XML Duplicate Detection Using Sorted Neighborhoods

Conference Paper *in* Lecture Notes in Computer Science · March 2006

DOI: 10.1007/11687238_46 · Source: DBLP

3 authors, including:

Melanie Herschel
Universität Stuttgart
54 PUBLICATIONS **1,416** CITATIONS
SEE PROFILE

Felix Naumann
Hasso Plattner Institute
272 PUBLICATIONS **6,766** CITATIONS
SEE PROFILE

XML Duplicate Detection using Sorted Neighborhoods

Sven Puhlmann, Melanie Weis, and Felix Naumann

Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
sven.puhlmann@alumni.hu-berlin.de,
{mweis,naumann}@informatik.hu-berlin.de

Abstract. Detecting duplicates is a problem with a long tradition in many domains, such as customer relationship management and data warehousing. The problem is twofold: First define a suitable similarity measure, and second efficiently apply the measure to all pairs of objects. With the advent and pervasion of the XML data model, it is necessary to find new similarity measures and to develop efficient methods to detect duplicate elements in nested XML data.
A classical approach to duplicate detection in (flat) relational data is the sorted neighborhood method, which draws its efficiency from sliding a window over the relation and comparing only tuples within that window. We extend the algorithm to cover not only a single relation but nested XML elements. To compare objects we make use of XML parent and child relationships. For efficiency, we apply the windowing technique in a bottom-up fashion, detecting duplicates at each level of the XML hierarchy. Experiments show a speedup comparable to the original method and the high effectiveness of our algorithm in detecting XML duplicates.

1 Introduction

The problem of duplicate detection has been considered under many different names, such as record linkage[1], merge/purge[2], entity identification [3], and object matching [4]. It generally addresses the problem of finding different representations of a same real-world object, which we refer to as duplicates. Various representations are due to errors, such as typographical errors, inconsistent representations, synonyms, and missing data. Examples for applications where data cleansing and hence duplicate detection are a necessary (pre)processing step include data mining, data warehouses, and customer relationship management. Another scenario where duplicates naturally occur and need to be identified is data integration, where data from distributed and heterogeneous data sources are combined to a unique, complete, and correct representation for every real-world object.

Most approaches address the problem for data stored in a single relation, where a tuple represents an object and duplicate detection is performed by comparing tuples. In most cases however, data is stored in more complex schemata,

e.g., in relational database tables that are related through foreign key constraints, or, in the case of XML data, as elements that are related through nesting. Only recently has duplicate detection been considered for data other than an instance of a single relation [5–8].

The work presented in this paper focuses on duplicate detection in XML. More specifically, we present an approach that adapts the sorted neighborhood method (SNM)—a very efficient approach for duplicate detection in a single relation—to complex XML data consisting of several types of objects related to each other through nesting. Similarly to the relational SNM, we compare XML elements describing the same type of object. First, our XML adaptation to SNM, called SXNM, generates a key for every element subject to comparisons in the XML data source. This phase is referred to as *key generation*. In the second phase, namely the *duplicate detection* phase, the elements are sorted using these keys and a sliding window is applied over the sorted elements. Assuming that the key-order places duplicates close to one another, we drastically improve efficiency while maintaining good effectiveness by comparing elements within the window. Relationships between different types of objects are exploited by our similarity measure, which considers duplicates among descendants, in addition to the information defined manually to describe the particular object (a so called *object description*). Therefore, we compare XML elements in a bottom-up fashion. Experiments show that SXNM is an effective and efficient algorithm for duplicate detection in XML data.

The remainder of this paper is structured as follows: In Sec. 2, we describe related work with a special focus on the relational sorted neighborhood method. Section 3 describes how sorted neighborhoods are used in XML. In Sec. 4, we show results of evaluating our algorithm. To conclude, we provide a summary of the paper and directions for further research in Sec. 5.

2 Related Work

2.1 Duplicate detection

The problem of duplicate detection, originally defined by Newcombe [9] and formalized in the Fellegi-Sunter model for record linkage [10] 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 [3, 14].

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. This top-down approach, however, is not well-suited for 1:N parent-child relationships. As an example, let us consider `<movie>`

elements nesting `<actor>` elements. The top-down approach prunes comparisons of children not having the same ancestors, an assumption that misses duplicates for an M:N relationship between parent and child such as movie and actor, because an actor can play in several different movies. Our bottom-up approach overcomes these issues. In the example of movies nesting actors, we first compare actors independently of movies, and then compare movies also considering duplicates found in actors. Consequently, duplicate actors can play in different movies whereas duplicate movies are detected through co-occurring actors. We compensate the additional comparisons by using sorted neighborhoods.

Work presented in [6, 15] describes efficient identification of similar hierarchical data, but it does not describe how effective the approaches are for XML duplicate detection. At the other end of the spectrum, we have approaches that consider effectiveness (in terms of recall and precision) [7, 16, 17]. 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. In [8], we presented the domain-independent DogmatiX algorithm, 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.

2.2 The Sorted Neighborhood Method

The Sorted Neighborhood Method (SNM) is a well known algorithm for the efficient detection of duplicates in relational data [13]. We describe it in detail, as the method introduced in this paper is based on SNM. Given a relation with possibly duplicate tuples, the algorithm consists of three main steps:

1. *Key Generation*: For each tuple in the relation a key is extracted according to a given key definition specified by an expert. Normally a generated key is a string consisting of concatenated parts of the tuple's contents. Each key is linked with a reference to its tuple. Consider a relation `MOVIE(TITLE, YEAR)` and let a tuple of the relation be `Mask of Zorro, 1998`. Let a key be defined as the first four consonants of the title and the third and fourth digit of the year. Then, the key value for the sample tuple movie is `MSKF98` (underlined characters).
2. *Sorting*: The keys generated in Step 1 are sorted lexicographically.
3. *Duplicate Detection*: A window of fixed size slides over the sorted keys and searches duplicates only among the tuples referenced in the window, thus limiting the number of comparisons. The size of the window is crucial for

the effectiveness of the algorithm and the quality of the result. With a small window only a small set of elements are compared, leading to a relatively fast duplicate detection, though with possibly poor recall. A large window results in a slower algorithm, but the chance to find duplicates is better as more comparisons are performed.

To compare tuples referenced by keys in the window SNM uses an equational theory combined with a similarity measure. The equational theory defines under which circumstances two tuples are considered duplicates, e.g., if a person's name and address are sufficiently similar.

Using the transitive closure on the detected duplicates increases the number of duplicates found. Moreover, the multi-pass method, which executes SNM several times with different keys, significantly increases recall [13]. For large amounts of data as well as for repeatedly updated data there exists an incremental version of the method dealing with how to combine data that have already been deduplicated with new data packets. The basic SNM is very effective for duplicate detection in relational data and achieves high recall and precision values. Next, we adapt the method to XML.

3 SXNM – The Sorted XML Neighborhood Method

We apply the idea of the SNM to nested XML data and call our algorithm the Sorted XML Neighborhood Method (SXNM). It consists of two independent phases: The *key generation* and the *duplicate detection* phase. Figure 1 shows the basic workflow of SXNM with its two phases. The key generation algorithm uses the XML data source and some configuration as its input and returns the generated keys. Note that our algorithm assumes that the XML data has a common schema. That is, elements having the same XPath represent the same type of object, and elements with different XPath have different object types and are not compared. This assumption can be satisfied by applying schema matching and data integration into a common target schema prior to SXNM. During the duplicate detection phase, elements are sorted according to their generated keys and a sliding window is applied over the elements, possibly using multiple passes if multiple keys have been defined. To detect duplicates for every element in the document, which is traversed in a bottom-up fashion, information about previously detected duplicates, i.e., duplicates in descendants, is used. Details about each step are provided in separate subsections as indicated in Fig. 1, but first, we illustrate our approach with an example.

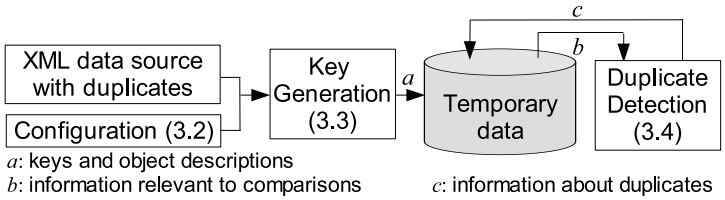
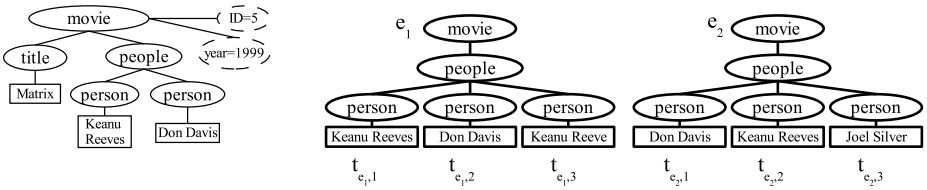


Fig. 1. The SXNM workflow (with section numbers).

3.1 Illustrative Example

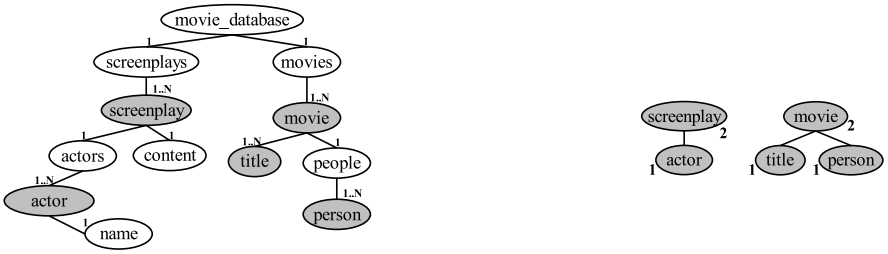
As input SXNM requires some configuration in addition to the XML data on which duplicate detection is applied. The configuration includes (i) the definition of what object types are subject to deduplication, so called *candidates*, (ii) the definition of what data describes an object, its *object description (OD)*, and (iii) the definition of keys. To illustrate the configuration, we consider the `<movie>` element in Fig. 2(a). The ellipses, rectangles, and dashed ellipses depict XML elements, text nodes, and attributes, respectively. `Matrix`, the `<title>` content of the `<movie>` element, is referenced by the relative path `title/text()`, the text node of the `<title>` child. Further relative paths might include `people/person[1]/text()` and `@year`.



(a) A single movie element. (b) Two movies with children.

Fig. 2. Two examples for XML data.

In the configuration, which is itself an XML document, we define all candidates using relative paths. For the bottom-up traversal, the algorithm considers only the subtrees consisting of candidates. Consider a simplified structure of an XML data source depicted in Fig. 3(a). Candidates are shaded, and for only these are keys and ODs defined as we will see shortly. The numbers and ranges at the elements indicate the possible number of children. Note that for the XML elements `<actor>`, `<title>`, and `<person>` only the object descriptions can be used for duplicate detection, whereas for the XML elements `<screenplay>` and `<movie>` information about duplicates in descendants can be used additionally. Fig. 3(b) shows the subtrees consisting of candidates extracted only from the XML structure in Fig. 3(a). The numbers at the elements indicate the order in which the duplicate detection is executed.



(a) XML tree structure. (b) Candidate trees.

Fig. 3. XML candidates in an XML data source (a) and extracted subtrees (b).

For every candidate the object description and key are defined as exemplarily shown in Tab. 1 for the particular candidate `<movie>`. The tables hold the defin-

ition for two keys for `<movie>` elements (relations $KEY_{movie,1}$ and $KEY_{movie,2}$) and the object descriptions definition (in OD_{movie}). In the *pattern* attribute of the key relations `K`, `C`, and `D` stand for consonants, characters, and digits respectively. The number after the character type indicates its position in the text value of the relative path referenced by the *pid* attribute and stored in *relPath*. For example, the first key for `<movie>` elements uses relative paths 1 and 3, which are `title/text()` and `@year`. From the text value of path 1, the key definition defines a key consisting of the first two consonants from path 1 concatenated with the third and fourth digit from path 3. Applying both key definitions to the `<movie>` element of Fig. 2(a) we obtain the keys `MT99` and `5MA`.

Table 1. Relations defining the keys and object descriptions for `<movie>` elements.

(a) $PATH_{movie}$

id	*relPath*
1	title/text()
2	@ID
3	@year

(b) OD_{movie}

pid	*relevance*
1	0.8
3	0.2

(c) $KEY_{movie,1}$

pid	*order*	*pattern*
1	1	K1,K2
3	2	D3,D4

(d) $KEY_{movie,2}$

pid	*order*	*pattern*
2	1	D1
1	2	C1,C2

Using all these definitions provided in the configuration, we begin key generation. To save an extra pass of the XML data, we simultaneously extract the object descriptions that are necessary for the second phase. The result of the first phase is a temporary relation GK (Generated Keys) for every candidate storing the generated keys as well as the corresponding object descriptions. For example, using the definitions for the movie candidate in Tab. 1, we obtain GK_{movie} shown in Tab. 2(a). The sample tuple describes the movie of Fig. 2(a). Further tuples represent other movies stored in the XML document.

Table 2. Temporary tables used by SXNM

(a) Subset of the GK_{movie} relation.

elID	key_1	key_2	od_1	od_2
1	MT99	5MA	Matrix	1999
...	...	...	...	...

(b) Several clusters in CS_{person}.

cluster ID	`<person>` elements
1	$\{t_{e_1,1}, t_{e_1,3}, t_{e_2,2}\}$
4	$\{t_{e_1,2}, t_{e_2,1}\}$
8	$\{t_{e_2,3}\}$

During each pass of the duplicate detection phase (there is one pass for every defined key), GK is sorted according to a key. A sliding window is then applied over the sorted table, and pairs of tuples within the same window are compared using a similarity measure. The similarity measure is a combination of the similarity of object descriptions and the similarity of children sets, if applicable. The similarity measure is defined in Sec. 3.4. By applying the transitive closure over duplicate pairs over all passes, we obtain clusters of duplicates. For every candidate, cluster sets are stored in a temporary table CS together with information that can be used for computing similarities of ancestors. As an example, consider the `<movie>` elements e_1 and e_2 in Fig. 2(b). Information about duplicates in `<person>` elements helps to detect duplicates in `<movie>` elements. As the result of duplicate detection in `<person>` elements, Tab. 2(b) shows clusters in CS_{person}. We observe that e_1 and e_2 have two actors in common, namely `Keanu`

`Reeves` and `Don Davis`. Consequently, we conclude that e_1 and e_2 are similar enough based on children data to be duplicates.

For every candidate, the result of duplicate detection can be retrieved from the corresponding CS table for further processing. The following sections provide formal definitions and descriptions for every phase of Fig. 1.

3.2 Configuration

In addition to the XML data to be deduplicated, our algorithm requires some configuration. The configuration contains information about

- *Candidates*: the XML elements for which duplicates should be detected, and which therefore need a generated key.
- *Object description*: which information (text elements) of candidates is used for comparisons.
- *Key definition*: which information (text elements) of candidates is used to generate keys.
- *Key patterns*: which parts of this information comprise the keys.

To distinguish specific XML elements and their corresponding XML schema elements in the following, we use s as an element in an XML schema and e as an instance of s.

Candidates are specified by their absolute XPath and are given a unique name, which is required to associate configuration tables with temporary tables. For example, the `<movie>` candidate of Fig. 3(a) is specified with the XPath `movie_database/movies/movie` and is assigned *name = movie*. To specify information necessary for key generation and duplicate detection for a single candidate, we use *relative paths (relPath)*, i.e., XPath structures relative to the candidate. Relative paths identify text nodes or attribute values that belong to either the key or the object description of the candidate.

We construct separate relations for paths, keys, and object descriptions relevant for comparisons of instances of an XML schema element s. We show examples in Table 1.

- $PATH_s(id, relPath)$ is the *path relation* containing all relative paths that refer to information of an XML element, used for key definitions and object descriptions. The *id* attribute contains a unique id of the relative path.
- The relation $KEY_{s,i}(pid, order, pattern)$ defines the i^{th} key of s. The *pid* attribute is a foreign key to *id* in $PATH_s$ and refers to the relative path of the information that build parts of the key. The *order* attribute indicates the position of the information in the key, and *pattern* describes what characters to extract from a description referenced by the relative path. Note that we allow an arbitrary number of keys for each relevant XML element, enabling the use of the multi-pass variant of SNM [13].
- $OD_s(pid, relevance)$ is the *object description relation (OD relation)* indicating the information that is compared between two instances of s. The *relevance* attribute constitutes the relevancy (weighting) of the information, which is used by our similarity measure, and *pid* references the *id* of $PATH_s$.

In summary, several parameters are needed in conjunction with the XML data source to provide input for the key generation algorithm. For an XML schema element s, its parameters $P_s = \{PATH_s, OD_s, KEY_{s,1}, \ldots, KEY_{s,n}\}$ (where n is the number of keys defined for s) contain all relations needed for the key generation algorithm. For the set S of all XML schema elements for which definitions are made, the *parameter set* $P = \bigcup_{s \in S}\{P_s\}$ denotes the complete set of parameters.

3.3 Key Generation

P provides the input for the key generation algorithm. Whilst this task for the original SNM was to extract only the keys, the key generation algorithm of SXNM extracts the keys as well as the object descriptions needed for comparisons, reading the given XML data in a single pass. The result of the key generation algorithm is GK, the set of generated keys, again stored in a relation. Let S be the set of all XML schema elements, for whose instances duplicates should be detected, and $s \in S$. The relation $GK_s = (eid, key_1, \ldots, key_n, od_1, \ldots, od_n)$ denotes the result of the key generation. The attribute *eid* contains the ID of the respective XML element—for instance the position of the element in the data source; $key_1, \ldots, key_n$ and $od_1, \ldots, od_n$ contain the keys generated for this XML element and the extracted object descriptions respectively. $GK = \bigcup_{s \in S}\{GK_s\}$ denotes the combination of all generated keys.

3.4 Duplicate Detection

In the duplicate detection step the generated keys in GK are processed. Along with GK, the duplicate detection process takes several parameters:

- The parameter set P containing object description and their relevancies,
- the window sizes to use for the XML elements,
- thresholds needed to classify XML elements as duplicates and non-duplicates,
- information about when not to use descendants for duplicate detection.

We expect that a domain expert is able to set these parameters. We experienced that performing duplicate detection both manually and automatically on a small sample can help determine suitable parameters values.

As the main idea of SXNM is to use information about duplicates in descendants, the order in which candidates are processed hast to be defined accordingly. In the following, we start with a description of how duplicates for a single candidate are detected. Thereafter we describe the order in which candidates are processed.

The general duplicate detection process. For each key attribute in GK_s, e.g., key_1, the GK_s relation is sorted according to the appropriate key attribute. A sliding window of a specified size w_s slides over the tuples in the sorted relation, in analogy to the window in the relational SNM. For each pair of tuples in the window, a similarity is computed based on their object description and descendants, if available. If the similarity exceeds a given threshold, the corresponding XML elements are classified as duplicates. The result of this multi-pass

method executed for s is a set of element ID pairs that represent duplicates. A transitive closure algorithm is applied to the duplicates, resulting in the cluster set CS_s.

Definition 1 (Cluster Set). *Let s be an element of an XML schema. $CS_s = \{C_1, \ldots, C_m\}$, $m \leq n$ is a* cluster set *where each cluster represents a real-world object o, holds a unique cluster ID, and contains references to all XML data instances of s represented by o. Each instance of s belongs to exactly one cluster of the cluster set.*

A cluster set is created for every candidate XML schema element. The cluster sets can then be used to create a de-duplicated version of the XML data source. Moreover, cluster sets help to detect and verify duplicates in other XML elements, using a bottom-up duplicate detection process.

Bottom-up duplicate detection. In SXNM, the similarity of two XML elements can consist of (i) the similarity of their object descriptions (Def. 2) and (ii) the similarity of their descendants (Def. 3).

Using information about key elements stored in P, the tree structure of the entire XML document can be split into a set of trees by extracting all elements s ($P_s \in P$) from the XML document and preserving the ancestor-descendant relationships. This was demonstrated in Figure 3. We need this tree set structure to execute duplicate detection in a bottom-up fashion. The duplicate detection process as described above can be executed on an extracted tree independently from other extracted trees. For each tree, the process starts with the nodes having the largest distance δ to the root node. It continues with the nodes having distance $\delta - 1$ etc. up to the root node.

Lacking descendants of their own, the similarity of elements that are instances of the XML schema elements (represented by the leaf nodes of the extracted tree structures) is based on the similarity of their object descriptions alone. This is true also for other schema elements, for which the expert decided that descendants should not be taken into account during duplicate detection.

Definition 2 (OD Similarity). *Let e_1 and e_2 be two instances of schema element s occurring together in a sliding window. Let OD_s contain n entries $od_{e_j,1}, \ldots, od_{e_j,n}$; r_i indicates the relevancy of path i as defined in the OD_s relation. With ϕ_i^{OD} being a similarity function for the i^{th} entry in OD_s, the similarity of the object descriptions of e_1 and e_2 is $sim_{e_1,e_2}^{OD} = \sum_{i=1}^{n} r_i \phi_i^{OD}(od_{e_1,i}, od_{e_2,i})$.*

An example for a ϕ^{OD} function is the edit distance [18], with computes the minimum number of operations needed to convert one string into another. Using domain-knowledge, more accurate ϕ^{OD} functions can be used, e.g., a numeric distance function for numerical values.

Except for XML elements that are leaf nodes where only the object description is available, duplicate detection for XML elements can be performed using the similarity of their object descriptions and their descendants. As an XML element can have descendants of several types, we start with the similarity of

individual descendants and combine the similarities of all descendants of this XML element thereafter.

For two instances e_1 and e_2 of an XML schema element s having a descendant schema element t, $t_{e_j,i}$ denotes the i-th instance of t descendant of e_j ($j \in \{1,2\}$). As our duplicate detection is a bottom-up process, duplicates in the instances of t have already been detected, leading to the cluster set CS_t, which helps to detect whether e_1 and e_2 are duplicates. The function cid returns the unique cluster ID of a cluster in a cluster set (cf. Def. 1), given a cluster set and an instance of an element in the cluster set. Using cid we define lists of cluster IDs for e_1 and e_2 and with them the descendant-based similarity of two elements:

$$l_{e_1} = (\text{cid}(t_{e_1,1}, CS_t), \ldots, \text{cid}(t_{e_1,i}, CS_t)) = (id_1, \ldots, id_i)$$
$$l_{e_2} = (\text{cid}(t_{e_2,1}, CS_t), \ldots, \text{cid}(t_{e_2,j}, CS_t)) = (id_1, \ldots, id_j)$$

Definition 3 (Descendants Similarity). *The similarity of two instances e_1 and e_2 of an XML schema element s regarding a single descendant schema element t is calculated using the ϕ_t^{desc} function: $sim_{e_1,e_2,t}^{desc} = \phi_t^{desc}(l_{e_1}, l_{e_2})$.*

Let $t_1, \ldots, t_n$ be the n descendant schema elements of s. We use $agg()$ to obtain the combined similarity sim_{e_1,e_2}^{Desc} for all instances of the descendants of e_1 and e_2: $sim_{e_1,e_2}^{Desc} = agg(sim_{e_1,e_2,t_1}^{desc}, \ldots, sim_{e_1,e_2,t_n}^{desc})$

There are numerous possibilities for the ϕ^{desc} and $agg()$ functions. One example for the first would be to calculate the ratio between the cardinalities of the intersection and the union of l_{e_1} and l_{e_2}; this is our current implementation. The $agg()$ function could simply calculate the average of its arguments, or it could weigh the importance of different descendants. Currently, we calculate the average; future implementations will have declarations of different weights in the configuration.

Consider the `<movie>` elements e_1 and e_2 in Fig. 2(b). Information about duplicates in `<person>` elements helps to detect duplicates in `<movie>` elements. As the result of duplicate detection in `<person>` elements, Tab. 2(b) shows clusters in CS_t, $t = person$. This leads to $l_{e_1} = (\text{cid}(t_{e_1,1}, CS_t), \text{cid}(t_{e_1,2}, CS_t), \text{cid}(t_{e_1,3}, CS_t)) = (1, 4, 1)$ and $l_{e_2} = (4, 1, 8)$. Using the similarity function proposed above we have $sim_{e_1,e_2,t}^{desc} = \frac{|l_{e_1} \cap l_{e_2}|}{|l_{e_1} \cup l_{e_2}|} = \frac{2}{3}$.

Finally, to gain the resulting similarity for the XML elements e_1 and e_2 of the same schema element s we combine sim_{e_1,e_2}^{OD} and sim_{e_1,e_2}^{Desc}. The result is sim_{e_1,e_2}^{comb}, reflecting the final combined similarity of both XML elements. An example for calculating the combined similarity is to weigh sim_{e_1,e_2}^{OD} and sim_{e_1,e_2}^{Desc} to gain sim_{e_1,e_2}^{comb}. Our current implementation calculates the average of the two values.

Having executed the duplicate detection process for all instances of the defined XML schema elements, we have a resulting cluster set for each of these schema elements. What to do with this information remains up to the domain specific application. A typical approach selects a *prime representative* for each cluster and discards the others. More sophisticated approaches perform data fusion by resolving conflicts among the different representations.

4 Evaluation

In this section we present various experimental results of SXNM and show that this method is ready to detect duplicates in complex, large, and nested XML data structures.

4.1 Data Sets

We use three different data sets for our experiments—both artificial and real-world XML data. To generate artificial data, we use two tools consecutively: The first is ToXGene[1], which, using a template similar to an XML schema, generates clean XML data sets. We assign an unique ID to the data objects for identification. The second tool is the Dirty XML Data Generator[2]. It uses the clean XML data and some parameters, e.g., the duplication probability, the number of duplicates, and the errors to introduce into the duplicates, as its input and generates dirty XML data according to the parameters. To observe the recall, precision, and f-measure values we use the unique IDs of the clean data objects. Of course these IDs are not made available to SXNM. The data sets are described below. When not specified, the OD of a candidate is its text node with relative path `text()` and relevance 1. Key definitions used in our experiments are provided in Tab. 3.

Table 3. Configurations for the three data sets

(a) Data set 1 (art. movies)

candidate	*key relPath*	*pattern*
movie	title/text()	K1-K5
	@year	D3,D4
	@length	D1,D2
	title/text()	K1,K2
	@genre	C1,C2
	title/text()	K1-K4

(b) Data set 2 (CDs)

candidate	*key relPath*	*pattern*
disc	artist[1]/text()	K1-K4
	year/text()	D3,D4
	did/text()	C1-C4
	dtitle[1]/text()	C1-C4
	genre/text()	C1,C2
	year/text()	D3,D4
	artist[1]/text()	K1,K2
	did/text()	C1,C2
disc/tracks/title	text()	C1-C6

(c) Data set 3 (real-world movies)

candidate	*key relPath*	*pattern*
disc	dtitle[1]/text()	K1-K6
	artist[1]/text()	K1-K4
	did/text()	C1-C4
	dtitle[1]/text()	C1-C4
disc/dtitle	text()	C1-C6
disc/artist	text()	C1-C6
disc/tracks/title	text()	C1-C6

Data set 1: Artificial movie data. We generate various data sets of different sizes consisting of artificially generated `<movie>` data using ToXGene and the Dirty XML Data Generator. The resulting `<movie>` elements in the data sets contain several `<title>`, `<person>`, and `<review>` descendants. The `<person>` elements can contain one `<lastname>` and several `<firstname>` elements. A

[1] http://www.cs.toronto.edu/tox/toxgene/
[2] http://www.informatik.hu-berlin.de/mac/dirtyxml/

<movie> element has two attributes, namely year and length. As a candidate, we consider the movie schema element only. As its object description, we use title/text() and @length with respective relevancies 0.8 and 0.2.

Data set 2: Real-world CD data, artificially polluted. Here we use real-world CD data consisting of 500 clean CD objects extracted from the FreeDB data set[3] and 500 artificially generated duplicates (one duplicate for each CD; using the Dirty XML Data Generator) as a test data set. Each <disc> element contains several <title> descendants nested under a <tracks> element and at least one <artist> and <dtitle>. Optional children of <disc> are <year>, <did>, a disc id that FreeDB provides and <genre>. As candidates, we use disc schema elements and their descendant /tracks/title. The object description of a disc consists of did/text(), artist/text() and dtitle/text() with respective relevancies of 0.4, 0.3, and 0.3.

Data set 3: Real-world movie data. For precision tests of larger bodies of XML data we use real-world movie data consisting of 10,000 CDs selected from FreeDB. Having the same schema as Data set 2, the candidates are disc, disc/title, disc/artist and disc/tracks/title.

4.2 Experimental Results

We now show the results of a variety of experiments. In the first set of experiments we examine SXNM in terms of recall, precision, and f-measure. The second set of experiments deals with the scalability of our duplicate detection method. Finally, in the third set of experiments we show how and when duplicates in descendants help to detect duplicates in higher levels of the hierarchy. For all experiments we only show a selection of the result graphs.

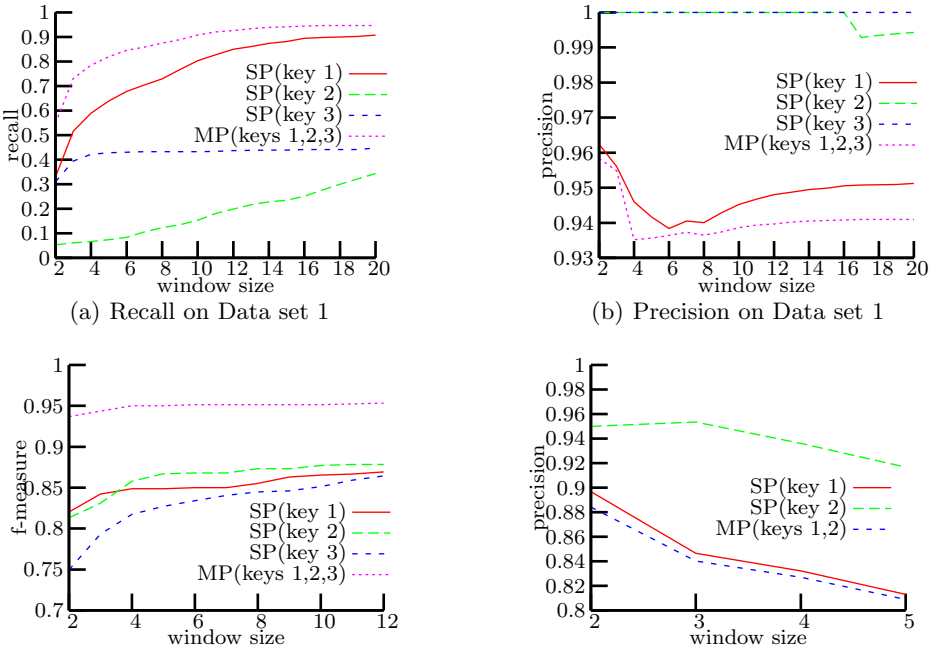
Experiment set 1: Single- vs. Multi-Pass with varying window sizes.

Purpose: In these experiments we show the overall effectiveness of our method by examining recall, precision, and f-measure results on different data sets. We use varying window sizes and different keys. Moreover, the advantage of the multi-pass vs. the single-pass method is shown.

Methodology: We use all three data sets for this experiment. For Data sets 1 and 2, we can evaluate recall and precision and therefore calculate the f-measure, because we know the true duplicates in these data sets. This is not the case for Data set 3, for which we determine only precision. For all experiments in this subsection, we used threshold values that we consider sensible based on our experience. Results are shown in Figure 4. Each line in a graph represents the use of a different key in single-pass SXNM (SP), or the combination of all keys for the multi-pass SXNM (MP).

Discussion Data set 1: The results for this experiment on artificial movies can be seen in Figures 4(a) and 4(b). We used the three different keys shown in Tab. 3(a). As an example of how to read the table, consider the first key defined for the relative path "title/text()" as "K1-K5". The relative path of the key definition points to the movie's title, from which the first five consonants are used as key.

[3] http://www.freedb.de



(a) Recall on Data set 1

(b) Precision on Data set 1

(c) F-measure on Data set 2

(d) Precision on Data set 3

Fig. 4. Results for experiment set 1 on effectiveness.

We can see in Fig. 4(a) that for the single keys as well as for the multi-pass method the recall increases with increasing window size. The individual keys lead to very different results. In terms of recall, Key 2 leads to the worst results. This is explained by the fact that the first part of this key consists of the year of the movie, which results in poorly sorted keys when the year is missing or contains severe errors. With increasing window size, more pairs are compared and more duplicates are found, which increases the recall of Key 2, while precision is not considerably compromised. The same argument can be used to explain the results of Key 3, however, the development with increasing window size is not as pronounced as for Key 2.

Key 1 performs best and almost as good as the multi-pass method which uses all keys. This is because the first five consonants of a movie's title are very distinguishing and lead to a very good order after sorting. The precision curve of Key 1 (and consequently of MP) in Figure 4(b) shows an at first surprising decrease in precision for small window-sizes (2-6) and increases afterwards to converge to a precision around 0.95. This can be explained as follows: Because we introduced artificial errors in titles that constitute the key, the good order is compromised. Indeed, 5% of the titles were polluted in such a way that their keys are sorted far apart. These duplicates are not detected for small window sizes but can be found with larger windows. We observe that with increasing window size, the precision converges to 0.95. In fact, the precision for large window sizes converges to the precision the similarity obtains when comparing all pairs. In terms of recall, the multi-pass method performs best (as already shown in [13]) but not much better SP for Key 1, because Keys 2 and 3 do not increase the

number of detected duplicates much (low recall values). However, in terms of precision the multi-pass method performs worst (although overall the values are still high between 0.93 and 0.96). This is because the multi-pass method executes the largest number of comparisons and there is an increased probability of false positives.

Discussion Data set 2: We discuss the results on Data set 2 for the disc candidate only, using three different keys shown in Tab. 3(b). Figure 4(c) shows the result of this experiment in terms of f-measure. The individual keys perform in a similar range between 0.75 and 0.87. Key 3 leads to the worst results because genre and year are not very distinctive attributes (same reason as for Key 2 in Data set 1). Key 1 yields better results than Key 3 because an artist's name is more distinctive than the genre. Key 2 consists of the first characters of the CD's ID, which in only some cases is incorrect and missing and therefore leads altogether to the best results. The multi-pass method results show that even the smallest window size (2) leads to much better results in terms of f-measure than the largest tested window size of 12 for the single keys. Larger windows give only slight improvement, so in this case window size 4 is sufficient. For all keys (single-pass and multi-pass) we observe that the f-measure increases with increasing window size. This is explained by the fact that with increasing window size, the recall increases because more pairs are compared. At the same time, the precision settles at large window sizes because its degradation is limited by the similarity measure.

Discussion Data set 3: In Fig. 4(d) we show precision for different window sizes obtained using SXNM on 10,000 disc candidates and the keys of Tab. 3(c). Recall could not be measured, because we do not know the true duplicates in this data set. We observe that Key 2, which is the same as Key 2 used on Data set 2 again yields the highest precision. At window size 5, we detect 48 duplicates. Key 1 results in a lower precision but detects far more duplicates, e.g., at window size 5, it finds 289 duplicates. Using multi-pass SXNM with both Key 1 and Key 2 results in the worst precision, because the false positives of both keys are cumulated. In this real-world data set, we observe that most duplicate clusters consist of two elements only, and that our algorithm detects false duplicates mainly due to two reasons. Between 54% and 77% (decreasing with increasing window size) of false duplicates are pairs of CDs that are part of a series and differ in a single number only, e.g., `Christmas Songs (CD1)` and `Christmas Songs (CD2)` or that feature various artists (the two cases are often correlated). Between 19% and 36% (increasing with increasing window size) of false duplicates are CDs whose text is provided in a format that failed to enter the database (e.g., Japanese or Russian). Comparisons were then only performed on "readable" attributes (year and genre). For any window size, less that 10% of false duplicates are due to other reasons.

To summarize the experiments for all three data sets, SXNM achieves overall high precision and recall, comparable to or exceeding related approaches. Also as expected, the multipass method outperforms the single-pass method. Finally, the choice of good keys is of course very decisive to achieve good results.

Experiment set 2: Scalability.

PURPOSE: In the second series of experiments, we show how the individual phases of SXNM scale with the amount of data and the number of duplicates. The distinguished phases are key generation (KG), sliding window (SW), transitive closure (TC) as currently provided in our implementation, and overall duplicate detection (DD), which is the sum of the SW and TC.

METHODOLOGY: We use artificially generated movie data so we can generate data sets with different sizes and numbers of duplicates. With ToXgene we generated 9 XML files containing from 100 to 2000 movies. Each movie has one to three title- and three to ten person-descendants. Using the Dirty XML Data Generator, we polluted the clean movie data using two configurations with different duplication probabilities (dupProb) and different numbers of duplicates for `<movie>` and `<person>` elements to obtain two different pollution degrees:

- *few duplicates*: 20% dupProb for `<movie>`, `<title>`, and `<person>` elements each producing exactly one duplicate.
- *many duplicates*: 100% dupProb for `<movie>` and `<person>`, each generating up to two duplicates, and 20% dupProb for `<title>` elements each generating exactly one duplicate object.

We polluted the text nodes of the duplicate elements by deleting, inserting, or swapping characters as described for the Dirty XML Data Generator. The window size used in these experiments is 3.

DISCUSSION: Figure 5 shows the results for this series of experiments. To enable a comparison to the clean XML data resulting from ToXGene, we executed SXNM over the set of clean movie data to show the difference to the least possible time needed to detect duplicates in a specific data set (Fig. 5(a)). In all graphs, the *duplicate detection* time DD is the sum of the *transitive closure* TC and the *sliding window* SW time. SXNM's comparisons are performed in SW.

The overall time (duplicate detection) needed for the largest clean data set is 129 s. Although there are no duplicates and the transitive closure algorithm is expected to need almost no time, there is an increasing possibility for false duplicates when the file size increases, leading to some pairs of duplicates for which the transitive closure algorithm is executed. Altogether, the key generation is a linear process, compared to the comparisons in the sliding window, which is polynomial.

We can also see from Figures 5(a) and 5(b) that the duplicate detection for the file with "few duplicates" performs almost as well as for the clean data. Looking on Fig. 5(d), which shows the time overhead of the sum of key generation and sliding window for both few and many duplicates, compared to the time needed on clean data, we observe a time overhead of below 20% for few duplicates. For the file with "many duplicates" the time needed for the transitive closure exceeds the time needed for key generation, as the transitive closure algorithm has to process many duplicate pairs (Fig. 5(c)). Additionally, for the largest file size, this file needs almost 20 minutes for duplicate detection (the dirty data is about four times the size of the clean data) and represents a considerable time overhead compared on clean data.

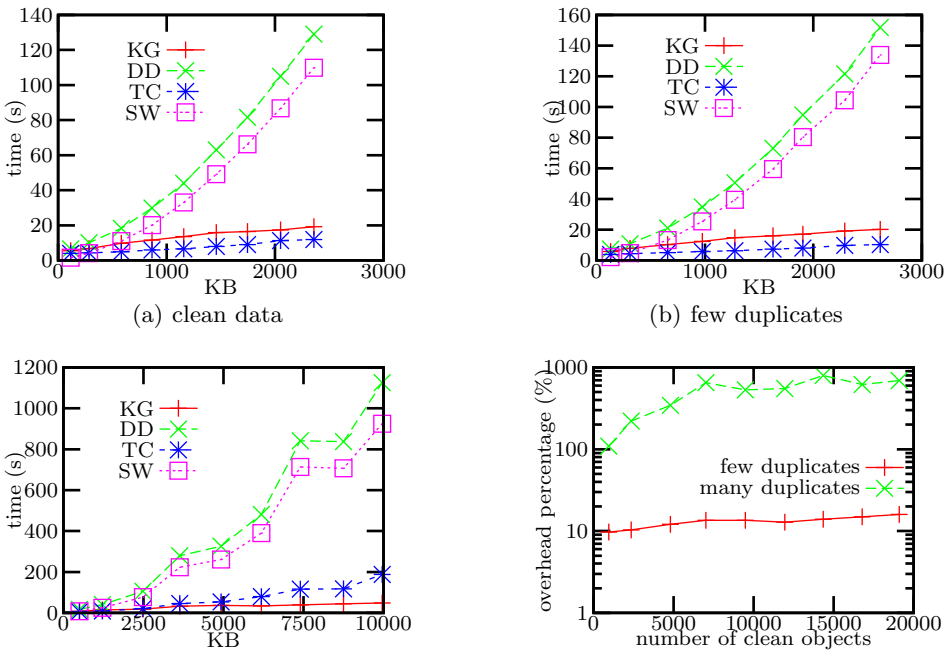


Fig. 5. Results for experiment set 2 on scalability.

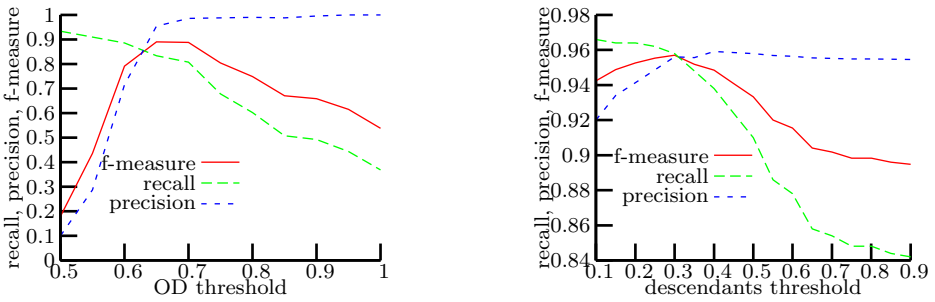
Experiment set 3: Threshold impact.

PURPOSE: In our third experiment we evaluate the effect of different thresholds on recall, precision, and f-measure. The two thresholds are the OD threshold that is used for comparisons of two element's object description, and the descendants threshold that is used for the similarity measurement of children. It shows how descendants help duplicate detection, depending on thresholds.

METHODOLOGY: Using Data set 2, we first detect duplicates in `<disc>` elements using only the object descriptions of the CDs, namely the disc ID, the artist and the CD title. We vary the OD threshold from 0.5 to 1. Afterwards, we use a fixed threshold for the OD and take the descendants `<title>` elements of the `<disc>` elements into account for duplicate detection. For these, we vary the descendants threshold from 0.1 to 0.9. Results are summarized in Figure 6.

DISCUSSION: Figure 6(a) shows the results for different object description thresholds. Using a low threshold of 0.5 results in a large amount of detected duplicates, leading to high recall but also to low precision, as many false positives occur. When the threshold increases, precision increases and recall decreases as expected. The f-measure peaks at a threshold of 0.65.

For the varying descendants threshold we use the OD threshold of 0.65 determined as optimal from the last experiment. We vary the descendants threshold from 0.1 to 0.9 and observe two things. First, Fig. 6(b) shows that the best f-measure obtained using descendants is higher than the best f-measure obtained when only considering the object description. Thus, we can conclude that it is worthwhile to take into account descendant information when detecting dupli-



(a) Varying OD thresholds (b) Varying descendants thresholds

Fig. 6. Results for experiment set 3 on threshold impact.

cates. Second, we observe that a very high descendants threshold downgrades the results, whereas the low descendants threshold of 0.3 leads to the best result of almost 0.96 in terms of f-measure. Choosing a low descendants threshold yields better results because it implies that a small overlap in children is already sufficient to consider children sets as similar. This compensates the effect of non-overlapping children, which drastically reduces similarity in our similarity measure (Def. 3).

5 Conclusions and Outlook

The Sorted Neighborhood Method is a very efficient method to detect duplicates in relational data. We have shown that our extension of this method to XML data, combined with new approaches in duplicate detection, is a reasonable alternative for XML duplicate detection for large amounts of data. However, there remain several open issues to further improve our method both in efficiency and in effectiveness.

Efficiency. In previous work we have shown that filters are quite effective to avoid comparisons, especially with the edit distance operations [17]. The work presented in the paper at hand also performs filtering but based on the generated keys and the sliding window. It will be interesting to see how the two filters interact. Moreover it could be useful to include the ideas of the Duplicate Elimination Sorted Neighborhood Method (DE-SNM) of [19] in our algorithm.

Effectiveness. In our current algorithm we use a simple approach of similarity function and threshold to determine whether two elements are duplicates. However, our algorithm is ready for the usage of equational theory, which was used for the relational SNM. We believe that the domain knowledge considered using the equational theory will yield even better results. Also, the choice of the thresholds yet remains an open issue. In [5] the authors propose a corresponding learning technique, which we plan to adapt to our problem of more than one type of descendant. Another knob to turn is the window size. In [20], a method to dynamically adapt the window size using distance measures on the keys is proposed. We plan to examine how sampling techniques can help determine an appropriate window size for each data set.

Acknowledgment. This research was supported by the German Research Society (DFG grant no. NA 432).

References

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