An Efficient use of a Mapping Knowledge Base

Boubaker Kahloula, Karim Bouamrane
Department of Computer Science
University of Oran, Algeria
IGMO, 31000 Oran
Algeria
{boubaker.kahloula, k_bouamrane}@yahoo.fr

Journal of Digital Information Management

ABSTRACT: *In a wide range of commercial and industrial applications data from different sources - web, databases or documents - are integrated into databases for the purpose of querying and analysis. It is then determined for each of the information contained in the source schema, what should be the corresponding column in the target relational database, that is to say, to establish a correspondence (Mapping) between the data source and the target column. A number of tools and prototypes of Schema Matching have been developed in order to automate the process of finding a suitable mapping. One technique employed is to use mapping decisions previously taken by the user manually. These decisions are stored in a Mapping Knowledge Base. The problem with this procedure is that the volume of the Mapping Knowledge Base increases very significantly over time. Processing large volumes of information may take several hours or even days. The prototype described in this article, keeps processing time at acceptable levels.*

Categories and Subject Descriptors:
H.2.1 [Database Logical Design]: Schema and Subschema;
I.2.6 [Learning]: Knowledge Acquisition

General Terms:
XML, Database Processing, Database Schema

Keywords: XML, Relational Databases, Data Integration, Schema Matching, Similarity Coefficients

Received: 19 April 2013, Revised 2 June 2013, Accepted 9 June 2013

1. Introduction

XML (eXtensible Markup Language) is considered a universal medium for data exchange and the best way to integrate data from different sources and of different types in a database (Figure 1). The data extracted from different sources are converted to XML. The prototype presented here is used to load (semi-) automatically data contained in an XML file into a relational database. It is based on several concepts, the first being Schema Matching[1], that is to say, the processing used to establish a correspondence between the names of the elements of the input XML file and those of the columns of the tables in the target database. With regard to this aspect, several algorithms or prototypes such as SemInt [2], Cupid [3] , Clio [4], Coma++ [5,6], etc. have been developed so far, some of them specific to a given domain, for example for the biomedical field [7], others being generics. Some tools, such as Altova MapForce [8] are commercially available. In 2001, Erhard Rahm and Philip A. Bernstein [9] developed a classification of these prototypes, depending on whether the mapping takes place according to the scheme or instances (data), the individual elements or structures, the names of these elements or constraints that these elements must meet, the auxiliary information used as thesaurus, dictionary, etc. Schema Matching is also used, as in our case, in data migration, integration and exchange [4].

We can see in Figure 1, which shows the whole process schematically, at what level is the user intervention, which is expected to be over time less and less necessary. The prototype takes into account the processing time, keeping it at acceptable levels in case of large XML files.

In this article we first present the different process sequences through which an XML file must pass, and the role of each of the different modules of the prototype. Then we focus on the key component of the system architecture, which is the Matcher. We will then review all the elements that reduce the processing time.

2. Process sequences

As can be seen in Figure 2, the processing starts when the data are retrieved from the web, from databases or documents. They are then converted to XML and stored in a text file.

- The first module of the system, referred to as Matcher, begins by reading the XML file, extracts the XPaths and stores them in a table, called here Mapping Table (2a). What we call XPath is the path to reach each information (instances) contained in the XML file.

Example: /Article/JournalIssue /pubDate/Year

The Mapping Table contains two columns. It represents the relationship between the XPaths contained in the XML file and the column names of the table in the target database.

Example: /Article/JournalIssue/pubDate/Year ® Year

- The second column is filled in if the matcher has found an identical or similar XPath in the Mapping Knowledge Base which is accessed simultaneously (2b). An identical mapping to the example given above is /Article/ JournalIssue/pubDate/Year while a similar mapping would be /Article/ Journal/Date/Year.

- The data contained in the XML file, instances as well as attributes, are stored for purely technical reasons in a temporary table (2c), which allows indexing and sorting operations.

- The mapping, that is to say, the correspondence between the XPaths and column names of the table in the target database, being validated or completed by the user (3a), is stored in the Mapping Knowledge Base (3b).

- A second module, the Loader, then takes the data contained in the temporary tables (4a) and loads them into transient tables (4b). Transient tables derive their name from serving as transit for the records to be loaded into the target database. Their structures are static and defined in advance by the user.

- The last module is the Rules Engine. By using Rules, which are SQL commands or procedures stored in the repository (5a), it will read the records from the transient tables (5b) and dispatch them onto the different target database tables.

The prototype we have developed contains the following programs:

- The Matcher, whose role is to extract the XPaths from the XML file and search the mapping Knowledge Base for identical or similar XPaths. Similarity is evaluated according to a calculated coefficient. The Matcher can potentially access thesauri.

- The Loader, which will load the data into a transient table.

- The Rules Engine, which will read the data from the transient table and load them into the target database.

- The Archiver will archive the XML file once the processing

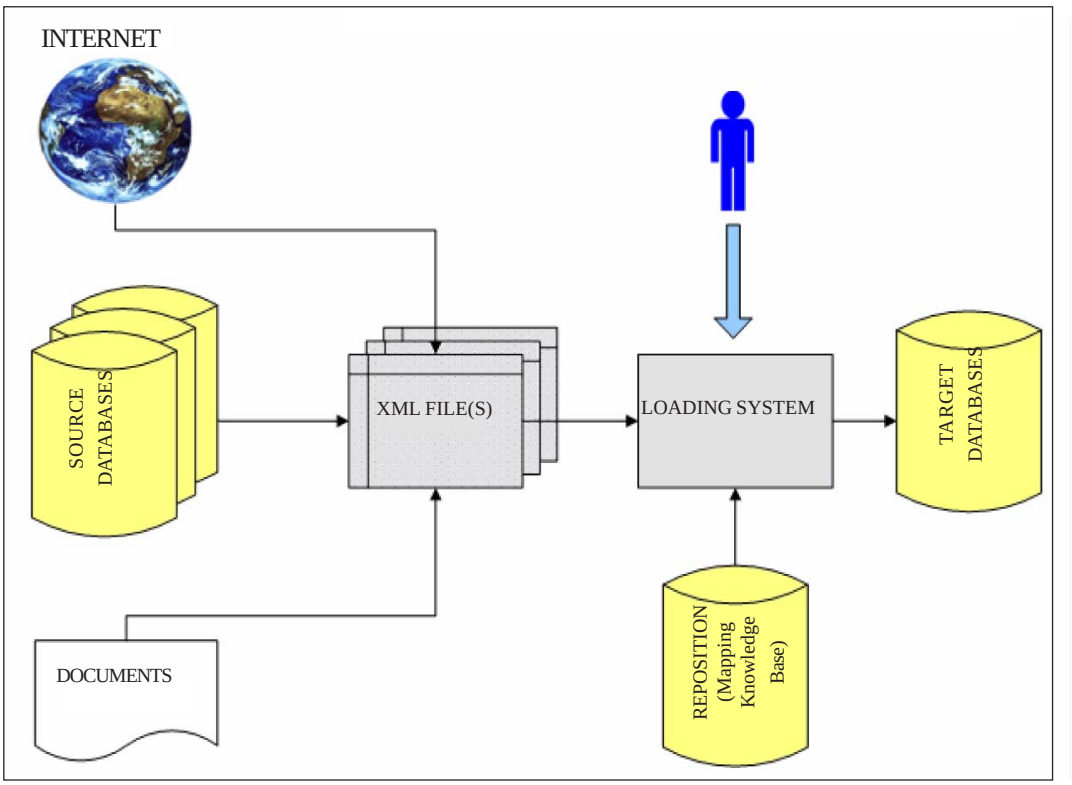

Figure 1. Data Integration

is completed.

A Graphical user interface (GUI) allows the user to

- enter, complete or validate the mapping,

- input the Rules to be applied to the transient tables.

The Repository is a database containing:

- The Mapping Knowledge Base,

- The Rules to be applied to the transient table for the dispatch of the data contained therein onto the tables of the target database,

- The table containing the mapping to be eventually completed by the user,

- The potential thesauri,

-A temporary table used for purely technical purposes.

3. The Matcher

In our prototype we use Name Matching [9]. Name Matching considers equality or similarity of names, canonical names (e.g., EmpID for employee identity) or synonyms.

The resulting Mapping of the Name Matching is proposed for validation to the user. Mapping decisions taken by the user are stored in the Mapping Knowledge Base to be reused, in form of relationships "*XPath* → *Column Name*". Assuming that $S1$ is the set of XPaths contained in an XML file to be processed and the Mapping Knowledge Base contains "$S2 \rightarrow S3$" where $S2$ is a set of XPaths and $S3$ a set of column names, we perform a Schema Matching between $S1$ and $S2$ then deduce by transitivity the mapping "$S1 \rightarrow S3$".

The matching algorithm is described in pseudo code below (Algorithm 1). The Matcher, after extracting the XPaths from the XML file will look for each one of these XPaths for a similar XPath in the Mapping Knowledge Base. A similarity coefficient is calculated and the mapping with the highest similarity coefficient is proposed to the user. The search stops if the calculated similarity coefficient is equal to 1, meaning that an exactly equal XPath was found.

Calculating the similarity coefficient occurs after pre-processing of the XPaths. Special characters commonly used in XML element names, such as underscore, hyphen and colon are removed.

4. Process efficiency

Regardless of the similarity coefficient used, the more XPaths are contained in the XML file or the Mapping Knowledge Base, the longer will processing time be.

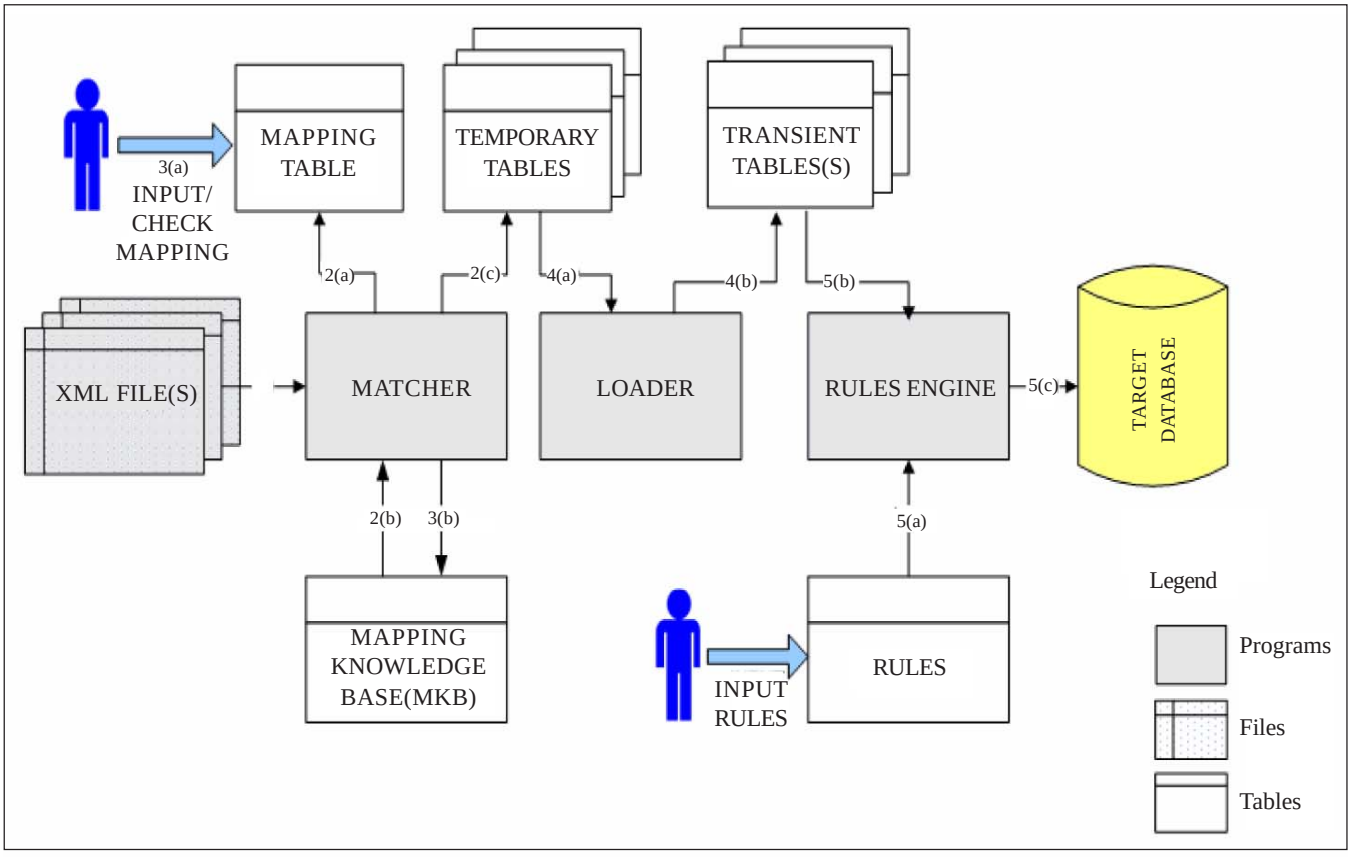

Figure 2. Process sequences

Algorithm 1:

```
// FP[i] are the XPaths extracted from the
XML-file (i=1,n)
// MP[j] are the XPaths contained in the
Mapping Knowledge Base (j=1,m)
// MCol[j] are the columns of the transient
table mapped to the MP[j]
// We search for the FCol[i], the columns
of the transient table to map
// to FP[i]

FOR i=1,n {
// Loop on the XPaths extracted from the
XML-file
  maxSim = 0 ;
  FOR j=1,m {
// Loop on the XPaths contained in the
Mapping Knowledge Base
    CALCULATE sim(FP[i],MP[j]);
    IF (sim(FP[i],MP[j] > maxSim) {
      maxSim = sim(FP[i],MP[j]);
      sim[i] = maxSim;
      FCol[i]= MCol[j];
      IF (sim[i]=1) EXIT LOOP;
    }
  }
}
```

Processing time, which may be several hours or even several days, is definitely a problem in the processing of large files [10,11]. In order to improve this time

- First, we used StaX, a Java API for extracting XPaths from XML files

- Secondly, we chose Jaro-Winkler as method of similarity calculation

- Thirdly, we opted for PathSim, an efficient technique for computing similarity coefficients of two XPaths

- And finally, and this is the most important point, we adopted a strategy when the Knowledge Base is "*full enough*". We called this developed strategy ZeroOne algorithm.

4.1 StAX

The Java API for XML Processing, or JAXP [12], is one of the Java XML APIs. It is a standardized API for validating, parsing, generating and transforming XML documents. The three basic interfaces are:

- The Document Object Model or DOM parser interface

- The Simple API for XML or SAX parser interface

- The Streaming API for XML or StAX interface (added in JDK 6, available separately as a jar for JDK 5)

To read XML files and extract the XPaths we used the StAX API (Streaming API for XML), the technique for streaming. The StAX API is the last of the JAXP family. It is an alternative to SAX and DOM. DOM creates memory objects representing complete trees, within which navigation is easy. But the disadvantage of this technique is that it consumes a lot of memory and CPU time. As with SAX, XML documents are read with StAX sequentially, so that even large volumes of material can be processed. While the SAX Parser call registered callback functions (Push-Parsing), the reverse is true for StAX; the application calls iteratively the parser (Pull-Parsing). The advantage is that, unlike SAX, the control is held by the programmer who can write simple iterative loops.

4.2 Calculating similarity of Xpaths

To determine if there are any XPaths in the Mapping Knowledge Base which are similar to those contained in the XML files, we use the notion of similarity coefficient. The coefficient of similarity of two XPaths $x1$ and $x2$ is the value of the function f of similarity $f(x1, x2)$, $0 \le f \le 1$. The closer the value of f is to 1, the more XPaths are similar. A value of 1 indicates that XPaths are identical. The methods of calculating coefficients of similarity are numerous, varying from simple string matching functions [13,14,15] to functions specific to XML [16]. The prototype can be configured in such a way as to use different similarity functions, including but not limited to:

- Cosine, which measures the cosine of the angle between the strings represented as vectors.

- Jaccard, which calculates the similarity between two strings A and B as $J(A, B) = |A \cap B| / |A \cup B|$

If we consider for example the two XPaths

A = /Article/JournalIssue/pubDate/Year and B = /Article/Journal/Date/Year

the similarity of these two XPaths according to Jaccard will be calculated as follows:

$|A \cap B|$ is equal to 2 and $|A \cup B|$ to 6. $J(A, B)$ will be therefore equal to 0.33.

- Sorensen-Dice for which the similarity between A and B is $S(A, B) = 2|A \cap B| / |A| + |B|$

- Jaro-Winkler, based not only on the number of common characters between A and B but also on their order. Jaro-Winkler is a variant of the Jaro function. Unlike Jaro, it only considers the first few characters of the strings to compare.

Studies [17,18] based on the recall/precision measures, the classical measures in the Information Retrieval (IR) field, do not determine the effectiveness of similarity functions. However among the methods proposed in the prototype, Jaro-Winkler, as confirmed by the study in [19] was, in our simulations, more effective (Figure 5).

4.3 PathSim

It is possible to calculate the coefficient of similarity of two XPaths to proceed in different ways:

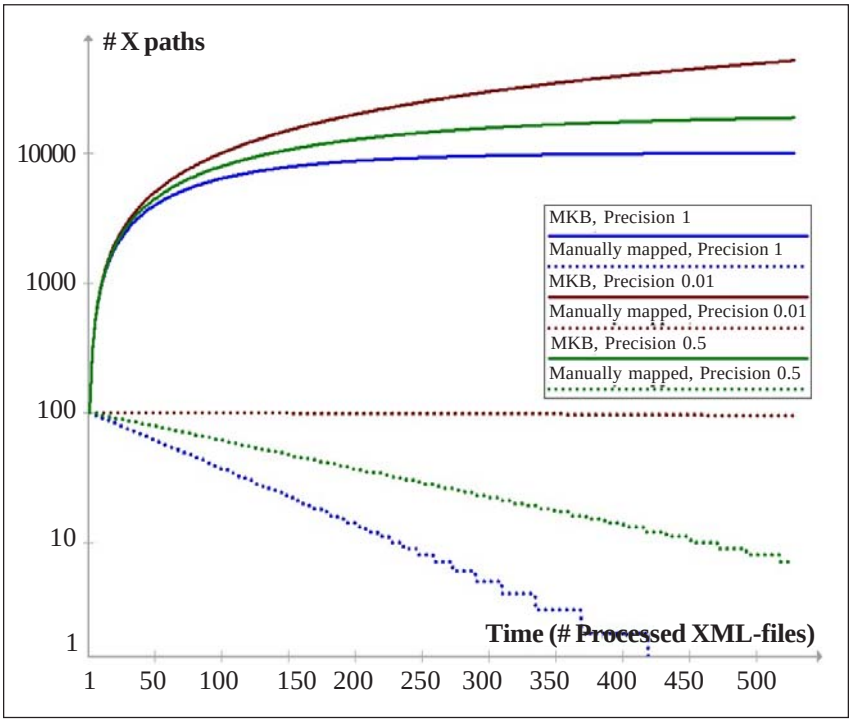

Figure 3. Volume of the MKB and manually mapped XPaths over time

- Adopting the so-called namePath [20] or Context Path Coefficient [21], which is to form strings from the concatenation of the elements of the XPaths, then compute the similarity between strings,

- Adopting that called PathSim [22], also used in [23] and [24], which treats XPaths as lists of names of elements and calculates the similarity between lists.

Finding a string in a string collection such that it has similarity with a query string, is a problem adressed in a broad range of applications such as spelling correction, duplicate detection, flexible dictionary look-up and record linkage [25,26]. Some algorithms such as the so called CPMerge [27] are intended for Approximate Dictionary Matching. It collects candidate strings that satisfy the constraint of a certain overlap. Since we look for only one candidate, namely the most similar to the query string, we can not use such an algorithm.

We have therefore opted for the method called PathSim, which proved significantly better than namePath.

4.4 The ZeroOne algorithm

Over time, the volume of the Mapping Knowledge Base increases, along with the probability of finding in it similar XPaths to the XPaths contained in the XML file to be processed. To illustrate what we have said, let us consider the hypothetical example obtained from a simulation: the XML files have to deal with a constant number of XPaths equal to 100 and the number of XPaths of the Mapping Knowledge Base used in the mapping is 1 percent. We see in Figure 3 that after a period of "*filling*", the volume of the Mapping Knowledge Base tends to stabilize over time, whereas the manual work is linearly decreasing.

We also vary the Precision during the simulation. Precision is a measure used to evaluate performance. The elements for calculating the Precision in this case are given in Figure 4. It is determined by the formula:

$$Precision\ (A, C) = XML\text{-}Files \cap Mapping\ Knowledge\ Base\ /\ Mapping\ Knowledge\ Base = B\ /B + C$$

We note in Figure 3 that the volume of the Mapping Knowledge Base is much larger and manual work much more significant when Precision is lower. Precision reflects the heterogeneity of the processed XML files. When they are more heterogeneous, the volume of the Mapping Knowledge Base is larger. But in all cases, the volume of the Mapping Knowledge Base stabilizes over time and manual work tends to decrease.

When considering Algorithm 1, we also note that the calculation of similarity takes place inside of two nested loops, one traversing the XPaths of the XML file, the other one those of the Mapping Knowledge Base. If n is the number of XPaths contained in the XML file and m is the number of those contained in the Mapping Knowledge Base, calculating the similarity coefficient is performed $n * m$ times. Regardless of the method of calculating the similarity used, the more XPaths are contained in the Mapping Knowledge Base, the longer processing will take.

So we included in our prototype an option to avoid this calculation. This is the option we have called *ZeroOne*. This option allows searching the Mapping Knowledge Base for an exactly equal XPath, which may be found or not. Access to the Mapping Knowledge Base, which is a table of the repository, is in this case through an index, which

can significantly speed up the processing time. The user has the choice between using a quick and exact search or a slower fuzzy search, the probability of results being greater in the second case. Whether to choose one or the other option depends on the volume of the XML file and that of the Mapping Knowledge Base. As the volume of the Mapping Knowledge Base is not very important at the beginning (Figure 3) the user can choose an approximate search and add mappings to the Mapping Knowledge Base manually and later choose the ZeroOne-mapping to speed up processing time.

Algorithm 2 (ZeroOne):

```
// FP[i] are the XPaths extracted from the
XML-file (i=1,n)
// MP[j] are the XPaths contained in the
Mapping Knowledge Base (j=1,m)
// MCol[j] are the columns of the transient
table mapped to the MP[j]
// We search for the FCol[i], the columns
of the transient table to map
// to FP[i]

// Loop on the XPaths extracted from the
XML-file
FOR i=1,n {
// Search for FP[i] in the Mapping Knowledge
Base
  SEEK FP[i];
  IF FOUND {
   // MP[j] = FP[i] found and its
corresponding MCol[j]
    FCol[i]= MCol[j];
  }
}
```

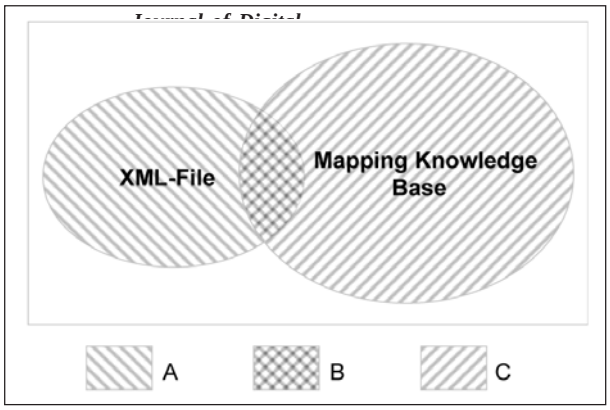

Figure 4. Performance measure

We see in Figure 5 that ZeroOne is the algorithm that takes less time. We also see that the larger the XML file to process, the greater the difference between the ZeroOne option and options using the calculation of similarity coefficients. This is to be expected since, when applying ZeroOne, there are no calculations of similarity coefficients, and access to the Mapping Knowledge Base is not done sequentially but through an index.

5. Conclusion

We used to (semi-)automatic loading of data into a relational database a Mapping Knowledge Base, which stores mappings established or validated manually by the user. Whatever the heterogeneity of XML source files, the volume of the Mapping Knowledge Base tends to stabilize; manual mapping tends to decrease.

We can distinguish two periods, the first be-ing that of "*filling*" the Mapping Knowledge Base. During this period the user can opt for the calculation of a coefficient of

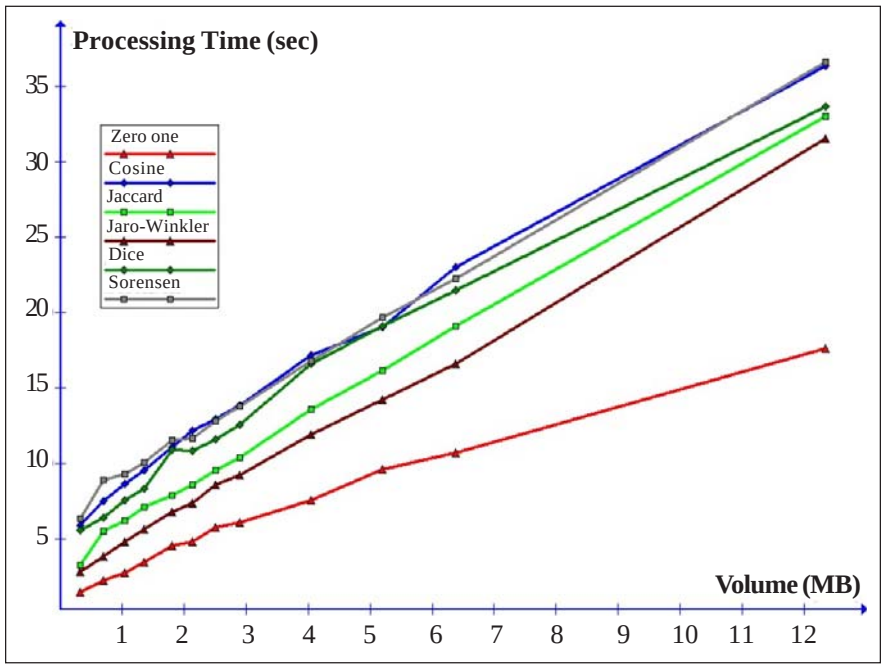

Figure 5. Processing Time

similarity, which will provide assistance in the completion of the manual mapping, but over time the volume of the Mapping Knowledge Base will become more important and the user can then choose the coeffi-cient *ZeroOne*, which avoids the calculation and reduces processing time. For this purpose, the prototype includes a statistics module to help the user make a decision.

The Schema Matching algorithm used here is generic; the prototype can be operated both in e-commerce, in industry, in banking, in the medical sector and elsewhere.

6. Acknowledgment

We would like to thank Dr. Angelika Krüger for reviewing this paper.

References

[1] Bellahsène, Z., Bonifati, A., Rahm, E. (2011). Schema Matching and Mapping. *Data-Centric Systems and Applications*. Springer.

[2] Li,W. -S., Clifton, C. (2000). Semint: A tool for identifying attribute correspondences in heterogeneous databases using neural networks, *Data Knowl. Eng.*, 33 (1) 49–84.

[3] Madhavan, J., Bernstein, P. A., Rahm, E. (2001). Generic schema matching with cupid. In VLDB, p. 49–58.

[4] Fagin, R., Haas, L. M., Hernández, M. A., Miller, R. J., Popa, L., Velegrakis, Y. (2009). Clio: Schema mapping creation and data exchange, *In*: Conceptual Modeling: *Foundations and Applications*, p. 198–236.

[5] Aumüller, D., Do, H. H., Massmann, S., Rahm, E. (2005). Schema and ontology matching with coma++, *In*: SIGMOD Conference, p. 906–908.

[6] Maßmann, S., Raunich, S., Aumüller, D., Arnold, P., Rahm, E. (2011). Evolution of the coma match system, *In*: OM.

[7] Rahm, E., Bernstein, P. A. (2001). A survey of approaches to automatic schema matching, *VLDB J.*, 10 (4) 334–350.

[8] Shvaiko, P., Euzenat, J., Giunchiglia, F., Stucken schmidt, H., Noy, N. F., Rosenthal, A., editors. (2009). *In*: Proceedings of the 4th International Workshop on Ontology Matching (OM-2009) collocated with the 8th International Semantic Web Conference (ISWC-2009) Chantilly, USA, October 25, 551 of CEUR Workshop Proceedings. CEUR-WS.org.

[9] Jianguo, L., Wang, J., Shengrui, W. (2007). XML Schema Matching, *International Journal of Software Engineering and Knowledge Engineering*, 17 (5).

[10] Da Silva, R., Stasiu, R. K., Orengo, V. M., Heuser, C. A. (2007). Measuring quality of similarity functions in approximate data matching, *J. Informetrics*, 1 (1) 35–46.

[11] Do, H. H., Rahm, E. (2002). Coma - a system for flexible combination of schema matching approaches, *In*: VLDB, p. 610–621.

[12] Lee, M. L., Yang, L. H., Hsu, W., Yang, X. (2002). Xclust: Clustering xml schemas for effective integration, p. 292–299. ACM Press.

[13] Vinson, A. R., Heuser, C. A., da Silva, A. S., de Moura, E. S. (2007). *In*: WIDM, p. 17–24.

[14] Boukottaya, A., Vanoirbeek, C. (2005). Schema matching for transforming structured documents, *In*: ACM Symposium on Document Engineering, p. 101–110.

[15] Carmel, D., Efraty, N., Landau, G. M., Maarek, Y. S., Mass, Y. An extension of the vector space model for querying xml documents via xml fragments, *In*: Workshop On XML and Information Retrieval, SIGIR, V. 36.