2010 Second International Conference on Computer Engineering and Applications

An Efficient Duplicate Detection System for XML Documents

Thandar Lwin & Thi Thi Soe Nyunt
University of Computer Studies, Yangon, Myanmar.
thandarlwin72@gmail.com, ttsoenyunt@gmail.com

***Abstract*— Duplicate detection, which is an important subtask of data cleaning, is the task of identifying multiple representations of a same real-world object and necessary to improve data quality. Numerous approaches both for relational and XML data exist. As XML becomes increasingly popular for data exchange and data publishing on the Web, algorithms to detect duplicates in XML documents are required. Previous domain independent solutions to this problem relied on standard textual similarity functions (e.g., edit distance, cosine metric) between objects. However, such approaches result in large numbers of false positives if we want to identify domain-specific abbreviations and conventions.**

In this paper, we present the process of detecting duplicate includes three modules, such as selector, preprocessor and duplicate identifier which uses XML documents and candidate definition as input and produces duplicate objects as output. The aim of this research is to develop an efficient algorithm for detecting duplicate in complex XML documents and to reduce number of false positive by using MD5 algorithm. We illustrate the efficiency of this approach on several real-world datasets.

***Keywords*— XML, Data Cleaning, Duplicate Detection, MD5 Algorithm.**

I. INTRODUCTION

The problem of identifying duplicate records has been considered under many different names, such as record linkage [1], merge/purge [2], entity identification [3], and object matching [4]. Typically, standard string similarity metrics such as edit distance or vector-space cosine similarity [21] are used to determine whether two values or records are alike enough to be duplicates.

Several problems arise in the context of data integration, where data from distributed and heterogeneous data sources is combined. One of these problems is the possibly inconsistent representation of the same real-world object in the different data sources. When combining data from heterogeneous sources, the ideal result is a unique, complete, and correct representation for every object. Such data quality can only be achieved through *data cleansing*, where the most important task is to ensure that an object only has one representation in the result. This requires the identification of duplicate objects, and is referred to as *object identification* or *duplicate detection*.

The problem has been addressed extensively for relational data stored in tables. However, relational data only represents a small portion of today's data. Indeed, XML is increasingly popular as data representation, especially for data published on the World Wide Web and data exchanged between organizations. Therefore, we need to develop methods to detect duplicate objects in nested XML data. There are two types of data heterogeneity [24]: structural and lexical. Structural heterogeneity occurs when the fields of the tuples in the database are structured differently in different databases. For example, in one database, the customer address might be recorded in one field named, say, addr, while, in another database, the same information might be stored in multiple fields such as street, city, state, and zipcode. Lexical heterogeneity occurs when the tuples have identically structured fields across databases, but the data use different representations to refer to the same real-world object (e.g., StreetAddress = 44 W. 4th St. versus StreetAddress = 44 West Fourth Street).

In this paper, we focus on the problem of lexical heterogeneity and present a novel, domain-independent approach that overcomes these problems to efficiently and effectively detect duplicates in an XML document. The objectives of our proposed system are to detect duplicated XML documents relating to the same entities, to improve data quality and integrity, to allow re-use of exiting data sources for new studies and to reduce costs and effort in data acquisition for research studies. More specifically, our approach detects duplicate objects in a single XML document, assuming the structure of elements with equal name may differ. Without further domain-specific information, we are able to detect duplicate objects with typographical errors (data differs only slightly), equivalence errors (data differs significantly but has same meaning), and missing data.

The outline of this paper is as follows. In Section II, we present related work. In Section III, we discuss the main concept of MD5 algorithm on which our approach is based. Section IV provides a detailed description of our approach. In Section V, we present experimental study and we conclude in Section VI.

II. RELATED WORK

The problem of duplicate detection, originally defined by Newcombe [9] and formalized in the Fellegi-Sunter [10] model for record linkage has received much attention in the relational world and has concentrated on efficiently and effectively finding duplicate records.

Some approaches are specifically geared towards a particular domain, including census, medical, and genealogical data [1, 11, 12], and require the help of a human expert for calibration [13]. Other algorithms are domain-independent, e.g., those presented in [14, 22]. All these approaches have in common that description

978-0-7695-3982-9/10 $26.00 © 2010 IEEE
DOI 10.1109/ICCEA.2010.189

selection and structural heterogeneity are not considered because the problems do not arise in relational duplicate detection.

Recent projects consider detecting duplicates in hierarchical and XML data. This includes DELPHI [5], which identifies duplicates in hierarchically organized tables of a data warehouse using a top-down approach along a single data warehouse dimension. The algorithm is efficient because it compares only children with same or similar ancestors. In this context, instance heterogeneity has to be considered but description selection and schematic heterogeneity are still not an issue. There is work on identifying similar hierarchical data and XML data. To minimize the number of pairwise element comparisons, an appropriate filter function is used [23], where three filtering methods is used.

Work presented in [15] describes efficient identification of similar hierarchical data, but it does not describe how effective the approaches are for XML duplicate detection. For example, Dong et al. present duplicate detection for complex data in the context of personal information management [7], where different kinds of entities such as conferences, authors, and publications are related to each other, giving a graph like structure. The algorithm propagates similarities of entity pairs through the graph. Any similarity score above a given threshold can trigger a new propagation of similarities, meaning that similarities for pairs of entities may be computed more than once. Although this improves effectiveness, efficiency is compromised. The domain-independent DogmatiX algorithm [8], which considers both effectiveness by defining a suited domain-independent similarity measure using information in ancestors and descendants of an XML element, and efficiency by defining a filter to prune comparisons. However, in the worst case, all pairs of elements need to be compared, unlike the sorted neighborhood method that has a lower complexity. The word presented in [23] which uses the sorted neighborhood method, which trades off effectiveness for higher efficiency by comparing only objects within a certain window.

However, most work based on similarity matrix to classify pairs of objects as duplicates or non-duplicates using an appropriate threshold value which may cause false positives.

III. MD5

MD5 stands for Message Digest 5, which is developed by Ron Rivest of the MIT Laboratory for Computer Science and RSA Data Security, Inc. MD5 is an algorithm which takes an input of any length and outputs a message digest of a fixed length. The generation of a digest is very fast and the digest itself is very small and can easily be encrypted and transmitted over the Internet. It is very easy and fast (and therefore cheap) to check some data for validity. The algorithms are well known and implemented in most major programming languages, so they can be used in almost all environments. MD5 uses the same algorithm every time. Hence it will always generate the same message digest for the string (data).

A. Message Padding

The message to be fed into the message digest computation must be a multiple of 512 bits (sixteen 32-bit words). The original message is padded by adding a 1 bit, followed by enough 0 bits to leave the message 64 bits less than a multiple of 512 bits. Fig 1 illustrates the padding for MD5.

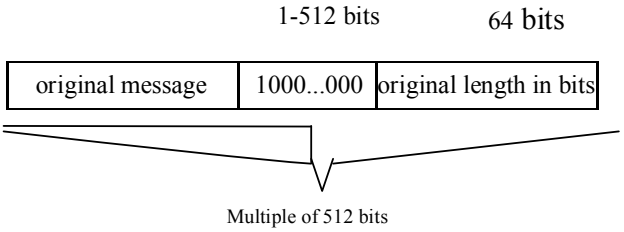

Figure 1. Padding for MD5

B. Overview of MD5 Computation

In MD5 the message is processed in 512-bit blocks (sixteen 32-bit words). (See Fig2.) The message digest is a 128-bit quantity (four 32-bit words). Each stage consists of computing a function based on the 512-bit message chunk and the message digest to produce a new intermediate value for the message digest. The value of the message digest is the result of the output of the final block of the message.

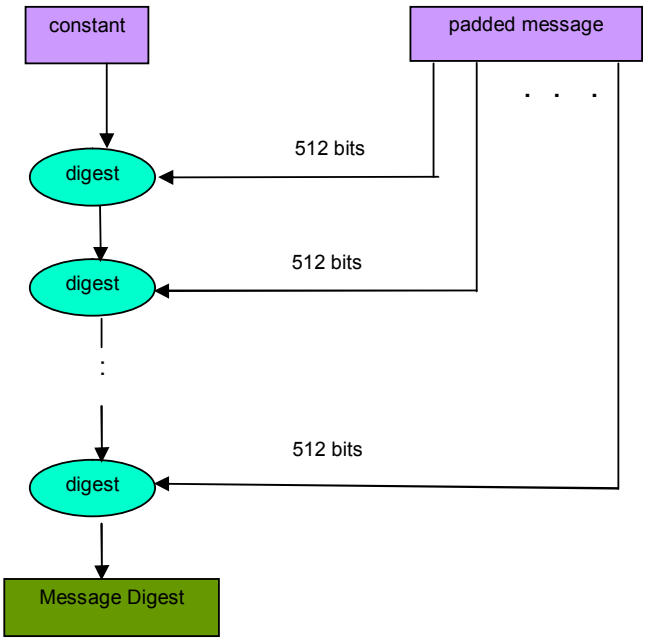

Figure 2. Overview of MD5.

C. MD5 Algorithm

MD5 consists of 64 of these operations, grouped in four rounds of 16 operations. F is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation.

$\lll_s$ denotes a left bit rotation by s places; s varies for each operation. $\boxplus$ denotes addition modulo 2^{32}.

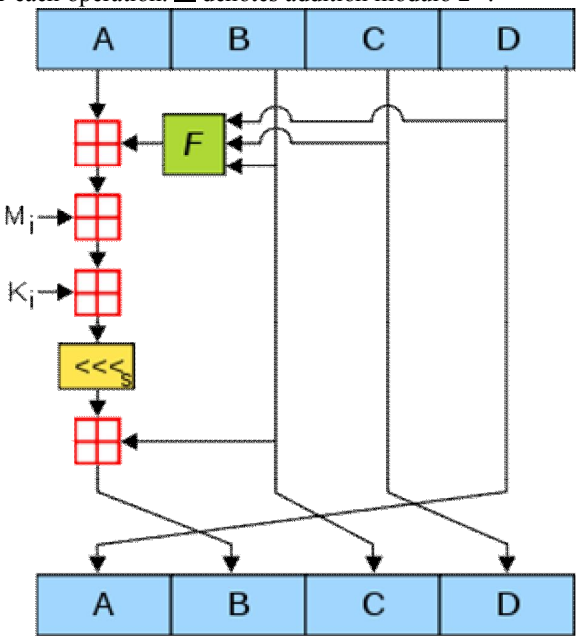

Figure 3.One MD5 operation

MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit little endian integers); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with a 64-bit integer representing the length of the original message, in bits.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted A, B, C and D. These are initialized to certain fixed constants. The main algorithm then operates on each 512-bit message block in turn, each block modifying the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function F, modular addition, and left rotation. Fig 3 illustrates one operation within a round. There are four possible functions F; a different one is used in each round:

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

$\oplus, \wedge, \vee, \neg$ denote the XOR, AND, OR and NOT operations respectively.

Pseudocode

Pseudocode for the MD5 algorithm follows.

```
//Note: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating
var int[64] r, k

//r specifies the per-round shift amounts
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21}

//Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) × (2 pow 32))

//Initialize variables:
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476

//Pre-processing:
append "1" bit to message
append "0" bits until message length in bits ≡ 448 (mod 512)
append bit /* bit, not byte */ length of unpadded message as
64-bit little-endian integer to message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
    break chunk into sixteen 32-bit little-endian words w[i], 0 ≤ i ≤ 15

    //Initialize hash value for this chunk:
    var int a := h0
    var int b := h1
    var int c := h2
    var int d := h3


  //Main loop:
    for i from 0 to 63
        if 0 ≤ i ≤ 15 then
            f := (b and c) or ((not b) and d)
            g := i
        else if 16 ≤ i ≤ 31
            f := (d and b) or ((not d) and c)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47
            f := b xor c xor d
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63
            f := c xor (b or (not d))
            g := (7×i) mod 16

        temp := d
        d := c
        c := b
```

```
        b := b + leftrotate((a + f + k[i] + w[g]) , r[i])
        a := temp

    //Add this chunk's hash to result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d

var int digest := h0 append h1 append h2 append h3
//(expressed as little-endian)
    //leftrotate function definition
    leftrotate (x, c)
      return (x << c) or (x >> (32-c));
```

D. MD5 hashes

The 128-bit (16-byte) MD5 hashes (also termed *message digests*) are typically represented as a sequence of 32 hexadecimal digits. The following demonstrates a 43-byte ASCII input and the corresponding MD5 hash:

MD5("The quick brown fox jumps over the lazy dog")
= 9e107d9d372bb6826bd81d3542a419d6

Even a small change in the message will (with overwhelming probability) result in a completely different hash, due to the avalanche effect. For example, adding a period to the end of the sentence:

MD5("The quick brown fox jumps over the lazy dog.")
= e4d909c290d0fb1ca068ffaddf22cbd0

The hash of the zero-length string is:

MD5("") = d41d8cd98f00b204e9800998ecf8427e

IV. Duplicate Detection

The duplicate detection algorithm is introduced in this section. Section A describes the overall architecture which consists of several phases, which are discussed in detail in the Sections B through C.

A. Outline of the Duplicate Detection

The architecture of duplicate detection is show in Fig 4. The system begins with the XML data and candidate definitions as input. The Selector module selects candidate objects from the XML documents using candidate definition. The candidate objects are then preprocessed to get standardized objects. The outputs of Preprocessor are stored in a relational database to identify duplicate objects. The Duplicate Identifier uses the output of Preprocessor to compute their corresponding hash codes by using MD5 algorithm. Finally, the duplicate objects can be identified according to their same hash values.

B. Candidate Definition

Defining which object data values actually describe an object, i.e., which values to consider when comparing two objects. It is based on three observations.

(i) A data source may store information about various types of real-world objects, not all of which need to be considered for duplicate detection.

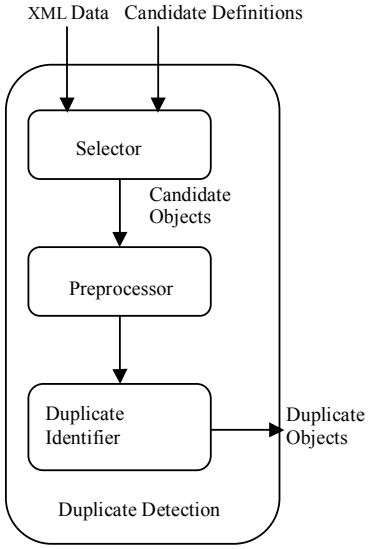

Figure 4. Duplicate Detection Architecture

(ii) Among the elements relevant to object identification, some may represent the same type of real-world object, just represented differently. These should be compared with each other.

(iii) On the other hand, it makes no sense to compare objects of different real-world type, as they cannot be duplicates of each other.

C. Data Preprocessing

The data preprocessing module includes a parsing, a data transformation step and a standardization step all of which are performed first for improving the quality of in-flow data and second for making the data comparable and more usable.

Parsing is the first critical component in the data preparation stage of record matching. This process locates, identifies and isolates individual data elements in the source files. Parsing makes it easier to correct, standardize, and match data because it allows comparing individual components, rather than long strings of data. For example, the appropriate parsing of name and address components is a crucial part in the duplicate detecting process.

Data transformation refers to simple conversions that can be applied to the data in order for them to conform to the types of their corresponding domains. The most common form of a simple transformation is the conversion of a data element from one data type to another.

Data standardization refers to the process of standardization the information represented in certain fields with some special content. This is used for information that can be stored in many different ways in different data sources and must be converted to a uniform presentation before the record matching process starts. One of the most common formatting needs is for address information. Without standardization, many true

matches could erroneously be designated as non-matches, based on the fact that common identifying information can not be compared.

V. Experimental Study

In the current state of our work, we have implemented a prototype to evaluate our proposal for duplicate detection in complex XML data. We used a Pentium(R) 4 computer with 3.20 GHz processor and 1GB main memory for the experiments. The experiments were tested using Java 1.6 software development kit. We performed some experiments with this prototype, by using a real world bibliography datasets such as DBLP and SIGMOD.

Then, we compared the elements from these document sets. We hope that the experiments produced good results, showing that our approach is capable of dealing with divergences in the documents' contents and also in their structures.

VI. Conclusion

In this paper, we presented an approach to detect duplicate objects in an XML document, which is an important data cleansing task necessary to improve data quality. Our approach overcomes the problems of element scope and structural diversity within an XML document, and efficiently identifies duplicate elements by using MD5 algorithm. We tend to address the optimality of our algorithm by comparing its effectiveness to other similarity measures when applied to XML and explore the automation of the selection of candidates, so that no domain-knowledge whatsoever is required.

REFERENCES

[1] Winkler, W.E.: Advanced methods for record linkage. Technical report, Statistical Research Division, U.S. Census Bureau, Washington, DC (1994).

[2] Hern´andez, M.A., Stolfo, S.J.: The merge/purge problem for large databases. In: SIGMOD Conference, San Jose, CA (1995) 127–138

[3] Lim, E.P., Srivastava, J., Prabhakar, S., Richardson, J.: Entity identification in database integration. In: ICDE Conference, Vienna, Austria (1993) 294–301

[4] Doan, A., Lu, Y., Lee, Y., Han, J.: Object matching for information integration: A profiler-based approach. IEEE Intelligent Systems, pages 54-59 (2003)

[5] Ananthakrishna, R., Chaudhuri, S., Ganti, V.: Eliminating fuzzy duplicates in data warehouses. In: International Conference on VLDB, Hong Kong, China (2002)

[6] Guha, S., Jagadish, H.V., Koudas, N., Srivastava, D., Yu, T.: Approximate XML joins. In: SIGMOD Conference, Madison, Wisconsin, USA (2002) 287–298

[7] Dong, X., Halevy, A., Madhavan, J.: Reference reconciliation in complex information spaces. In: SIGMOD Conference, Baltimore, MD (2005) 85–96

[8] Weis, M., Naumann, F.: DogmatiX Tracks down Duplicates in XML. In: SIGMOD Conference, Baltimore, MD (2005)

[9] Newcombe, H., Kennedy, J., Axford, S., James, A.: Automatic linkage of vital records. Science 130 (1959) no. 3381 (1959) 954–959

[10] Fellegi, I.P., Sunter, A.B.: A theory for record linkage. Journal of the American Statistical Association (1969)

[11] Jaro, M.A.: Probabilistic linkage of large public health data files. Statistics in Medicine **14** (1995) 491–498

[12] Quass, D., Starkey, P.: Record linkage for genealogical databases. In: KDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation, Washington, DC (2003) 40–42

[13] Hern´andez, M.A., Stolfo, S.J.: Real-world data is dirty: Data cleansing and the merge/purge problem. Data Mining and Knowledge Discovery **2(1)** (1998) 9–37

[14] Monge, A.E., Elkan, C.P.: An efficient domain-independent algorithm for detecting approximately duplicate database records. In: SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, Tuscon, AZ (1997) 23–29

[15] Kailing, K., Kriegel, H.P., Schnauer, S., Seidl, T.: Efficient similarity search for hierarchical data in large databases. International Conference on EDBT (2002) 676–693

[16] Carvalho, J.C., da Silva, A.S.: Finding similar identities among objects from multiple web sources. In: CIKM Workshop on Web Information and Data Management, New Orleans, Louisiana, USA (2003) 90–93

[17] Weis, M., Naumann, F.: Duplicate detection in XML. In: SIGMOD Workshop on Information Quality in Information Systems, Paris, France (2004) 10–19

[18] Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. In: Journal of Molecular Biology. Volume 147. (1981) 195–197

[19] Hern´andez, M.A.: A Generalization of Band Joins and The Merge/Purge Problem. PhD thesis, Columbia University, Department of Computer Science, New York (1996)

[20] Lehti, P., Fankhauser, P.: A precise blocking method for record linkage. In: International Conference on Data Warehousing and Knowledge Discovery (DaWaK, Copenhagen, Denmark (2005) 210–220

[21] Bilenko, M., Mooney, R.J.: Adaptive Duplicate Detection Using Learnable String Similarity Measures. The 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003), Washington DC, pp.39-48.

[22] Jin, L., Li, C., and Mehrotra, S.: Efficient record linkage in large data sets. In Proc. Of DASFAA, Kyoto, Japan(2003).

[23] Puhlmanu, S., Weis, M., and Naumann, F.: XML Duplicate Detection Using Sorted Neighborhoods. International conference on Extending Database Technology (EDBT), (2006)

[24] Elmagarmid, A., K.: Duplicate Record Detection: A Survey. IEEE Transactions on Knowledge and Data Engineering,Vol.19, No.1, January(2007)