mappings using XPath as well, and furthermore the join operator is implemented by binding semantically related values. We rewrite queries on the global schema as queries on the data sources by using not only the mappings, but also the target constraints.

Rewriting queries using views is required by multiple data management tasks (Halevy, 2001). In the context of XML, Xu and Ozsoyoglu (2005) discuss the problem of deciding whether there exists a rewriting of a query using XPath views, and finding the minimal rewriting if possible. In our setting, we consider the problem of virtually querying the global instance, as the global instance is not materialized in advance. Onose et al. (2006) use different methods to define views on an instance level. The input and output are all XQuery queries. Yu and Popa (2004) study the problem of answering queries through a target schema, and target constraints may be incorporated in the query rewriting. It defines the semantics of target query answering by constructing a canonical target instance. In their setting, the target constraints are assumed to be always satisfied, which is quite different from our problem statement.

To the best of our knowledge, neither of the former works considers global integrity constraint violations with XML as a global schema. In this paper, we discuss the query rewriting using constraints, and consistent query answering from virtually integrated XML data, which bring about new challenges.

1.2. Organization

The rest of the paper is organized as follows. Section 2 provides the basic notations. Section 3 gives a general constraint model for XML. In Section 4, we give a formal definition of consistent query answers from virtually integrated XML data in the LAV approach. Section 5 discusses the constraint based rewriting of queries for consistent query answers. We propose an approach to define XML views in Section 6. We provide a method to reverse rules in view definition for global query answering in Section 7. The prototype implementation and experimental results are given in Section 8. Finally, Section 9 draws some conclusions.
2. Preliminaries

In this section, we review the definitions of DTD, XML document, symbol mapping and the class of XPath queries considered in this paper.

**DTD (Fan and Bohannon, 2008).** We represent a DTD by (Ele, P, r), where Ele is a finite set of element types; r is a distinguished type in Ele, called the root type; P defines the element types. For each A in Ele, we say A → P(A) is the production of A. P(A) is a regular expression of the form: \( \alpha \) := str | \( B_1 \cdot \ldots \cdot \cdot \cdot B_n \cdot B \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \..
2. The combination of name and K is a key for customers.

\((\epsilon, \text{Global}/\text{Customer}, \langle \text{Name}, K \rangle | [[x_1, y_1], [x'_1, y'_1]]) \models \epsilon = x'_1 = y'_1 = y_1 = y'_1 = 0 = 1\)

3. The zip in address determines the city. \((\epsilon, \text{Global}/\text{Customer}/\text{Address}, \langle \text{Zip}, \text{City} \rangle | [[x_1, y_1], [x'_1, y'_1]]) \models x_1 = x'_1 = y_1 = y'_1\)

4. For customers with name “Tom”, the city in address determines the zip. \((\epsilon, \text{Global}/\text{Customer}, \langle \text{Name}, \text{Address}, \text{Zip}, \text{City} \rangle | [[x_1, y_1], z_1, z'_1, x'_1, y'_1]]) \models x_1 = x'_1, x_1 = “Tom”, z_1 = z'_1 \Rightarrow y_1 = y'_1\)

5. Customers with name “Mary” must be in city “Paris”. \((\epsilon, \text{Global}/\text{Customer}, \langle \text{Name}, \text{Address}/\text{City} \rangle | [[x_1, y_1]]) \models x_1 = “Mary” \Rightarrow y_1 = “Paris”\)

The introduction of variables, constants and symbol mapping in the constraint definition improve its semantic expressiveness and generalize the definition. Constraints 1 and 2 are examples of keys, and constraint 3 is a functional dependency. For example, constraint 1 says that in the whole document, two Account nodes cannot agree on values of their child node. The semantics is restricted to only those customer nodes that match the patterns. The “conditional” constraints make weaker assertions than the traditional ones, and are hence more widely applicable in real applications.

4. Consistent query answers from integrated XML data

In this section we give a formal definition of consistent query answers from integrated XML data under the LAV approach. In a data integration environment, we generally neglect the order of sibling nodes in the global instance. For example, in Fig. 1, the two customers in virtual global instance \(l\) come from different data sources, and the order of these two customers is usually irrelevant in the integrated result. Recall that the mappings between the global schema and the data sources are usually incomplete, and some variables may be introduced by skolem functions to node values; thus we also need to accommodate different symbol sets.

Definition 2. Let \(T, T'\) be two XML documents. Let the root node for \(T\) be \(r_T\), the root node for \(T'\) be \(r_{T'}\), the child nodes of \(r_T\) be \(\{v_1, \ldots, v_n\}\), and the child nodes of \(r_{T'}\) be \(\{u_1, \ldots, u_m\}\). We use \(T_{uj}\) to denote the subtree rooted at \(v_i\) in \(T\), and \(T_{uj}\) to denote the subtree rooted at \(u_j\) in \(T'\). We write \(T \subseteq T'\) iff

1. \(r_T = r_{T'}\);
2. For each different subtree \(T_{uj}\), there exists a different subtree \(T'_{uj}\), such that \(T'_{uj} \subseteq T_{uj} : \forall i \in [1, n], j \in [1, m]\)

Definition 3. Let \(T, T'\) be two XML documents, we write \(T \ll T'\) iff there exists a symbol mapping \(h\) from \(T\) to \(T'\), such that \(h(T) \subseteq T'\). If \(T \ll T'\) and \(T' \ll T\), we write \(T \ll T'\). If \(T \ll T'\) and \(T' \not\ll T\), we write \(T \not\ll T'\). It is clear that if \(T \ll T'\), then \(T \ll T'\). Note that \(\ll\) ignores the order of sibling nodes, and \(<\ll\) allows different symbol sets for node values. We need the partial order \(<\) with this semantics, as new variables may be introduced in data integrations.

We assume that in a global integration system \(G\), each data source of schema \(S_i\) is associated with data \(T_i\) and defined as a view \(\eta_i\) of the global schema \(S\). The global system does not have its own data, however, the possible virtual global instance \(T\) conforming to \(S\) is restricted. That is, by the view definition \(\eta_i\), the instance \(T\) must at least produce all the data in \(T_i\). We use \(\eta(T)\) to denote the view instance generated by \(\eta_i\) from \(T\). We first give some definitions about the global instances, following Bertossi et al. (2002).

Definition 4. Given a global system \(G\) with schema \(S\), and a list of data sources with schema \(S_i\), instance \(T_i\) and view definition \(\eta_i\) of \(S\), the set of legal global instances is \(\text{linst}(G) = \{ T | T \text{ conforms to } S, \eta(T) \text{ conforms to } S_i, T \not\ll \eta_i(T) \}\). We would like to characterize a global system as consistent or not. We must do this based on the data at hand, the one that is forced to be in the system, avoiding the inconsistencies caused by data that is only potentially contained in the global system. For this reason, we concentrate on the minimal instances.

Definition 5. Given a global system \(G\), a minimal global instance of \(G\) is an instance \(\text{linst}(G)\), and \(T \not\ll \text{linst}(G)\) such that \(T \ll T\). We denote by \(\text{mininst}(G)\) the set of minimal global instance of \(G\).

A global system is consistent w.r.t. a constraint \(\sigma\) if all the minimal global instances are consistent with \(\sigma\).

Definition 6. A global system \(G\) is consistent w.r.t. a set \(\Sigma\) of global integrity constraints, if \(\forall T \in \text{mininst}(G)\), \(T \not\ll \Sigma\).

The notion of repair is used as an auxiliary concept to define consistent query answers in the framework proposed by Arenas et al. (1999) for relational databases. Basically, a repair satisfies the constraints, and differs minimally from the original data. However, there are many different versions of repair in the literature even for relations, when different constraint models and update operations are considered. We next extend this concept to the area of XML. This is necessarily difficult, for we are coping with a totally different data model.

We first outline our approach to repair generation. We get repairs in the following way: starting from the original XML document \(T\), (1) we first try to find a document \(M\) that \(\text{sub-satisfies } \Sigma\). Here “sub-satisfies” means that (2) we can further find a document \(R\) based on \(M\), and \(R \not\ll \Sigma\).

To be specific, to obtain \(M\) from \(T\), we “remove” the conflicting data w.r.t. \(\Sigma\) from \(T\). This is done either by node value modifications (to replace a constant by a variable), or by node deletions. It is clear that this is not. We must do this based on the data at hand, the one that is forced to be in the system, avoiding the inconsistencies caused by data that is only potentially contained in the global system. For this reason, we concentrate on the minimal instances.

To be specific, to obtain \(M\) from \(T\), we “remove” the conflicting data w.r.t. \(\Sigma\) from \(T\). This is done either by node value modifications (to replace a constant by a variable), or by node deletions. It is clear that this is not. We must do this based on the data at hand, the one that is forced to be in the system, avoiding the inconsistencies caused by data that is only potentially contained in the global system. For this reason, we concentrate on the minimal instances.
to reconcile the inconsistent data w.r.t. the functional dependency, which says that the zip determines the city. 2) we further modify W to “Beijing” to produce a repair. This is actually implemented by applying the functional dependency to the mend, to enforce the equality between W and “Beijing” by replacing W with “Beijing”.

Similarly, we can modify the node value “Beijing” to “Shanghai” for a repair, or we can modify either zip value “100000” to a variable to generate a repair, e.g., the $R_3$ in Fig. 2. Note that the mend $M_3$ and the repair $R_3$ are the same, because the mend $M_3$ itself already satisfies the constraints. Theoretically, the chase result of applying constraints to $M_3$ is $R_3$, the same as $M_3$.

Also note that the deletion of a city or zip node is not allowed according to the schema $S$. We can delete a whole customer subtree to get an XML document satisfying global constraints and still conforming to $S$. However, this result is not a repair, because there exists a repair “nearer” to the original document based on the partial order $\prec$.

Earlier approaches in relations have confined the repair work to the deletion and insertion of entire tuples (Arenas et al., 1999). Wijsen (2005) proposes a theoretical framework that also covers updates as a repair primitive for relations. We find it necessary to support value modifications in the repair definition for XML, as the deletion and insertion of nodes in XML are usually restricted by the schema definition.

Note that as far as our constraint model is concerned, the insertion of nodes is not used in the repair. It can be verified that if an XML document violates a constraint in our model, the insertion of nodes will not “repair” it. For example, if an XML document violates a key constraint, or a (conditional) functional dependency, it is useless to insert nodes to repair the document.

The number of repairs can grow exponentially when node value modifications are supported in the definition of repair. However, there is no actual need to compute repairs in our framework, and the notion of repair is only used as an auxiliary concept to define consistent query answers.

**Definition 8.** Given a global system $G$, and a set $\Sigma$ of global integrity constraints, a repair of $G$ w.r.t. $\Sigma$ is an instance $R$ conforming to the global schema, such that $\exists T \in \text{mininst}(G), R$ is a repair for $T$ and $\Sigma$.

If $G$ is consistent, then the repairs are exactly the ones in $\text{mininst}(G)$. Also note that a repair is not required to be a legal global instance. Because different data sources may provide conflicting data w.r.t. global constraints, some nodes may be deleted, or node values may be modified in a repair to make it consistent. Thus a repair may not contain all the data from the data sources.

When posed to an XML document $T$, an XPath query $Q$ qualifies a set of nodes $\{v_1, \ldots, v_n\} = \{Q\}$ in $T$. The answers of $Q$ is the set $\{T_{v_1}, \ldots, T_{v_n}\}$ of subtrees rooted at $v_i$ in $T[i \in [1, n]]$. We use $Q(T)$ to denote the query answers of $Q$ on $T$. Before presenting the definition of consistent query answers, we give one more notation. Given a set $T^1$ of XML trees $\{T_1, \ldots, T_n\}$, we say that a tree $T$ is subcontained in $T^1$ (written $T \triangleleft T^1$), if there exists $T_i$ such that $T \triangleleft T_i$.

**Definition 9.** Given a global system $G$, a set $\Sigma$ of global integrity constraints, and a query $Q$, we say that $T$ is a general consistent query answer to $Q$ w.r.t. $\Sigma$, if for every repair $R$ of $G$, $T \triangleleft Q(R)$.

Please note that sometimes leaf nodes in a general consistent query answer may contain variable values. These values are suggested as placeholders by skolem functions for incomplete mapping or introduced as replacements of conflicting node values in repairs. They should generally be removed from the final answers to the users. Recall that we assume each element node is followed either by a list of child element nodes, or by a text node. Given an XML tree $T$, we use $\text{ground}(T)$ to denote the tree that results after (1) all the variable-valued text nodes are removed; (2) if all the child nodes of a non-leaf element node are removed, then it is removed as well.

**Definition 10.** Given a global system $G$, a set $\Sigma$ of global integrity constraints, and a query $Q$, we say that $T$ is a consistent query answer to $Q$ w.r.t. $\Sigma$, if there exists a general consistent query answer $T'$ to $Q$ w.r.t. $\Sigma$, and $T = \text{ground}(T')$.

**Example 5.** Consider the XPath query //Customer posed on the global system in Fig. 1. The consistent query answers will be the two customer subtrees, with only the name information. The four repairs given in Fig. 2 can facilitate understanding. We are sure that there are customers named “Tom” and “Jerry”, but their cities

![Fig. 2. Mends and repairs.](Image)
and zip codes are uncertain, as different repairs provide different results.

For another XPath query `//Account` on the global system, the consistent query answers will be the two account subtrees, with only nodes of `no` and `balance`. There is no constraint violation, and the `R` nodes are removed because they contain variable values caused by incomplete mappings.

We should once again stress that the global instances and repairs are not actually computed and that they are given for illustration purpose only.

5. Computing consistent query answers

We have given a theoretical framework of repair and consistent query answers from virtually integrated XML data. In this section we approach the problem of computing consistent query answers. Note that in this section we process the global query as if the global instance is materialized. However, as mentioned before, the global instance is actually a virtual one. This problem will be solved in Section 7, where we further rewrite the query on the virtual global instance to new queries on the underlying data sources by reversing the rules in the view definition.

5.1. Query processing based on constraint rewriting

Negation is required in the rewriting method. Note that we rely on minimal global instances for the definition of consistent query answers; thus the global system is “closed” to make the negation computation possible. We use the notation – to denote negation.

We start with the general form of constraints. A constraint `σ` of the form `(P, P, (P₁, ..., Pₙ))` and (X₁, ..., Xₙ) [y₁ = y₂, ..., yₙ₋₂ = yₙ₋₁ = y₁ = yₙ] is rewritten in the form ¬((P, P, (P₁, ..., Pₙ)) | X₁, ..., Xₙ) [y₁ = y₂, ..., yₙ₋₂ = yₙ₋₁ = ¬y₁ = yₙ]. Intuitively speaking, we first find the violating target nodes by using negation on the conditions, and then compute the consistent query answers by excluding the violating target nodes using another negation.

Example 6. In the proposed constraint model, a key constraint can always be given in the form: `(P, P, (P₁, ..., Pₙ)) | X₁, X₂, ..., Xₙ = x₁₁ = x₂₁ = ... = x₁ₙ = x₂ₙ = 0 = 1`, which is to say that two target nodes of a key constraint cannot agree on the values of all the value nodes of this constraint, or there is a contradiction “0=1”. Recall that we use `Xᵢ` to denote the `i`th node in variable `X` in `X₁, ..., Xₙ`.

The result of rewriting is ¬((P, P, (P₁, ..., Pₙ)) | X₁, X₂, ..., Xₙ = x₁₁ = x₂₁ = ... = x₁ₙ = x₂ₙ, 0 ≠ 1). As 0 ≠ 1 is trivial, it can be removed. The final result is ¬((P, P, (P₁, ..., Pₙ)) | X₁, X₂, ..., Xₙ = x₁₁ = x₂₁ = ... = x₁ₙ = x₂ₙ), which means that target nodes that have the same values for all the value nodes should be removed from the consistent query answers.

Given a query `Q` and a set `Σ` of constraints on the global system, we process `Q` based on constraint rewriting to compute consistent query answers. The approach is summarized as follows:

1. The nodes `{v₁, ..., vₖ}` qualified by `Q` are selected as usual;
2. The constraints in `Σ` whose target nodes have `vᵢ` as ancestor nodes should be identified. These constraints can be determined by considering `Q`, the context and target paths of the constraints;
3. The constraints are verified one by one, ordered by the concatenate of their context and target paths. Note that `vᵢ` is the domain for constraint validations. That is, at least one target node of the validated constraints should be selected from the descendant nodes of `vᵢ`. However, because the constraints may have context beyond `vᵢ`, the original document may be accessed again in the process of constraint validations;
4. The resulting XML document is generated top-down from `vᵢ`, excluding the violating value nodes w.r.t. constraints in `Σ`.

The details of the approach are given in Algorithm 1. We validate the constraints from `σ list` one by one for each original result `Tᵢ`. During the validation, we mark the violating nodes. When all the related constraints are validated, we remove the marked nodes from `Tᵢ`, and `Tᵢ` is the final output. Note that in real applications, because the resulting XML document is processed in a top-down fashion both for the result generation and for the constraint validation, the result can actually be output together with the validation. This will be further illustrated by the examples in the following subsection.

**Algorithm 1**

**input**: the query `Q`, the virtual global instance `T` with DTD `D`, the global constraint set `Σ`  
**output**: consistent query answering of `Q` on `T` w.r.t. `Σ`  
1 let `σ list` be empty initially;  
2 foreach `σ = (P, P, (P₁, ..., Pₙ)) | X₁, X₂, ..., Xₙ = x₁₁, x₂₁, ..., x₁ₙ = x₂ₙ` do  
3 if any simple XPath path in the query `Q` on `D` is a prefix of `P, P` then  
4 insert `σ` into `σ list` properly;  
5 [for `σ = (S, S, (..., (X, Y, Z)) ∈ σ list`, if `S/S` is a prefix of `P, P` then ensure that `σ` is before `σ` in `σ list`];  
6 `σ list` will be a list of constraints that should be validated for the query `Q`;  
7 `endif`  
8 evaluate the query `Q` on the global instance `T`, and let the nodes qualified be `{v₁, ..., vₖ}`;  
9 foreach `vᵢ ∈ {1, k}` do  
10 `σ list` = `σ list` do  
11 validate `σᵢ` (during the validation, at least one target node for `σᵢ` should be selected from `Tᵢ`);  
12 if `σᵢ` do not hold then  
13 mark the violating value nodes of `σᵢ` in `Tᵢ`;  
14 `endif`  
15 `endif`  
16 Output `Tᵢ`, top-down, excluding the marked nodes.  
17 `endif`  

The query `Q` is then evaluated on the XML document `T` with complexity |`T`|×|`Q`|, where |`T`| is the size of the document. Let the nodes qualified by `Q` on `T` be `{v₁, ..., vₖ}`. For each subtree rooted at `vᵢ` (i ∈ {1, k}), the related constraints are then validated. Let the number of target nodes qualified by a given constraint `σ` be `nᵢσ`, and let the number of target nodes simultaneously bound in the definition of `σ` be `mᵢσ`. The complexity to validate `σ` in `T` is `nᵢσ mᵢσ`. Here `nᵢσ` is usually much smaller than |`T`|. We also require at least one target node to be selected from the subtree rooted at `vᵢ`, so the selectivity of `Q` also affects the number of related nodes, which will be further illustrated in the experimental evaluations in Section 8. The value of `mᵢσ` is determined only by the definition of `σ`. For our general constraint model, we do not restrict the possible values of `mᵢσ`. However, practical constraints can be expressed by binding a small number of target nodes simultaneously. For example, as far as keys and functional dependencies are concerned, `mᵢσ` is actually 2. Constraints are validated one by one for each `vᵢ`, so the total complexity is `I × k × nᵢσ`. Here `I` is the number of related constraints, `k` is the number of qualified nodes by `Q`, `nᵢσ` is the maximum one in all the related `nᵢσ`, and `mᵢσ` is the maximum of all the related `mᵢσ`. Finally, the query result is output, excluding the violating nodes. Each node in `T`
is visited and output at most once, so this step has data complexity $|T|$.

5.2. Implementing in XQuery

In this subsection, we focus on XPath queries posed on the global system. With an XPath query $Q$ on the global system, we can rewrite the query as an XQuery query $Q_x$. For clarity, we use $Q$ to denote a query in XPath, and $Q_x$ to express a query in XQuery.

Without loss of generality, we illustrate the rewriting of an XPath query in XQuery using the following example. In the XQuery implementation, the resulting XML document is processed in a top-down fashion only once, both for the result generation and for the constraint validation.

Example 7. The XPath query $Q$: $//Customer$ is given to get all the customers. The global constraints are then considered to generate an XQuery $Q_x$ on the global system for consistent query answers.

1. let $doc := doc(T)$
2. let $S\langle List \rangle :=$
3. for $s$ in $doc/Global/Customer$
4. return $s$
5. let $S\langle Violating List \rangle :=$
6. for $s1$ in $S\langle List \rangle$, $s2$ in $S\langle List \rangle$
7. where $s1/Name = s2/Name$ and $s1/K = s2/K$
8. return $s1$
9. let $S\langle Result List \rangle := S\langle List \rangle[not(\ldots S\langle Violating List \rangle)]$
10. for $s$ in $S\langle Result List \rangle$
11. return
12. $< Customer >$
13. let $S\langle List \rangle := s/Address$
14. let $S\langle Violating List \rangle :=$
15. for $s1$ in $S\langle List \rangle$
16. $s2$ in $doc/Global/Customer/Address$
17. where $s1/Zip = s2/Zip$ and $s1/City = s2/City$
18. return $s1$
19. $< S\langle Name \rangle \ldots S\langle Violating List \rangle[] >$
20. $< /Customer >$

By considering the query $//Customer$, the context and the target paths of the constraints, constraints 2 and 3 are taken into account to generate the new query. Constraints 4 and 5 are also involved, but we can neglect them as they are given for illustration only. They can also be treated, as we have discussed the general way to rewrite constraints above.

In lines 2–4, all the customers are first collected in $S\langle List \rangle$, as the value domain. Lines 5–8 deal with the key constraint, which is an absolute constraint. There is no need to query the original document, as all the customers are available in $S\langle List \rangle$. Finally, all the violating customers are listed in $S\langle Violating List \rangle$. Then we take the negation in line 9, which returns the customers in $S\langle List \rangle$ that do not appear in $S\langle Violating List \rangle$. It is given in the XQuery style, which may be a little tricky.

Inside a legal customer, the functional dependency regarding city and zip is processed. Note that the possible resulting $Address$ node is always extracted under the given $Customer$ node in $S\langle Result List \rangle$, as suggested by line 13. The functional dependency is absolute, which is evaluated inside the whole document by lines 14–17. The domain of possible resulting address is given by $S\langle List \rangle$, so there is no need to compare two addresses if neither of them occurs in $S\langle List \rangle$. However, to validate the absolute constraint, we must query the document for all the possible addresses in line 15 using another value binding with $S\langle List \rangle$. If the address satisfies the constraint, line 18 will output it.

6. XML view definition

In this section, we propose an approach to defining XML views to be used by the XML data integration settings. We define an XML view as a mapping $\eta$. Given a document DTD $D$, a view DTD $D_v$, and an XML document $T$ of $D$, the XML view $\eta$ can generate an XML view instance that conforms to $D_v$ from $T$.

We require the following basic features in an XML view definition:

1. Type Mapping: Type mapping is used to rename labels from $D$ to $D_v$. It always maps the root type of $D_v$ to the root type of $D$. Given a type $A$ in $D_v$, below we use $\eta(A)$ to denote the type in $D$, which is mapped from $A$.

2. Edge-Path Mapping: $D$ and $D_v$ may have different structures, especially for a semi-structured data model like XML. We use a DTD annotating method similar to that in Fan and Bohannon (2008) to define edge-path mappings. To be specific, for each element type $A$ and its child type $B$ in $D_v$, $\eta$ maps the edge $(A, B)$ to a path in the simple XPath from $\eta(A)$ to $\eta(B)$ in $D$.

3. Value Binding: The type and edge-path mappings cannot fully express the relationships between node values in $D$ and $D_v$, for example, it cannot implement the commonly used $join$ operation. We find it necessary to incorporate bindings of semantically related values in the definition of views, and below we introduce a compact form to combine edge-path mappings and value bindings in a single expression. Prior works like Fan et al. (2007) and Fan and Bohannon (2008) lack this feature.

Example 8. In Fig. 1, the source schema $S_0$ is defined as a view of the global schema $S$. The root type $Bank$ of $S_0$ is mapped to the root type $Global$ of $S$, and the types $DAccount$ and $LAccount$ in $S_0$ are both mapped to the type $Account$ in $S$. Finally, the types $No$ and $Balance$ in $S_0$ are mapped to $No$ and $Balance$ in $S$, respectively.

We then illustrate the edge-path mappings in the form of Fan and Bohannon (2008), using XPath queries with predicates. Note that for each pair of an element type $A$ and its child type $B$ in $S_0$, the edge $(A, B)$ is mapped to an XPath path evaluated from the element $\eta(A)$ in $S$.

$edge(Bank, DAccount) \rightarrow Account[Balance > 0]$

$edge(Bank, LAccount) \rightarrow Account[Balance < 0]$

$edge(DAccount, No) \rightarrow No$

$edge(LAccount, Balance) \rightarrow Balance$

$edge(LAccount, Balance) \rightarrow Balance$

We illustrate the semantics of a view definition. Given an instance $T$ of a document DTD $D$, the corresponding view instance $T_V$ of the view DTD $D_v$ is built in a top-down computation. First the root of $T$ is extracted as the root of $T_V$, and then $T_V$ is iteratively expanded by generating children nodes of the current leaf nodes, starting from the root at the first step. Given a current leaf node $v$, assume the element type of $v$ is $A$ and $D_v$ has the production for $A$ as $P(A) = A \rightarrow \alpha$. The children of $v$ are generated by extracting nodes from $T$ via the XPath annotation for each child type $B$ in $\alpha$.

We can give the view definitions in the former example in a more compact form:

\[
\begin{align*}
(DAccount, (No, Balance))[(x_1, x_2)] & \rightarrow (Account[Balance >= 0], (No, Balance))[(x_1, x_2)], \text{ and} \\
(LAccount, (No, Balance))[(x_1, x_2)] & \rightarrow (Account[Balance > 0], (No, Balance))[(x_1, x_2)].
\end{align*}
\]

The type mapping, edge-path mapping and value binding are combined together. As described before, the root element $Bank$ of $S_0$ is mapped to the root element $Global$ of $S$. The two child nodes of $Bank$ are mapped using XPath queries with different predicates. For the two leaf nodes $No$ and $Balance$, variables $x_1$ and $x_2$ are used to bind their node values. The same variables occur in the rule body as well. Intuitively, this means that the view instance has the same values for the leaf nodes $No$ and $Balance$ as the document instance.
To be more formal, the rules in the view definition express the semantics of data-value relationships by symbol mapping. That is, given an instance $T$ of the document DTD $D$, it determines a symbol mapping $h$ from the rule body to $T$, which maps the variables in rule body to the corresponding node values in $T$. The same symbol mapping $h$ is then applied by the rule head, generating node values in the view instance $V_T$ of view DTD $D_V$.

The introduction of symbol mapping in a view definition makes it very flexible to express relationships between node values. We give one more example to illustrate it.

**Example 9.** We give the view definition of source schema $S_2$ using the compact expression.

$$
\text{(AC, \langle Account/No, Account/Balance, Customer/Name, Customer/Address/City, Customer/Address/Zip \rangle) (\{x_1, x_2, x_3, x_4, x_5\})}
\rightarrow
\text{(Account/No, Account/Balance) (\{x_1, x_2, x_3, x_4, x_5\})}
$$

This view defines the relationships between data values in a join manner, which has the same function as the following XQuery fragment.

```xml
let $doc := doc(T)
for $a1 in $doc/Global/Account,
  $a2 in $doc/Global/Customer
where $a1/R = $a2/K
return <AC >
  <Account > $a1/No, $a1/Balance
  </Account >
  <Customer > $a2/Name, $a2/Address
  </Customer >
</AC >
```

7. Computing query answers with data integration settings

We have discussed the constraint based rewriting of a query on the global system in Section 5. When posed to the global instance, it will generate consistent query answers. To really compute query answers in a virtual XML data integration environment, we must further rewrite queries on the virtual global instance as new queries on the underlying data sources. This is achievable, as we can map the XPath paths mentioned in the query to queries on the data sources by reversing the view definitions. We use the view definition approach proposed in Section 6 as the foundation for our discussion.

7.1. Reversing value bindings

We need to reverse the value bindings supported by our definition of views. A mapping rule is given in the form $\chi = R_0(\tilde{X}_0) \rightarrow R_1(\tilde{X}_1), \ldots, R_m(\tilde{X}_m)$, where $\tilde{X}_i (i \in \{0, m\})$ is a list of constants or variables. For each $i \in \{1, m\}$, we generate a reversed rule $R_i(\tilde{X}_i) \rightarrow R_0(\tilde{X}_0)$. We modify $\tilde{X}_i$ to obtain $\tilde{X}_i'$ as follows: for a variable $x_j$ in $\tilde{X}_i$, if $x_j$ is also in $\tilde{X}_0$, then $x_j$ is unchanged in $\tilde{X}_i'$. Otherwise, $x_j$ is a variable appearing in the body of $\chi$, but not in the head. We replace $x_j$ by a function $f(\chi, j, \tilde{X}_0)$ in $\tilde{X}_i'$.

The value of the function $f(\chi, j, \tilde{X}_0)$ is determined by the rule name $\chi$, the variable name $x_j$, and the values in the head of rule $\chi$. The introduction of this function not only binds all the values in the head of reversed rules, but also preserves the connections between values.

**Example 10.** We have two reversed rules for the rule $\chi$ in the definition of source schema $S_2$ (in Section 6):

$$
\text{(Account/No, Account/Balance) (\{x_1, x_2, x_3, x_4, x_5\})}
$$

$$
\text{(AC, \langle Account/No, Account/Balance, Customer/Name, Customer/Address/City, Customer/Address/Zip \rangle) (\{x_1, x_2, x_3, x_4, x_5\})}
$$

7.2. Computing global query answers in LAV approach

For each XPath query $Q$ on the virtual global instance, we can enumerate all the possible simple paths $\{Q_1, \ldots, Q_m\}$ that are in $Q$ w.r.t. the global DTD. The result of evaluating $Q$ on the global system is the union of evaluating each of the $Q_i (i \in \{1, m\})$ on the global system. Thus, in what follows we assume that $Q$ is a simple path.

We can convert the reversed view definitions to the following form:

$$
P_1(\tilde{X}_P_1) \rightarrow R_1(\tilde{X}_{R_1})
\ldots
P_n(\tilde{X}_P_n) \rightarrow R_n(\tilde{X}_{R_n})
$$

where $R_i$ is an edge in the source schema, and $P_i$ is a path starting from the root in the global schema. The reversed rules given before can be easily converted into this form, by binding one variable each time and unfolding paths in the global DTD. Given a query $Q$, the rules involved in the computation are defined in terms of $P_i$, which has $Q$ as a prefix. We assume that the rules are ordered, if $R_i$ is an edge following $R_j$ in the source schema, we have $i < j$. Rules ordered with this requirement will assure that if $P_i$ is a prefix of $P_j$, then $i < j$.

We introduce Algorithm 2 to compute answers of $Q$ on the global system, which is called for each data source in the form evaluate $(T, \eta, Q, v, \text{"true"})$. Here $T$ is the given data source instance, $\eta$ is the reversed view definition, and $v$ is the root of the XML tree containing the query answers. The Boolean value “true” is used as an indicator for the function at the first calling for a data source.

**Algorithm 2.**

**input:** the source instance $T$; the reversed view definition $\eta$; the query $Q$; node $v$ is given as the root node of the query answer; ind is “true” at the first calling, otherwise “false”

**output:** the query answer of $Q$ on $T$

1. If in the definition of $\eta$, some path $P_i$ matches $Q$ then
2. foreach $P_i$ that matches $Q$ do
3. evaluate $P_i$ on the global DTD, and let the result be node $E_i$; 4. with the rule $P_i(\tilde{X}_P_i) \rightarrow R_i(\tilde{X}_{R_i})$, assume $R_i$ is an edge from element $E_{\text{root}}$ to element $E$ in the source schema;
5. if ind is true then
6. let $P_{\text{new}}$ be the path from the root of source DTD to $E$; 7. //the first edge is absolute
8. endif
9. else
10. $P_{\text{new}} = E_i$; //other edges in $R_i$ are relative;
11. endif
12. foreach $v_j(i \in \{1, m\})$ do
13. copy $v_j$ to be a new node $v'_j$, rename label and assign value for $v'_j$ when required by $\eta$;
14. assign $v'_j$ to be a child node of $v$;
15. foreach child node $E_i$ of $E$ in the global DTD graph do
16. Evaluate($T, \eta, Q, E_i, v'_j, \text{false}$)
17. //iteratively expand the result
We give a brief explanation of Algorithm 2. In line 2, there may be more than one $P_i$ that matches $Q$ in the reversed view definition, because different edges in the source schema can be mapped to the same paths in the global system, for example, $DAccount$ and $LAccount$ in source schema $S_B$. As indicated by lines 5–10, except for the first edge matching in the rules, all the following edges are relative. This means that when a node is extracted from the source instance, its child nodes are extracted iteratively by the succeeding rules. Algorithm 2 is somewhat simplified to make the main idea clear. For example, the value assignment in line 13 should use the method of reversing value bindings as described before. $Q$ is evaluated first, and in lines 15–17 all the child nodes of the element node qualified by $Q$ in the global schema are evaluated iteratively. All the evaluations are conducted via reversed rules on the source instance.

We need lines 21–28, in which an edge in the source is mapped to a path in the global schema, so it is possible that some nodes in the global DTD do not have corresponding reversed rules. Some necessary default elements must be added to make the result conform to the global schema. For concatenation, we generate one node for each type. For disjunction, we choose an arbitrary type to generate a node. For the Kleene star, we generate only one node for the type.

Computational Complexity. Algorithm 2 constructs the result for query $Q$ on the global system in a top-down fashion, by using the reversed view definition $\eta$ on the source instance $T$. It can be seen that each node in source instance $T$ can be visited at most once, because each edge in the source schema is mapped to one simple XPath path in the global schema. Once a node in source instance is visited, the algorithm will iteratively visit its child nodes. Algorithm 2 may generate nodes that are not in the source instance $T$, because the mapping may be incomplete. The algorithm will generate default element nodes if necessary according to the global DTD. As described before, it will generate one node for each type for concatenation; choose an arbitrary type to generate a node for disjunction and generate only one node for the Kleene star. Thus, the number of generated nodes is bound by the size of global DTD. Algorithm 2 generates nodes in the query result one by one, and the upper bound of the data complexity is $|T|+|D|$, the sum of the size of the source instance $T$ and the size of the global schema $D$.

7.3. Implementing in XQuery

We can convert a given query $Q$ in XPath on the global system to a query $Q_x$ on the data sources in a richer language, e.g., XQuery. The query $Q_x$, when posed to the underlying data sources, will get the answers for $Q$. The implementation in XQuery is possible because the construction of XML elements is supported in XQuery, which can be used to introduce necessary default elements as suggested by Algorithm 2. User defined functions can also be imported to assign variable values to nodes if necessary. We illustrate it in the following example.

Example 11. Given the XPath query $//Account$ to get all the accounts from the global system, it will be translated into the following XQuery query.

```xquery
let $doc1 := doc('h1'), $doc2 := doc('l1'), $s1 := $doc1/Bank/DAccount, $s2 := $doc1/Bank/LAccount, $s3 := $doc2/Bank/AC/Account
return <Answer>
  <Account>
    <Balance/> <No/> <Bank/> 
  </Account>
</Answer>
```

We have three related reversed rules for the query $//Account$ in the global system, and the child nodes of Account are extracted iteratively. The node $<Answer><Account></Account>$ is introduced as the root node of the query answer, which is denoted $v$ in the function $\text{Evaluate}$. Essentially, for non-leaf nodes, only the mapped labels are generated iteratively. For leaf nodes, the values are also copied.

We use the function $f(\ldots)$ to reverse value bindings, and it generates a distinct variable each time when called with different parameters. We omit the details of $f$ here, as its semantics have been thoroughly discussed before.

8. Experimental Evaluation

8.1. Settings

To perform the experiments, we implement a rewriting module that translates XPath queries on the global system to XQuery queries on the underlying data sources, based on the reversed view definition and constraint rewriting. The transformed queries will produce consistent query answers when running on the data sources.

Given an XPath query $Q$ on the global system, we first rewrite $Q$ as a new XQuery $Q_x$ on the global system based on target constraints, which is discussed in Section 5. $Q_x$ is then recomposed by transforming all the paths mentioned in $Q_x$ into queries on the underlying data sources, as illustrated in Section 7. Finally, we run the generated XQuery queries on the source XML documents using an XQuery system built on Saxon. To the best of our knowledge, this is the first implementation of consistent query answering for XML data integration.

The experiments are conducted on a PC with a 3.0 GHz Pentium 4 CPU and 2 GB RAM, running Windows XP(SP3). We perform our experiments with artificial data sets generated by the IBM XML Generator. The schema used is an extension of Fig. 1. We generate XML documents that conform to the source DTDs with different sizes.

In the experiments, we compare the time of ordinary query answering with the time of consistent query answering. For time measurement, each query is run five times and the average time is recorded. The consistent query answering program first searches the source data for the domain of all qualified nodes by the query, just as the ordinary query answering program does. It then identifies the violating nodes w.r.t. global constraints. Because the constraints considered may have different contexts than the current resulting nodes, the data sources may be accessed again to validate constraints. Finally, the negation is computed to get the result.

8.2. Results

We run various types of XPath queries, for example, simple filters on data values, unions of queries, and Boolean combinations of filters. Below we give the evaluation time for queries with filters on the data values to illustrate the results. This kind of query is commonly seen in real applications, and is the foundation for the construction of more complex queries.
To simulate the data inconsistency, we may introduce noise to leaf nodes involved in the constraints for the experimental data set. When noise is introduced, with a given probability, nodes will have values violating the constraints. It is clear that the noise factor has no effect on the running time of ordinary query answering. The consistent query answering process also actually takes almost the same time on a consistent XML document, or on a badly inconsistent XML document. The number of violating nodes scarcely affects the query answering process, and the only difference is the negation computation in the final step of the consistent query answering process, which is usually trivial. The number of possible repairs increases dramatically with increasing noises, but the time for consistent query answering remains almost unchanged. This is an important advantage of using query transformation to compute consistent query answers for inconsistent data sets.

We further consider three factors when evaluating our method.

**Scalability.** In Fig. 3, we vary the size of source data sets. Each data source grows at the same rate, and here the total size of the data sources is given. We compare the time of ordinary query answering with the time of consistent query answering. Both times grow with larger data sets, and consistent query answering takes more time, which implies that most of the time is spent on getting violating nodes w.r.t. global constraints. This is understandable, as a very large portion of the leaf nodes are related to constraints in the experimental data set. Value comparisons in constraint validation need a lot of time, especially when the values themselves are of tree structures. Moreover, with larger data sets, the time for constraint validation grows quadratically for functional dependencies and keys, faster than the time for ordinary query answering.

**Selectivity of query.** In Fig. 4, with fixed data sources, we run queries with decreasing selectivity by using different conditions in filters. A lower selectivity means that fewer nodes are qualified. Because we have not constructed an index, the time for ordinary query answering is not sensitive to the selectivity of the queries. However, the set of qualified nodes is the domain for constraint validations. We do not compare two target nodes if neither of them is qualified by the query. That is, at least one target node in the comparison is selected from the qualified nodes. Thus, the time for consistent query answering decreases dramatically with lower selectivity.

**Impact of constraint sets.** In Fig. 5, we give the consistent query answering time on a fixed data set, when different constraint sets are considered. One constraint is added each time to the experiments. Rewriting based on constraints is processed in the resulting XML tree in a top-down fashion. The time to get violating nodes w.r.t. two constraints is the sum of the times to get the violating nodes of each individual constraint. Thus, Fig. 5 does not show an exponential growth in time with increasing constraints.

**9. Conclusions**

In this paper, we consider the problem of defining and computing consistent query answers when queries are posed to inconsistent virtually integrated XML data. An extended abstract of this paper was presented in Tan et al. (2009).

In our discussion, the global system and each data sources use XML as their schemas, and the mappings between the global schema and the data sources are defined in local-as-view approach. We propose an approach to defining XML views, which supports edge-path mappings and data-value bindings. We also provide a constraint model for XML to define global constraints, which can express the most commonly discussed constraints, including functional dependencies and keys. It naturally extends the recently discussed relational conditional functional dependencies to XML, as well.

We give a formal definition of consistent query answers from virtually integrated XML data, by redefining the concept of *repair* and *consistent query answers* for XML and for the data integration settings. We also provide a method to compute consistent query answers. Given a query on the global system, we first transform it with the global constraints considered for consistent query answers. As the global XML data instance is not materialized, we then transform the queries on global instance into queries on the underlying data sources for the actual query answers. This is achieved by reversing rules in the view definitions. For a given XPath query on the global system, we illustrate that both transformations mentioned above can be implemented in XQuery.

We also implement prototypes of our method and evaluate our algorithms in the experiments. Different parameters, including the scalability of the data sets, the selectivity of the queries, and varying constraint sets, are considered in the experiments.

For future work, we are interested in extending the concept of consistent query answering to the interactions between XML constraint satisfaction and schema validity (Arenas et al., 2008). The study of the XML schema mapping language also requires more
attention, e.g., the newly introduced mapping language based on tree patterns (Amano et al., 2009). We also intend to discuss the consistent query answering problem with the XML data exchange settings (Arenas and Libkin, 2008).

Acknowledgements

This work is supported by the National Natural Science Foundation of China under grant no. 60603043, and Australian Research Council Discovery Project under grant no. DP0878405.

References

Ng, W., 2003. Repairing inconsistent merged XML data. DEXA, 244–255.