Towards Cleaning XML Databases: Experience and Performance Evaluation*

Wai Lup Low†
DSO National Laboratories
Singapore
lwailup@dso.org.sg

Wee Hyong Tok, Mong Li Lee, Tok Wang Ling
School of Computing
National University of Singapore
{tokwh,leeml,lingtw}@comp.nus.edu.sg

January 13, 2003

Abstract

With the increasing popularity of data-centric XML, data warehousing and mining applications are being developed for the rapidly burgeoning XML data repositories. Data quality will no doubt be a critical factor for the success of such applications. Data cleaning, which refers to the processes used to improve data quality, has been well researched in the context of traditional databases. In this work, we present a novel attempt to clean XML databases. We discuss the new challenges that arise in XML data cleaning and propose solutions to overcome these problems. Our experimental dataset is the DBLP database, a popular online XML bibliography database used by many researchers. The DBLP database is a large collection of small XML documents. Our study shows the benefits of performance gains, flexibility and maintainability that can be achieved by leveraging on the use of a relational database management system to clean XML data. We also investigate the conventional practice of using XML parsers when the structure of the XML data is simple and static, and compare their performance against string matching approaches.

*An earlier version of this work has been presented in the poster session of the 18th International Conference on Data Engineering, February 26-March 1, 2002 San Jose, California (ICDE 2002).

†This work was done while the author was on a research scholarship from the National University of Singapore.

1 Introduction

The eXtensible Markup Language (XML) [BPSMM00] is now the *lingua franca* for data exchange and information interchange on the Internet [Wid99]. XML is a simplified descendant of the Standard Generalized Markup Language (SGML), which is a more complex markup language that has been around for more than a decade. This W3C-backed development has received overwhelming response from the industry and academia alike. Groups from the health care, music, publishing, education, scientific and finance industries, just to name a few, are working on standardizing XML formats for information exchange. For the academia, work on XML query languages, storage, representation, semantics among other topics have been the focus of many research papers. It is a forgone conclusion that XML is the de-facto language of the Internet tomorrow.

The basic idea behind XML is very simple. XML embeds information within tags to give meaning to the associated data fragments. Although verbose, XML has the advantages of being simple, interoperable and extensible, which make it popular for information interchange over the Internet. Currently, there are two main uses of XML : Presentation-Oriented Publishing (POP) and Message-Oriented Middleware (MOM). There are on-going projects which aim to make XML a *repository technology*. Several of these projects (e.g. [MAG+97] [The01]) even host XML data in its native form, which means that the underlying structure and form of the XML document is preserved or mapped to the physical storage. With the numbers and sizes of such repositories set to grow quickly in the coming years, researchers are already looking into ways to warehouse and obtain knowledge from such databases [Xyl01]. From the teething (and very expensive) problems dirty data has caused in the past, it is not difficult to foresee that the data quality of these XML databases will be a crucial factor for the success of such activities.

Data cleaning refers to a series of processes used to improve data quality and it has been well-researched in the context of relational data [HGS99] [RH00] [LLL01]. There are many causes for dirty data: misuse of abbreviations, data entry mistakes, duplicate records, missing fields, etc. Such data has proven to be very costly to organizations. In the context of data mining and warehousing, such data results in inaccurate and inconsistent information, commonly known as the "*garbage in, garbage out*" principle. These same problems will plague the fast growing XML data repositories which are beginning to be warehoused, mined and used in decision support systems.

Many studies have shown that *domain knowledge* is required to identify and eliminate data anomalies effectively. [LLL01] propose a knowledge-based framework and demonstrate its effectiveness in cleaning relational databases. The framework captures domain knowledge in the form of rules, which are subsequently used in a reasoning process to identify data anomalies. In this work, we present a novel attempt to clean XML databases. The semistructured XML data presents new problems and challenges to data cleaning, and we investigate methods and techniques to solve them efficiently and effectively. In particular, we report our experience and results of detecting the errors and anomalies in the DBLP database, which is a large database of XML-style bibliographical records of computer science publications. The contributions of this paper are:

1. We present a novel approach to improve the quality of XML data by using a knowledge-based system to clean the data.
2. Evaluate the necessity of using of XML parsers for simple XML structures and compare their performance with parsing via regular expression matching.
3. Examine different ways to map DTD specifications to fact templates in knowledge bases.
4. Investigate the performance of various ways to index the data in XML documents, and the feasibility of replicating the data in the index for faster processing.

The rest of the paper is as follows. Section 2 surveys related research in this area. Section 3 discusses our knowledge-based framework for cleaning relational databases. In Section 4, we discuss the problems encountered when the knowledge-based framework is applied to XML databases and explore possible solutions. We detail our experiments and performance studies in Section 5. Finally, we conclude in Section 6.

2 Related Works

Knowledge discovery and data mining (KDD) has been the focus of many researchers and industries for the past few years. Efficient data mining algorithms have been developed and powerful tools have been built to discover useful and interesting patterns in data. Related to these developments is data warehousing, which builds massive collections of key pieces of information to support decision making processes. These works have been done in the context of traditional structured data formats, including relational tables and object databases. The growing popularity of XML on the World Wide Web (WWW) has initiated KDD activities that are looking into using semistructured data as the underlying data format.

Web mining, which refers to mining information from the WWW, is a budding area of research. Recognizing that the WWW is a huge repository of semistructured information, data mining methods are being developed for unstructured and semistructured data sources (such as HTML documents). Such methods are often more complex than their traditional counterparts for structured sources as the lack of structure and source heterogeneity give rise to semantic ambiguity. XML, whose tags give meaning to the associated data fragments, simplifies KDD processes for semistructured data. [BMBA00] presents a survey which describes potential synergy between data mining and XML. There are architectures specifically designed for semistructured data mining [SCH$^+$98]. Besides the data instances, these architectures also need structural information of the data to achieve a semantically richer model.

In [CFP00], XML has been identified as a repository technology. [CFP00] reasons that XML offers applications a portable interface for data access and management, and hence is a convenient format for repositories. However, it is inefficient to store XML as text, and it is difficult to efficiently map its structure to other existing data models (e.g. relational).

To the best of our knowledge, this is the first quantitative work in cleaning XML databases. Data cleaning, however, has been studied extensively for many years for the traditional data models. There are two main approaches : domain independent data cleaning, and data cleaning with domain knowledge. Domain independent data cleaning [ME97] generally treats database records as strings and computes record similarity using string comparison algorithms. Although simpler, faster and usually cheaper to implement, domain independent data cleaning processes often lack the power to identify and eliminate complex data anomalies. For the approaches leveraging on domain knowledge, data cleaning is considered a complex inferential process. Domain expertise is required for identifying data anomalies and the ways to eliminate them [HS98] [LLL01] [WM89]. Since cleaning with domain knowledge is more effective, we will employ this approach to clean XML data.

3 A Knowledge-Based Framework for Data Cleaning

In this section, we briefly review our knowledge-based framework for intelligent data cleaning. This framework takes a systematic approach and provides a complete strategy for standardizing, anomaly detection and removal, and duplicate elimination in dirty databases. We omit the details here due to space constraints. Interested readers can refer to [LLL01].

Figure 1 shows our proposed framework. It consists of three stages :

1. Pre-processing Stage. Data records are first conditioned and scrubbed of any anomalies we can detect and correct at this stage. Examples of conditioning processes that we can do here include data type checks, abbreviations unification and format standardization. The output of this stage will be a set of conditioned records which will be input to the processing stage.

2. Processing Stage.

   The conditioned records are next fed into an expert system engine together with a set of rules. Each rule will fall into one of the following categories :

   (a) Duplicate Identification Rules. These rules specify the conditions and criteria for two records to be classified as duplicates. Functions to compare text similarity, procedures to perform string manipulation, and complex logic for determining record equivalence can be coded into or referenced by the rules.

   (b) Merge/Purge Rules. These rules specify how duplicate records are to be handled. A simple rule might specify that only the record with the least

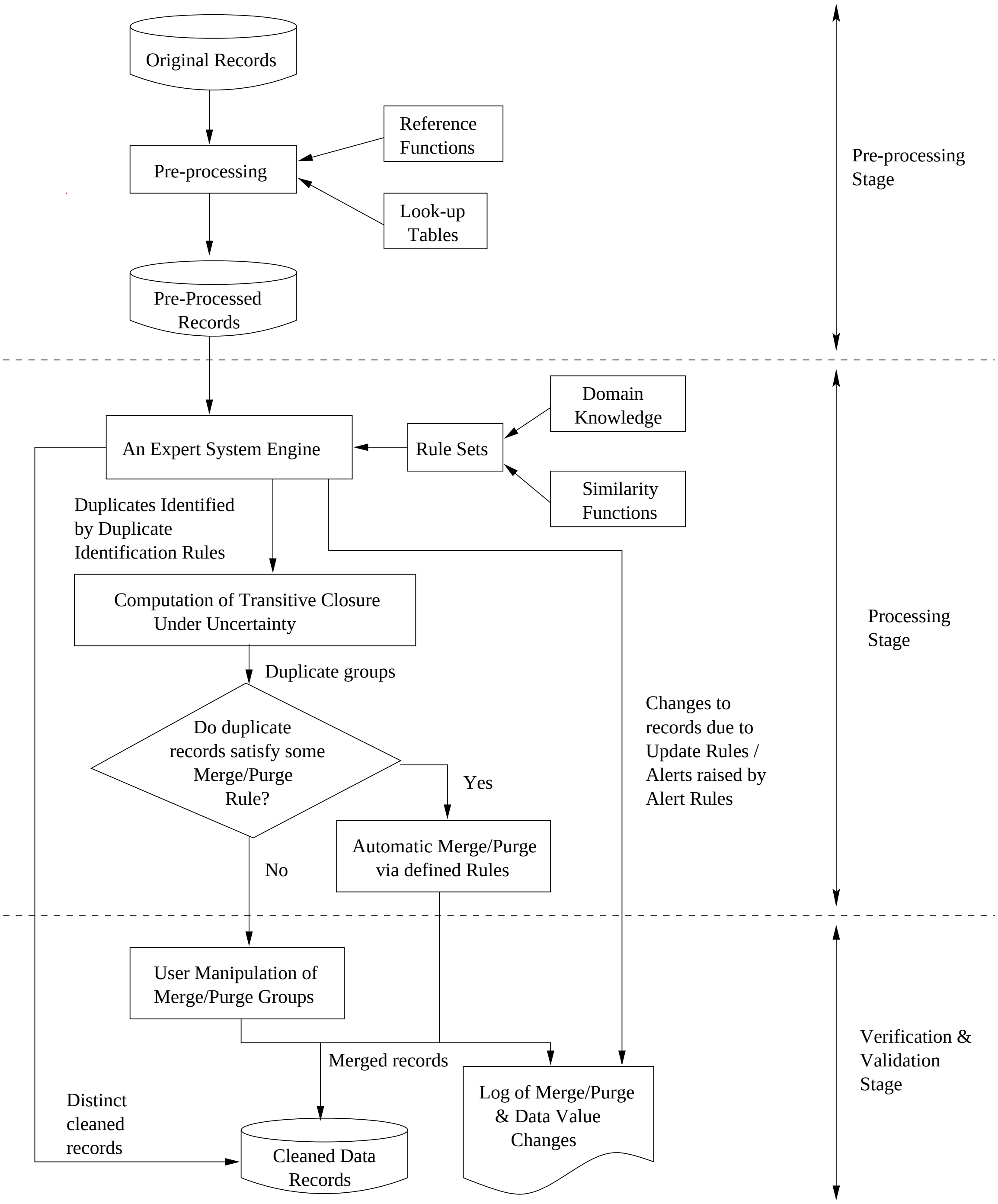


Figure 1: A Knowledge-Based Framework for Data Cleaning

number of empty fields is to be kept in a group of duplicate records, and the rest be deleted. Much more complex actions can be specified in a similar fashion.

(c) Update Rules. These rules specify the way data is to be updated when specified conditions are satisfied.

(d) Alert Rules. The user might want an alert to be raised when certain events occur. User-defined actions can be specified in the alert rules. Alert rules can also be used for the checking and warning of constraint violations.

All the rules will be fired in an opportunistic manner when the pre-processed records are fed into the expert system engine. After the duplicate record groups are identified, we compute the transitive closure under uncertainty [LLL01] for these groups to increase the recall. The merge/purge rules will then act on the duplicate record groups that satisfy their conditions.

To avoid pairwise comparison for each and every record in the database (which may be infeasible in large databases), a sliding window scan is employed as follows:

(a) Create Keys. A key is computed for each record in the database by extracting relevant fields or portions of fields which form an important discriminating attribute. The selection of a "good" key requires domain knowledge of the data concerned.

(b) Sort Data. Sort the records in the database using the key computed in Step 1.

(c) Sliding Window. Move a fixed size window through the sequential list of records limiting the comparisons for matching records to those records in the window. If the size of the window is w records, then every new record entering the window is compared with the previous $w - 1$ records to find "matching" records. The first record in the window then slides out of the window as shown in Figure 2.

Hence, the rules will only act on one window of records in the expert system engine at any point in time. Compared to the naive way of comparing every record with every other record, which is an $O(N^2)$ operation, the sliding window scan requires only wN comparisons, where w is the window size.

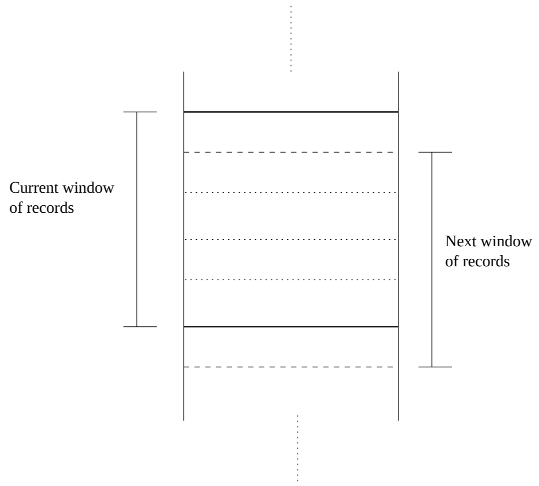


Figure 2: Sliding Window Scan

3. Human Verification and Validation Stage. In this stage, human intervention will be required to manipulate the duplicate record groups for which merge/purge rules are not defined. Upon human inspection, false positives can be taken out of these groups and appropriate merge/purge procedures can be carried out.

4 Cleaning XML Data

In this section, we will describe what is involved in cleaning XML databases. To make our discussion and work more concrete, we will use the DBLP database as an example. We first introduce the DBLP database and highlight its characteristics. Following that, we discuss the new challenges faced when we attempt to clean the database using the knowledge-based framework described in the previous section. We will also present the solutions to overcome these problems.

4.1 The DBLP database

The Digital Bibliography & Library Project (DBLP) database is a large collection of bibliographical records of computer science publications. It provides bibliographic information on major computer science journals and proceedings and has been described as the *portal to computer science publications*. The DBLP database stores its records in XML-style formats and an example is shown in Figure 3. As of March

```
<inproceedings key="BertinoCS98">
<author>Elisa Bertino</author>
<author>Barbara Catania</author>
<author>Boris Shidlovsky</author>
<title>Towards Optimal Indexing for Segment Databases.</title>
<pages>39-53</pages>
<year>1998</year>
<booktitle>EDBT</booktitle>
<url>db/conf/edbt/edbt98.html#BertinoCS98</url>
</inproceedings>
```

Figure 3: A sample bibliographical record in the DBLP database.

2001, the database contains more than 200,000 such records with each record being stored in a different file. The files are organized hierarchically in the file system, with the records for the same publication being stored under the same directory. The database takes up about 800 MB of disk space when stored as text files.

The DBLP database indisputably provides high quality bibliographic information. However, collecting and maintaining such a massive amount of semistructured data from diverse sources is no simple job and errors are bound to creep in. The name variation problem has been highlighted as the most time consuming issue in the maintenance of the DBLP and is described as a "never ending story".

4.2 Issues and Solutions

Cleaning XML databases poses new challenges and problems which do not arise in cleaning relational databases. We highlight these problems and our proposed solutions in the cleaning of the DBLP.

- **Issue 1: Lack of well-defined structures.** Unlike relational data which has a well-defined tabular structure, XML databases can have much more complex structures [DFS99, STZ^{+}99]. In the knowledge-based framework, the expert system requires the definition of fact templates, which serve the same purpose as schemas in databases. This is straightforward when cleaning relational data. For each relational table, we define a fact template. Each column in the table is mapped to a slot-value of the corresponding fact template. Unfortunately, XML databases can have very irregular structures that makes this mapping much more difficult.

Data-centric XML repositories (such as the DBLP database) often come with a schema. While there has been some work done on mapping an XML schema[1] to object-based models and relational schemas [Bou01a, Bou01b], mapping an XML schema to expert system fact templates has never been dealt with before.

Algorithm 1 maps the DBLP DTD specification to a collection of corresponding fact templates that can be used in the knowledge base. Although the algorithm is developed with the DBLP DTD specifically in mind, it is generic enough to accommodate other DTDs with minor or no modifications.

Algorithm 1: Algorithm to map DTD/XML Schema to a collection of fact templates.

```
   begin
1      S ⟵ schema of the XML database
2      for each element E in S do
3          create a fact template, F
4          create parent pointer in F
5          for each attribute e of E do
6              if e is single-valued then
7                  create a slot e in F
8              else
9                  create a multi-slot e in F
10         for each allowed child element c of E do
11             if c occurs at most once then
12                 create a slot c in F
13             else
14                 create a multi-slot c in F
   end
```

In Algorithm 1, each element defined in the schema of the XML database (i.e. S) is mapped to a fact template. To facilitate the upward navigation during the reasoning process, a parent reference is created for the template (Line 4). This reference will contain the pointer to the fact which holds the parent of the element. Each single-valued attribute (e.g. `ID`) in the element is then mapped to a slot (Line 7). Each multi-valued attribute(e.g. `IDREFS`) is mapped to a multi-slot (which can hold multiple values). The child elements are handled

[1] We use XML schema (note the lower case 's') to refer to schemas of XML databases in general. We use "XML Schema" to refer to the W3C Recommendation at [Fal01].

similarly (Lines 10-15). The slot values of the child elements do not store the actual child nodes. Instead, they store the references to the facts storing the child nodes.

For simplicity, the algorithm focuses on the key building blocks of XML, which are elements, attributes and references. This is sufficient for the simple DTD used by the DBLP. We will now illustrate the use of the algorithm with a simplified version of the DBLP DTD. The simplified DTD is shown in Figure 4 and the corresponding mapped fact templates are shown in Figure 5. The `dblp` and `article` elements in the DTD are mapped to fact templates. Each attribute in the article element is mapped to a slot with the same name. The entity declaration in Figure 4 is expanded and each allowed field is mapped to a multislot in the article fact template. This mapping is done directly from the DTD and there is room for optimization. Firstly, we know (as domain knowledge) that the only field in the article template that needs to be a multislot is the author. Secondly, depending on the rules used in the reasoning process, not all information need to be fed into the knowledge base and only the required information need to be mapped and loaded. In such situations (we will see one in Section 5), optimized templates may be handcrafted for best performance.

The Java Expert System Shell (JESS) language [FH97] is used in the reasoning engine in our framework. Note that additional constraints such as nullable, default values, and allowed values can also be set and checked as necessary. Some of the constraints are directly supported by the deftemplate construct in JESS, while others will have the be enforced by the domain rules. Although we do not consider XML databases without a schema in this work, but we note that in such cases, the schema may be hand-built by domain experts or mined from the data instance [Min00].

Two issues to highlight here are that of *data typing* and *ordering of data*:

1. **Data type**. Due to the lack of information from the DTD, all slot values in the fact templates are assumed to be of `String` type. In some schema languages (e.g. XML Schema), typing information can be obtained from the schema definition. Such constraints can also be considered to be part of the domain expertise used in the cleaning of the data, and coded as an alert rule. Hence, warnings will be issued during the cleaning process if the data is of the wrong type. This method can be used in cases where the schema language does not support type constraints (e.g. DTD).

```
<!ELEMENT dblp (article)*>
<!ENTITY % field "author|title|pages|year|journal|volume">
<!ELEMENT article (%field;)*>
<!ATTLIST article
      key CDATA #REQUIRED
      reviewid CDATA #IMPLIED
      rating CDATA #IMPLIED
>
```

Figure 4: A simplified DBLP DTD.

```
(deftemplate dblp "template for element dblp"
   (multislot       article)
)

(deftemplate article "template for article element"
   (multislot       author)
   (multislot       title)
   (multislot       pages)
   (multislot       year)
   (multislot       journal)
   (multislot       volume)
   (slot key)
   (slot reviewid)
   (slot rating)
)
```

Figure 5: Fact templates for the simplified DBLP DTD in Figure 4.

2. **Order**. Another issue is that of sibling order in the data. Although order is significant in XML, it is generally not a concern in the DBLP database (except for the ordering of authors in a publication). For instance, whether the `year` element comes before or after the `pages` element is not important. This can be seen from the design of the DBLP DTD. We note that in cases where the order is significant, the constraint on the ordering should be imposed in the schema definition (e.g. DTD, XML Schema, etc). Although the constraint can be expressed as rules in the reasoning engine, it will be complex and likely to be less efficient.

- **Issue 2: XML parsing versus Text parsing**. Since the data stored by the

DBLP database is in XML, the obvious way to parse the data will be to rely on an existing XML parser API to process the files. The most common parser APIs currently are the Simple API for XML (SAX) [Meg01] and the Document Object Model (DOM) [W3C01].

However, given the simplicity of the DBLP DTD, we have an an alternative way of treating the small XML files as simple text files. The information extraction process can then be carried out using regular expressions to match the mark-up tags. As a result, this may mean faster processing speed since the overhead of XML parser interfaces is avoided. However, such methods are likely to be less maintainable and scalable with the introduction of new tags and changes to tags. This is the reason why the XML parser APIs were developed. The use of an XML parser allows easy maintenance and updates. We compare the performance of parsing simple XML structures using an XML parser API with text parsing methods in the next section.

- **Issue 3: Sorting of XML data**. Since the knowledge-based framework makes use of a sliding window of sorted data to detect anomalies within the neighbourhood, the ability to efficiently feed the ordered data into the knowledge base is very important. In relational databases, this process can be as simple as issuing an *order by* clause as part of the SQL command. However, this becomes problematic when XML data is stored as native text files. For example, the DBLP database is made up of more than 200000 small files stored in the file system.

  A possible solution is to build a customized index based on the defined key (e.g. a customized B+ tree index). However, such a solution is unlikely to be flexible enough. Instead, we propose to leverage the power and flexibility of existing relational database management systems (RDBMS) as follows. We create the sort keys (derived from domain knowledge) and store them in relational tables, backed by a RDBMS engine. As a result, the data in the RDBMS indexes the XML data stored in text files. We will describe in detail this method in Section 5. The advantages of this approach include flexibility and efficiency when new key entries are added and new sort keys are defined. Scalability and reliability issues have also been addressed in the decades of research done on RDBMS. However, there are several drawbacks, including the maintenance costs and performance overheads of a RDBMS.

- **Issue 4: Should we replicate data in the index for efficiency**? After building the index, it is only natural to ask if it is more efficient to store the data to be cleaned in the index. On the one hand, this simplifies many teething problems such as sorting/ordering of input for the sliding window scan. In fact, if only a very small portion of the XML data is involved in the data cleaning logic, it may be much more efficient to store all data required into the index and perform the data cleaning process using the data from the index only. This reduces cleaning XML databases to cleaning relational data as described in [LLL01]. Undoubtedly, this gain in efficiency is due to avoiding the expensive file operations and disk accesses required when the data is not in the index.

However, storing the data in the index implies data replication. Additional storage becomes a consideration if the XML database is large. Updates to the underlying XML databases will also have to be propagated to the replicated information in the index, failing which will result in inconsistency. We compare the performance gains and costs of this approach in Section 5.

5 Performance Study.

In this section, we present the results of the experimental studies done to investigate the various issues raised in the previous section. The experiments are performed on a Pentium III 886MHz machine with 256 MB RAM running Mandrake Linux 8.0. The expert system shell used is JESS [FH97].

5.1 Experiment 1 : XML parsers a must for XML data?

Parsing a file refers to the process of breaking up the contents of the file into meaningful constituent parts. If the data has a fixed imposed structure, then it is easy to write an application specific parser to break up the data into the required parts as defined by the structure. The most common technique is to read the input file line-by-line and use regular expressions to match the delimiters. However, [Boh01] argues that this is not a good way to parse XML as line boundaries carry no structural meaning in XML. This technique of using regular expressions is difficult to maintain when new tags are introduced or existing tags are changed. XML applications making use of XML parsers have the advantages of maintainability and flexibility. Undoubtedly, XML parsers provides a layer of abstraction, making it easy and convenient to parse

XML.

However, XML parsers may add a layer of processing overhead when processing XML. Thus, we argue that for the case of XML data with **static** and **simple** structures, it may be more efficient to use simple regular expression matching techniques to extract the required information. This is especially effective when XML-specific features (such as Namespaces, XPointers etc.) are not required. Extracting the data from the DBLP database for cleaning is one such example. The DBLP data has a very simple structure and we need to extract the text for the cleaning process. We perform the data extraction using 4 methods: Perl regular expression matching, an XML SAX-based parser in Perl, regular expression matching in Java, and an XML SAX-based parser in Java.

Perl and Java are chosen for their popularity with XML application developers. Perl is a widely used language with strong regular expression support, and is a popular language for text processing tasks. Java is currently one of the most popular languages used for XML applications.

Regular expressions (Regex) are used to extract values of the XML tags via pattern matching. An example of a Perl regular expression is as follows :

```
/<author>(.*)<author>/
```

This example matches the author tags in an XML document and retrieves the data between the opening and closing tags. This will give us the author's name.

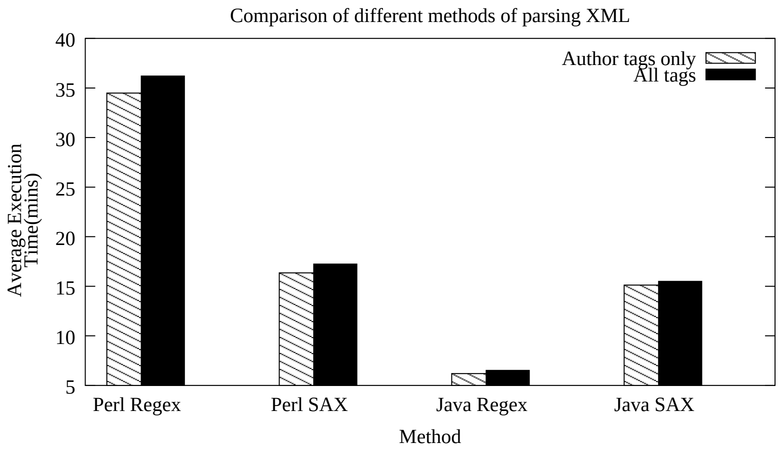
SAX (Simple API for XML) [Meg01] is a popular interface for XML parsing, with the other being the Document Object Model (DOM) interface [W3C01]. We choose SAX-based XML parsers to represent the generic XML parser for its scalability and efficiency. It is widely agreed that the event-based SAX parsing is more efficient and suitable when parsing large XML databases. The details of SAX and DOM parsing are not discussed here. Interested readers can refer to [Meg01] and [W3C01] for more details.

The results of our experiments are shown in Figure 6. We perform the parsing twice using each method. For one pass, we are only interested in extracting the data from `author` tags. We extract the information from all tags in the second pass.

	Avg. Execution Time for Perl (mins)		Avg. Execution Time for Java (mins)	
	Regex	SAX-Based Parser[a]	Regex [b]	SAX-based Parser[c]
Author tags	34.38	16.36	6.20	15.12
All tags	36.20	17.24	6.52	15.49

[a]The package `XML::Parser::PerlSAX` is used.
[b]OROMatcher [Sav00], a regular expression package in Java, is used.
[c]The Xerces [Pro01] Java XML parser is used.

(a)



(b)

Figure 6: Comparison of different methods of parsing XML.

The first observation we make is that the time taken for processing only the author tags is almost the same as the time taken for processing all tags. This is due to the fact that the bulk of the time is spent on file I/O operations. Another reason is that all the tags have to be parsed regardless of whether we are interested in the contents, as we have to get the name of the tag before we know if we are interested in the contents. The time on spent processing the content (e.g. processing the authors' name) is negligible.

The next observation is that in terms of performance, there is no clear winner between using SAX to parse XML and using regular expression matching. For Perl, using the SAX parser is much faster than using regular expression matching. The reverse is true for the experiments using Java[2]. Our view is that it is viable to use regular expression matching techniques to parse XML data with **simple** and **static** structures (where only the very basic XML structures are used). This is especially true if it offers a large performance gain over XML parsers. However, using an XML parsing API is definitely the way to go if the XML structure is complex and uses advanced features (such as Namespaces, XPointers, etc.). Using an XML parser API also makes the applications easier to maintain (as the authors found in the course of the experiments).

Lastly, Java is the clear winner over Perl for the simple reason that the Perl code is interpreted while the Java code is compiled.

5.2 Experiment 2: Indexing structures for sorted data.

As highlighted in the previous section, the cleaning process requires the data to be sorted based on a user-defined key. This is not easily achieved for this dataset (see Issues 3 and 4 in Section 4). Our solution is to make use of indexes. The goal of this set of experiments is to identify efficient, scalable and flexible indexing techniques to retrieve the authors' names from the DBLP database (which consists of many small XML files) in various sorted orders. We compare the performance of the following 3 setups for indexing the authors' names :

1. Using a RDBMS and replicating the name (Setup 1). This set-up examines the

[2]The author has, as far as possible, used standard programming techniques in all the codes to ensure a fair comparison.

use of a relational database as an index mechanism. We define a relational table with the schema : `(pubid,name,sortkey1,sortkey2,...)`. The attribute `pubid` contains a reference to the physical XML file from which this author's name is extracted. The `name` stores the author's name. The `sortkey`s store the "keys" used by the SNM. Since we store the actual name in the table, we do not need to refer back to the XML files anymore. We simply issue an SQL statement which selects all the names ordered by the sort key (e.g. `sortkey1`), and feed them into the reasoning engine using the SNM. We note here that the name is replicated (stored in both the RDBMS and the XML files).

2. Using a sorted flat file as index (Setup 2). This set-up models the case where the name cannot be replicated (either as a result of storage constraints or to avoid data inconsistency problems). Each line in the index file contains the location of the XML file containing the publication, and the location of the author within the file. The index is physically sorted in the file with respect to the authors' names that the entries are referencing. For every entry in the flat file index, the base XML file is accessed once to retrieve the author's name.

3. Using a flat file as index and replicating the name (Setup 3). This set-up is a hybrid of the first two set-ups. The name is replicated in the flat file, but no RDBMS is used.

For illustration purposes, suppose we define a sort key which changes all letters in the name to lowercase and sorts them lexicographically. Assume we have 2 authors : "Tan Ah Kow" is the sole author of a paper stored at /conference/abc/TAK2001. "John Lim" is an author of a paper stored at /conference/xyz/JL2001, and his name appears on the third line. The way this data is stored by the 3 setups is depicted in Figure 7.

The runtimes of cleaning the names using the different index setups with varying window sizes are shown in Figure 8. For Setup 1, the RDBMS used is Oracle 8 running on a remote server connected by a LAN. Setup 1 is slightly slower than Setup 3 due to the overheads of the RDBMS and the communication costs between the client program and the database server. However, it is much faster than Setup 2 (by up to 40 times). This is largely due to the I/O costs incurred by Setup 2. In Setup 2, since the name is not stored in the index, we need to access the physical XML document and parse it to retrieve the author's name. The number of files that need

pubid	Name	Sortkey1
/conference/abc/TAK2001	Tan Ah Kow	aahknotw
/conference/xyz/JL2001	John Lim	hijlmno

(a) Setup 1.

```
/conference/abc/TAK2001,1
/conference/xyz/JL2001,3
```

(b) Setup 2

```
/conference/abc/TAK2001,1|Tan Ah Kow
/conference/xyz/JL2001,3|John Lim
```

(c) Setup 3

Figure 7: Three indexing techniques for author names.

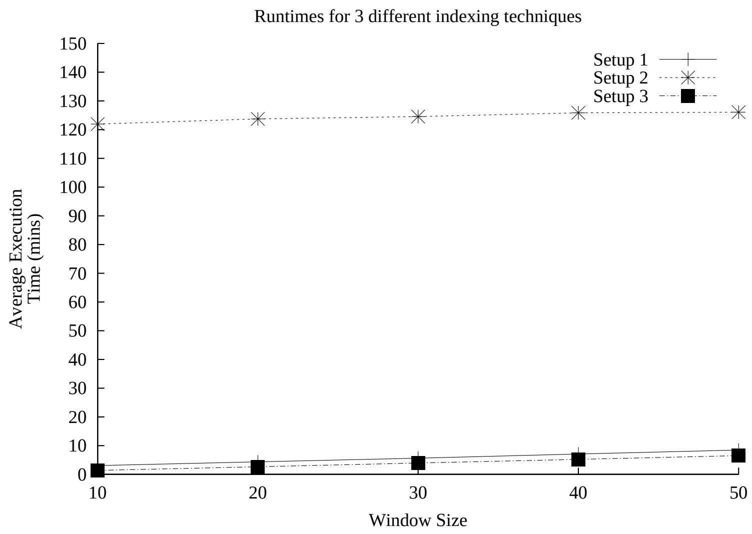
to be accessed and searched is equal to the number of authors in the index (i.e. the same file may be accessed more than once). The storage requirements of the 3 setups are listed in Figure 9. The extra storage required by Setup 3 compared to Setup 2 is due to the name replication in the index.

Although Setup 3 does not require the support of a RDBMS and is the fastest among the 3 techniques, Setup 1 is the most scalable and maintainable among them. Consider the case when multiple sort keys are defined for the SNM. Since the index file for Setups 2 and 3 need to be physically sorted, one index file can only support one sort key. That is to say, for each additional sort key defined, there is a need for another index file. For Setup 1, adding new sort keys involves only adding a new column and queries changed to "*order by*" the new sort key.

We leave this set of experiments with a note about *incremental data cleaning* using the indexes. Consider the scenario when we add a name to the XML database. If we have an index on a key, we can generate the key for this new addition and check if this "new" name is actually a spelling variation of an existing name in the database. This can be done efficiently by comparing the "new" name with the neighbouring names (defined by the key). For example, a paper written by "J. Smith" is to be added to the DBLP. We generate the key for this name and compare with similar keys in the index. This might bring to our attention that this entry may be a variation of the name "John Smith", which is already in the database.

Window Size	Running Time (mins) Setup 1	Setup 2	Setup 3
10	3.07	121.95	1.39
20	4.37	123.73	2.66
30	5.64	124.57	3.94
40	7.10	125.85	5.21
50	8.47	126.10	6.52

(a)



(b)

Figure 8: Runtimes for 3 different indexing techniques.

Storage Requirements (kB) Setup 1	Setup 2	Setup 3
24640	10450	16996

Figure 9: Storage requirements for the 3 indexing techniques.

5.3 Experiment 3: Effectiveness.

In this final set of experiments, we investigate the effectiveness of the knowledge-based approach to cleaning XML databases. We design 2 rules to identify if 2 inexact author names actually refer to the same person. Both rules are based on string similarity measures and the results are shown in Figure 10.

Our technique managed to identify quite a number of inexact duplicates in the author names, within a reasonable amount of time. Some examples of inexact duplicate author names are shown in Figure 11. The sampled precision is relatively constant throughout the various runs of the experiments with varying window sizes and rules. Higher precision can be achieved easily by raising the similarity threshold [LLL01]. It is important to note that there are numerous cases where a definite conclusion cannot be reached on whether the author names are duplicates, even after doing a detailed check on the authors' research interests and respective co-authors manually.

The 2 duplicate identification rules used here are based on string similarity. As shown in [LLL01], the introduction of domain knowledge will increase both recall and precision of the duplication identification process. One such rule for this dataset might be : *For 2 similar names, that we suspect belong to the same person, we can check on their co-authors. Confidence of the 2 names referring to the same person will increase if they have some common co-authors.* Implementation-wise, this will involve finding the set of co-authors for both names, and performing a set intersection on the 2 sets of co-author names. This is an expensive process and adding this rule to the duplicate identification process can increase the time taken by 80 times! There are ways to reduce the computation required, but that is beyond the scope of this paper. Another piece of domain knowledge might be : *If the 2 names are very similar, and they have the same research interests, chances are that they are the same person.* The difficulty lies in how to determine if their research interests are *similar*.

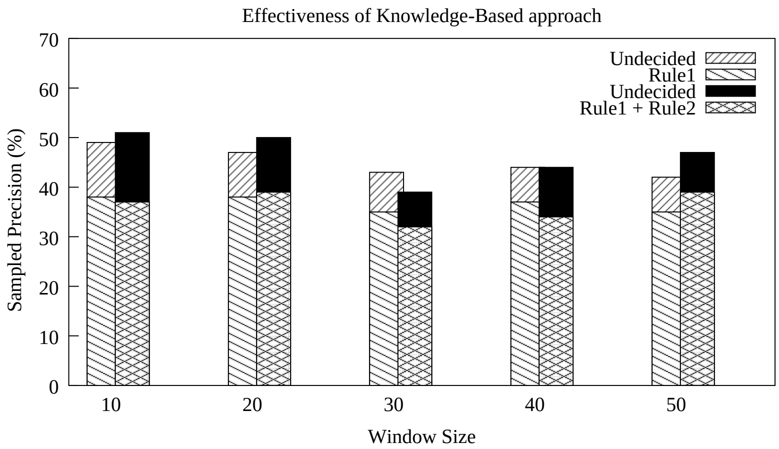
So far, we have focused on detecting inexact duplicate author names. We now see how another *business rule* can be verified. Most journals are yearly publications, with all issues of the same volume being published in the same year. Hence, we can write this rule as : *All articles belonging to the same volume of the same journal should be published in the same year.* We note that there are a few journals whose issues from the same volume span over different years (e.g. AI Magazine). To identify

Window Size	Rule 1				Rule 1 + 2			
	Time (mins) (mins)	No. of pairs	Precision[a]	Undecided[b]	Time (mins)	No. of pairs	Precision	Undecided
10	3.07	2534	38%	11%	3.55	12292	37%	14%
20	4.37	2961	38%	9%	5.67	14460	39%	11%
30	5.64	3224	35%	8%	7.65	15714	32%	7%
40	7.10	3413	37%	7%	9.67	16619	34%	10%
50	8.47	3590	35%	7%	11.81	17401	39%	8%

[a]Based on a sample size of 100.

[b]When manual verification (making use of information such as the group of co-authors, year of publications, research topics etc.) cannot affirm if the pair of names belong to the same person, it is classified as "undecided".

(a)



(b)

Figure 10: Effectiveness of Knowledge-Based Approach.

Takashi Matsuyama	Takashi Matsyama
Francisco V. Cipolla Ficarra	Francisco V. Ciolla Ficarra
N. Chandrasekharan	Chandra N. Sekharan

Figure 11: Examples of inexact duplicate author names in the DBLP.

```
(deftemplate journalVolYr
        "template for checking Journal,Volume-> Year"
   (slot journal)
   (slot volume)
   (slot year)
   (slot count)
)
```

Figure 12: Fact templates for the simplified DBLP DTD in Figure 4.

these journals, we can list out all these journals and mark them as non-errors. In our experiment, we make use of the fact that in the case of an error, the number of articles that are not published in the same year should be small (which we set at 10%). For journals with issues from the same volume spanning over different years, the percentage of publications in the same volume spanning the years should be sizable.

To identify this class of error, we make use of the fact template obtained in Figure 5. The reasoning engine then processes the facts to the summarized format shown in Figure 12. The additional count slot is used to keep track of the number of articles with the same journal name, volume and year combination. This value can help determine if the journal is one whose issues from the same volume span over different years, or if it is likely to be an error. This experiment took about 15.5 minutes to process about 80,000 journal entries and identified 7 errors with 100% precision (i.e. all cases identified are indeed errors). Two examples of the errors detected are shown in Figure 13. Note that we do not need to keep all the journal records in the reasoning engine. The information is summarized, and the unnecessary facts are taken out. There were at most 3195 facts in the reasoning engine at any one time. These results show that the framework is flexible enough to identify various types of data anomalies effectively.

6 Conclusion

This work presents a first attempt to clean XML databases with a knowledge based framework. We experimented with the DBLP database, a large bibliographical database

```
Journal : ACM SIGMOD Digital Review
Volume  : 1
Year    : 2000 (count 1)
Year    : 1999 (count 68)
Percentage 0.014492753623188406

Journal : IEEE Computer
Volume  : 33
Year    : 1997 (count 9)
Year    : 2000 (count 137)
percentage  :0.06164383561643835
```

Figure 13: Examples of Volume-Year mismatch anomalies in journal records.

in XML. The knowledge based framework was originally designed for relational data, and using it to clean XML databases presents new challenges. We examine the problems that arise and investigate various ways in which they can be overcome. In particular, we have shown how XML data can be mapped to expert system facts, and how indexing can enable sorted XML data to be fed into the reasoning engine. We compare the performance of various methods to index the data, including the use of a RDBMS, and highlight the benefits of each method.

We argue that for XML data that has a simple and static structure which is known priori, it may be more efficient to parse them using regular expression matching. This is especially effective if complex XML features are not required. However, it should be noted that if the structure of the XML database is complex or changes frequently, XML parsers should be used instead.

We note that every issue raised in this work, including XML indexing, mapping XML to other data models, and the efficiency of XML parsers etc., is a large area of research in its own right and requires much further work. For cleaning the DBLP specifically, it is worth studying the error patterns and deriving domain knowledge from them for more effective cleaning. It is also interesting to study how to efficiently introduce such knowledge into the framework. We are also looking into testing this framework with XML databases having complex structures. We used simple indexing techniques in this work. It will be interesting to see how recent work in indexing XML (e.g. Index Fabric [CSF^{+}01]) can be applied to this work.

Acknowledgements

We thank Michael Ley for his kind permission to use the DBLP database in our work, and his feedback on the errors discovered when cleaning the data.

References

[BMBA00] A.G. Büchner, M. Baumgartenand M.D. Mulvenna, R. Böhm, and S.S. Anand. Data Mining and XML: Current and Future Issues. In *Proceedings of the 1st International Conference on Web Information Systems Engineering, Hong Kong*, pages 127–131, 2000.

[Boh01] Eric Bohlmann. Parsing xml, part 1. *Perlmonth*, June 2001. Available at URL `http://www.perlmonth.com/columns/perl_xml/perl_xml.html`.

[Bou01a] Ronald Bourret. Mapping dtds to databases. Available at `http://www.xml.com/lpt/a/2001/05/09/dtdtodbs.html`, May 2001.

[Bou01b] Ronald Bourret. Mapping w3c schemas to object schemas to relational schemas. Available at `http://www.xml.com/lpt/a/2001/05/09/dtdtodbs.html`, May 2001.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000.

[CFP00] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. XML: Current Developments and Future Challenges for the Database Community. In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust, editors, *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2000.

[CSF$^+$01] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the 27th International Conference on Very Large Data Bases, VLDB.*, 2001.

[DFS99] A. Deutsch, M. Fernandex, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999.

[Fal01] D. C. Fallside (Eds). "XML Schema Part 0: Primer". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-0.

[FH97] Ernest J. Friedman-Hill. Jess, the Java Expert System Shell. *Sandia National Laboratories report SAND98-8206*, 1997. Available at URL `http://www.prod.sandia.gov/cgi-bin/techlib/access-control.pl/1998/988206-r.pdf`.

[HGS99] Dennis Shasha Helena Galhardas, Daniela Florescu and Eric Simon. An extensible framework for data cleaning. *INRIA Technical Report*, 1999.

[HS98] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery, Vol. 2, No. 1*, pages 9–37, 1998.

[LLL01] Wai Lup Low, Mong Li Lee, and Tok Wang Ling. A Knowledge-Based Approach for Duplicate Elimination in Data Cleaning. In Maurizio Lenzerini Mokrane Bouzeghoub, editor, *Information Systems : An International Journal. Special Issue on Data Extraction, Cleaning, and Reconciliation.* Elsevier Science, 2001.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(3), 1997.

[ME97] Alvaro E. Monge and Charles P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of the ACM-SIGMOD Workshop on Research Issues on Knowledge Discovery and Data Mining.* Tucson, AZ, 1997.

[Meg01] David Megginson. SAX: The Simple API for XML. Available at `http://www.megginson.com/SAX/`, 2001.

[Min00] Minos N. Garofalakis and Aristides Gionis and Rajeev Rastogi and S. Seshadri and Kyuseok Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 165–176. ACM, 2000.

[Pro01] The Apache XML Project. Xerces Java Parser. Available at `http://xml.apache.org/xerces-j/index.html`, 2001.

[RH00] Vijayshankar Raman and Joseph M. Hellerstein. Potters wheel: An interactive framework for data cleaning and transformation, 2000. `http://control.cs.berkeley.edu/abc/index.html`.

[Sav00] Daniel F. Savarese. Oromatcher^{TM} 1.0. Available at `http://www.savarese.org/oro/`, 2000.

[SCH+98] Lisa Singh, Bin Chen, Rebecca Haight, Peter Scheuermann, and Kiyoko Aoki. A Robust System Architecture for Mining Semi-Structured Data. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining, New York, USA.*, pages 329–333, 1998.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314, 1999.

[The01] The dbXML Group. dbXML. Available at `http://www.dbxml.org`, 2001.

[W3C01] W3C DOM Working Group. Document Object Model (DOM). Available at `http://www.w3.org/DOM/`, 2001.

[Wid99] Jennifer Widom. Data Management for XML: Research Directions. *IEEE Data Engineering Bulletin*, 22(3):44–52, 1999.

[WM89] Y. Richard Wang and Stuart E. Madnick. The inter-database instance identification problem in integrating autonomous systems. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 46–55. IEEE Computer Society, 1989.

[Xyl01] Lucie Xyleme. A dynamic warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin, Special Issue on XML Data Management*, 24(2):40–47, June 2001.