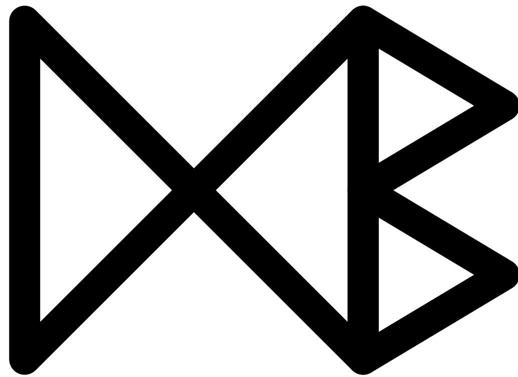


Synchronizing XPath Views

Dennis Pedersen and Torben Bach Pedersen

April 30, 2004

TR-7

A DB Technical Report

Title	Synchronizing XPath Views
	Copyright © 2004 Dennis Pedersen and Torben Bach Pedersen. All rights reserved.
Author(s)	Dennis Pedersen and Torben Bach Pedersen
Publication History	April 2004. a DB Technical Report A shorter version will appear in *Proceedings of the Eighth International Database Engineering and Application Symposium*, Coimbra, Portugal, July 2004

For additional information, see the DB TECH REPORTS homepage: ⟨www.cs.auc.dk/DBTR⟩.

Any software made available via DB TECH REPORTS *is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.*

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

The increasing availability of XML-based data sources, e.g., for publishing data on the WWW, means that more and more applications (data consumers) rely on accessing and using XML data. Typically, the access is achieved by defining *views* over the XML data, and accessing data through these views. However, the XML data sources are often independent of the data consumers and may change their schemas without notification, invalidating the XML views defined by the data consumers. This requires the view definitions to be updated to reflect the new structure of the data sources, a process termed *view synchronization*. XPath is the most commonly used language for retrieving parts of XML documents, and is thus an important cornerstone for XML view definitions. This paper presents techniques for discovering schema changes in XML data sources and synchronizing XPath-based views to reflect these schema changes. In many cases, this allows the XML data consumers to continue their operation without interruption. Experiments show that the techniques work well even if both schema and data change at the same time. To our knowledge, this is the first presented technique for synchronizing views over XML data.

1 Introduction

Modern information systems increasingly rely on using several diverse data sources to satisfy their information need. The data sources are often outside the IT system's own organization, and thus allow only restricted access, i.e., the IT system is a *federation*. The increasing availability of XML-based data sources means that the desired information will most often be available in an XML-based format, and accessed using XML technologies, e.g., XML query languages. The most commonly used XML query language is XPath [14], a compact, simple "Unix-like" language for retrieving parts of XML documents, which despite of its simplicity allows very powerful query definitions. The access is typically achieved by defining *views* over the XML data, and accessing data through these views. We assume that the views will be specified using a certain class of XPath queries.

However, as the XML data sources are often independent, they may change their schemas without notification. This requires the view definitions to be updated to reflect the new structure of the data sources, a process termed *view synchronization*. For example, a schema change in the XML source containing city information may invalidate the view that identifies the population data. In the worst case, the data is completely removed from the XML document. However, in many cases the change will be minor, such as adding information about who is the city's mayor to the document. Thus, it is often possible to automatically *synchronize* the view with the new schema, thereby making the federation more adaptable.

This paper presents heuristic techniques for discovering schema changes in XML data sources and synchronizing XPath-based views to reflect these schema changes. In many cases, but of course not all, this allows the XML data consumers to continue their operation without interruption. The techniques include ways of discovering schema changes, determining new views, and comparing XML schemas. Prototype implementations have been made of the fundamental algorithms, and a set of experiments have been performed on these prototypes demonstrating the power and practical usefulness of the presented techniques. Experiments with the techniques show that the techniques work well even if both schema and data change at the same time. The work is motivated by our work on a system for federating OLAP and XML data sources [10], but the techniques are completely general, and are applicable to all situations where views are defined over XML data, e.g., Web Services, B2B applications, and XML-based websites. We believe that this paper is the first to present generally applicable techniques for *synchronizing views* over XML data in case of schema changes.

There has been much previous work on *data changes* in XML sources, most notable in the Xyleme project [17]. Here, [7] describes an algorithm for determining the differences, a so-called *delta*, between different versions of XML documents, while [9] describes an architecture for subscribing to changes in

XML documents. Zhuge et al. [18] have considered the more general problem of maintaining materialized views over graph-structured data when the sources change. However, common to this work is that it only deals with *data changes* and not *schema changes*. Similarly, the Harmony project [6] considers synchronizing the *data* in different copies of a tree-structured/XML data source, but does not consider schema changes. *View synchronization* has been considered for relational sources by the EVE project [8, 12], Apart from not considering XML views, this work differs from ours in that it assumes explicit notification of schema changes by the sources, only handles small, atomic changes, and requires extensive metadata to be specified along with the view. In contrast, we discover the schema changes ourselves (no notification by sources needed), consider more complex (restructuring) changes, and do not require additional metadata to be specified. Adam et al. [1] present techniques to *detect* schema changes in semi-structured documents, but consider only special scientific documents, and do not view synchronization in any detail.

The rest of the paper is structured as follows. Section 2 introduces the problem setting and discusses the issues in XML view synchronization. Section 3 describes the discovery of schema changes, while Section 4 describes how to determine new views. Section 5 describes the comparison of XML schemas, while Section 6 provides an evaluation of the techniques. Section 7 gives an overview of a concrete usage context for the technique, namely a system for integrating OLAP and XML data sources. Finally, Section 8 summarizes the paper and points to future work.

2 Problem Setting

XML (Extensible Markup Language) is a flexible yet relatively simple way to represent information, which permits easy exchange of both the information and its schema. An XML document consists of a number of *elements*, which are identified by a *start tag* and an *end tag*, and may contain other elements, textual data, and attributes. Thus, elements are organized in a tree structure. We will use the term *node* to refer to both elements and attributes of an XML document. The basic schema formalism is the DTD (Document Type Definition), which is a context-free grammar describing the legal nesting of elements and attributes. However, other more complex formalisms exist, e.g. XML Schema [15].

The XPath language [14] is one of the most widely used languages for querying XML documents. XPath is not as powerful as some other XML query languages, such as XML Query [16], but it is sufficient for many purposes, including basic view definition. The basic syntax of an XPath query resembles a Unix file path where a full path expression is given as a number of locations separated by a "/". For example, /Authors/Publisher/Author/Book returns book elements nested under the Authors, Publisher, and Author tags. Each step selects a (multi-)set of nodes relative to the previous step and may *filter* the set using a selection predicate appended to the node name. For example, /Authors/Publisher/Author[@AName = "Kimball"]/Book only returns book elements written by Kimball. Here, "@" indicates that "AName" is an attribute. A variety of functions can be used in selection predicates, such as position(), which returns the position of a node when these are counted according to the order in which start tags appear in the document. XPath queries can use any number of *variables* of the form $Var, which must be bound to a valid value. For example, given a value for $A, such as "Thomsen", the path /Authors/Publisher/Author[@AName = $A]/Book returns books written by authors with the name bound to $A. Many other types of queries can be expressed, e.g., //Author returns *all* "Author" elements in the document, //Title/text() returns only the text part of the title elements without the surrounding tags, and /Authors/Publisher/Author | /Authors/Author concatenates the results of the two sub-queries.

The main challenge of view synchronization is the vast number of different schema aspects that can be used to characterize XML documents. For example, XML Schemas [15] allow the specification of detailed node occurrence constraints, complex data types, and inheritance. Only the most essential types of changes are considered here, namely those that deal with the structure and naming of nodes in an XML document.

These fundamental schema characteristics are captured by the *XML Schema Graph* defined below, which is essentially a DTD without the choice and sequence operators, somewhat similar to the *Data Guide* for semi-structured data [5]. To simplify the presentation, we assume in the following that elements in an XML document contain *either* a textual value or a sub-element, but not both. Mixed content is permitted in the standard, but its use is often discouraged, and the following can easily be extended to cover it. Also, we require that elements are not recursively defined. This requirement is not strictly necessary, since recursive definitions to some fixed maximal depth can be handled with small extensions to the following, but this would complicate the presentation unnecessarily. Furthermore, recursive definitions are not much used in practice. Thus, a schema graph for an XML document is defined as follows:

Definition 1 A *schema graph* S for an XML document D is a single-source DAG where vertices correspond to different types of nodes (elements or attributes) in D, such that:

- each vertex v has a name, denoted by $\text{Name}(v)$, which is the element identifier for elements and a combination of the parent element's identifier and the attributes identifier for attributes,
- the source v_s of S is labelled with the same name as the root node n_r of D, i.e., $\text{Name}(v_s) = \text{Name}(n_r)$,
- if node n_c is a child of node n_p in D, then vertex v_p in S with $\text{Name}(v_p) = \text{Name}(n_p)$ has a child v_c with $\text{Name}(v_c) = \text{Name}(n_c)$. □

In the following, we will simply use v as an abbreviation for $\text{Name}(v)$. The set of all vertices in a schema graph S will be referred to as V. Also, we will use $n \in v$ to mean $\text{Name}(v) = \text{Name}(n)$. As a compact way of representing schema graphs, we will use a notation similar to that of [2] and write $v_1(v_2, \ldots, v_n)$ to mean that $v_2, \ldots, v_n$ are children of v_1. Notice that only the first occurrence of a vertex is expanded with its children. A similar notation is used for nodes of XML documents.

Example 2.1 Two examples of schema graphs are shown in Figure 1. Imagine that the Bibliography schema is transformed to the Books schema. The XPath query /Bibliography/Publisher[@Name=$P]/Author[Last-Name=$A]/Book/Title identifies book titles in a Bibliography document given a publisher and an author. If the document changes such that it conforms to the Books schema instead, this query would be invalid. □

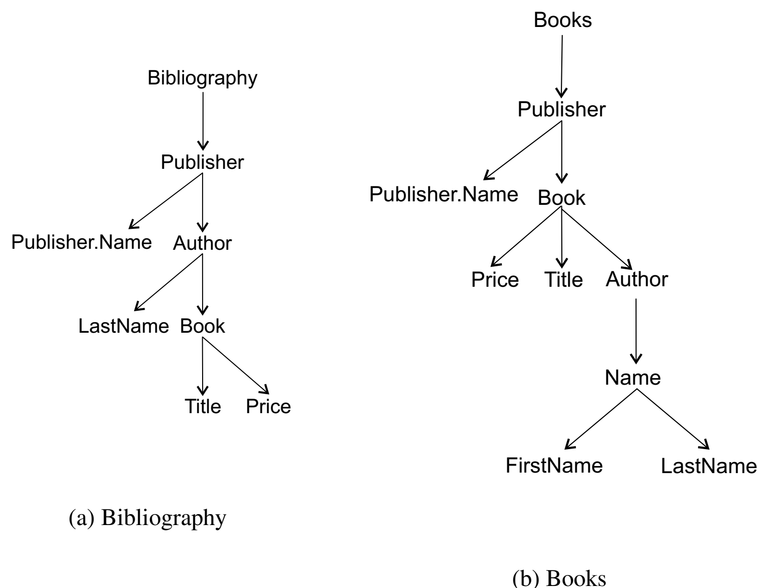
The XPath queries we consider are of the form:

$$P = /v_1[p_1(v_{1,1}/\ldots/v_{1,k_1})]/\ldots/v_n[p_n(v_{n,1}/\ldots/v_{n,k_n})],$$

where each v_i is a vertex and p_j is a predicate, and can also be combinations of results of such queries using the concatenation operator "|".

An XML document's schema can either be given explicitly as a separate file, which is identified by a URL in the document, or it can be implicitly given by the document contents. In both cases, a change in the schema may invalidate views over the document, requiring these to be synchronized with the new schema. Doing this manually is typically not a difficult task, but the process often includes a thorough analysis of the schema changes, and thus, it may interrupt the users' work-flow for hours or even days. To avoid this, the necessary changes should preferably be made *automatically*, and fortunately, it is often possible to do so. Hence, the problem discussed in this paper is that of automatically updating a parameterized XPath query. i.e., an XPath view definition, such that it identifies the same information as before the XML document's schema changed.

When *monitoring* changes in an XML document's schema, it is important that only a small amount of work is needed during regular operation. In general, it should be possible to perform any run-time checks for schema changes on-the-fly while the external data is being retrieved. Although a reasonable response time



(a) Bibliography

(b) Books

Figure 1: Two Schema Graphs.

is always important, the time requirements are not as strict when it comes to the *synchronization actions* that must be performed when a schema change has occurred, since this will be a relatively rare event and since the alternative is having no data at all. Thus, speed is generally of less concern when *handling* a schema change than when *checking* for changes.

It is, of course, not possible to make an equivalent view for an arbitrary change to an XML component's schema. However, even if it is possible, it may not always be possible to determine the new view query automatically, because there most often will not be an explicit notification about what changes have occurred. In that case, the changes have to be inferred from implicit knowledge, e.g., discovered from the original document and the changed document.

3 Discovering Schema Changes

Often, schema changes will not be explicitly notified to the federation, which makes view synchronization more difficult. Besides the obvious problem of detecting what changes have occurred and deriving an equivalent XPath query when nothing is known for certain about the changes, it may also be a problem to even detect that *any* changes have occurred. Nevertheless, techniques are described below that find a good alternative XPath query for many situations, and allow the DBA to specify the *certainty* with which the new view should be equivalent to the old one. This certainty is based on a comparison of the query result before and after the schema change. All that can be *guaranteed* in this way, is that the views are equivalent for the current document contents, but, in line with the work of [12], this will often be much better than having no view at all. The only schema changes of concern here are those that affect the result of the XPath query. Thus, we are e.g. not interested in changes in parts of the XML document that are not referred to in the XPath query.

To detect a change in an XML document's schema, the old and the new schema can often be compared

directly to see if a schema change has occurred. The overhead of doing this is limited, since schemas will usually be small (perhaps 10-50 vertices), and since, using common hashing techniques, each vertex need only be considered once. However, sometimes this approach it insufficient. First, if the query is not invalidated by the schema change, it may be perfectly correct for the schema to change. The change may not affect the result of the query, or the change may have been anticipated when the XPath query was formulated, and thus, the result after the change will also be semantically correct. Trying to update such a query automatically could destroy a cleverly defined view. Of course, distinguishing a legal schema change from an illegal one is impossible to do in general, as it would require the ability to accept or reject individual result values, an assumption that cannot be made without severely limiting the usefulness of the federated system. However, as discussed below, we can do better than to act on all kinds of schema changes. Second, the document may not have an explicitly defined schema, even though most data sources, which are intended for use by others, most likely will. Although a schema can always be generated from the document, this would be a very time-consuming way to check for schema changes, since it should, ideally at least, be done before every use of the document. Also, if queries are evaluated remotely, additional data and possibly the entire document would have to be retrieved in order to generate the schema. The approach adopted here is to use the *query result* as an indication that a schema change has invalidated the query. An *empty* result is then seen as an indication that the schema has changed such that the query is no longer valid with respect to the document. Thus, if a schema exists and it has changed, the query result is examined to determine whether the result is empty and the query is likely to have been invalidated by the change.

However, a schema change may also produce a non-empty but *incorrect* result. This can be handled by always keeping the previous result in the cache and comparing this to the new query result, the assumption being that, although a schema change can be accompanied by a data update, there should be a considerable overlap between the old result and the new result. If none of the result values after the schema change overlaps with the result values before the change, it is likely that the new values are invalid. This assumption is, of course, only valid for some data sources. The main problem with the approach is that all values in the old and the new query result must be compared each time a new query is issued, a time-consuming operation for large results. However, with the use of hashing techniques, this can be done by iterating through the values only once. Furthermore, the old result values can be hashed beforehand and the new values can be hashed on-the-fly, while processing the result. Thus, the overhead of comparing the values will typically be small. Because this approach only looks at query results and not at the full documents, it is useful both when queries are evaluated remotely and locally.

For data sources where there will not necessarily be a large overlap between the values, the *size* of the query results can be compared instead. Here, the assumption is that if the number of values are almost the same, then it is unlikely that a schema change has occurred that has invalidated the result. This can also be used as an optimization to the first approach, such that the actual result values are only compared if the number of values varies significantly. What constitutes a significant variation is highly dependent on the type of data source; Geographical information such as country populations typically has a very stable size, whereas click-stream data may not. Thus, the allowed size variation as well as the basis for comparing results must be specified for each document.

Sometimes an XML document may not have a schema. In that case, a schema may be *generated*, but, as mentioned, this can be a time-consuming process, and thus, it should only be performed when a schema change is already suspected, and not e.g. each time the document is used. If queries are evaluated remotely, it will generally be infeasible to generate the schema, since the full document will have to be retrieved in order to do this. Thus, if no schema exists, the query result is examined and used to determine whether a schema should be generated, if possible.

The combined approach is outlined in Algorithm 1. The algorithm returns a boolean answer indicating whether a given query is assumed to have been invalidated by a schema change, based on the old and the new query results, and, if they exist, the old and new schemas. In the algorithm, *Invalid* is a real number

between 0.0 and 1.0 indicating the certainty with which the query is assumed to be invalid. We assume a function *Compare* that compares two result sets and produces a real number indicating either, e.g., the percentage of overlapping values, the percentage difference in the number of values or a combination of the two. If the XML document has a schema and the schema has not changed, Algorithm 1 returns *false*. However, if the schema has changed and it is found to invalidate the query, either by comparing it directly to the query or because the result is empty, *true* is returned. If the query is not directly invalidated and the result is non-empty, the old and new results are compared as described above. If the document does not have a schema, the result is used to determine whether a schema should be generated. If it should and if it *can* be generated, it is compared to the old schema. Hence, in the typical case, when a schema exists and no schema change has occurred, the new and old schemas are compared, which is a very fast operation. If the schemas do not exist, the results must be compared, but as discussed above, this can be done mostly on-the-fly when processing the result.

Algorithm 1 SchemaInvalidated(*NewSchema, OldSchema, NewResult, OldResult, Query*): boolean

```
 1: real Invalid := 0.0
 2: if NewSchema exists then
 3:    if Changed(NewSchema, OldSchema) then
 4:       if Inconsistent(Query, NewSchema) or NewResult is empty then
 5:          Invalid := 1.0
 6:       else
 7:          Invalid := Compare(NewResult, OldResult)
 8: else
 9:    if NewResult is empty or Compare(NewResult, OldResult) > THRESHOLD then
10:       Generate NewSchema
11:       if Changed(NewSchema, OldSchema) or NewSchema cannot be generated then
12:          Invalid := 1.0
13: Return (Invalid > THRESHOLD)
```

Example 3.1 Consider again the schema change and XPath query from Example 2.1. Assuming that the schema exists, Algorithm 1 would then determine whether the query is consistent with the new schema. This is obviously not the case, since it refers to the root element “Bibliography” rather than “Books”. Consequently, the algorithm would set *Invalid* to 1.0 and return *true*.

If the schema did not exist, the query would return an empty result, causing the new schema to be generated. This schema would clearly have changed, and, again, *Invalid* would be set to 1.0 and *true* would be returned. □

4 Determining New Views

After it has been identified that a schema change has occurred that invalidates the XPath query, the task is to find a new query that is equivalent to the old one. As discussed above, this is not generally possible without help from a human domain expert that can guarantee that the semantics of the two queries are the same. Given only the old document and the new document, it is, of course, very difficult to automatically generate a semantically equivalent query. The identifiers may have been renamed in the new document such that equivalence between an element in the old and the new document can only be determined from the *data values*. However, there is no guarantee that the data values have not been updated in the new document, in addition to the schema change. More importantly, the XPath query may have been formulated based on implicit knowledge, such as “there will never be any duplicate values”. This need not be true after the schema change even though the current data values do not reflect this, and thus, it can be impossible to automatically make an equivalent query. However, often it will be possible to generate a query that *is*

equivalent to the old one, and it is even possible to provide a good quality measure based on the similarity of the old and the new query results, which can be used to decide whether or not to accept the new query.

We assume in the following that an XPath query returns a list of *values*, rather than a list of subtrees as is generally possible. However, the technique can easily be extended to the more general case.

Basically, a new equivalent XPath query is derived by searching the new schema for the nodes that are referenced in the old query, and using the new names and structure of these nodes to construct the new query. The matching nodes are identified by comparing their names, data values, and structure in the old and the new document and assigning a *matching score* to each pair of an old and a new node. The pair with the best score, which exceeds some threshold, are assumed to be equivalent. Thus, the threshold is used to ensure that a new query is only generated when the result will be sufficiently close to that of the old query. In many situations, more than one query can be constructed. In that case, the results of these potential new queries are compared to the result of the old query and the one with the best match is chosen.

Since the data values for each vertex in a schema graph must be considered when trying to automatically derive a new equivalent XPath query, we extend Definition 1 such that each vertex is associated with a list (actually a histogram) of data values that occur for that vertex in the document. This is similar to the value-added schema graphs used in [1]. However, our schemas are generic descriptions of the structure of an XML document, whereas those of [1] are restricted to special scientific documents. Thus, the schema graph definition is extended as follows:

Definition 2 A *value-added schema graph* S for an XML document D is a schema graph, where for all leaf nodes n_l in D, vertex v_l with $\text{Name}(v_l) = \text{Name}(c_l)$ is associated with a list of node values V such that each value $\text{Value}(n_l) \in V$. □

Algorithm 2 uses these value-added schema graphs (simply referred to as schema graphs in the following) to construct a close-to-equivalent XPath query. The inputs to the algorithm are the old and the new XML documents, the old and the new schemas, which are generated from the documents, and the old XPath query. The algorithm returns the new equivalent query if one is found.

Algorithm 2 GenerateXPath(D^{old}, S^{old}, D^{new}, S^{new}, Q^{old}): XPathQuery

1: Search through all leaf vertices v_l^{new} in S^{new} and locate v_{output}^{new} such that $m_{output} = \text{Match}(v_{output}^{old}, v_{output}^{new})$ is maximal, and locate all $v_{selection,i}^{new}$ such that $m_{selection,i} = \text{Match}(v_{selection,i}^{old}, v_{selection,i}^{new})$ is maximal.
2: **if** one of the m's is smaller than THRESHOLD **then**
3:     **Return** NOT FOUND.
4: Let P^{old} be the path from v_{output}^{old} up to v_{source}^{old} in S^{old} as given by Q^{old}.
5: Find a path P^{new} from v_{output}^{new} up to v_{source}^{new} in S^{new} with the best combined match when compared to P^{old}.
6: **for** all $v_{selection,i}^{old}$ in S^{old} **do**
7:     Let P_i^{old} be the path from $v_{selection,i}^{old}$ up to a vertex on P^{old} in S^{old} as given by Q^{old}.
8:     Find a path P_i^{new} from $v_{selection,i}^{new}$ up to a vertex on P^{new} with the best combined match when compared to P_i^{old}.
9: Construct Q^{new} from P^{new} and all P_i^{new}.
10: **Return** Q^{new}.

Example 4.1 The XPath query /Bibliography/Publisher[@Name="Wiley"]/ Author[@Name=$N]/Book/Title identifies book titles in a document conforming to the schema in Figure 2(a). The titles returned are all from the publisher "Wiley" and written by the given author.

Assume that the schema of this document changes into the one shown in Figure 2(b), but still represents the same information (except for the missing author address). That is, the "Bibliography" element is renamed to "BookList", "Book" is renamed to "Volume" and the "Author" element now contains the same information as the "Author.Name" attribute did. In the following, we will show how the algorithm automatically finds an equivalent query that locates the same book titles as the query above. □

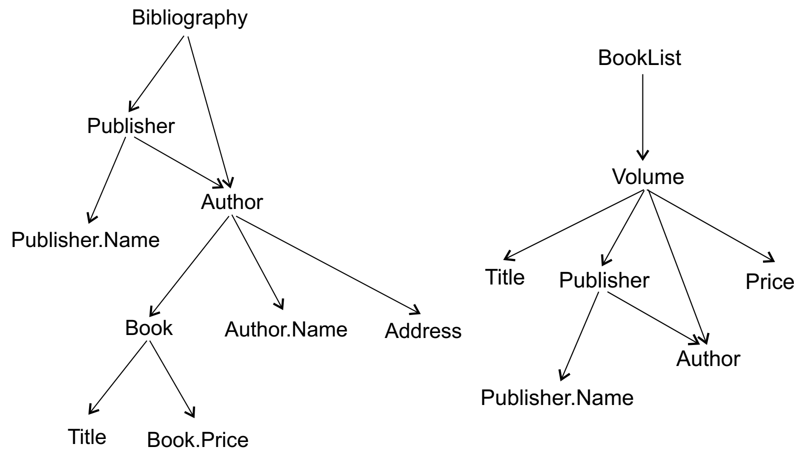


Figure 2: Bibliography (a) and Booklist (b) Schema Graphs

Before applying the algorithm, the schema graphs for the old and the new document are constructed by a single iteration of the documents. This is done in advance for the old schema. When the schema graphs have been constructed, the vertices that are used directly in the old query are identified in the new schema graph. This is the *output vertex* (v^{new}_{output}), which contains the values from which the result is a subset, and the *selection vertices* ($v^{new}_{selection,i}$) that are used to filter the output values. In this example, it is the "Title" vertex and the "Publisher.Name" and "Author" vertices in the new schema graph, respectively. These nodes are identified by comparing the output and selection nodes of the old schema graph (v^{old}_{output} and $v^{old}_{selection,i}$) to each leaf vertex in the new schema graph. This comparison is made using the Match-function, which outputs a real value between 0.0 and 1.0 indicating the certainty with which two vertices represent the same element or attribute. This function, which is described in detail below, considers both the name, the structure and the data values of the two vertices. If one of the vertices is not identified with the required certainty, the query construction is aborted.

When the referenced vertices have been identified, the paths leading from the source of the schema graph to these vertices are chosen. As Figure 2 illustrates, this choice is vital for producing an equivalent query. Here, the path /BookList/Volume/Author obviously does not produce the same result as the query /BookList/Volume/Publisher/Author. Which path should be chosen? In general, we cannot know this for certain, but again we can use the result of the old query to make the decision. Using this heuristic, we simply try all the paths and compare their results to that of the old query. The one that produces a result closest to the old query's result is chosen. Considering all paths will be realistic for most, if not all, situations, because any schema will be created by a human designer, who would never e.g. make a schema graph where most of the vertices are connected. Thus, the schema graph will typically not be large and only few of the vertices will be connected compared to the worst case. A typical schema will probably contain at most 5-10 paths. The paths that must be considered are not only those from the root to the output vertex, but also those from that path to the selection vertices. This is illustrated in Figure 3, where four paths can be found from the output path to the selection vertices. Still, the number of path combinations will typically be quite manageable. However, in general, the number of paths of a DAG is exponential in the number of vertices. For this reason, although it is somewhat theoretical, a greedy technique can be used to select a path, in which a sub-path is chosen independently from the rest of the path, by comparing the sub-paths directly to the path of the old query. This heuristic is based on the observation that when several paths can be chosen for the new query, the correct one is often the one that is most "similar" to the old query, as

discussed in Section 5 below. This is also the case in the example in Figure 2(b), where the correct path is the one through the “Publisher” vertex, which is similar to the old query. This technique can be used to simply choose a single path or to choose a subset of the possible paths, which are then compared on their results. The details of how (sub-)paths are compared are discussed in Section 5, and the general complexity of Algorithm 2 is discussed in Section 6.

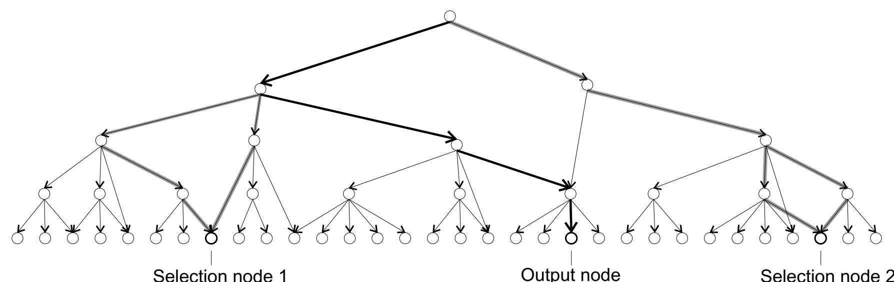


Figure 3: Schema graph with output and potential selection paths

When the paths have been found from the output vertex to the source, and from all selection vertices to this main path, construction of the new query is straight-forward. The selection paths are simply inserted as selections in the main path at the places where the selection paths intersect.

Example 4.2 In Figure 2(b), there is only one path from the output vertex “Title” to the source vertex “BookList”. Similarly, there is only one path from “Publisher.Name” up to the output path, i.e. to “Volume”. However, there are two paths from “Author” to “Volume”. Thus, two different queries can be constructed and compared on their results. In this case, it is very likely that the query containing the selection path /Volume/Publisher/Author returns a result that is closer to that of the old query, than the query with the selection path /Volume/Author. The new XPath query is created directly from the old query and the paths that have been chosen. The result is the following query: /BookList/Volume/[Publisher/@Name="Wiley"] [Publisher/Author/text()="$N"]/Title. This query produces the same result as the one in Example 4.1. □

To take full advantage of the algorithm, the data values are needed for comparing schema vertices. Thus, a local copy of the external document must be maintained, and the new document must be retrieved when the schema changes. However, as discussed below, the overhead of maintaining the old document will often be small and the cost of retrieving the new document is acceptable, because schema changes are rare and will otherwise disrupt the system.

Unless the XML source supports data subscriptions, it will be necessary to retrieve the full document periodically. However, as will also be demonstrated by experiments in Section 6, the old data values need not be entirely fresh for the algorithm to work well. Thus, the local copy will typically have to be refreshed much less frequently than data is fetched. This interval, of course, depends on the particular source, but for many sources it will be days or weeks before a significant part of the document has changed. Also, this retrieval can typically be performed when the system is not used, e.g. at night. For both the old and the new documents, only the part of the document that is used by the respective query is needed to do the comparison. This part is easily determined for the old document, since the XPath query is available, and consequently, only a small part of the old document needs to be maintained. For the new document, this is not generally possible. However, for large documents where the schema is available, the vertex names in the schema graph can be used to choose a subset of the values for retrieval. Thus, when searching for the output and selection vertices in the new schema, only the vertices most likely to match are actually compared on their data values, unless no good matches are found. For example, when looking for the old schema’s “Title” vertex in the new schema, only vertices with names that contain the string “Title” are considered at first. If

none of these produce a match better than some threshold, the full document is retrieved. Consequently, it will often be possible to significantly reduce the amount of data retrieved. Finally, for very large documents, it is also possible to use the algorithm without comparing the data values. This, of course, leads to a less precise result.

5 Comparing XML Schemas

In the absence of a human domain expert that can determine which vertices in two XML schemas correspond to the same real world entities, this must be done automatically by analyzing the information that is available in the two documents and their schemas. To be really useful, the degree to which two vertices "match" should be given as a numerical value, where a higher value indicates a stronger belief in the match. This is similar in spirit to the matching of data values in [4], except that they distinguish only between full, partial, and no match. Thus, in the following, we provide a set of formulas that produce a real-numbered value between 0.0 and 1.0 based on our confidence that two vertices match. Together, these formulas constitute the definition of the "Match" function mentioned in Algorithm 2. A similar approach is described for path comparisons. These values for the vertices and the paths as well the output of the resulting query provide the basis for an overall *certainty* for the generated query, which is also defined.

Essentially, a pair of vertices in two schema graphs can be compared on four accounts: their name, their data values, their relationships to other vertices, i.e. their structure, and their type, i.e. whether it is an element or an attribute. Only leaf vertices contain data directly, whereas only non-leaf vertices contain other vertices. Also, only leaf vertices can be of attribute type. For these reasons, we distinguish between leaf and non-leaf vertices in the Match function.

5.1 Comparing Leaf Vertices

For a leaf vertex the most safe matching parameter will generally be its data values. Its name and type can easily change, but as discussed earlier, at least some of the data values will typically be the same. For example, it is very likely that the "Author" and the "Author.Name" vertices contain some overlapping values, if they describe the same information. When this is not the case, e.g. because the values have changed, the *number* of values may provide evidence of a match. Following this idea, the name can, of course, also be a significant indication of a match, but it should not be considered more important than a high number of overlapping data values. Thus, only if two vertices contain almost the same number of overlapping values when compared to a third vertex, should the name be the determining factor. The same can be said about the type of the vertex, as this can easily change. The type, of course, provides less evidence about a match than the name. A further observation about the occurrence of vertices in schema graphs is that when a vertex has a different name than the one it is compared to, its *parent* will often have the same name. This is e.g. the case when a vertex, which contains a data value directly, is transformed into a vertex with e.g. a "Name" or a "Value" child, because additional sub-information is suddenly required. For example, a "Book" element directly containing the textual title of the book could be changed into a non-leaf element by adding a "Title" element and an "Author" element below it. To facilitate this knowledge, a vertex should produce a better match if its parent has the same name as the vertex it is compared to. In general, everything else being equal, a vertex that has a predecessor with an identical name should produce a better match than a vertex without such a predecessor. However, the relative importance of the predecessor's name should decrease rapidly as its distance to the leaf vertex increases. In particular, the fact that a predecessor has the same name should not be as significant as if the leaf vertex itself has the same name. In order to make these heuristic ideas operational they must be expressed as a formula that always outputs a value between 0.0 and 1.0 indicating the confidence in the match. This is accomplished by presenting a sequence of partial formulas that each

adjusts the output value according to some tests on the two vertices, which are compared (referred to as v and u). The formulas presented in the following have been chosen because they operationalize the ideas discussed above in a compact way, but other formulas could, of course, represent the same ideas. First, the number of *overlapping values* should be a significant factor. Consequently, the output value of the Match function should be close to 1.0 when there is a large overlap and close to 0.0 when there is only a small overlap. That is, the first step in calculating the result of the Match function is given by the function $LM_{overlap}$ (LM stands for Leaf Match):

$$LM_{overlap} = LW_{overlap} \frac{O}{Max(n_v, n_u)}$$

where O is the number of common values and n_v and n_u are the number of values in the vertex v and u, respectively. $LW_{overlap} \leq 1.0$ is a user definable (leaf) *weight* expressing the relative importance of the number of overlapping values (default is $LW_{overlap} = 0.95$). The *number of values* should be significant only if the overlap, i.e. $LM_{overlap}$, is relatively small. Even if there is only a small number of overlapping values and there is a very similar number of values, the certainty should not be as high as if there were a large number of overlapping values. However, since this depends on the type of document, this relative importance should be redefinable. These requirements are satisfied by the following formula:

$$LM_{number} = LM_{overlap} + LW_{number} \cdot V(1 - LM_{overlap}),$$

where $V = 1 - \frac{|n_v - n_u|}{Max(n_v, n_u)}$ with n_v and n_u denoting the number of values in the vertex v and u, respectively. LW_{number} is a weight expressing the importance of the number of values compared to the number of overlapping values (default is $LW_{number} = 0.5$). If the leaf vertex has the *same name* as the vertex it is compared to, the match value should increase significantly. However, since a smaller number of overlapping values may be as good a match as if the name is identical, it should not simply increase the Match value with a fixed amount. Instead, it should increase relative to the current match value, i.e. LM_{number}. Moreover, there should also be an increase if only a predecessor has the same name. This increase is set to be very small if there is more than 2-3 levels up to the predecessor, indicating that such a predecessor's name says only little about the match of a leaf node. This is satisfied by the formula:

$$LM_{name} = LM_{number} + LW_{name} \cdot S \cdot LM_{number}(1 - LM_{number})$$

Here $S = \frac{1}{e^l}$, where l is the number of levels up to a predecessor with the same name and e is the natural logarithm base. $l = 0$ if the leaf node has the same name and $l = \infty$ if no predecessor is found with a matching name. Again, LW_{name} is a weight indicating the relative importance of the name (default is $LW_{name} = 0.5$). Finally, the *type* of the vertex should have a small impact on the match value. However, this should only be the case if there is some other indication of a match, i.e. if LM_{name} is not close to zero. The fact that two vertices has the same type does not alone indicate a match. Moreover, the impact should also be small when LM_{name} is large, since one does not go from 95% to 100% certain of a match just because the types are identical. Only if one is relatively uncertain about the match should the type be significant. This is reflected in the following formula:

$$LM_{type} = LM_{name} + LW_{type} \cdot T \cdot LM_{name}(1 - LM_{name})$$

where $T = 1$ if the types are identical and $T = 0$ otherwise. The weight LW_{type} has a default value of 0.1. Given this, the value of the Match function for leaf vertices can be computed as:

$$Match_{leaf}(v, u) = LM_{type}$$

Example 5.1 Assume two leaf vertices v_1 and v_2 of an XML document are compared to a third vertex u in another document producing the following results:

Parameter	v_1	v_2
Number of overlapping values (O)	180	3
Number of values (n_{v_i})	200	300
Levels to a predecessor with name match (l)	0	∞
Same type (T)	0	1

Furthermore, assume the number of values in u is $n_u = 250$. Also, the number of values is decided to be insignificant for this type of document and thus, $LW_{number} = 0.1$ Based on these values the match can be calculated for the two vertices:

v_1:
$M_{overlap} = 0.95\frac{180}{Max(200,250)} = 0.684$
$M_{number} = 0.684 + 0.1(1 - \frac{|200-250|}{Max(200,250)})(1 - 0.684) = 0.709$
$M_{name} = 0.709 + 0.5 \cdot \frac{1}{e^0} \cdot 0.709(1 - 0.709) = 0.810$
$M_{type} = 0.812 + 0.1 \cdot 0 \cdot 0.812(1 - 0.812) = 0.812$

v_2:
$M_{overlap} = 0.95\frac{3}{Max(300,250)} = 0.010$
$M_{number} = 0.010 + 0.1(1 - \frac{|300-250|}{Max(300,250)})(1 - 0.010) = 0.092$
$M_{name} = 0.092 + 0.5 \cdot \frac{1}{e^\infty} \cdot 0.092(1 - 0.092) = 0.092$
$M_{type} = 0.092 + 0.1 \cdot 1 \cdot 0.092(1 - 0.092) = 0.100$

That is, $Match_{leaf}(v_1, u) = 0.812$ and $Match_{leaf}(v_2, u) = 0.100$. This corresponds quite well to the intuitive conception of the quality of the matches, given the weights of the different match parameters. □

5.2 Comparing Non-leaf Vertices

For non-leaf vertices, their names can, of course, also be used for comparison. However, the vertex does not directly contain any data values that can be compared. Also, a non-leaf vertex can only be of element type and thus, the type is of no interest. Instead, the *structure* of the vertex can be used for comparison, e.g. by looking at the children of the two vertices being compared. An obvious measure is the number of children with the same name, i.e. the overlap between the children of the two vertices. For example, the "Book" and "Volume" vertices both have a "Title" and a "Price" child, which, in this case, is a good indication that the two vertices correspond to the same real-world entity. However, some of the children could have been renamed, and for that reason, the *number* of children may also be an indication of equality, although intuitively, this should not be as significant as the children's names. However, data values will typically be a more safe basis for comparison than the names, since the name can easily change. For this reason, all leaf-vertex descendants can also be taken as an indication of equality. For example, the "Bibliography" and "BookList" vertices have a significant overlap in their descendant data values. (Only the "Address" values differ.) As discussed for leaf-vertices, the strongest indication is from the descendants that are closest to the vertex, because it is common to use "Id" or "Name" values as identifiers and these will typically be children or perhaps grand-children of the vertex. For example, the values of "Publisher.Name" in Figure 2(b) is a stronger indication of the identity for "Publisher" than the values of "Title". Thus, data values at the first 1-2 levels below a vertex should have much greater weight than values that are 5-6 levels down. In general, these data values should each have a smaller impact on the match value than the data values for leaf-vertices. To determine the match between these descendants, the Match function for leaf-vertices described above can be used directly.

To operationalize these heuristics, the formulas described in the following can be used. First, to compare two non-leaf vertices, the leaf descendants of the first vertex are compared to the leaf descendants of the second vertex. The pairs that provide the best matches are used to determine the initial match value of the non-leaf vertices. As discussed above, the effect of these descendants should be weighted according to their distance from the vertices being compared. That is, a weight function must be provided that outputs a value close to 1.0 for vertices 1-2 levels down and a value close to, but not equal to, 0.0 for vertices more than 4-5 levels down. These requirements are satisfied by the weight function w:

$$w_{leaf}(L) = \frac{0.9}{e^{0.01 \cdot L^4}} + 0.1$$

where L is the number of levels between the non-leaf vertex and its leaf descendant. This weight function is then used to determine the effect of each leaf descendant v_i of a vertex v when compared to the best matching leaf descendant $u_{i,best}$ of another vertex u. The effect should, of course, also depend on the quality of this match. If there is no good match between a pair of leaf descendants, i.e. if the match value is smaller than some threshold t, then a penalty value is subtracted from the combined match value. In this way the matching descendants add to the combined match value and the descendants that have no matching vertex in the other schema graph reduce the combined match value. This is satisfied by the *Non-leaf Match* formula (abbreviated *NM*) $NM_{desc} = LeafMatch(v_n)$, where n is the number of leaf descendants of v and $LeafMatch$ (abbreviated LM) is a recursive function defined as follows:

$$LM(v_1) = \begin{cases} w_{leaf}(v_1) \cdot w_{leaf}(u_{1,best}) \cdot Match(v_1, u_{1,best}) \\ \quad \text{if } Match(v_1, u_{1,best}) \geq t \\ 0 \quad \text{otherwise} \end{cases}$$

$$LM(v_i) = \begin{cases} LM(v_{i-1}) + w_{leaf}(v_i) \cdot w_{leaf}(u_{i,best}) \cdot \\ \quad Match(v_i, u_{i,best})(1 - LM(v_{i-1})) \\ \quad \text{if } Match(v_i, u_{i,best}) \geq t \\ LM(v_{i-1}) - w_{leaf}(v_i) \cdot w_{leaf}(u_{i,best}) \cdot \\ \quad P \cdot LM(v_{i-1}) \quad \text{otherwise} \end{cases}$$

Here, P is the penalty value that is subtracted for each non-matched descendant (the default is $P = 1.0$). As for leaf vertices, the name is also an important factor, which is operationalized by the formula:

$$NM_{name} = NM_{desc} + NW_{name} \cdot S \cdot NM_{desc}(1 - NM_{desc})$$

where $S = 1$ if the names of the two non-leaf vertices are identical and $S = 0$ if they are not. The default value of the weight NW_{name} is 0.8.

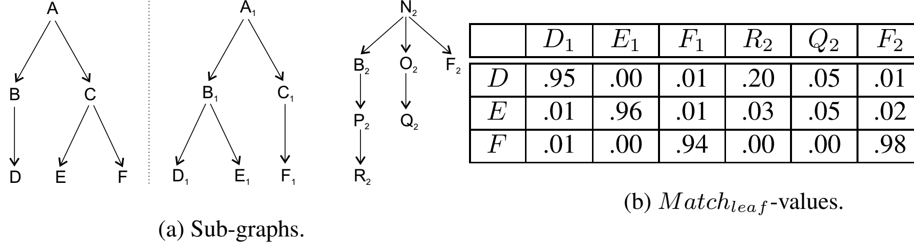
The structure of the two vertices being compared is considered by taking into account the number of children with identical names as well as the total number of children for each vertex:

$$NM_{struct} = NM_{name} + C \cdot NM_{name}(1 - NM_{name}),$$

where $C = NW_{struct1} \frac{N}{Max(c_v, c_u)} + NW_{struct2}(1 - \frac{|c_v - c_u|}{Max(c_v, c_u)})$. Here, N is the number of children with the same name, while c_v and c_u are the number of children for each of the two vertices v and u, respectively. The weights have the defaults $NW_{struct1} = 0.5$ and $NW_{struct2} = 0.1$. The Match function for non-leaf vertices can now be computed as:

$$Match_{non-leaf}(v, u) = NM_{struct}$$

Example 5.2 Assume two non-leaf vertices A_1 and N_2 are compared to a third vertex A. The three relevant sub-graphs are shown in Figure 4(a). Further assume that comparing the leaf vertices produces the results shown in Figure 4(b).



(a) Sub-graphs.

	D_1	E_1	F_1	R_2	Q_2	F_2
D	.95	.00	.01	.20	.05	.01
E	.01	.96	.01	.03	.05	.02
F	.01	.00	.94	.00	.00	.98

(b) $Match_{leaf}$-values.

Figure 4: Sub-graphs (a) and Match-values for the leaf-vertices (b).

Also, we have the following weight values for different values of L: $w_{leaf}(1) = 0.991$, $w_{leaf}(2) = 0.867$, $w_{leaf}(3) = 0.500$, $w_{leaf}(4) = 0.170$. Names with the same letter are assumed to be identical. This gives the following results:
A_1:
$LM(D_1) = 0.500 \cdot 0.500 \cdot 0.950 = 0.238$, $LM(E_1) = 0.238 + 0.500 \cdot 0.500 \cdot 0.960 \cdot (1 - 0.238) = 0.421$, $LM(F_1) = 0.421 \cdot 0.500 \cdot 0.500 \cdot 0.940 \cdot (1 - 0.421) = 0.557$, $NM_{desc} = 0.557$
$NM_{name} = 0.557 + 0.8 \cdot 1 \cdot 0.557 \cdot (1 - 0.557) = 0.755$
$NM_{struct} = 0.755 + (0.5 \cdot \frac{2}{Max(2,2)} + 0.1 \cdot (1 - \frac{0}{Max(2,2)})) \cdot 0.755 \cdot (1 - 0.755) = 0.866$
N_2:
$LM(R_2) = 0.170 \cdot 0.500 \cdot 0.200 = 0.017$, $LM(Q_2) = 0.017 - 0.500 \cdot 0.500 \cdot 1 \cdot 0.238 = 0.013$, $LM(F_2) = 0.013 \cdot 0.867 \cdot 0.500 \cdot 0.980 \cdot (1 - 0.013) = 0.432$, $NM_{desc} = 0.432$
$M_{name} = 0.432 + 0.8 \cdot 0 \cdot 0.432 \cdot (1 - 0.432) = 0.432$
$M_{struct} = 0.432 + (0.5 \cdot \frac{1}{3} + 0.1 \cdot (1 - \frac{1}{3})) \cdot 0.432 \cdot (1 - 0.432) = 0.490$

Thus, the results are: $Match_{non-leaf}(A_1, A) = 0.866$ and $Match_{non-leaf}(N_2, A) = 0.490$. As expected from the graphs, A_1 scores substantially better than N_2 when compared to A. The reason why N_2 does score reasonably well is because of F_2, since it matches quite well with F and is an immediate child of N_2. □

5.3 Comparing Paths

Given the above definition of the Match function, entire paths in a schema graph can also be compared. Such a comparison should consider not only the number of matched vertices in the two paths and how well they match, but also the order of the matched vertices, since two paths in the same order is, of course, a better match than if they were not in the same order. Also, two paths that contain the exact same vertices and no additional ones, should result in a better match than if one of them contains additional vertices. Consequently, there should be a penalty for unmatched vertices. The following formula calculates the match of two paths:

$$PathMatch(P_1, P_2) = \frac{q \cdot \sum_{i=1}^{m} Match(v_{P_1,i}, u_{P_2,i})}{s^2}$$

where s is the number of steps in the longest of P_1 and P_2, m is the number of matched pairs such that the match value is above some threshold (each vertex can only be matched once), q is the length of the longest sequence of matched pairs in the same order.

Thus, the optimal match between two paths is when $q = s = m$, which also implies that the two paths have equal length. In that case, the formula is simply the average Match value:

$$PathMatch(P_1, P_2) = \frac{\sum_{i=1}^{m} Match(v_{P_1,i}, u_{P_2,i})}{m}$$

5.4 Certainty of an XPath Query

Given formulas for matches between vertices and between paths, the *certainty* with which an XPath query is correct can now be stated. As mentioned, this certainty value should be based mainly on the similarity between the result of the old query and the result of the new query. However, two queries could by coincidence return the same result, even though one of them is incorrect. Thus, the certainty value must also take the Match and PathMatch values used to find the query into account. In this way, if two queries produce the same result, the query with the best matching output and selection vertices as well as the best matching paths will be chosen. The combined formula for calculating the certainty of an XPath query produced by Algorithm 2 is given by:

$$\begin{aligned} Certainty = W_{result} \frac{R}{Max(r_1, r_2)} \\ + W_{vertices} \cdot Avg(Match_o, Match_{s_1}, \ldots, Match_{s_n}) \\ + W_{paths} \cdot Avg(PathMatch_o, PathMatch_{s_1}, \ldots, PathMatch_{s_n}) \end{aligned}$$

where R is the number of overlapping result values r_1 and r_2 are the number of result values, $Match_o$ and $PathMatch_o$ are the match values for the output vertex and the path to it, respectively, and $Match_{s_i}$ and $PathMatch_{s_i}$ are the match values for the i'th selection node and the path to it, respectively. The weights have the defaults $W_{result} = 0.5$, $W_{vertices} = 0.3$ and $W_{paths} = 0.2$.

In summary, the heuristic view synchronization algorithm's ability to identify new equivalent queries is based on the many different aspects that are considered when comparing vertices and paths in different schema graphs, as well as on the fact that the query result is used to evaluate the quality of the query. When two leaf vertices are compared, four aspects are considered: the overlap between their data values, the number of data values, the names of the vertices, and the types of the vertices. Two non-leaf vertices are also compared on their names, but since they do not directly contain data, their leaf descendants are compared instead. The names of their children as well as the number of children are also taken into account. Two paths are compared by considering how many vertices occur on the paths, the order of these vertices, how well the vertices match, and also how many vertices on one path are *not* found on the other path. The query constructed by the algorithm is given with a *certainty* that takes all these aspects into consideration, in addition to the number of overlapping values in the query result. A prototype implementation of the basic synchronization algorithm, including the formulas discussed above, has been made, and a set of experiments has been performed on the prototype. These experiments are discussed next.

6 Evaluation

In this section, we evaluate the algorithm for synchronizing XPath queries and its limitations based on a set of experiments with the prototype implementation. We also discuss the running time of the algorithm. The three main aspects used in the synchronization are the structure of the two schema graphs, the names used in the two schemas, and the data values of the two documents. For this reason, experiments are conducted for different, realistic combinations of structural schema changes and subsequent data and schema name changes.

Table 1 presents the prototype's output for three different structural schema changes to an XML document and different types of changes made to the document after this transformation. Four combinations of

Data change		0%		50%	
Name change		**0%**	**50%**	**0%**	**50%**
Schema 2	XP1	+ (99,2%)	+ (98,9%)	+ (92,6%)	+ (90,8%)
Schema 2	XP2	+ (99,2%)	+ (97,9%)	+ (93,2%)	+ (88,9%)
Schema 3	XP1	+ (91,8%)	+ (90,6%)	+ (83,2%)	+ (80,4%)
Schema 3	XP2	+ (91,8%)	+ (90,5%)	+ (81,2%)	+ (79,7%)
Schema 4	XP1	+ (83,0%)	+ (77,6%)	+ (82,6%)	- (9,4%)
Schema 4	XP2	+ (82,9%)	+ (76,1%)	- (11,7%)	- (5.0%)

Table 1: Experimental Results

changes are considered in the experiments, namely where *no data or name changes* have been made after the structure change, *50%* of the *data* has been randomly updated after the structure change, *50%* of the *names* have been randomly changed after the structure change, and *50%* of the *data* has been randomly updated and 50% of the *names* have been randomly changed after the structure change. The experiments are all performed for two different XPath queries (XP1 and XP2), which are shown in the top two lines of Table 2. The results indicate whether the new XPath query, found by the algorithm, is *correct* or *incorrect*, and with what *certainty* the new queries are given by the algorithm.

The XML documents are generated from MS SQL Server's *pubs* database, which contains sample information about books and their authors and publishers. The documents use approx. 25 different elements or attributes and have between 5 and 13 levels of nesting, which we consider realistic for real-world XML data. Relatively small documents are used (50-100 KB), since these documents are not used for performance studies. Moreover, data changes on small documents will sometimes have a more invalidating effect than on large documents. Figure 5 shows the initial schema graph (Schema 1) and the three transformed schema graphs (Schema 2-4). Schema 2 is a basic transformation, where the nesting order is changed and Schema 3 is basically a subgraph of Schema 1. Schema 4 is a complete transformation of Schema 1, where the natural hierarchies of the geographical information have been unfolded, and some additional elements have been added. Intuitively, Schema 3 and, in particular, Schema 4 should pose the greatest challenges to the synchronization algorithm.

The XPath queries, shown in Table 2, retrieve book titles given the values of different elements, referring to the abbreviations used in Figure 5. Intuitively, query XP2 should be more difficult to handle than XP1, because it uses two elements with similar data values for selection (AuthorState and PublisherState).

Schema	Query	Correct Query
Schema 1	XP1	/As/A[FN=$F and LN=$L]/B[P/PN=$N]/PTi
Schema 1	XP2	/As/A[AS=$ASt]/B[P/PS=$PSt]/PTi
Schema 2	XP1	/Ps/P[PN=$N]/A[FN=$F and LN=$L]/B/PTi
Schema 2	XP2	/Ps/P[PS=$PSt]/A[AS=$ASt]/B/PTi
Schema 3	XP1	/As/A[FN=$F and LN=$L]/B[P2/PN=$N]/PTi
Schema 3	XP2	/As/A[AS=$ASt]/B[P2/PS=$PSt]/PTi
Schema 4	XP1	/Ps/PC/PS/P[PN=$N]/As/AS/AC/A[FN=$F and LN=$L]/Bs/BTy/B/BTi
Schema 4	XP2	/Ps/PC/PS[PS.N=$PSt]/P/As/AS[AS.N= $ASt]/AC/A/Bs/BTy/B/BTi

Table 2: Correct queries for the different schemas.

The experiments in Table 1 show that almost all queries were synchronized, even for large changes to the documents' data and names. Only three queries were not reported correctly by the prototype. The two wrong answers in the last column occur because the algorithm chooses the wrong path in Schema 4

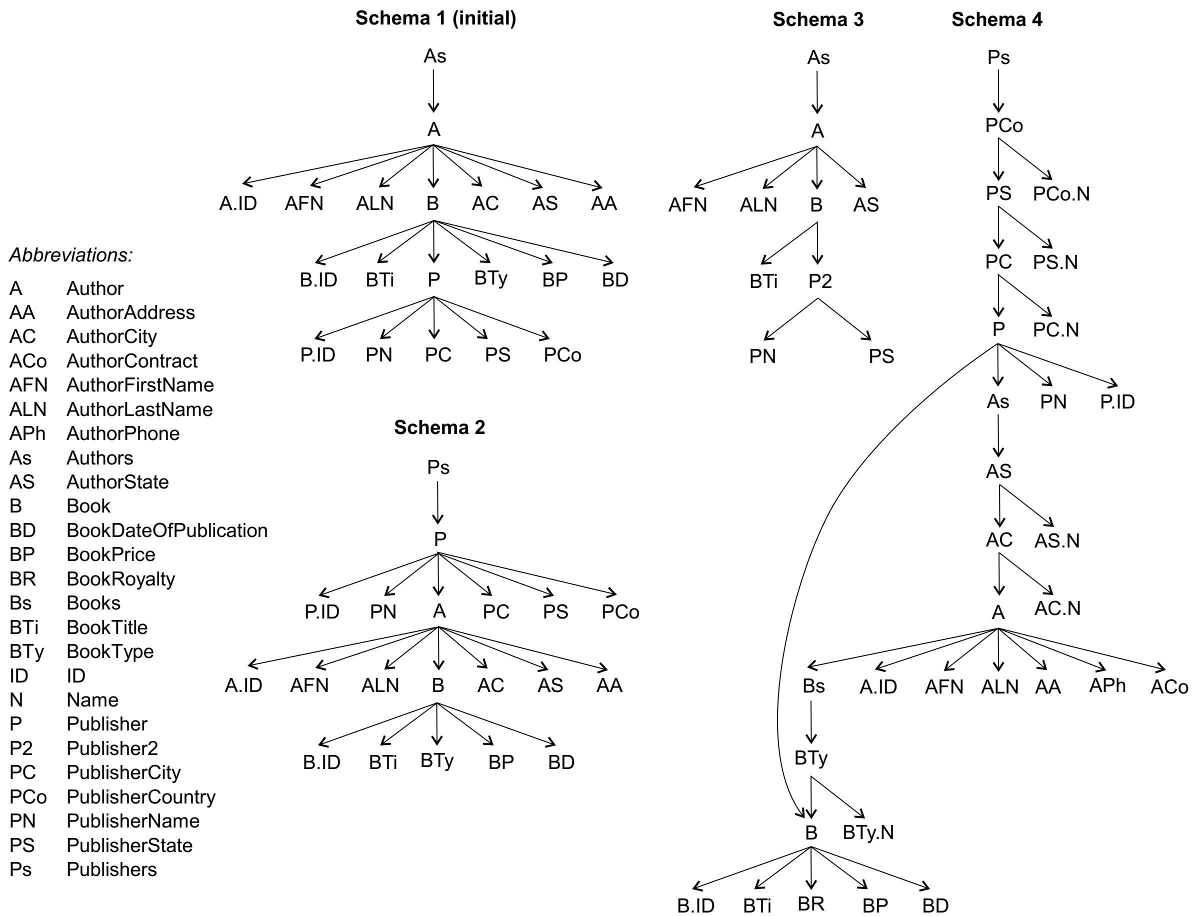


Figure 5: Initial (1) and transformed (2-4) schema graphs.

(the direct path from Publisher to Book) as a consequence of the large name and data changes. The third wrong answer is caused by the algorithm's false choice between the two selection nodes used in query XP2, which contain similar values already before the data changes. Notice that this false choice could also have occurred for Schema 2 or 3, if the subsequent changes had caused the two nodes to be indistinguishable. These wrong answers just tell us that when schema changes are accompanied by large changes to the schema names, and in particular to the data values, the identification of equivalent nodes becomes very difficult to do automatically (and most likely also manually).

Another important aspect of the results is the *certainty* values. When no subsequent changes are made to the documents, the certainty is relatively high, even for Schema 4, and it can also be seen that data changes reduces the certainty more than name changes. This is in line with the intentions behind the design of the matching formulas above. However, the certainty will be less affected by data changes for large documents, since even after a 50% data change they will most likely still contain many overlapping values. For small documents it is more likely that all or almost all values are replaced. For Schema 2 and 3 the certainty is generally higher than for Schema 4, which is clear when comparing their structure to that of Schema 1. What is significant here is that the certainty values, when the algorithm comes up with a wrong answer, is much smaller than when it finds the correct query. Thus, at least for these experiments, the certainty values can directly be used to distinguish between false and wrong answers by choosing a *threshold value* of around 50%.

However, since the algorithm is based on several heuristic assumptions, there are also certain situations that are not handled well by the algorithm. First, if the schema change is accompanied by a major, say 90%, *data* change, the basis for choosing one potential query over the other, namely the query results, becomes less useful. This problem can be reduced by ensuring that the cached copy of the document, i.e. the old document used for comparisons, is a more recent copy. In certain special situations it may not be possible to cache the document, such as if the data is only made available for browsing, and not for storing, and in that case only the schema comparisons can be made. The result will, of course, be much less accurate, but it will still provide a useful indication of what changes have occurred. Second, different leaf vertices with (almost) identical values can cause a problem, in particular, if none of their names match the name of the old vertex. Such vertices will always be difficult to distinguish from each other, even manually. Finally, legal schema changes that causes the query result to change significantly, may be interpreted as an illegal change. However, as discussed earlier, this is a rather special situation and it can often be prevented by adjusting some of the comparison thresholds.

Being able to quickly construct a new query in case of a schema change is not as important as being able to check for schema changes quickly. However, the time it takes to create a new query should still be comparable to the average query evaluation time when no schema changes have occurred. Experiments show that even for a relatively large XML document of approx. 10 MB the experiments in Table 1 each takes less than 2 seconds on a 800 MHz Duron PC. Thus, the running time is indeed satisfactory in this case. However, the general running time will also be acceptable. Given the optimizations described earlier, including the use of hashing techniques when comparing vertices and a greedy approach for choosing paths, the complexity of Algorithm 2 is $O(D_n + D_o + P_n)$, where D_n is the number of data values in the new document, D_o is the number of values in the selection vertices and the output vertex of the old document, and P_n is the number of paths in the new document going from the source to the output vertex or from this path to the selection vertices. As described earlier, P_n will usually be small (less than 10), and consequently this term is not significant. The total number of values will also be manageable. For example, with 10 MB of data, where each (string) value uses 50 bytes on average, Schema 4 would contain approx. 200000 values, when not counting the XML encoding overhead. Also with 10 MB data, the output and selection vertices of Schema 1 would contain approx. 37000 values. Thus, even for quite large schemas and documents, this will certainly be feasible.

As previously discussed, the algorithm is based on both the new and the old document, and consequently, a part of the external document must be retrieved both periodically and when a schema change occurs. However, only a part of the old and the new document is needed for the comparison, and, as the experiments show, the algorithm works even when schema changes are accompanied by relatively large data changes, which permits a rather old version of the document to be used. Because of this and because periodical retrievals can be performed when the system is not used, the document can be maintained with little burden on the system.

In conclusion, good results have been shown for the algorithm developed for synchronizing XML views after schema changes, even when these are accompanied by large changes to the contents of the documents. Also, in the experiments it was easy to distinguish the wrong answers from the correct ones based on the certainty measure. The time performance of the algorithm was also shown to be acceptable, both for the experiments and in general.

7 Employing View Synchronization: The Extended TARGIT System

This section gives an overview of one particular usage context for the XML view synchronization functionality, namely a system for integrating OLAP and XML data sources. The system started out as a research project at Aalborg University [10] and has later transitioned into industry [11] in a collaboration with the

Danish OLAP client vendor TARGIT. The industrial collaboration required us to find ways to make the system more adaptive towards changes, thus initiating the work on view synchronization for XML sources. The main part of the TARGIT system is an OLAP client, TARGIT ANALYSIS [13], which is known primarily for its ease of use.

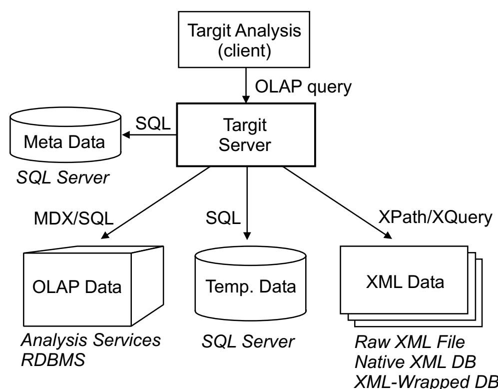


Figure 6: The extended TARGIT system.

The TARGIT system has been extended with federation facilities by adding the ability to handle external dimensions and measures without having to integrate the data physically in the OLAP database. To users, the external dimensions and measures are indistinguishable from ordinary dimensions and measures. Thus, the goal of the federation is to offer users complete *transparency* with respect to the *location* of data, the external data's *schema*, and the methods that are used to *access* the external data.

The general structure of the extended TARGIT "federation" system is shown in Figure 6. Instead of having only an OLAP data source as in the original system, three different data sources, also known as *components* in federation terminology, participate in the federation: an OLAP component, an XML component, and a relational component for performing temporary calculations during the evaluation of a query. The definition of external dimensions and measures are based on the contents of views over the XML data sources, specified using XPath expressions of the form given in Section 2. Thus, a robust view synchronization technique is very important for keeping the federation system running when external data sources change their schema, most often without informing the federation about the changes. We note that although the view synchronization technique has been developed within this project, it is completely general and can be used in other contexts, e.g., e-business systems.

8 Conclusion and Future Work

Motivated by the increasing reliance on views specified on external XML data sources, this paper considered the problem of *synchronizing* a certain class of XPath views when the schema of the external XML data changes. The paper presented heuristic techniques for discovering schema changes in XML data sources and synchronizing XPath-based views to reflect these schema changes, often allowing the XML data consumers to continue their operation without interruption. The techniques included ways of discovering schema changes, determining new views, and comparing XML schemas. A prototype implementation of the fundamental algorithms has been made, and experiments showed that the techniques worked well even if both schema and data changed at the same time.

In future work, it would be interesting to consider a wider class of XML views, e.g., XPath queries containing wildcards/general regular expressions or XML views specified using XQuery, and to utilize

the extra information available in an XML Schema. Although the experiments with the heuristic schema synchronization algorithm indicate that the chosen weight values are reasonable, even better values may be found by utilizing common techniques for experience-based weight adjustment, such as neural networks. Finally, as mentioned, the techniques could be applied to a wide range of other applications, e.g. Web Service applications.

References

[1] N. Adam et al. Detecting Data and Schema Changes in Scientific Documents. In *Proc. of ADL*, pp. 160–172, 2000.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[3] G. Cobéna, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proc. of ICDE 2002*.

[4] S. Chawathe et al. Change Detection in Hierarchically Structured Information. In *Proc. of SIGMOD*, pp. 493–504, 1996.

[5] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of VLDB*, pp. 436–445, 1997.

[6] M. B. Greenwald et al. The Harmony Project. `www.cis.upenn.edu/ bcpierce/harmony/`, current as of Jan. 31, 2004.

[7] A. Marian et al. Change-Centric Management of Versions in an XML Warehouse. In *Proc. of VLDB*, pp. 581–590, 2001.

[8] A. Nica et al. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proc. of EDBT*, 1998.

[9] B. Nguyen et al. Monitoring XML Data on the Web. In *Proc. of SIGMOD*, 2001.

[10] D. Pedersen, K. Riis, and T. B. Pedersen. XML-Extended OLAP Querying. In *Proc. of SSDBM*, pp. 195–206, 2002.

[11] D. Pedersen, J. Pedersen, and T. B. Pedersen. Integrating XML Data in the TARGIT OLAP System. To appear in *Proc. of ICDE*, 2004.

[12] E. Rundensteiner et al. Evolvable View Environment (EVE). In *Proc. of SIGMOD*, pp. 553–555, 1999.

[13] TARGIT Corporation. TARGIT homepage. `www.targit.com`, current as of Jan. 31, 2004.

[14] W3C. XML Path Language (XPath) Version 1.0. `www.w3.org/TR/xpath`. Current as of Jan. 31, 2004.

[15] W3C. XML Schema Part 0: Primer. `www.w3.org/TR/xmlschema-0`. Current as of Jan. 31, 2004.

[16] W3C. XQuery 1.0: An XML Query Language. `www.w3.org/TR/xquery`. Current as of Jan. 31, 2004.

[17] L. Xyleme. Xyleme: A Dynamic Warehouse for XML Data of the Web. In *In Proc. of IDEAS*, pp. 3–7, 2001.

[18] Y. Zhuge and H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *Proc. of ICDE*, pp. 116–125, 1998.