IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITJ2782

A Methodology for Building XML Data Warehouses

Laura Irina Rusu, La Trobe University, Australia
J. Wenny Rahayu, La Trobe University, Australia
David Taniar, Monash University, Australia

ABSTRACT

Developing a data warehouse for XML documents involves two major processes: one of creating it, by processing XML raw documents into a specified data warehouse repository; and the other of querying it, by applying techniques to better answer users' queries. This paper focuses on the first part; that is identifying a systematic approach for building a data warehouse of XML documents, specifically for transferring data from an underlying XML database into a defined XML data warehouse. The proposed methodology on building XML data warehouses covers processes including data cleaning and integration, summarization, intermediate XML documents, and updating/linking existing documents and creating fact tables. In this paper, we also present a case study on how to put this methodology into practice. We utilise the XQuery technology in all of the above processes.

Keywords: *data warehouse; star schema; XML; XML schema*

INTRODUCTION

In the last few years, building a data warehouse for XML documents has become a very important issue, when considering the continual growth of representing different kinds of data as XML documents (Widom, 1999; World Wide Web Consortium). This is one of the reasons why researchers became interested in studying ways to optimise processing of XML documents and to obtain a better data warehouse to store optimised information for future reference.

Many papers have analysed how to design a better data warehouse for XML data from different points of view (e.g., Widom, 1999; Goffarelli, Maio & Rizzi, 1998; Vrdoljak, Banek & Rizzi, 2003; Zhang, Ling, Bruckner & Tjoa, 2003) and many other papers have focused on querying XML data warehouse or XML documents (e.g., Fernandez, Simeon & Wadler, 1999; Deutch, Fernandez, Florescu, Levy

This paper appears in the journal *International Journal of Data Warehousing and Mining* edited by David Taniar.

& Suciu, 1999), but almost all of them have considered only the design and representations issues of XML data warehouse or how to query them and very few have considered optimisation of data quality in their research.

In this paper, we propose a practical methodology for building XML documents data warehouses. We ensure that the data warehouse is one where the occurrences of dirty data, errors, duplications or inconsistencies are minimized as much as possible and a good summarisation exists. The steps cover two stages: (A) data cleaning and (B) data summarization, creating fact documents and linking all documents to create data warehouses. We use XQuery in all of the above processes. The main purpose of this paper is to show systematic steps to building an XML data warehouse as opposed to developing a model for designing a data warehouse. However, it is important to note that our proposed steps for building an XML data warehouse are generic enough to be applied to different XML data warehouse models.

The rest of this paper is organised as follows: After discussing related work, we present our proposed methodology for building XML data warehouses, then a case study exemplifies our methodology and the final section gives the conclusions.

RELATED WORK

There is a large amount of work in the data warehouse field. Many researchers have studied how to construct a data warehouse, first for relational databases (Goffarelli et al., 1998; Galhardas, Florescu, Shasha & Simon, 2000; Roddick et al., 1999; Song, Rowen, Medsker & Ewen, 2001) but in the last few years, for XML documents (Vrdoljak et al., 2003; Zhang et al., 2003), considering the spread of use for this kind of documents in a vast range of activities. Furthermore, if we think of steps in our proposed methodology, there were few attempts to solve the problem of data cleaning automation, too, but most researchers concentrated on databases field analysis.

A concrete methodology on how to construct an XML data warehouse analysing frequent patterns in user historical queries is provided in Zhang et al. (2003). The authors start from determining which data sources are more frequently accessed by the users, transform those queries in *Query Path Transactions* and, after applying a rule mining technique, calculate the *Frequent Query Paths* which stay at the base of building data warehouse schema. It was also mentioned that the final step in building a data warehouse would be to acquire clean and consistent data to feed to the data warehouse. However, there is not enough detail on how to ensure this. Although it seems to be a simple thing to do in the whole process, this is the place where corrupted or inconsistent data can slip into the data warehouse.

Another approach is proposed in Vrdoljak et al. (2003), where an XML data warehouse is designed from XML schemas, proposing a semi-automated process. After pre-processing an XML schema, creating and transforming a schema graph, the designer chooses facts for the data warehouse and, for each fact, follows a few steps in order to obtain star-schema: building the dependency graph from schema graph, rearranging the dependency graph, defining dimensions and measures and creating logical schema. In this approach, XQuery is used to query XML documents in three different situations: (i) examination of convergence and shared

hierarchies, (ii) searching for many-to-many relationships between the descendants of the fact in schema-graph and (iii) searching for one-to-many relationship toward the ancestors of the fact in the schema-graph. The authors specify that in the presence of many-to-many relationships, one of the logical design solutions proposed by Song, Rowen, Medsker and Ewen (2001) is to be adopted.

The aspect of data correctness is considered by Galhardas et al. (2000) and they propose a solution where a data-cleaning application is modelled as a directed acyclic flow of transformations, applied to the source data. This framework consists of a platform offering three services: data transformation, multi-table matching and duplicate elimination—each service being supported by a shared kernel of four macro-operators consisting of high-level SQL-like commands. The framework proposed by these authors for data cleaning automation addresses three main problems: object-identity, data entry errors and data inconsistencies across overlapping autonomous databases, but the method covers only data cleaning in the relational databases aspect.

Roddick, Mohania and Madria (1999) survey various summarisation procedures for databases and provide a categorisation of the mechanisms by which information can be summarised. They consider information capacity, vertical reduction by attribute projection, horizontal reduction by tuple selection, and horizontal/vertical reduction by concept ascension as being a few very good methods of summarisation, but as they only analysed implementation on database projects, future work should be done for implementing some specific technique of summarisation for XML documents. Kim and Park (2005) describe data reduction and summarisation of data streams by using a flexible adjustment of time section size.

Ram and Park (2004) present a hybrid of two approaches related to data cleaning, one considering federated schema and another one about using domain ontology, and their paper is based on using an ontology that explicitly captures knowledge about different types of semantic conflicts and propose ways to resolve them. Their ontology, named SCROL, presents an efficient way to identify and resolve these conflicts among multiple heterogeneous databases, but only in the "conclusions" section, do the authors propose a future extension of SCROL to cover XML documents.

Our proposed method focuses on practical aspects of building XML data warehouses through several practical steps, including data cleaning and integration, summarization, intermediate XML documents, and updating/linking existing documents and creating fact tables. The first part of it, related to data cleaning and integration, has clear rules and steps to follow and it is a very practical approach, useful for most of those who would like to clean their documents before starting any processing but have not identified any systematic techniques to do it comprehensively. What makes the main difference is that, in our methodology we have developed generic methods, whereby the proposed method is able to be applied to any collection of XML documents to be stored in an XML data warehouse.

PROPOSED METHOD ON BUILDING XML DATA WAREHOUSES

Our paper proposes a systematic approach on how to feed data from an un-

derlying XML database into a XML data warehouse. We emphasise the fact that the numerous processes needed to build a data warehouse are structured and optimised in our approach. Also, a methodology of building a data warehouse from an initial XML document is provided, developing necessary fact and dimensions.

The steps involved by this methodology are as follow:

Stage A. Data cleaning;
Stage B. Data summarisation, creating fact document and linking all together to obtain data warehouse, with the following steps:

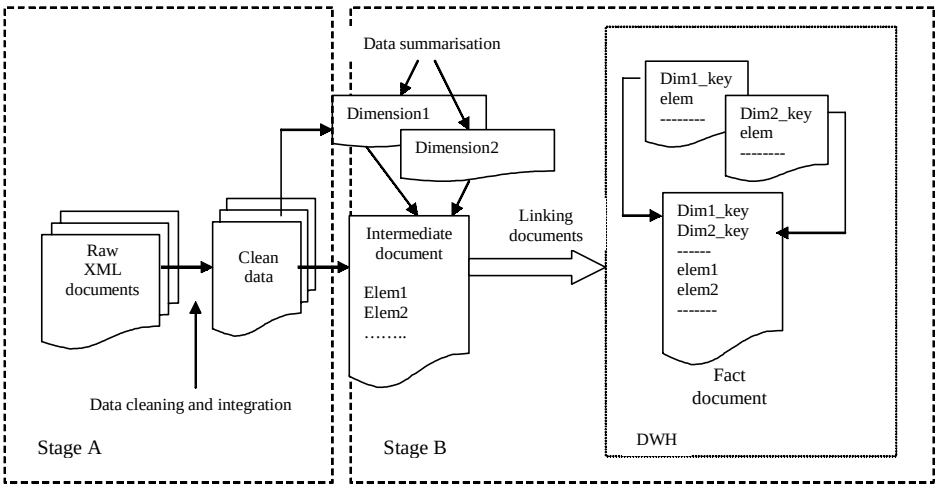
1. Data summarization: creating dimensions
2. Creating intermediate XML documents
3. Updating/linking existing documents and creating the complete data warehouse

Each of these steps is described in the next sections of our paper. Generalisation of the proposed methodology is extremely important and the aim is to be successfully applied to different XML data warehouses. Therefore for each of these steps we propose general rules and/or techniques and provide some examples on how to apply them. Figure 1 gives a graphical presentation of the entire process.

Stage A: Data Cleaning

Cleaning data is a very important step, so it should be analysed very carefully, as it can save a lot of future workload and time during the following steps. Two different situations can appear: whether an XML schema exists for the document or not. It is understandable that, in the first case, it would be easier to clean the data if we already have an XML schema developed, as we can use it for validation of the data in the XML documents. But in the other case, where a schema does not exist, we can also apply the same rules as provided below. The main difference is that without an XML schema, the rules below will have to be applied iteratively throughout the XML document.

Figure 1. Building XML data warehouse general process



In their paper, Ram and Park (2004) identify a large number of possible conflicts which can appear during an integration process. We adopt some of their conflict definitions here, trying to specifically analyse in our paper those types of conflicts that can appear in XML data warehouses. We will present a methodology for solving those conflicts which can be semi-automated. In some cases, user collaboration is vital to determine which element should be kept or removed.

Following we will analyse two major types of conflicts:

1. *schema level conflicts* (an XML schema exists but the document is not fully in accordance with the schema's specifications); and
2. *data level conflict* (regardless of whether a schema exists or not, wrongly typed data into the documents can appear). We will analyse each of these types of conflicts, establishing rules to follow and giving examples on techniques which should be used for eliminate them in the next sections.

Schema Level Conflicts

The main purpose of an XML schema is to provide a structure on how to construct a specific XML document. In an XML schema, requirements regarding elements, attributes, hierarchy, data types, possible values (restrictions), order indicators, occurrence indicators, etc., are clearly defined. When there are inconsistencies from these requirements within the XML documents, we can say that we have "schema level conflicts". Table 1 contains a list of the most significant set of schema conflicts.

When the documents will be validated against its schema, an error will be raised for each inconsistency. But the user may not be able to correct them one by one as they are raised, because in the case of very large documents, a lot of precious time would be lost. Following we present a few steps and examples on how this schema level conflict-solving activity can be automated.

(a) Schema Conflict S1: Name Conflict

This type of conflict appears when names assigned to elements and/or attributes vary during the document and do not correspond with names established in the schema. In this section, we present the description, a rule to follow, procedure steps and some examples of this type of conflict.

In XML, each element is identified by a name. Graphically it is represented quite similar with a *tag* in HTML (e.g., start tag <elem>, end tag </elem>), but the dif-

Table 1. Schema level conflicts

Code	Conflict	Description
*S*1	Name conflict	Names assigned to elements and/or attributes vary during the document
*S*2	Data type conflict	Elements and/or attributes do not respect data type established in the schema
*S*3	Schema restrictions conflict	Order indicators, number of occurrences different from specifications

ference is while in HTML a tag specifies just the element format, in XML, name can specify the content of that element so it can be easily understood what it refers to. Names of elements and attributes in XML documents are case sensitive. If an XML schema exists, it will specify these names, and the document(s) must always observe them.

Rule: Name of elements and attributes in the document should match names specified in the schema.

Steps: The steps are divided into four, as follows:

Step 1. For each element in the document, we verify if its name exists in the list of elements and attributes in XML schema; and at the end we create a document that contains elements that are not found in the schema;

Step 2. For elements identified at Step 2, we will compare their uppercase-transformed names against the uppercase names from schema.

Step 3. If we find upper-case-equal names, there are two possibilities:

1. it was a mistyping lowercase/uppercase → ask user if he wants to replace the wrong name with the correct one; or
2. there are two different elements → user should be asked if he wants to extend the document schema or not;

Step 4. For elements which are not found after upper-case search (Step 3) which are possibly new elements → ask user if he wants to extend the document schema or not.

(b) Schema Conflict S2: Data Type Conflict

This conflict appears when the actual contents of elements and/or attributes do not respect the data type assigned for each of them in the schema. Following we are giving some details, a rule to follow, procedure steps and some examples.

In an XML schema, an element content type can be specified. For example, it can be a string (e.g., a name of a person), an integer (e.g., street number, number of children, etc.), a decimal number (e.g., a price), a date (e.g., date of birth, purchase date, etc.) so on. There can be situations where for different reasons (e.g., mistyping), an element is wrongly introduced in the document, for example, a name "John2", which contains a number, or a date of birth "15/O5/250O", which contains the letter "O" instead of "0" (zero).

Rule: Data types must be the same with the specified data types in the schema. Furthermore, natural logic should be respected (for example, a telephone number will probably have "string" data type but it can have only digits, not letters, a name cannot contain digits, but only characters, etc…).

Steps: The steps are divided into three, as follows:

Step 1. Identifying types of elements in an XML schema and what conditions they should fulfil.

Step 2. Starting from an XML schema, we will create a document that will contain elements and their data types.

Step 3. We use the document created at Step 2 to check if data type of elements in our raw XML document are re-

spected. If not, we return a document, containing all wrong-type elements found in the initial document, to be corrected.

(c) Schema Conflict S3: Schema Restrictions Conflict

This type of conflict appears when the structure of our document does not entirely respect schema specifications, specifically: how many apparitions can an element have, how elements are embedded one into another, which is the correct sequence for them to appear, and so on. Following we give some details, a rule to follow, procedure steps and few examples.

In a XML document, elements can be embedded one into another. Container element is named "parent" and the contained element is named "child". But the user can decide, for example, that not all children elements should appear at a time or one element should appear at least twice, and so on. Furthermore, a user can be interested in how children elements appear in their parent, in a specific order or not.

Rule: Order indicators and number of occurrences, if specified in the schema, should be respected.

Steps: The steps are divided into three, as follows:

Step 1. Verify if *order indicators* are respected. They can be: all, choice, or sequence. *"All"* indicates that child elements declared in it can appear in any order and each child element must occur only once. *"Choice"* indicates that either one element or another can occur, and if *"sequence"* indicator exists, elements inside it should appear only in the specified order;

Step 2. In an XML schema, the umber of occurrences of an element can be specified, for example minOccur="1" & maxOccur="20" means that at least one apparition of element should occur and no more than 20 occurrences;

- if minOccur=0 → we don't look for it; it may not appear
- if minOccur=1 → search if find at least 1 apparition
- if minOccur=n → search if find at least n occurrences
- if maxOccur=1 → search if only 1 apparition
- if maxOccur=n → search if number of occurrences lower than n

Step 3. We verify how schema-specified hierarchy is observed and analyse the relationship between elements (child-parent, sibling, etc.).

Data Level Conflicts

Data level conflicts appear when there are differences on how elements are entered in our XML documents. In Table 2 we present three types of data level conflicts.

These kinds of conflicts can appear in both situations described at the beginning of our paper (with or without existence of XML schemas), and they can be solved in the same way.

First, we show a few examples of each data level conflict type and we present steps to follow in removing them and examples on how to apply those steps.

(a) Data Level Conflict D1: Data Value

This kind of conflict appears when the actual content of an element is entered differently in the document. Sometimes it

Table 2. Data level conflicts

Code	Conflict	Description
*D*1	Data value conflict	Different ways to instantiate a certain element
*D*2	Data unit conflict	Same element can be instantiate using different measuring units in the document or in different documents in XML database
*D*3	Data representation conflict	Different structure used for an element instantiation (e.g., date format)

can be due to mistyping but more often it is just another way to express the same thing. If the same information is entered in the document in multiple ways, this can produce a high level of inconsistency when building the data warehouse.

Below there are few examples:

- a customer name was entered two or more times, by different departments in a store, in a different manner/order (surname&firstname, firstname&surname, surname&thefather's initial &firstname, etc.);
- a country can be entered using different conventions, as entire name ("Australia") or as an acronym ("Au" or "AU"); this can be the case of suburbs, too;
- "price" can be entered in different ways, e.g., "10" or "$10" or "AUD10";
- a date (e.g., "orderDate") can have different formats (British, Australian, American or short year — yy, long year – yyyy, etc.).

(b) Data Level Conflict D2: Data Unit

In the same way as a data value conflict, a data unit conflict appears because of using different measuring units when talking about some entities (e.g., volume, length, temperature). Examples include:

- "size" can be expressed using different measuring units (meters, centimetres, etc.);
- volume can be expressed in litres, cube-meters, etc.; and
- the temperature can be expressed in Fahrenheit or Celsius degrees.

(c) Data Level Conflict D3: Data Representation

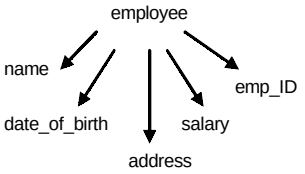
There are situations when different documents can contain different ways to use XML elements to express the same specific information. Figure 2a and 2b contain an example of how a date can be represented: as a single element, "date_of_birth", or as a complex element, with children elements for "day", "month" and "year".

(d) Steps for Solving Conflicts D1, D2, D3

For all data level conflicts described above, the procedure steps and examples of how to implement the code to solve them are quite similar, so we will discuss them generally, during this section and the following one.

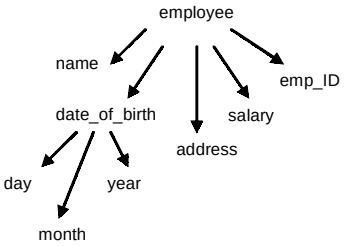
In solving these kinds of conflicts, our methodology is little different from solving schema level conflicts, because at this stage we don't know against what we should

Figure 2a. Date, represented as an element, and code example




```
<employee>
        <emp_ID>123</emp_ID>
        <name>John</name>
        <date_of_birth>15/05/1971</date_of_birth>
        <address>12, Swanston St.<address>
        <salary>$1000</salary>
</employee>
```

Figure 2b. Date, represented as an element with children, and code example




```
<employee>
        <emp_ID>123</emp_ID>
        <name>John</name>
        <date_of_birth>
                <day>15</day>
                <month>05</month>
                <year>1971</year>
        </date_of_birth>
        <address>12, Swanston St.<address>
        <salary>$1000</salary>
</employee>
```

correct our data. A few researchers have tried to solve this in databases, and they proposed using a "federated schema" or "ontology" for verifying their data (Ram & Park, 2004).

We propose to solve it by using a "dictionary" that is built considering those elements and attributes that can have different representation. For example, our dictionary will contain countries names (e.g., "Australia") and their acronyms/abbreviations (e.g., "AU"), easy to transform from one to another. The dictionary will stand as a new document and the user will decide, at the beginning of cleaning activity, what sort of information this document should include. We say that each possible value of each element/attribute name appearing in dictionary is an *"instance"*. The operation of finding an instance of a concept in the dictionary (*"source"*) and taking its correspondent instance value (*"destination"*) is named *"translation"*. A translation can consist of one or more *operations*.

If we consider Figure 2a as "source" and Figure 2b as "destination", the translation will consist in three similar operations:

- get-month-from-date (date_of_birth);
- get-day-from-date (date_of_birth);
- get-year-from-date (date_of_birth)

Each of these operations' result will be "packed" in some new elements, future children of "date_of_birth" destination element.

If we consider Figure 2b as "source" and Figure 2a as "destination", the translation will consist in two operations:

- Concatenate string values extracted from <day>, <month> and <year> elements.
- Use a constructor for building a date from the result of concatenation, which will be "date_of_birth" element in the destination; one example of how constructor acts is:

xsd:date ("2000-01-01") → date representing 1st of January, 2000.

In the same way, using the dictionary and translation operation, we obtain the fullname of a country if we know the acronym and vice versa.

Stage B: Data Summarization, Fact Document and Creating Data Warehouse

Data Summarization: Creating Dimensions

Because not the entire volume of existing data in the underlying database will be absorbed into the XML data warehouse, this step describes how to perform *data summarisation*. We must extract only useful and valuable information, so we will create another XML document(s) which will be, at the end, part of the data warehouse. Following, we will present how to construct dimensions using summarisation. We'll give some examples to show different kinds of dimensions that can appear and we will develop general techniques for constructing both *"constructed"* or *"extracted"* dimensions.

Generally, we can separate this step into two different approaches. Depending of how many levels of summarisation we will have for a specific dimension, we will either (1) create and populate new documents that contain *extraction* from initial data; or (2) create *special-constructed* values.

For example (Figure 3):

a. If we need to create a "part-of-the-day" dimension (e.g., query: "What are the dynamic figures of sales during the day: morning, afternoon, and evening") we will need to create and populate the dimension as a new document — *a "constructed" dimensio*n;

Figure 3. Dimensions created and populated as new XML documents

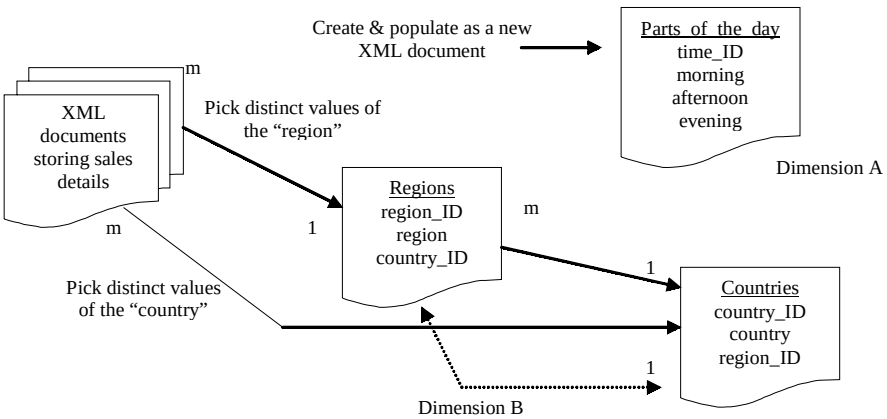


Figure 4. A general way to construct a dimension with n elements

```
for $a in (1,2,…n)
document {
       <new_element>
              <element_ID>{$a}</element_ID>
              <element_name1>{value}</element_name1>
              <element_name2>{value}</element_name2>
              ……..
       </new_element>
```

Figure 5. Example of creating a dimension

```
document {
       <semester>
              <semID>1</semID>
              <start_month>1</start_month>
              <end_month>6</end_month>
       </semester>
       <semester>
              <semID>2</semID>
              <start_month>7</start_month>
              <end_month>12</end_month>
       </semester>
},
```

b. If we need "country" or "region" as a level of summarisation (e.g., query: "What are the sales of product X by countries/regions"), we can find this information by querying directly into the primary document and searching for distinct values of "country" and/or "region" element — thus *an "extracted dimension"*.

Techniques to Create "Constructed" Dimensions

If necessary data do not exist in our document, the first step is to identify which elements need to be created. Secondly, a key for linking the dimension with the fact document in the data warehouse should exist. This key must be unique as it will serve to identify a specific requested element and for extracting it in future queries. A general way to construct a dimension with "n" new elements is as shown in Figure 4.

In Figure 4, <new_element> is a tag representing a new node in the new created dimension, <element_name1>, <element_name2> etc are names of node's children, taking specific *values*, and <element_ID> will be the unique identifier of <new_element>.

To demonstrate the above general technique with a concrete case, we will create and populate a "semester" dimension as shown in Figure 5.

In the "semester" example, when we will have to link semester dimension with the fact data, we will only need to determine which semester each date corresponds, by extracting month-from-date and comparing it with <start_month> and <end_month> values.

Figure 6. A general way to build an extracted dimension

```
let $a:=0
document {
for $t in distinct-values(doc("doc_name.xml")//element)
let $a:=$a+1
return
       <new_element>
              <elementID>{$a}</elementID>
              <element_name>{$t}</element_name>
              ……
       </new_element>
},
```

Figure 7. A general way to build a partial-extracted dimension

```
let $a:=0
document {
for $b in distinct-svalues(doc(doc_name.xml")/element)
       let $a:=$a+1
       return
            <new_element>
              <element_ID>{$a}</element_ID)
              <element1>{function1($b)}</element1>
              <element2>{function1($b)}</element2>
              <element3>{function2($b)}</element3>
              ……..
            </new_element>
}
```

Techniques to Create "Extracted" Dimensions

If necessary data already exist in the document, we may be interested in distinct values of the element involved and therefore we will need to extract them in a newly created document. At the same time, a key (e.g., <element_ID>) will be easily created, using a variable for incrementing its unique value. (See Figure 6.)

In Figure 6, <new_element> is a tag representing a new created element in the dimension. It contains a key (<elementID>, which takes predetermined values), actual value which is the value of interest (that is <element_name>, e.g., values of "country") and any other elements that can be helpful in the dimension.

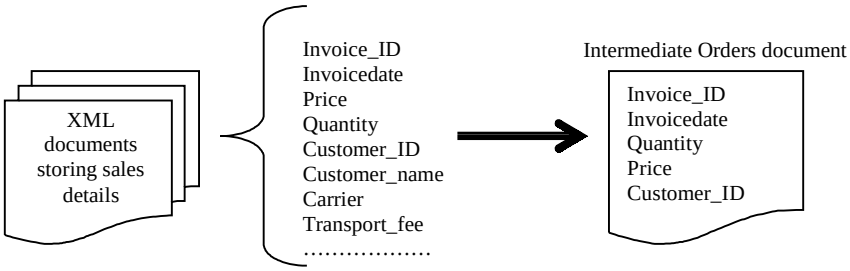
There can be situations where the desired data do not exist in the initial document, but they can be extracted from other existing elements, using specific functions (e.g., we may have the entire date element, but only month would be needed for a dimension). (Applying a specific XQuery function (get-month-from-date()) we can obtain what we need.)

A general way to construct such a *partial-extracted* dimension is described in Figure 7 where <new_element> is a node in the new constructed dimension, <element1>, <element2> etc are children of <new_element>, containing desired extracted values and <element_ID> is an unique identifier of the <new_element>.

Figure 8. Example of building a time dimension

```
let $a:=0
document {
for $b in distinct-values(doc("doc_name.xml")/date_node)
let $a:=$a+1
return
    <time_node>
       <timeID>{$a}</timeID>
       <date>{$b}</date>
       <month>{get-month-from-date($b)}</month>
       <year>{get-year-from-date($b)}</ year >
  </time_node>
}
```

Figure 9. Extracting important information as a new XML document



As an example, a time dimension is shown in Figure 8, using two date functions in XQuery.

In this case, *function1* and *function2* are get-month-from-date() and respectively, get-year-from-date(). There are many functions for date and time operations, available in XQuery (http://www.w3.org/TR/xpath-functions), so a large range of time level summarisation can be analysed and created.

Creating Intermediate XML Documents

In the process of creating a data warehouse from collection of documents, creating intermediate documents is a common way to *extract valuable & necessary information* (refer to example in Figure 9).

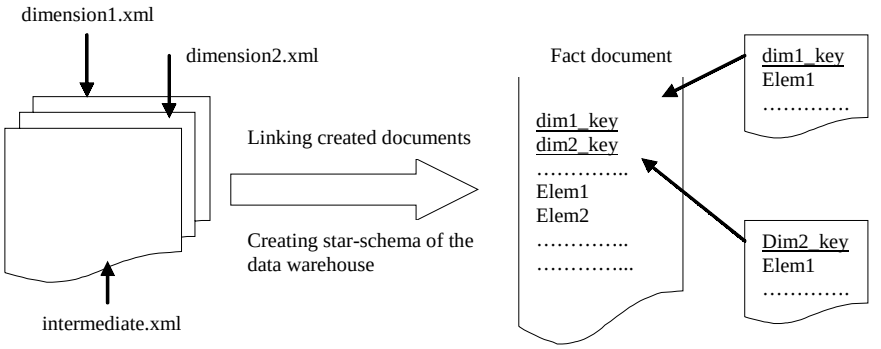
Which information in the initial documents is most important and necessary and should be kept in the data warehouse is a very good question and researchers have attempted to answer it by determining different complex techniques (detecting patterns in historical user queries (Zhang et al., 2003) or detecting shared hierarchies and convergence of dependencies (Fernandez et al., 1999). Still, for general users the analysis of possible queries in the domain remains a common way to do it.

During this step, we are only interested in data representing activity, which include data involved in queries, calculations etc., from our initial document. At the

Figure 10. Generic way to create the intermediate document

```
document {
for $t in (doc("doc_name.xml"))
return
       <temp_fact>
              <elem1_name>{$t//elem1_content}</elem1_name>
              <elem2_name>{$t//elem2_content}</elem2_name>
              ……..
       </temp_fact>
},
```

Figure 11. Linking documents and creating star-schema of data warehouse



same time, we will bring in the intermediate document elements from our initial document, which are keys to dimensions, if they already exist (e.g., "customerID" in Figure 9, which references "customer" dimension). Actual fact document in data warehouse will be this intermediate document, but linked to the dimensions. (See Figure 10.)

In Figure 10, <temp_fact> is a tag representing a new element in the intermediate document, containing <elem1_name> (name of the element) and <elem1_content> (value of element which is valuable for our fact document), etc.

Updating/Linking Existing Documents; Creating Fact Document

At this step all intermediate XML documents created in the earlier steps should be linked, in such a way that relationships between keys are established (Figure 11).

If linking dimensions to intermediate documents and obtaining facts are processed altogether, the number of iterations through our initial document will be lower, so it subsequently reduces the processing time. A general way to do it is shown in Figure 12.

Figure 12. Linking dimensions to fact document

```
let $a:=doc("dimension1.xml")          →e.g. time dimension
let $b:=doc("dimension2.xml")          →e.g. customer dimension
document
{
for $t in (doc("intermediate.xml")/node)
return
       <dim1_key>{for $p in $a
                  where $p//element=$t//element
                  return $p//dim1_key}
       </dim1_key>
       <dim2_key>{for $p in $b
                  where $p//element=$t//element
                  return $p//dim2_key}
       </dim2_key>
       ------------- ( for all dimensions ) ---------
       <elem1>{$t//elem1_name}</elem1>
       <elem2>{$t//elem2_name}</elem2>
       <elem3>{$t//elem1 * $t//elem2}</elem3>
             ---- (for all extracted & calculated elements ) ----

}
```

In Figure 12, we just obtained the fact, where <dim1_key>, <dim2_key> etc represent the new created keys elements which will link the fact to dimensions and <elem1>, <elem2>, <elem3>, etc., are elements of the fact, extracted from intermediate document. As can be seen in the <elem3> declaration, a large range of operators can be applied, in order to obtain desired values for analysis (e.g., price * quantity=income).

A CASE STUDY

Because the main purpose of this paper is to show systematic steps to build a XML data warehouse, we present a case study based on a XML document containing data about borrowed books in a library (see Figure 13). However, it is important to note that our proposed steps for building a XML data warehouse are generic enough to be applied on different XML documents.

In this section, we will show how the star-schema of data warehouse can be obtained from initial XML documents structure, following our proposal and steps presented in the previous section. An example of visual representation of mentioned XML document is shown in Figure 14.

Figure 15 shows the mapping from the visual representation as described in Figure 14 to the implementation in XML schema (during the case study, we name this schema "myschema.xml", as it is a XML document by itself):

Data Cleaning

The first step is to apply the specified rules described in the methodology, which include verifying correctness of all schema stipulations, eliminating duplicate records, inconsistencies and data entry errors. These rules are applied to the data that will be transferred to our data warehouse. This step is normally performed by a user who has a good understanding of the domain, as it supposes some interaction during the process, in vital moments, for deciding on

Figure 13. Example of XML document, considered in the case study

(Document example: libraryBooks.xml)

```
<libraryBooks>
       <book>
              <title>Calculus with analytic geometry</title>
              <publisher>
                     <name>Houghton Mifflin Co</name>
                     <address>Boston, US</address>
              </publisher>
              <ISBN>0395899206</ISBN>
              <publishing_date>1998</publishing_date>
              <author>
                     <name>Larson, Roland E.</name>
                     <affiliation>Boston University</affiliation>
              <author>
              <borrower>
                     <name>John John</name>
                     <identification>ID123456</identification>
                     <address>15, Melanie St.</address>
              </borrower>
              <borrowing_date>16/04/2003</borrowing_date>
              <returning_date>20/08/2003</returning_date>
       </book>
       <book>
              <title>Calculus</title>
              <publisher>
                     <name>Thomson Brooks/Cole</name>
                     <address>Belmont, CA,US</address>
              </publisher>
              <ISBN>053439339x</ISBN>
              <publishing_date>2003</publishing_date>
              <author>
                     <name>Stewart, James R.</name>
                     <affiliation>Belmont University</affiliation>
              <author>
              <borrower>
                     <name>Mary Fitzpatrick</name>
                     <identification>ID555666</identification>
                     <address>27/150 Cotham Rd.</address>
              </borrower>
              <borrowing_date>12/08/2004</borrowing_date>
              <returning_date>25/09/2004</returning_date>
       </book>
       --------
</libraryBooks>
```

Figure 14. Example of a XML document schema-graph

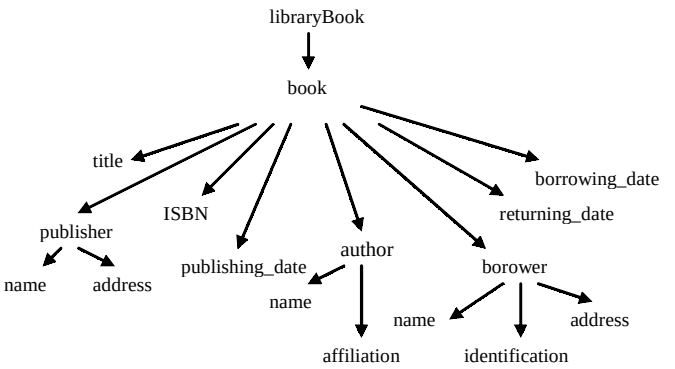


Figure 15. XML schema for libraryBooks.xml document example

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="libraryBooks">
 <xs:complexType>
    <xs:element name="book">
    <xs:complexType>
        <xs:element name="title" type=”xs:string”>
        <xs:element name="publisher">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="address" type="xs:string"/>
         </xs:sequence>
        </xs:complexType>
        <xs:element name="ISBN" type=”xs:string”>
        <xs:element name="publishing_date" type=”xs:date”>
       <xs:element name="author">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="affiliation" type="xs:string"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="borrower">
        <xs:complexType>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="identification" type="xs:string"/>
          <xs:element name="address" type="xs:string"/>
        </xs:complexType>
        </xs:element>
        <xs:element name="borrowing_date" type=”xs:date”>
        <xs:element name="returning_date" type=”xs:date”>
    </xs:complexType>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 16. Code example for Step 1 in verifying schema-level conflict S1

Step1:

```
let $root:=doc(“(“libraryBooks.xml””)  -> my document root
let $schema:=doc(“mySchema.xml”) -> my schema root
let $bfound:=false
document{
for $t in distinct-values($root//)  → for each descendant of root
       return {
              for $p in distinct values ($schema//)
              if $p/@name=$t
              let $bfound:=true}
       if not $bfound
       return
              <element>{$t}</element>
              <exist>false</exist>
}
```

Figure 17. Code example for Step2 in verifying schema-level conflict S1

```
Step2:
let d1:=doc("document1.xml")//
let $bfound:=false
document {
for $p in distinct-values(d1/element)
return {
       for $t in distinct-values($schema)
              if upper-case($p/@name)=uppercase($t)
              let $bfound:=true}
       if not $bfound
       return
              <element>{$t}</element>
              <exist>false</exist>
}
```

keeping some elements. Following, we show some code implementation using XQuery.

a. Verifying Schema Level: Name Conflict S1

Please refer to Figures 16 and 17 for code example.

Step 1 creates a new document with elements that are not found in the schema. We will name the new created document as "document1.xml". Step 2 applies the upper-case function on elements found at Step 1 and create another new document, containing all elements not found after upper-case search.

We have just obtained "document2.xml", containing those elements which exist in the primary document but not in the schema, and we did not find that a case-sensitive entering mistake would be the reason)

In Steps 3 and 4, the user decides whether the schema is to be extended with the new elements. If the decision is to extend the schema, user will establish which data type these elements should have.

b. Verifying Schema Level: Data Type Conflict S2

Step 1 identifies what data types are specified in the XML schema; at this step it will be decided what checks should be performed. Step 2 creates a new document where all elements and their data type are listed. Next, Step 3 goes through the document created at Step 2 and identifies all elements which do not have specified data type and lists them in another new document, for user to correct them.

In Step 1, we identified our schema the most significant data types used and decide what checks should be carried out against them:

- For *numeric type*, we take each elements declared as numeric in the XML schema, extract its value from the document and test to determine if it is numeric (do not appear in our case-study document & schema);
- Same procedure for *other types* (string, decimal, etc.)
- For *date type*, we can perform the followings checks: if day is between 1 and 31, if month is between 1 and 12 and,

Figure 18. Code example for Step 2 in verifying schema-level conflict S2

```
let $s:=doc("myschema.xml//")
document
{
       for $t in distinct-values ($s/xs:element)
       if $t/@type="xs:string"
       return
              <string_element>{$t}</string_element>
       if $t/@type="xs:integer"
       return
              <int_element>{$t}</int_element>
       if $t/@type="xs:decimal"
       return
              <decimal_element>{$t}</dec_element>
       if $t/@type="xs:date"
       return
              <date_element>{$t}</date_element>
}
```

Figure 19. Code example for Step 2 in verifying schema-level conflict S2-continued

```
<elementtype>
       <string_element>title</string_element>
       <string_element>name</string_element>
       <string_element>address</string_element>
       <string_element>ISBN</string_element>
       <date_element>publishing_date</date_element>
       <string_element>affiliation</string_element>
       <string_element>identification</string_element>
       <date_element>borrowing_date</date_element>
       <date_element>returning_date</date_element>
</elementtype>
```

Figure 20. Code example for Step 3 in verifying schema-level conflict S2

```
let $d:=doc("libraryBooks.xml")//
let $p:=doc("elementtype.xml")//
document
{
for $t in distinct-values($p/string_element)
       for $n in distinct-values($d/$t)
              if not ($n instance of xs:string)
              return
              <wrong_string>{$n}</wrong_string>
for $t in distinct-values($p/int_element)
       for $n in distinct-values($d/$t)
              if not ($n instance of xs:integer)
              return
              <wrong_int>{$n}</wrong_int>
for $t in distinct-values($p/dec_element)
       for $n in distinct-values($d/$t)
              if not ($n instance of xs:decimal)
              return
              <wrong_dec>{$n}</wrong_dec>
}
```

Figure 21. Code example for Step 1 in verifying schema-level conflict S3

```
document
{
for $t in distinct-values ($s/xs:choice)
       let $np:=$t/../@name
       let $n:=$t/@name
       let $counta:=0
       for $p in distinct-values($d/$np/$n)
       let $counta:=$counta+1
       if $counta<>0
              let $countb:=0
              for $p in distinct-values($t/..//)
                     if $p/@name<>$n
                            for $q in distinct-values($d/$np/$p)
                            let $countb:=$countb+1
                            if $countb<>0
                            return
                            <many_choices>{$q}</many_choices>
}
```

Figure 22. Code example for Step 2 in verifying schema-level conflict S3

```
document
{
for $t in distinct-values($s/xs:element)
       if not(empty($t/@minOccurs))
              let $m:=$t/@minOccurs
              let $n:=$t/@name
              let $countMin:=0
              for $p in distinct-values($d/$n)
                     let $countMin:=$countMin+1
              if $countMin<$m
              return
                     <too_few_occur>{$n}</too_few_occur>
       if not(empty($t/@maxOccurs))
                     let $m:=$t/@maxOccurs
              let $n:=$t/@name
              let $countMax:=0
              for $p in distinct-values($d/$n)
                     let $countMax:=$countMax+1
              if $countMax>$m
              return
                     <too_many_occur>{$n}</too_many_occur>
}
```

depending of what the document refers to, we can check validity of the year (e.g., if it is about issued invoices in a company or contains dates of birth, date cannot be greater than current date);

Step 2 creates "elementType.xml", which is a new document, with all elements and their data types, from the schema. (See Figure 18 for code example.) Starting from our document in the case study, libraryBooks.xml, the new "elementType.xml" is exemplified in Figure 19. At Step 3, we verify each element in our document against data types from previously created document during Step 2 (see Figure 20 for code details).

Figure 23. Code example for Step 3 in verifying schema-level conflict S3

```
document {
for $a in distinct-values($schema//)
       if not (empty($a//))
       let $p:=$a/@name
       for $m in distinct-values($a//@name)
       let $c:=$a//@name
       return
              <family>
                     <parent>{$a}</parent>
                     <child>{$c}</child>
              </family>
}
```

Figure 24. Code example for Step 3 in verifying schema-level conflict S3 (continued)

```
document {
let $d:=doc("document.xml")
       let $r:=doc("relationships.xml")
       for $c in ($r/family/child)
              let $p:=$c/../parent
              let $bfound:=false
              for $t in distinct-values($d//)
              if $t=$c and not($c/../=$p)
              return
                     <wrong_parent>{$c}</wrong_parent>
}
```

c. Verifying Schema Level: Schema Restrictions Conflict S3

Step 1 shows how to check one of possible order indicators, this is *choice*, returning a document with wrong chosen elements (see Figure 21 for code example). Step 2 shows how to verify the number of occurrences for each element, if specified in the schema (Figure 22). Finally, Step 3 shows how to verify one possible relationship, and this is *parent-child* (see Figure 23 and 24).

Once all the above rules have been applied to the data, the following step — data summarisation — is then performed.

Data Summarisation: Creating Dimensions

By analysing possible queries to the data warehouse for decision making, for example, we need to summarise borrowing details at a month level, to study how this activity is related to different periods of the year. Considering this, we will construct a "time" dimension, where one of the attributes will be "month" (in the same way, we can obtain year, semester, quarter level, etc.). At the same time, a summarisation by authors, publishers or by book titles could be necessary for further studies. For easy referencing, we bring again the schema-graph in attention.

Figure 25. Creating time dimension example

```
let $b:=0
document{
for $t in distinct-values
(doc("libraryBooks.xml")//borrowing_date)
       let $b:=$b+1
       return
         <borrowtime>
              <timeKey>{$b}</timekey>
              <borrowdate>{$t}</borrowdate>
              <month>{get-month-from-date($t)}</month>
         </borrowtime>
}
```

Figure 26. Creating authors dimension example

```
let c:=doc("libraryBooks.xml")/libraryBooks
let $b:=0
document{
       for $t in distinct-values ($c/author/name)
       let $b:=$b+1
       return
        <author>
              <authorKey>{$b}</authorKey>
              <name>{$t}</name>
              <affiliation>{$t/../affiliation}</affiliation>
       </author>
}
```

Figure 27. Creating title dimension example

```
let $b:=0
let $p:= doc("libraryBooks.xml")/libraryBooks
document {
       for $t in dictinct-values($p/title)
       let $b:=$b+1
       return
         <title>
              <titleKey>{$t/../ISBN}</titleKey>
              <description>{$t}</description>
              <publishing_date>{$t/../publishing_date}</publishing_dates>
         </title>
}
```

Creating Time Dimension

Because we need "month" level of summarization, we should link each particular borrowing date in our data with its month value, so we will extract only distinct values of "borrowing_date" from the initial document and will apply a "get-month-from-date" function, in order to find which month the date corresponds to. We name this new document "timeDim.xml". Each new document will be created using *"document"* instruction from XQuery. (See Figure 25.)

Creating Authors Dimension

To do this, we will extract only distinct (unique) authors, considering their names, from our initial document and we will name it "authorDim.xml". (See Figure 26.)

Figure 28. Creating intermediate document example

```
document {
       for $t in doc("libraryBooks.xml")//
       return
         <borrow>
              <ISBN>{$t/ISBN}</ISBN>
              <author>{$t/author/name}</author>
              <borrower>{$t/borrower/name}</borrower>
                     <borrowing_date>{$t/borrowing_date}</borrowing_date>
       <returning_date>{$t/returning_date}</returning_date>
         </borrow>}
```

Figure 29. Linking documents and creating data warehouse example

```
        let $p:=doc("authorDim.xml")
        let $c:=doc("titleDim.xml")
        let $t:=doc("timeDim.xml")
        document {
               for $a in doc("libraryTemp.xml")/borrow
               return
                      <authorKey>{for $b in $p/author
                             where $b/name=$a/author
                             return $b/authorKey }
                      </authorKey>
                      <titleKey> {for $b in $c/title
                               where $b/description=$a/title
                               return $b/ISBN}
                      </titleKey>
                      <borrowDateKey> {for $b in $t/borrowtime
                      where $t/borrowdate= $a/borrowing_date
                             return $t/timekey} </borrowDateKey>
                      <returning_date>{$a/returning_date}
                      </returning_date>
                      <borrower>{$a/borrower}</borrower>
  }
```

Creating Titles Dimension

The same as for "authors" dimension, we will create a "titles" dimension which contains only distinct book titles extracted from our initial document and we will consider "ISBN" element as a key, because it can uniquely identify a book. We will name it "titleDim.xml". (See Figure 27.)

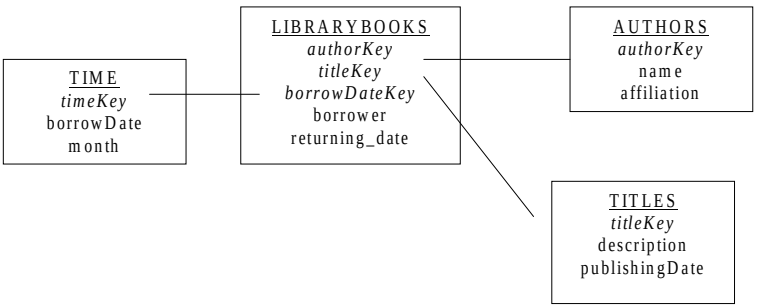
Creating Intermediate XML Documents

As our possible queries refer to *month, author and titles* as possible levels of summarisation, we need to extract from the cleaned "libraryBooks" document all records having borrowing_date, author and ISBN and, among them, borrower and returning_date as main tools for analysing this library activity. It is all we need for now and we will name it "LibraryTemp.xml". (See Figure 28.)

Linking Existing (Newly Created) Documents: Creating a Data Warehouse

It is straightforward that we now have to link "borrowing_date" from our inter-

Figure 30. Star-schema of libraryBooks XML data warehouse



mediate document ("libraryTemp.xml") with "time" dimension, "author" with "authors" dimensions and "ISBN" with "titles" dimension. We create now the final document, which will be *the fact of data warehouse* and will contain: keys for links with all three dimensions, "borrower" and "returning_date" as raw specific activity data. (See Figure 29.)

Following all the steps from 1 to 4, we have just obtained the following star-schema for the data warehouse to store "libraryBooks" XML documents, visually described in Figure 30.

CONCLUSIONS AND FUTURE WORK

This paper establishes a framework for building an XML data warehouse, taking into consideration the quality of data and practical procedures that can be followed by general data warehouse users. Our paper has presented a systematic approach on how to build a data warehouse for XML documents. We have presented this methodology in a generic way, so that the rules and techniques can be applied to a wide-range of XML data warehouse models and implementations.

Our proposed methodology has two major parts: first, we discuss data cleaning techniques, applied to our row XML documents. Because the sources of these documents are widely spread, the possibility to have dirty data, inconsistencies and errors is quite high. Automation of the cleaning activity is a very important issue, as it can save a lot of precious time and can substantially improve the quality of data obtained.

Secondly, techniques for data summarisation, creating dimensions and the fact documents are discussed and exemplified, building at the end a star-schema data warehouse for XML documents. By employing the appropriate technique for data summarisation when developing dimensions, the volume of data can be optimised. Therefore this may subsequently impact on the optimisation of access to the information in the data warehouse.

After covering all steps involved, we obtain not only an efficient processing of creating a data warehouse, but also high quality data and a low level of redundancy.

Examples in the paper are written using XQuery which represent a practical technique to implement the rules. It is a strong accomplishment that the steps of work and the examples are presented in a very clear and easy manner, so that people who do not have a vast knowledge of XQuery can iterate them, with adequate

and proper modifications, in order to obtain a data warehouse corresponding to their necessities. Queries will be easy to generate and process, as long as dimensions reflect all level of summarisation suitable for each specific case. For example, a query "Give all purchase orders which were billed to customers from UK" will only look in "customer" dimension and take keys for customers who have country="UK" and, for each of these keys will take order date, price, quantity and income from the fact document.

Because at this stage there are no automatic methods for covering data cleaning, our aim is to study this concept and try to identify some techniques that can be applied to automate the XML documents processing. Another interesting aspect to study is the integration of different XML schemas when data to be included in the data warehouse come from documents with different structures but with equivalent data content. Data mining for a XML data warehouse is another interesting field to research, as we do not want just to keep our data in a well-designed data warehouse, but to learn some interesting facts from it.

REFERENCES

Bouzeghoub, M., Fabret, F., & Matulovic, M. (1997). *Modeling data warehouse refreshment process as a workflow application, Workflow Handbook*. Chichester, UK: John Wiley & Sons.

Cooley, R., Mobasher, B., & Srivastava, J. (1999). Data preparation for mining World Wide Web browsing patterns. *Knowledge and Information Systems*, 1(1), 5-32.

Deutch, A., Fernandez, M., Florescu, D., Levy, A., & Suciu, D. (1999). A query language for XML. *Computer Networks*, 31, 1155-1169.

Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). Knowledge discovery and data mining toward a unifying framework. *Proceedings of the Second Int. Conference on Knowledge Discovery and Data Mining*, (pp. 82-88).

Fernandez, M., Simeon, J., & Wadler, P. (1999). *XML query languages: Experiences and exemplars, raft manuscript*. Online: http://homepages.inf.ed.ac.uk/wadler/topics/xml.html

Galhardas, H., Florescu, D., Shasha, D., & Simon, E. (2000). An extensible framework for data cleaning. *Proceedings of the International Conference on Data Engineering (ICDE)*, San Diego, CA.

Goffarelli, M., Maio, D., & Rizzi, S. (1998). Conceptual design of data warehouses from E/R schemes. *Proceedings of Hawaii Int. Conf. on System Sciences*, Kona, Hawaii (vol. VII, pp. 334-343).

Guyon, I., Matic, N., & Vapnik, V. (1996). Discovering informative patterns and data cleaning. *Advances in Knowledge Discovery and Data Mining* (pp. 181-203). Menlo Park, CA: AAAI Press/MIT Press.

Hernandez, M.A., & Stolfo, S.J. (1998). Real-world data is dirty: Data cleansing and the merge/purge problem. *Journal of Data Mining and Knowledge Discovery*, 2(1), 9-37.

http://www.w3.org/TR/xpath-functions

Kim, J., & Park, S. (2005). Periodic streaming data reduction using flexible adjustments of time section size. *International Journal of Data Warehousing and Mining*, 1(1), 37-56.

Ram, S., & Park, J. (2004, February). Semantic conflict resolution ontology (SCROL): An ontology for detecting and

resolving data and schema level semantic conflicts. *IEEE Transaction on Knowledge and Data Engineering*, (pp. 189-202).

Robie, J. (2004). XQuery, A Guided Tour. Online: http://www.datadirect.com/news/whatsnew/xquerybook/index.ssp

Roddick, J.F., Mohania, M.K., & Madria, S.K. (1999). Methods and interpretation of database summarisation. *Database and Expert Systems Application, Florence, Italy, Lecture Notes in Computer Science*, (vol. 1677, pp. 604-615). Springer-Verlag.

Rusu, L.R., Rahayu W., & Taniar D. (2004). On building XML data warehouses. *Proceedings of the Fifth International Conference on Intelligent Data Engineering and Automated Learning (IDEAL' 04)*, Lecture Notes in Computer Science (vol. 3177, p. 293). Springer-Verlag.

Song, I.Y., Rowen, W., Medsker, C., & Ewen, E. (2001). An analysis of many-to-many relationships between fact and dimension tables in dimensional modelling. *Proceedings DMDW*, Interlaken, Switzerland, (pp. 6.1-6.13).

Vrdoljak, B., Banek, M., & Rizzi, S. (2003). Designing Web warehouses from XML schema. *Data Warehousing and Knowledge Discovery, 5th International Conference DaWak 2003*, Prague, Czech Republic, September 3-5.

World Wide Web Consortium (W3C). XML Schema Part 0: Primer. Online: http://www.w3.org/TR/xmlschema-0/#emptyContent

Zhang, J., Ling, T.W., Bruckner, R.M., & Tjoa, A.M. (2003). Building XML data warehouse based on frequent patterns in user queries. *Data Warehousing and Knowledge Discovery, 5th International Conference DaWak 2003*, Prague, Czech Republic.

Widom, J. (1999). Data management for XML: Research directions. *IEEE Data Engineering Bulletin*, 22(3), 44-52.

Laura Irina Rusu received her BSc (computer science) in 1996 and her MEc in 1997, both from The Academy of Economic Studies, Bucharest, Romania. Currently, she is studying for a Master by Research degree at La Trobe University, Australia, where her areas of interest are the XML data warehousing and XML data mining. Rusu is working as an associate lecturer at La Trobe University.

Johanna Wenny Rahayu received a PhD in computer science from La Trobe University, Australia (2000). Her thesis has been awarded the 2001 Computer Science Association Australia Best PhD Thesis Award. Dr. Rahayu is currently an associate professor at La Trobe University. She has published two books and numerous research articles. Her research interests include Semantic Web and ontology, medical information systems, bioinformatics databases, and XML technologies.

David Taniar received his PhD in computer science from Victoria University, Australia (1997) and is currently working at Monash University. His research interests include databases and data mining, mobile information systems, Web information systems, and high performance grid computing. He has published more than 30 journal papers and 150 conference papers in these fields. He is an editor-in-chief of a number of international journals including Data Warehousing and Mining, Business Intelligence and Data Mining, Mobile Information Systems, Mobile Multimedia, Web Information Systems, *and* Web and Grid Services. *He is fellow of the Institute of Management Information Systems (FIMIS).*