PORSCHE: Performance ORiented SCHEma Mediation

Khalid Saleem Zohra Bellahsene

LIRMM - UMR 5506, Université Montpellier 2, 34392 Montpellier, France

Ela Hunt

GlobIS, Department of Computer Science, ETH Zurich, CH-8092 Zurich

Abstract

Semantic matching of schemas in heterogeneous data sharing systems is time consuming and error prone. Existing mapping tools employ semi-automatic techniques for mapping two schemas at a time. In a large-scale scenario, where data sharing involves a large number of data sources, such techniques are not suitable. We present a new robust automatic method which discovers semantic schema matches in a large set of XML schemas, incrementally creates an integrated schema encompassing all schema trees, and defines mappings from the contributing schemas to the integrated schema. Our method, PORSCHE (Performance ORiented SCHEma mediation), utilises a holistic approach which first clusters the nodes based on linguistic label similarity. Then it applies a tree mining technique using node ranks calculated during depth-first traversal. This minimises the target node search space and improves performance, which makes the technique suitable for large scale data sharing. The PORSCHE framework is hybrid in nature and flexible enough to incorporate more matching techniques or algorithms. We report on experiments with up to 80 schemas containing 83,770 nodes, with our prototype implementation taking 587 seconds on average to match and merge them, resulting in an integrated schema and returning mappings from all input schemas to the integrated schema. The quality of matching in PORSCHE is shown using precision, recall and F-measure on randomly selected pairs of schemas from the same domain. We also discuss the integrity of the mediated schema in the light of completeness and minimality measures.

Key words: XML schema tree, schema matching, schema mapping, schema mediation, tree mining, large scale.

Email addresses: saleem@lirmm.fr (Khalid Saleem), bella@lirmm.fr (Zohra Bellahsene), hunt@inf.ethz.ch (Ela Hunt).

1 Introduction

Schema matching relies on discovering correspondences between similar elements in a number of schemas. Several different types of schema matching[3,6,7,10,15,16,21,22] have been studied, demonstrating their benefit in different scenarios. In data integration schema matching is of central importance[2]. The need for information integration arises in data warehousing, OLAP, data mashups [13], and workflows. Omnipresence of XML as a data exchange format on the web and the presence of metadata available in that format force us to focus on schema matching, and on matching for XML schemas in particular.

Previous work on schema matching was developed in the context of schema translation and integration [3,6,11], knowledge representation [10,22], machine learning, and information retrieval [7]. Most mapping tools map two schemas with human intervention [3,7,6,10,9,15,16]. The goals of research in this field are typically differentiated as matching, mapping or integration oriented. A *Matching* tool finds possible candidate correspondences from a source schema to a target schema. *Mapping* is an expression which distinctly binds elements from a source schema to elements in the target schema, depending upon some function. And *integration* is the process of generating a schema which contains the concepts present in the source input schemas. Another objective, *mediation*, is mapping between each source schema and an integrated schema. The objective behind our work is to explore the ensemble of all these aspects in a large set of schema trees, using scalable syntactic and semantic matching and integration techniques. The target application area for our research is a large scale scenario, like WSML [1] based web service discovery and composition, web based e-commerce catalogue mediation, schema based P2P database systems or web based querying of federated database systems.

We consider input schemas to be rooted labelled trees. This supports the computation of contextual semantics in the tree hierarchy. The contextual aspect is exploited by tree-mining techniques, making it feasible to use automated approximate schema matching [7] and integration in a large-scale scenario. Initially, we create an intermediate mediated schema based on one of the input schemas, and then we incrementally merge input schemas with this schema. The individual semantics of element labels have their own importance. We utilise linguistic matchers to extract the meanings hidden within the labels. This produces clusters of nodes bearing similar labels, including nodes from the intermediate mediated schema. During the merge process, if a node from a source schema has no correspondence in the intermediate mediated schema, a new node in the intermediate mediated schema is created to accommodate

[1] Web Service Modeling Language, http://www.w3.org/submission/WSML

this node.

Tree mining techniques extract similar sub tree patterns from a large set of trees and predict possible extensions of these patterns. The pattern size starts from one and is incrementally augmented. There are different techniques [1,25] which mine rooted, labelled, embedded or induced, ordered or unordered sub-trees. The basic function of tree mining is to find sub-tree patterns that are frequent in the given set of trees, which is similar to schema matching activity that tries to find similar concepts among a set of schemas.

Contributions

We present a new scalable methodology for schema mapping and producing an integrated schema from a set of input schemas belonging to the same conceptual domain, along with mappings from the source schemas to the integrated (mediated) schema. The main features of our approach are as follows.

1. The approach is almost automatic and hybrid in nature.
2. It is based on a tree mining technique supporting large scale schema matching and integration. To support tree mining, we model schemas as rooted ordered (depth-first) labelled trees.
3. It uses node level clustering, based on node label similarity, to minimise the target search space, as the source node and candidate target nodes are in the same cluster. Label similarity is computed using tokenisation and token level synonym and abbreviation translation tables.
4. The technique extends the tree mining data structure proposed in [25]. It uses ancestor/ descendant scope properties (integer logical operations) on schema nodes to enable fast calculation of contextual (hierarchical) similarity between them.
5. It provides as output the mediated schema and a set of mappings (1:1, 1:n, n:1) from input source schemas to the mediated (integrated) schema and vice versa.
6. The approach was implemented as a prototype. We report on experiments using different real (OAGIS [2], xCBL [3]) and synthetic scenarios, demonstrating:
   a) fast performance for different large scale data sets, which shows that our method is scalable and supports a large scale data integration scenario;
   b) input schema selection options (smallest, largest or random) for the creation of initial mediated schema, allowing us to influence matching performance;
   c) match quality evaluation using *precision, recall and F-measure* as measures of mapping quality;

[2] http://www.openapplications.org
[3] http://www.xcbl.org

d) an analysis of the integrity of integrated schema with reference to completeness and minimality measures;
e) quadratic time complexity.

The remainder of the paper is organised as follows. Section 2 presents the desiderata and issues encountered in large-scale schema mediation. Section 3 defines the concepts used in the paper. Our approach, Performance ORiented SCHEma mediation (PORSCHE) is detailed in Section 4, comprising of the architecture, algorithms and data structures, supported by a running example. Section 5 presents the experimental results in the framework of the desiderata for schema mediation. Section 6 reviews related work and compares it to ours. Section 7 gives a discussion on the lessons learned and Section 8 concludes.

2 Desiderata for Schema Mediation

Schema Mediation can be defined as integration of a set of input schemas into a single integrated schema, with concepts mappings from the input schemas to the integrated schema, also called the mediated schema. There are numerous issues in the semantic integration of a large number of schemas. Beside mapping quality, the performance and integrity of the integrated schema are also very important. For example, the Semantic Web, by definition, offers a large-scale environment where individual service providers are independent. In such a situation the mappings can never be exact, rather they are approximate [7,10,12]. And with hundreds of services available, searching and selecting the required service needs to be fast and reliable enough to satisfy a user query. Following, is a brief discussion of the desiderata for schema integration and mediation.

2.1 Feasibility and Quality of Schema Mapping and Integration

The quality of mappings depends on the number and types of matching algorithms and their combination strategy, for instance their execution order. To produce high quality mappings, matching tools apply a range of match algorithms to every pair of source and target schema elements. The results of match algorithms are aggregated to compute the possible candidate matches which are then scrutinised by users who select the best/most correct candidate as the mapping for the source schema element(s) to the target schema element(s). COMA++[6] and S-Match[10] follow this technique and create a matrix of comparisons of these pairs of elements and then generate mappings. The final mapping selection can also be automated, depending upon some pre-defined criteria (match quality confidence), for example, choosing the candidate with

the highest degree of similarity. QOM[9] and RONDO[17] use a variation of these techniques. The mapping algorithms used there are applied to each pair of schemas of size amenable to human inspection. Large schemas, anything in excess of 50 elements, make human intervention very time consuming and error-prone. Therefore, automated selection of best mappings from the source to the target schema is a must in large scale scenarios.

The second aspect regarding large scale scenarios is the requirement for batch schema integration, where schemas may contain thousands of elements. Often, an integrated schema is to be created and mappings from source schemas to the integrated schema have to be produced, and this is to be done fast and reliably. This requires both matching and integration and is not supported by tools like COMA++, S-Match, QOM and RONDO.

2.2 Schema Integration Approaches and Integrity Measures

Integrating a **batch of schemas** is a specialised application of schema matching, and a natural next step in data integration. Schema integration can be holistic, incremental or iterative (blend of incremental and holistic), as surveyed in [2]. The binary ladder (or balanced incremental integration) approach can be implemented as a script which automates the merging process, based on algorithms like QOM and SF. For example, in a binary ladder implementation, the result of merging the first two schemas is consecutively merged with subsequent schemas. [12,24], on the other hand, discuss mining techniques and apply an n-ary integration approach (all schemas exploited holistically), to generate an integrated schema, along with approximate acceptable mappings. An iterative approach first finds clusters of schemas, based on some similarity criteria, and then performs integration at cluster level iteratively[14,20,23]. Although the iterative process is automatic and robust, the mediation aspect is very difficult to implement on top of those approaches.

The main purpose of schema integration in an information system is to hide the complexity of the underlying data sources and schemas from the user. The user accesses the integrated schema (mediated schema) to find a service or an answer to a query. Batista and colleagues [4] explain the quality aspects of the integrity of a mediated schema in an information system. They highlight three quality criteria.

Schema completeness, computed as $completeConcepts/allConcepts$ is the percentage of concepts modelled in the integrated schema that can be found in the source schema. Completeness is indicative of the potential quality of query results.
Minimality, calculated as $nonRedundantElements/allElements$, measures the

compactness of the integrated schema and is normally reflected in query execution time.
Type consistency, measured as *consistentElements/allElements* spans all the input and mediated schemas. It is the extent to which the elements representing the same concept are represented using the same data types.

2.3 Performance

Performance is an open issue in schema matching [21,22], governed by **schema size** and **matching algorithms**. The complexity of the matching task for a pair of schemas is typically proportional to the size of both schemas and the number of match algorithms employed, i.e. $O(n_1 n_2 a)$, where n_1 and n_2 are element counts in the source and the target and a is the number of algorithms used [6,10,16]. Selection of the best match from a set of possible matches increases the complexity of the problem. In schema integration and mediation, beside the parameters discussed above, as there are more than two schemas, we may consider instead the batch size (number of schemas) and the manner in which the schemas are compared and integrated. The performance of the integration and mapping process can be improved by optimising the target search space for a source schema element. Minimising the search space by clustering will improve performance.

Our three desiderata, outlined at the start of this section, motivated us to deliver a new robust and scalable solution to schema integration and mediation. Here, we focus on a large number of schemas, automated matching, and good performance. We explore mediated schema generation. For a given batch of large schemas and an initial mediated schema (derived from one of the input schemas), we efficiently construct a mediated schema which integrates all input schemas in an incremental holistic manner. To enhance the speed and lower the cost of data integration, we try to remove the need for human intervention. We present a new method for schema matching and integration which uses a hybrid technique to match and integrate schemas and create mappings from source schemas to the mediated schema. The technique combines holistic node label matching and incremental binary ladder integration [2]. It uses extended tree mining data structures to support performance oriented approximate schema matching. Our approach is *XML schema centric*, and it addresses the requirements of the Semantic Web.

3 Preliminaries

3.1 Match Cardinalities

Schema matching finds similarities between elements in two or more schemas. There are three basic match cardinalities at element level [21]. Since we are matching schema tree structures (elements are nodes), where the leaf nodes hold data, we place more emphasis on leaf node matching. Our categorisation of node match cardinalities is driven by the node's leaf or non-leaf (inner node) status.
i) 1:1 - one node of source schema corresponds to one node in the target schema; leaf:leaf or non-leaf:non-leaf.
ii) 1:n - one node in the source schema is equivalent to a composition of n leaves in the target schema; leaf:non-leaf, where a source leaf node is mapped to a subtree containing n leaf nodes in the target.
iii) n:1 - n leaves in source schema compositely map to one leaf in the target schema; non-leaf:leaf, allowing a subtree with n leaves in a source to be mapped to a target leaf.

Example 1 (Match Cardinality): For 1:1, cardinality is straightforward. For 1:n, consider Figure 1. A match is found between $S_{source}name[2]$, child of *writer*, and $S_{target}name[2]$, child of *author*, with children *first* and *last*. We have a 1:2 mapping $(name)_{source} : (name/first, name/last)_{target}$. Also, there is a match between $S_{source}publisher[4]$ and $S_{target}publisher[5]$, with a 1:1 mapping $(publisher/name)_{source} : (publisher)_{target}$.•

Semantically speaking, a match between two nodes is fuzzy. It can be either an equivalence or a partial equivalence. In a partial match, the similarity is partial. It is highlighted in the following example.

Example 2 (Partial Match): In source schema Name = 'John M. Brown', is partially matched to LastName = 'Brown' and FirstName = 'John' in the target, because Name also contains the MiddleInitial = 'M'. •

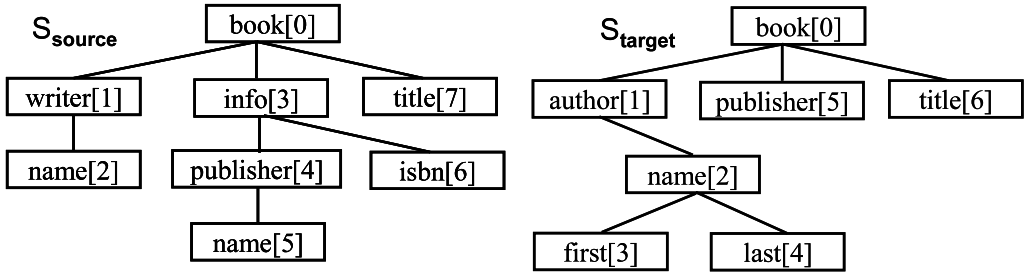

Fig. 1. Example schema trees showing labels and depth-first order number for each node.

Semantic matching requires the comparison of concepts structured as schema elements. Labels naming the schema elements are considered to be concepts and each element's contextual placement information in the schema further defines the semantics. For example, in Figure 1, $S_{source}name[2]$ and $S_{target}name[5]$ have similar labels but their tree contexts are different, which makes them conceptually disjoint. Considering the XML schema as a tree, the combination of the node label and the structural placement of the node defines the concept. Here we present the definitions used in schema matching and integration.

Definition 1 (Schema Tree): A schema $S = (V, E)$ is a rooted, labelled tree[25], consisting of nodes $V = \{0, 1, \ldots, n\}$, and edges $E = \{(x, y) \mid$ x,y $\in V\}$. One distinguished node $r \in V$ is called the root, and for all $x \in V$, there is a unique path from r to x. Further, lab:V $\rightarrow$ L is a labelling function mapping nodes to labels in $L = \{l_1, l_2, \ldots\}$.

Schema tree nodes bear two kinds of information: the node label, and the node number allocated during depth-first traversal. Labels are linguistically compared to calculate label similarity (Def. 2, Label Semantics). Node number is used to calculate the node's tree context (Def. 4, Node Scope).

Definition 2 (Label Semantics): A label l is a composition of m strings, called tokens. We apply the tokenisation function tok which maps a label to a set of tokens $T_l = \{t_1, t_2, \ldots, t_m\}$. Tokenisation [10] helps in establishing similarity between two labels.
$tok : L \rightarrow \mathcal{P}(T)$, where $\mathcal{P}(T)$ is a power set over T.

Example 3 (Label Equivalence): *FirstName*, tokenised as {first, name}, and *NameFirst*, tokenised as {name, first}, are equivalent, with 100 % similarity.•

Label semantics corresponds to the meaning of the label (irrespective of the node it is related to). It is a composition of meanings attached to the tokens making up the label. As shown by Examples 3-5, different labels can represent similar concepts. We denote the concept related to label l as $C(l)$.

Example 4 (Synonymous Labels): *WriterName*, tokenised as {writer, name}, and *AuthorName*, tokenised as {author, name} are equivalent (they represent the same concept), since 'writer' is a synonym of 'author'.•

Semantic label matching minimises the search space of possible mappable target nodes [10,25]. The derivation of concept similarity in two schemas is initiated by comparing their labels. Similarity between labels is either equivalence or partial equivalence, or no similarity, as defined below.

a. Equivalence : $C(l_x) = C(l_y)$
b. Partial Equivalence : $C(l_x) \cong C(l_y)$
   i) More Specific (Is part of) : $C(l_x) \subseteq C(l_y)$
   ii) More General (Contains) : $C(l_x) \supseteq C(l_y)$
   iii) Overlaps : $C(l_x) \cap C(l_y) \neq \emptyset$
c. No Similarity : $C(l_x) \cap C(l_y) = \emptyset$

Example 5 (Label Similarity): As *AuthorName* and *WriterName* are equivalent (Example 4), we write AuthorName = WriterName. Also, AuthorLastName ⊆ AuthorName, as *LastName* is conceptually part of *name*. Conversely, AuthorName ⊇ AuthorLastName. *MiddleLastName* and *FirstNameMiddle* overlap, as they share tokens {name, middle}.•

Definition 3 (Node Clustering): To minimise the target search space (see Sec. 2), we cluster all nodes, based on label similarity. The clustering function can be defined as $VT : L \rightarrow \mathcal{P}(V)$ where $\mathcal{P}(V)$ is the power set over V. VT returns for each label $l \in$ L a set of nodes $v_i \subseteq V$, with labels similar to l. Figure 2 illustrates node clustering. Here, the cluster contains nodes $\{x_{52}, x_{11}, x_{23}, x_{42}\}$ bearing synonymous labels {author, novelist, writer}.

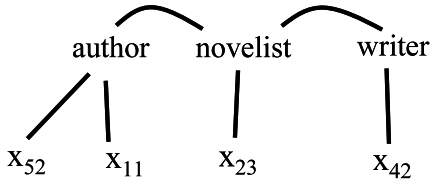

Fig. 2. Node Clustering. In x_{sn}, s is the schema number and n is the node number within the schema.

Definition 4 (Node Scope): In schema S each node $x \in V$ is numbered according to its order in the depth-first traversal of S (the root is numbered 0). Let *SubTree(x)* denote the sub-tree rooted at x, and x be numbered X, and let y be the rightmost leaf (or highest numbered descendant) under x, numbered Y. Then the scope of x is *scope(x)=[X, Y]*. Intuitively, *scope(x)* is the range of nodes under x, and includes x itself, see Fig. 3. The count of nodes in *SubTree(x)* is $Y - X + 1$.

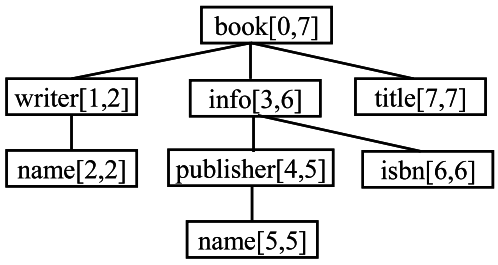

Fig. 3. Example schema tree with labels and [number,scope] for each node.

Definition 5 (Node Semantics): Node semantics of node x, C_x, combines

the semantics of the node label l, $C(l_x)$, with its contextual placement in the tree, *TreeContext(x)* [10], via function *NodeSem*.
$C_x : x \rightarrow$ *NodeSem*($C(l_x)$, *TreeContext(x)).*
TreeContext of a node is calculated relative to other nodes in the schema, using node number and scope (Example 7).

Definition 6 (Schema Mediation)
INPUT: A set of schema trees $SSet = \{S_1, S_2, \ldots S_u\}$.
OUTPUTS:
a) An integrated schema tree S_m which is a composition of all distinct concepts C_x in $SSet$ (see Def. 5, and[4]).

$$S_m = \bigwedge_{x \in S_i} \biguplus_{i=1}^{u} (C_x)$$

where $\uplus$ is a composition operator on the set of schemas which produces a tree containing all the distinct concepts (encoded as nodes). The tree has to be complete (see integrated schema integrity measure, Sec. 5 and[4]) to ensure correct query results.
b) A set of mappings $M = \{m_1, m_2, \ldots m_w\}$ from the concepts of input schema trees to the concepts in the integrated schema.

The integrated schema S_m is a composition of all nodes representing distinct concepts in *SSet*. During the integration, if an equivalent node is not present in S_m, a new edge $e\prime$ is created in S_m and the node is added to it, to guarantee completeness.

3.3 Scope Properties

Scope properties describe the contextual placement of a node[25]. They explain how structural context (within a tree) can be extracted during the evaluation of node similarity. The properties represent simple integer operations.

Unary Properties of a node x with scope [X,Y]:
Property 1 (Leaf Node(x)): $X{=}Y$.
Property 2 (Non-Leaf Node(x)): $X < Y$.

Example 6 (Leaf and Non-Leaf) : See Fig. 3. Property 1 holds for *title*[7,7] which is a leaf. Property 2 holds for *writer*[1,2] which is an inner node. Intuitively, properties 1 and 2 detect simple and complex elements in a schema.•

Binary Properties for x [X,Y], $x_d[X_d, Y_d]$, $x_a[X_a, Y_a]$, and $x_r[X_r, Y_r]$:
Property 3 (Descendant (x, x_d), x_d is a descendant of x): $X_d >$X and

$Y_d \leq$Y.
Property 4 (DescendantLeaf (x, x_d)): This combines Properties 1 and 3. $X_d >$X and $Y_d \leq$Y and $X_d = Y_d$.
Property 5 (Ancestor (x, x_a), x_a is an ancestor of x): (complement of Property 3) $X_a <$X and $Y_a \geq$Y.
Property 6 (RightHandSideNode (x, x_r), x_r is Right Hand Side Node of x with Non-Overlapping Scope): $X_r > Y$.

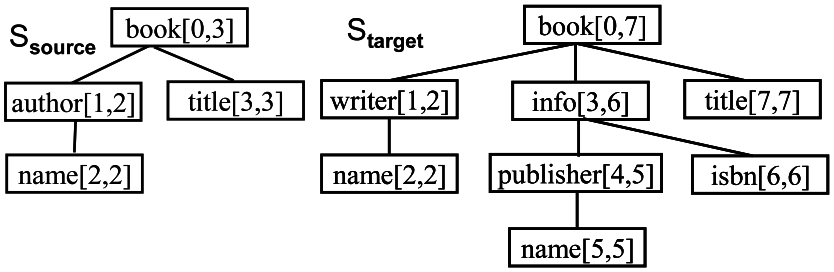

Fig. 4. Source and target schema trees.

Example 7 (Node Relationships): See Fig. 4. S_{target} – property 5 holds for nodes [4,5] and [5,5], as **Ancestor**([5,5],[4,5]), so *publisher*[4,5] is an ancestor of *name*[5,5]. Also **Ancestor**([4,5],[0,7]) holds for *book*[0,7] and *publisher*[4,5]. In S_{target}, **RightHandSideNode**([4,5],[6,6]) holds, implying node labelled *isbn* is to the right of node labelled *publisher*.•

Example 8 (Using Scope Properties) : The task is to find a mapping for S_{source}author/*name* in the target schema S_{target} (Fig. 4). S_{target} has two nodes called *name*: [2,2] and [5,5]. We assume synonymy between *author* and *writer*, top down traversal, and S_{source}*author* being already mapped to *writer* [1,2]. We perform the descendant node check on [2,2] and [5,5] with respect to *writer*[1,2]. Using Prop. 3, **Descendant**([5,5],[1,2])=false implies [5,5] is not a descendant of [1,2], whereas **Descendant**([2,2],[1,2]) is true. Thus, [2,2] is a descendant of [1,2], and author/*name* is mapped to writer/*name*.

4 PORSCHE

PORSCHE accepts a set of XML schema trees. It outputs an integrated schema tree and mappings from source schemas to the integrated schema.

4.1 PORSCHE Architecture

PORSCHE architecture (see Fig. 5) supports the complete semantic integration process involving schema trees in a large-scale scenario. The integration

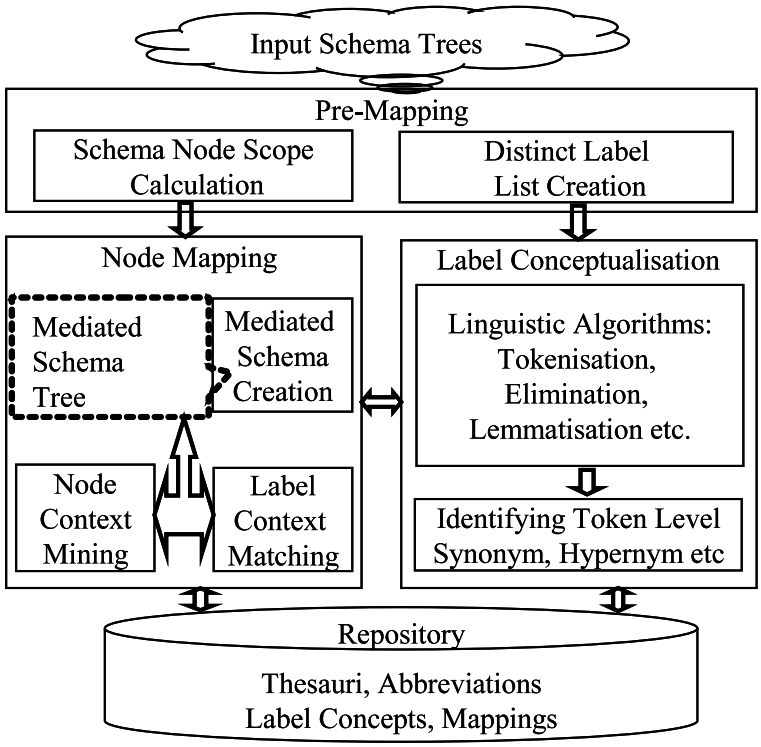

Fig. 5. PORSCHE architecture consists of three modules: pre-mapping, label conceptualisation and node mapping.

system is composed of three parts: i) *Pre-Mapping*, ii) *Label Conceptualisation* and iii) *Node Mapping*, supported by a repository which houses oracles and mappings.

The system is fed a set of XML Schema instances. *Pre-Mapping* module processes the input as trees, calculating the depth-first node number and scope (Def. 4) for each of the nodes in the input schema trees. At the same time, for each schema tree a listing of nodes is constructed, sorted in depth-first traversal order. As the trees are being processed, a sorted global list of labels over the whole set of schemas is created (see Sec. 4.2).

In *Label Conceptualisation* module, label concepts are derived using linguistic techniques. We tokenise the labels and expand the abbreviated tokens using an abbreviation oracle. Currently, we utilise a domain specific user defined abbreviation table. Further, we make use of token similarity, supported by an abbreviation table and a manually defined domain specific synonym table. Label comparison is based on similar token sets or similar synonym token sets. The architecture is flexible enough to employ additional abbreviation or synonym oracles or arbitrary string matching algorithms.

In *Node Mapping* module, the Mediated Schema Creator constructs the initial mediated schema from the input schema tree with the highest number of nodes augmented with a virtual root. Then it matches, merges and maps. Con-

cepts from input schemas are matched to the mediated schema. The method traverses each input schema depth-first, mapping parents before siblings. If a node is found with no match in the mediated schema, a new concept node is created and added to the mediated schema. It is added as the rightmost leaf of the node in the mediated schema to which the parent of the current node is mapped. This new node is used as the target node in the mapping. The technique combines node label similarity and contextual positioning in the schema tree, calculated with the help of properties defined in Section 3.

The *Repository* is an indispensable part of the system. It houses oracles: thesauri and abbreviation lists. It also stores schemas and mappings, and provides persistent support to the mapping process.

4.2 Algorithms and Data Structures

In this section we discuss the hybrid algorithms used in PORSCHE. We make assumptions presented in [12,24] which hold in single domain schema integration. Schemas in the same domain contain the same domain concepts, but differ in structure and concept naming, for instance, *name* in one schema may correspond to a combination of *FirstName* and *LastName* in another schema. Also, in one schema different labels for the same concept are rarely present. Further, only one type of match between two labels in different schemas is possible, for example, *author* is a synonym of *writer*.

Algorithm : preMap

Data: *SSet : Set of Schema Trees of size u*
Result: $\mathscr{V}$, $\mathscr{GLL}$, j
$\mathscr{V}$: List of lists of nodes (one list per schema) of size u, initially empty; each node list $\mathscr{SNL}$ sorted on depth-first order
$\mathscr{GLL}$: Global sorted list of labels
j : Schema tree identifier, initialized to 0

```
1  begin
2      for each schema S_i ∈ S do
3          V_i ← nodeScope(∅, RootNode_{S_i}, ∅)
4          𝒱 ← 𝒱 ∪ V_i
5          L_i ← lab(V_i)
6          if i = 1 then
7              𝒢LL ← sort(L_i)
8          else
9              𝒢LL ← mergeLabelLists(L_i,𝒢LL)
10     j ← initialMediatedSchema(𝒱_random|𝒱_smallest|𝒱_largest)
11 end
```

Fig. 6. Pseudocode of Pre-Mapping.

Pre-Mapping comprises a number of functions: 1) *depth-first, scope and par-*

ent node number calculation, 2) creation of *data structures for matching and integration*: schema node list ($\mathscr{SNL}$) for each schema and global label list ($\mathscr{GLL}$), and 3) *identification of the input schema* from which the initial mediated schema is created. $\mathscr{SNL}$ and $\mathscr{GLL}$ are later updated by the Node Mapping module (see Fig. 11). Pre-Mapping requires only one traversal of each input schema tree (Fig. 6).

```
Algorithm : nodeScope
   Data: p, c, V
   p : parent node list element, c : current node, V : nodes list
   Result: V
 1 begin
 2     x ← New nodesListElement(c)
 3     x.number ← length(V)
 4     x.parentNode ← p
 5     x.rightMostNode ← ∅
 6     Add x to V
 7     if c has no children then
 8         update(x, x)
 9     for each child of c do
10         nodeScope(x, child, V)
11 end
```

Fig. 7. Pseudocode for node scope calculation and node list creation.

Algorithm *nodeScope* (Fig. 7) is a recursive method, called from the *preMap* algorithm (L6.3)[4] for each schema, generating a schema node list ($\mathscr{SNL}$). It takes as input the current node, its parent reference, and the node list. In its first activation, reference to the schema root is used as the current node: there is no parent, and $\mathscr{SNL}$ is empty. A new node list element is created (L7.2-6) and added to $\mathscr{SNL}$. Next, if the current node is a leaf, a recursive method *update* (Fig. 8) is called. This adjusts the rightmost node reference for the current $\mathscr{SNL}$ element and then goes to adjust its ancestor entries in $\mathscr{SNL}$. This method recurses up the tree till it reaches the root, and adjusts all elements on the path from the current node to the root. Next, in *nodeScope* (L7.9-10), the method is called for each child of the current node.

After calculating node scope, *preMap* adds the new $\mathscr{SNL}$ to its global list of schemas and creates a global (sorted) label list ($\mathscr{GLL}$) (L6.5-9). (L6.10) chooses the largest (smallest or random) schema tree for subsequent formation of the initial mediated schema. $\mathscr{GLL}$ creation is a binary incremental process. $\mathscr{SNL}$ of the first schema is sorted and used as an initial $\mathscr{GLL}$ (L6.6-7) and then $\mathscr{SNL}$s from other schemas are iteratively merged with the $\mathscr{GLL}$, as follows.

[4] Lx.y refers to line y of algorithm in figure x

```
Algorithm : update
   Data: x_c, x_r
   x_c : Current nodes list element
   x_r : Right most node element of current nodes list element
1  begin
2      x_c.rightMostNode ← x_r
3      if x_c ¬ Root Node then
4          update(x_c.parentNode, x_r)
5  end
```

Fig. 8. Pseudocode for updating scope entries in $\mathscr{S}NL$.

mergeLabelLists (see Fig. 9) is a variant of merge sort. At (L9.7-8), we skip labels shared between $\mathscr{S}NL$ and $\mathscr{G}LL$ and add links between them, to keep track of shared labels. Multiple occurrences of the same label within an $\mathscr{S}NL$, however, are needed, as they help us in identifying labels attached to distinct nodes, e.g. author/*name* is different from publisher/*name*. If more distinct nodes with the same label are encountered in the matching process, they are handled in an overflow area. In the $\mathscr{G}LL$, multiple occurrences of the same label are equivalent to the largest number of their separate occurrences in one of the input schemas. This is further explained with the help of the following example.

```
Algorithm : mergeLabelLists
   Data: L_1, L_2 : Label lists for merging
   Result: GLL : Sorted list of merged label lists, initially empty
1  begin
2      sort(L_1)
3      while length(L_1) ≥ 0 ∧ length(L_2) > 0 do
4          if first(L_1) ≤ first(L_2) then
5              append first(L_1) to GLL
6              L_1 ← rest(L_1)
7              if first(L_1) = first(L_2) then
8                  L_2 ← rest(L_2)
9          else
10             append first(L_2) to GLL
11             L_2 ← rest(L_2)
12     if length(L_1) > 0 then
13         append rest(L_2) to GLL
14     if length(L_2) > 0 then
15         append rest(L_1) to GLL
16 end
```

Fig. 9. Pseudocode for label list merging, based on merge sort.

Example 9 (Repeated Labels) : Assume three input schema trees: S_1, S_2 and S_3. Their $\mathscr{S}NL$s before merge sort are: S_1{book, author, name, info, publisher, name, title}, S_2{book, writer, name, publisher, address, name, title}

```
Algorithm : labelSimilarity
   Data: 𝒢LL, simType
   𝒢LL : Global label list
   simType : Similarity type for labels
   Result: 𝒢LL : Global label list with inter-label similarity links
1  begin
2      var Set of token sets 𝒯
3      (𝒯,𝒢LL) ← tok(𝒢LL)
4      if simType = synonymTokenSetSimilarity then
5          adjustTokenSynonymSimilarity(Union of token sets ∈ 𝒯)
6      for each l_i ∈ 𝒢LL do
7          for each l_j ∈ rest(𝒢LL) do
8              if l_i ¬= l_j ∧ ∄ similarity(l_i,l_j) then
9                  if T_i = T_j then
10                     adjustLabelSimilarity(l_i,l_j)
11 end
```

Fig. 10. Pseudocode for label similarity derivation, based upon tokenisation (tok).

and S_3{book, author, address publisher, address, name, title}. The sorted $\mathscr{G}LL$ is {address, address, author, book, info, name, name publisher, title, writer}.•

Label conceptualisation is implemented in *labelSimilarity* (Fig. 10). The method creates similarity relations between labels, based on label semantics (Def. 2). This method traverses all labels, tokenises them (with abbreviations expanded), and compares them exhaustively. If similarity is detected, a link is added in $\mathscr{G}LL$ recording label similarity between two list entries.

After Pre-Mapping, PORSCHE carries out Node Mapping (see Fig. 11, mediation), which accomplishes both schema matching and integration. This handles the semantics of node labels, along with the tree context, to compute the mapping. The algorithm accepts as input a list of $\mathscr{S}NL$s ($\mathscr{V}$) and the identifier of the input schema with which other schemas will be merged (j). It outputs the mediated schema node list ($\mathscr{M}SNL$), V_m, and a set of mappings M, from input schema nodes to the mediated schema nodes.

First, the initial mediated schema tree which works as a seed for the whole process is identified. To this purpose, a clone of the largest $\mathscr{S}NL$ is created with a new ROOT node, V_m (L11.2). The new root node allows us to add input schema trees at the top level of the mediated schema whenever our algorithm does not find any similar nodes. This is required to support the completeness of the final mediated schema.

We use three data structures mentioned in the preceding paragraphs: (a) input $\mathscr{S}NL$s, $\mathscr{V}$ (b) $\mathscr{G}LL$, and (c) $\mathscr{M}SNL$, V_m. Here, we give a brief explanation of attributes associated with the node objects in each data structure. $\mathscr{V}$ holds

```
Algorithm : mediation
   Data: 𝒱, j
   𝒱 : List of lists of nodes (one list per schema) of size u; each node list
   𝒮𝒩ℒ sorted on depth-first order;
   j : Input schema identifier
   Result: M, V_m
   M : Set of mapping, initially empty
   V_m : Mediated schema nodes list
 1 begin
 2     V_m ← initialMediatedSchemaNodesList(V_j, x_ROOT)
 3     for each V ∈ 𝒱 do
 4         for each node x ∈ V do
 5             L_x ← lab(x)
 6             L_xsl ← similarLabels(L_x, V_m)
 7             if L_xsl ¬ ∅ then
 8                 V_t ← VT(L_xsl)
 9                 if |V_t| = 1 then
10                     (m, xt) ← oneMatch(x, V_t, V_m)
11                 else
12                     (m, xt) ← multipleMatch(x, V_t, V_m)
13             else
14                 (xt, V_m) ← addNewNode(V_m, x)
15                 (m, xt) ← mapOneToOne(x, xt)
16             M ← M ∪ m
17 end
```

Fig. 11. Pseudocode for schema integration and mapping generation.

u $\mathscr{SNL}$s, representing each input schema, where each element of the list has attributes {*depth-first order number, scope, parent, mapping data* {*mediated schema node reference, map type*}}. $\mathscr{GLL}$ element attributes are {*label, link to similar label*}. V_m's elements comprise attributes {*depth-first order number, scope, parent, list of mappings*{*input schema identifier, node number*}}. Data structures (a) and (c) are further sorted according to the order of (b), i.e. sorted $\mathscr{GLL}$, where each node object is placed aligned with its label (see Tab. 1). This helps in the clustering of similar nodes (Def. 3) and speeds up mapping, as similar nodes can be looked up via an index lookup. In *mapping data* in $\mathscr{V}$, the *mediated schema node reference* is the index number of the node in $\mathscr{MSNL}$ to which it has been matched, and *maptype* records mapping cardinality (1:1, 1:n or n:1).

During mediation a match for every node (L11.4) of each input schema (L11.3) is calculated, mapping it to the mediated schema nodes. For each input node x, a set V_t of possible mappable target nodes in the mediated schema is created, producing the target search space for x. The criterion for the creation of this set of nodes is node label equivalence or partial equivalence (L11.5,6). V_t can have zero (L11.13), one (L11.9) or several (L11.11) nodes.

If there is only one possible target node in the mediated schema, method

```
Algorithm : oneMatch
   Data: x, V_t, V_m
   x : Source node
   V_t : Set of one target node xt_1
   V_m : Mediated schema tree node list
   Result: m, xt
   m : Mapping
   xt : Target node in V_m
1  begin
2      if (Leaf(x)∧Leaf(xt_1)) ∨ (¬Leaf(x)∧¬Leaf(xt_1)) then
3          if ancestorMap(x,xt_1) then
4              (m, xt)← mapOneToOne(x, xt_1)
5          else
6              (xt, V_m) ← addNewNode(V_m,x)
7              (m, xt)← mapOneToOne(x, xt)
8      if Leaf(x)∧¬Leaf(xt_1) then
9          (m, xt)← mapOneN(x, xt_1)
10     if ¬Leaf(x)∧Leaf(xt_1) then
11         (m, xt)← mapNOne(x, xt_1)
12 end
```

Fig. 12. Pseudocode for matching one target node.

```
Algorithm : multipleMatch
   Data: x, V_t, V_m
   x : Source node
   V_t : Set of n(> 1) target nodes
   V_m : Mediated schema tree node list
   Result: m, xt
   m : Mapping
   xt : Target node in V_m
1  begin
2      descendantCheck ← false
3      for each node xt_i ∈ V_t do
4          if descendant(x,xt_i) then
5              (m, xt) ← mapOneToOne(x, xt_i)
6              descendantCheck ← true
7              break
8      if ¬descendantCheck then
9          (xt, V_m)← addNewNode(V_m,x)
10         (m, xt) ← mapOneToOne(x, xt)
11 end
```

Fig. 13. Pseudocode for matching several target nodes.

oneMatch (Fig. 12) is executed. (L12.2,8,10) compare the tree context of nodes x (input node) and xt_1 (possible target node in V_t), to ensure that a leaf node is mapped to another leaf node. Second check, *ancestorMap* (L12.3), ensures that at some point up the ancestor hierarchy of x and xt_1 there has been a mapping, in accordance with Prop. 5. This guarantees that subtrees containing

x and xt_1 correspond to similar concept hierarchies. This increases the match confidence of nodes x and xt_1.

Alternatively, if the target search space V_t has more than one node, algorithm *multipleMatch* (Fig. 13) is executed. Here, we check the descendant Prop. 3 (L13.4) for each node xt in V_t (L13.3). *Descendant* function verifies if xt lies in the sub-tree of the node to which the parent is mapped, that is within the scope. The function returns true for only one node or none.

Method *addNewNode* (L11.14,13.9,12.6) adds a new node xt as the rightmost sibling of the node to which the parent of x is mapped. Adding a new node requires a scope update: the ancestor and right hand side nodes of the new node have now a larger scope. Properties 5 and 6 are used to to find the ancestor and right hand side nodes. For ancestor nodes, scope is incremented, and for right hand side nodes, both node number and scope are incremented.

Mapping method *mapOneToOne*, creates a 1:1 map from x to xt, whereas *mapOneN* creates a mapping from x (a leaf) to a set of leaves in the subtree rooted at xt (non-leaf), which is a 1:n mapping. This mapping is considered to be an approximate mapping but follows the semantics of leaves mapping to leaves. And, similarly, *mapNOne* is a composite mapping of leaves under x, to leaf xt. Node Mapping algorithm (Fig. 11) integrates all input nodes into the mediated schema and creates corresponding mappings from input schemas to the mediated schema.

4.3 A Schema Integration Example

Figure 14 shows two trees after Pre-Mapping. A list of labels created in this traversal is shown in Table 1a. The two nodes with the same label *name* but different parents are shown (labels with index 2 and 3). The last entry in the list is the ROOT of the mediated schema. In the label list the semantically similar (equivalent or partially equivalent) labels are detected: *author* (index 0) is equivalent to *writer* (index 7).

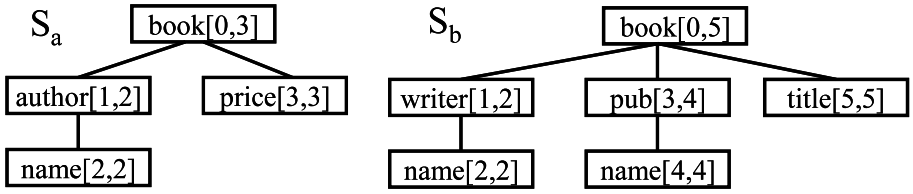

Fig. 14. Input Schema Trees S_a and S_b.

A matrix of size uq (here 2x9) is created, where u is the number of schemas and q the number of distinct labels in the $\mathscr{G}LL$, see Tab. 1b. Each matrix row represents an input schema tree and each non-null entry contains the

Table 1
Before Node Mapping. Schema column entry is (node number, scope, parent).

a. Global Label List

0	1	2	3	4	5	6	7	8
author	book	name	name	price	pub	title	writer	ROOT

b. Input Schema Matrix : Row 1 is Sa and Row 2 is Sb

1,2,0	0,3,-1	2,2,1		3,3,0				
	0,5,-1	2,2,1	4,4,3		3,4,0	5,5,0	1,2,0	

c. Initial Mediated Schema

	1,6,0	3,3,2	5,5,4		4,5,1	6,6,1	2,3,1	0,6,-1

Table 2
After Node Mapping. *Column entry is (node number, scope, parent, **mapping**). **Bold numbers refer to **sourceSchema.node**.

a. Global Label List

0	1	2	3	4	5	6	7	8
author	book	name	name	price	pub	title	*writer*	ROOT

b. Mapping Matrix *: Row 1 is Sa and Row 2 is Sb

1,2,0,**7**	0,3,-1,**1**	2,2,1,**2**		3,3,0,**4**				
	0,5,-1,**1**	2,2,1,**2**	4,4,3,**3**		3,4,0,**5**	5,5,0,**6**	1,2,0,**7**	

c. Final Mediated Schema **

	1,7,0, **1.0,2.0**	3,3,2, **1.2,2.2**	5,5,4, **2.4**	7,7,1, **1.3**	4,5,1, **2.3**	6,6,1, **2.5**	2,3,1, **1.1,2.1**	0,7,-1

node scope, parent node link, and the mapping, which is initially null. Matrix columns are alphabetically ordered. The larger schema, S_b, Fig. 14, is selected for the creation of the initial mediated schema S_m. A list of size q, Tab. 1c, is created to hold S_m, assuming the same label order as in Tab. 1a and 1b.

The node mapping algorithm takes the data structures in Tab. 1 as input, and produces mappings shown in Tab. 2b and the integrated schema in Tab. 2c. In the process, the input schema S_a is mapped to the mediated schema S_m, i.e. extended schema S_b. The mapping is taken as the column index (Tab. 2b, in bold) of the node in S_m. Saving mappings as column index gives us the flexibility to add new nodes to S_m, by appending to it. Scope values of some nodes may be affected, as explained previously, because of the addition of new nodes, but column indices of all previous nodes remain the same. Intuitively, none of the existing mappings are affected.

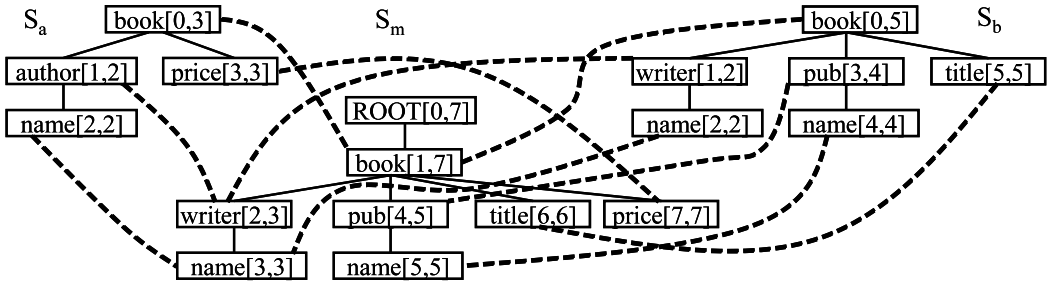

Fig. 15. Mediated Schema with mappings.

Node Mapping for input schema S_a (Tab. 1b row 1) starts from the root node S_a[0,3]*book*. It follows the depth first traversal of the input tree, to ensure that parent nodes are mapped before the children. S_a[0,3] has only one similar node in the mediated schema tree S_m i.e., node 1 at index 1. So entry **1** at index 1 for S_a records the mapping $(S_a[0,3], S_m[1,6])$, see Tab. 2b. Information regarding mapping is also saved in the mediated schema node as **1.0** (node 0 of schema 1). Next node to map in S_a is [1,2]*author*, similar to S_m[1,2]*writer*. Both nodes are internal and the method *ancestorMap* returns true since parent nodes of both are already mapped. The resulting mapping for node with label *author* is entry **7**. For node with label 2 *name*, there are two possibilities, label 2 (index 2) and label 3 (index 3) in the mediated schema. *Descendant* is true for node at index 2 (*author/name*,*writer/name*), and false for 3 (*author/name*,*pub/name*). Hence, **2** is the correct match.

The last node to map in S_a is [3,3]*price*. There is no node in S_m with a similar label, so a new node is added to S_m, recorded by an entry in the column with label 'price' in the mediated schema (Tab. 2c). A new node is created as the rightmost sibling of the node in the mediated tree to which the parent node of current input node is mapped, i.e. as child of *book*. The scope and parent are accordingly adjusted for the new node, and for its ancestors, and for right hand side nodes, scope and number are adjusted. There is no effect on the existing mapping information. Finally, a mapping is created from the input node to this new target node.

If a new node is added in the middle of the tree, its ancestor's scope is incremented by one. And, accordingly, right hand side nodes (satisfying Property 6) have their order numbers and scopes incremented by one. The implementation keeps track of the columns for the next node according to depth-first order. Thus, the final mediated schema tree can be generated from the final mediated schema row by a traversal starting from the ROOT. The mediated schema with mappings (dotted lines) is shown in Fig. 15.

4.4 Complexity Analysis

The worst-case time complexity of PORSCHE can be expressed as a function of the number of nodes in an input schema, n on average, where the number of schemas is u. The algorithm has the following phases.

- Data structure creation - all input schemas are traversed as trees (depth-first) and stored as node lists ($\mathscr{S}NL$s), a global list of node labels ($\mathscr{G}LL$) is created, and one of the schemas is selected as the base for the mediated schema, with time complexity $O(nu)$.
- Label List sorting - the number of labels is nu and sort time complexity for the global list is $O(nu \log(nu))$.
- Label similarity - we compare each label to the labels following it in the list. Time complexity of $O((nu)^2)$.
- Node contextual mapping - nodes are clustered, based on node label similarity. Worst case is when all labels are similar and form one cluster of size nu. We compare each node once to all other nodes, using tree mining functions. At worst, nu nodes are compared to nu nodes, with time complexity $O((nu)^2)$. Realistically, there are multiple clusters, much smaller than nu, which improves performance.

Overall time complexity is $O((nu)^2)$, with space complexity $O(n(u)^2)$, as the largest data structure is a matrix of size nu^2 used in the schema mapping process.

The theoretical time and space complexity of PORSCHE is similar to other algorithms we surveyed (Sec. 2). Because we perform only one full traversal of all the schemas and reuse the memory-resident data structures generated in the traversal of the fist schema, instead of performing repeated traversals used in alternative approaches which use a script to repeatedly merge any two schemas, we can generate a mediated schema faster than other approaches. Our experiments confirm the complexity figures derived above, see Section 5.

5 Experimental Evaluation

The prototype implementation uses Java 5.0. A PC with Intel Pentium 4, 1.88 GHz processor and 768 MB RAM, running Windows XP was used. We examine both the performance and quality of schema mediation, with respect to mapping quality and mediated schema integrity.

Table 3
Schema domains used in the experiment.

Domain	OAGIS	XCBL	BOOKS
Number of Schemas	80	44	176
Average nodes per schema	1047	1678	8
Largest schema size	3519	4578	14
Smallest schema size	26	4	5

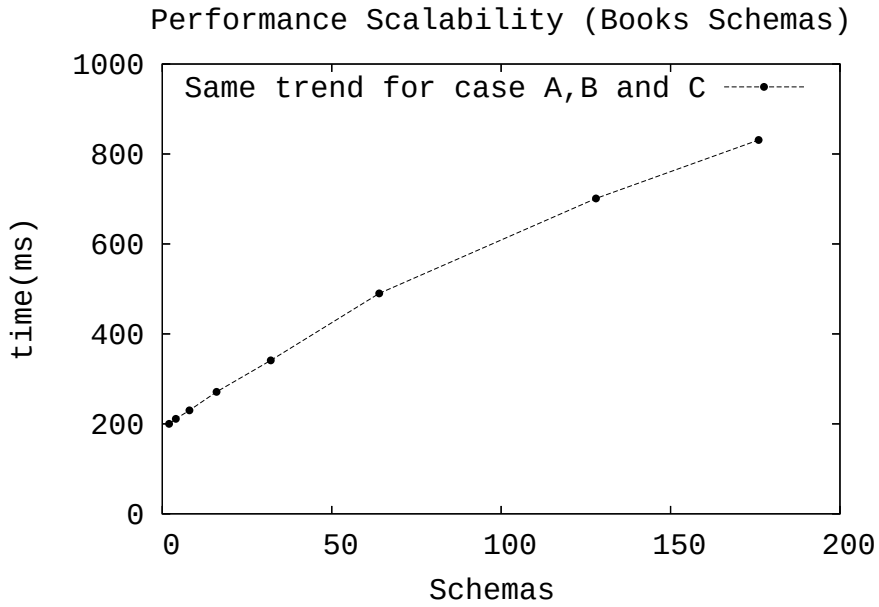

Fig. 16. BOOKS: integration time (ms) as a function of the number of schemas.

5.1 Performance Evaluation

Performance is evaluated as the number of schemas or nodes processed versus the time required for matching, merging and mapping. We selected three sets of schema trees from different domains, shown in Tab. 3. OAGIS and xCBL are real schemas, whereas BOOKS are synthetic schemas.

Experiments were performed with different numbers of schemas (2 to 176). Algorithm performance was timed under three different similarity scenarios:

A) Label String Equivalence,
B) Label Token Set Equivalence,
C) Label Synonym Token Set Equivalence.

Figure 16 shows a comparison of three similarity scenarios, A, B, and C, for sets of 2, 4, 8, 16, 32, 64, 128, and 176 schemas from BOOKS. There is no visible difference in the performance of various matchers. This is possibly due

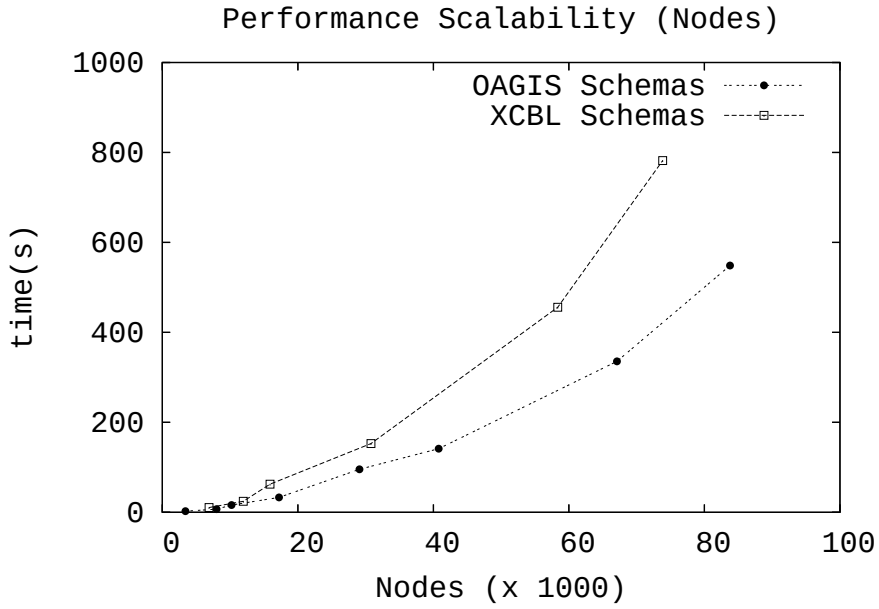

Fig. 17. Comparison of schema integration times (seconds) for real web schemas.

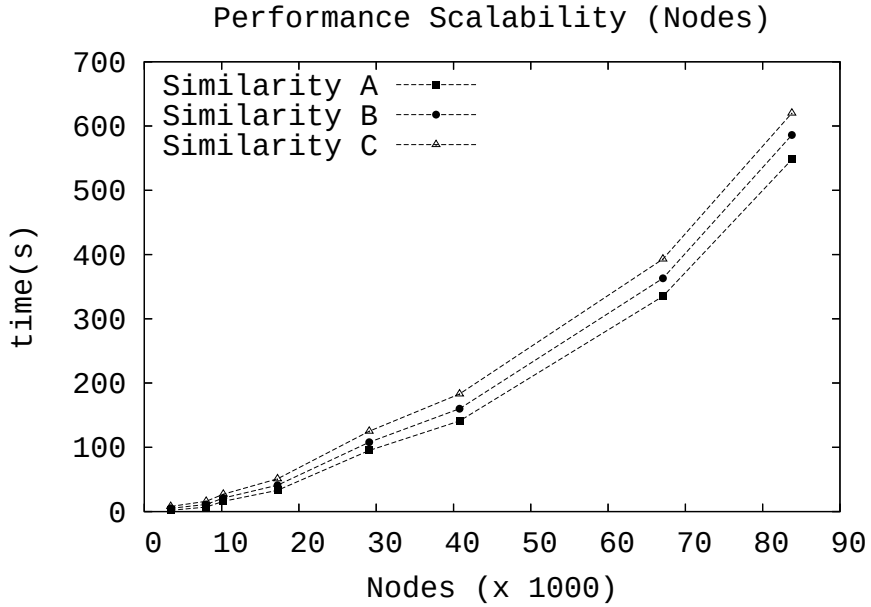

Fig. 18. Integration of OAGIS schemas.

to the fact that synthetic schemas vary little in their labels.

Figure 17 shows time in seconds for OAGIS and xCBL. The execution time for PORSCHE depends upon the number of schemas to be integrated, and appears to be quadratic in the number of nodes, as predicted by the complexity analysis (Sec. 4.4). Figures 18 and 19 show the time in seconds against the number of nodes processed for the three similarity methods (A, B, and C), for xCBL and OAGIS schemas. xCBL schemas (Fig. 19) are slower to match than OAGIS schemas (Fig. 18). This is due to the higher average number of nodes in xCLB schemas. It takes approximately 600 s to match 80 OAGIS schemas,

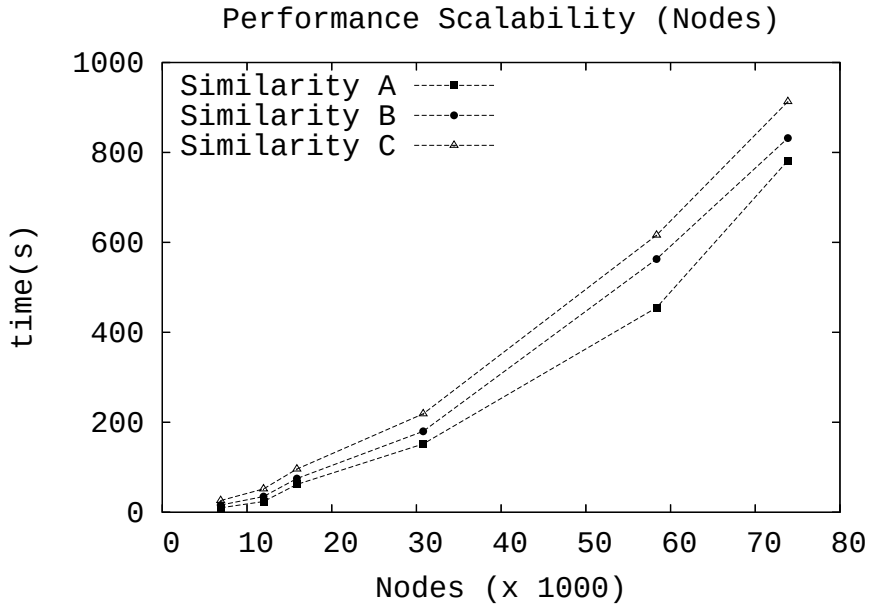

Fig. 19. Integration of xCBL schemas.

while 44 xCLB schemas require about 800 s.

In both cases there is a slight difference in matching times for categories A, B and C (Fig. 18 and 19), due to different label matching strategies. A is the fastest, as it works only on the labels. B is slightly slower, as labels have to be tokenised, and C is the slowest, as one needs to match synonyms as well. These evaluation cases show that PORSCHE has acceptable performance on an office PC for a large number of schemas.

5.2 Mapping Quality

As available benchmarks focus on two schemas at a time, and our approach targets a set of schemas, we can only compare mapping quality for two schemas at a time. From the set of 176 books schemas we randomly selected 5 pairs of schemas (sizes ranging between 5 and 14 nodes), from purchase order one schema pair (schema sizes 14 and 18 nodes), and from OAGIS two schema pairs (schema sizes from 26 to 52 nodes) [5]. We computed mapping quality for COMA++ and SF:Similarity Flooding (RONDO) and compared those to PORSCHE. The quality of mappings is evaluated using precision, recall and F-measure. The results are summarised in Figures 20, 21, 22 and 23. For PORSCHE, similarity uses token level synonym lookup.

The comparison shows almost equivalent mapping quality for PORSCHE and COMA++, since both are driven by similar user defined pre-match effort of

[5] Schemas and results detail at http://www.lirmm.fr/PORSCHE/results/

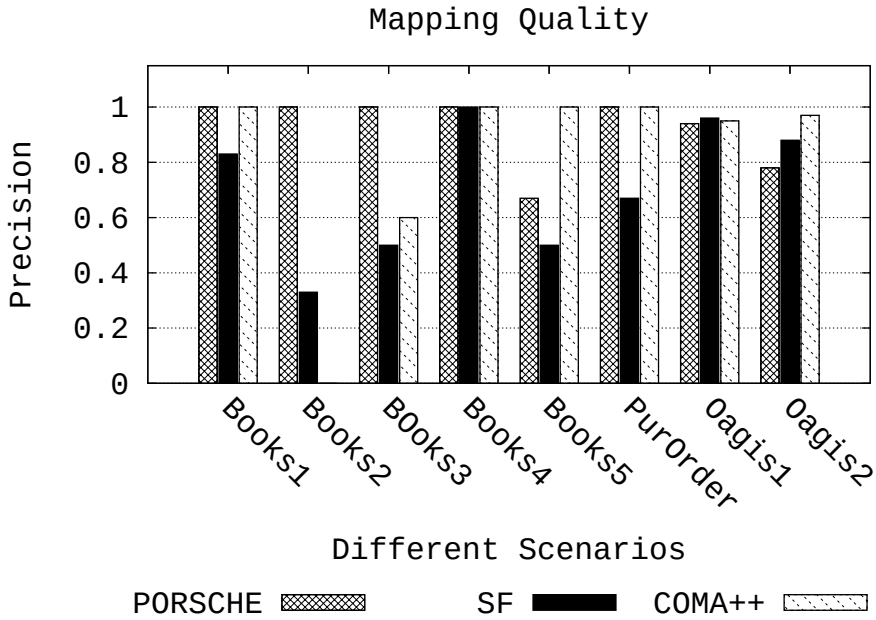

Fig. 20. Precision for PORSCHE, COMA++ and Similarity Flooding for 8 pairs of schemas.

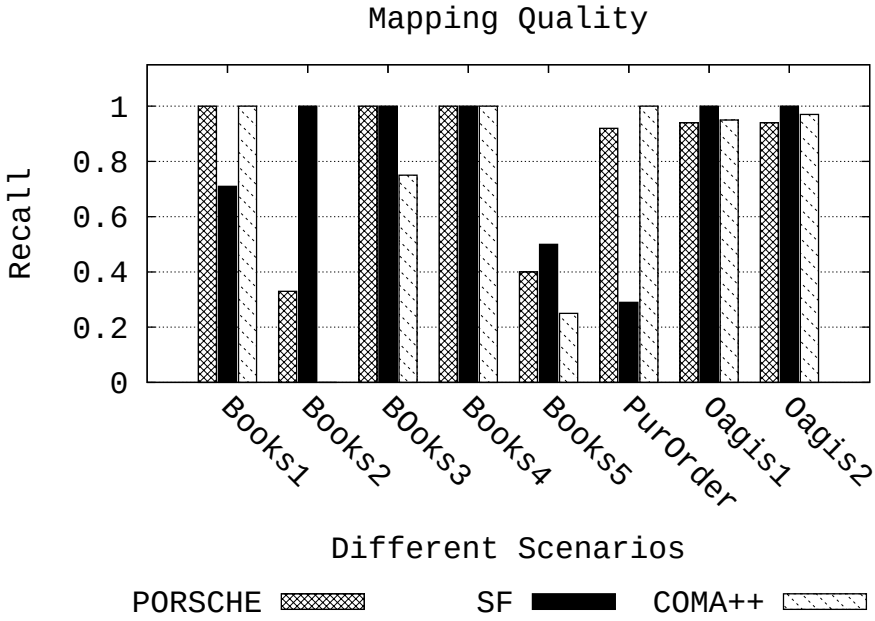

Fig. 21. Recall for PORSCHE, COMA++ and Similarity Flooding for 8 pairs of schemas.

constructing synonym tables used to derive label similarity. Similarity flooding demonstrates better recall than PORSCHE or COMA++. The results also demonstrate local schema differences. Books2 is the hardest case, and we discovered that it contained inverted paths (book/author is matched to author/book).

PORSCHE has also been evaluated within our benchmark, XBenchMatch[8]. The benchmark uses precision, recall, F-measure, overall measure, and other

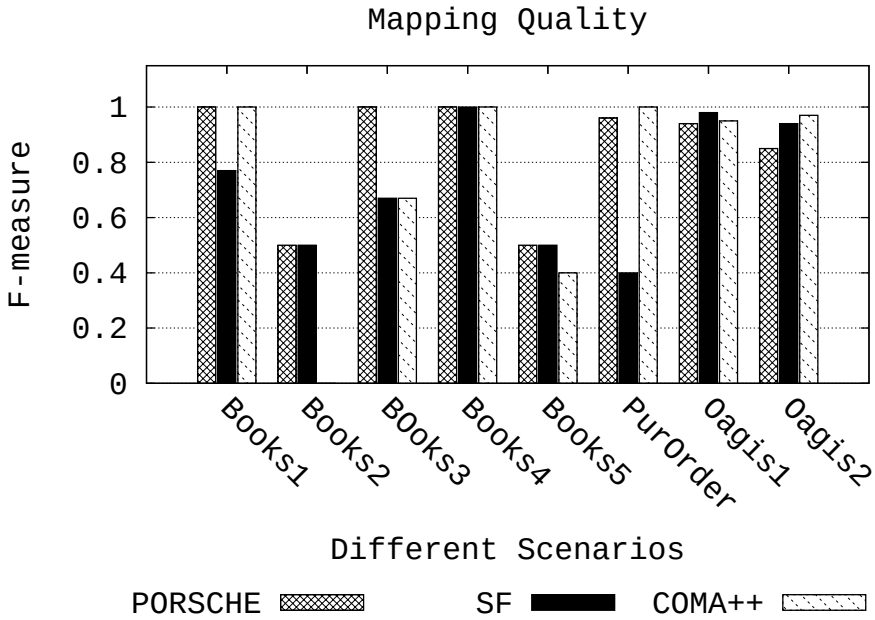

Fig. 22. F-measure for PORSCHE, COMA++ and Similarity Flooding for 8 pairs of schemas.

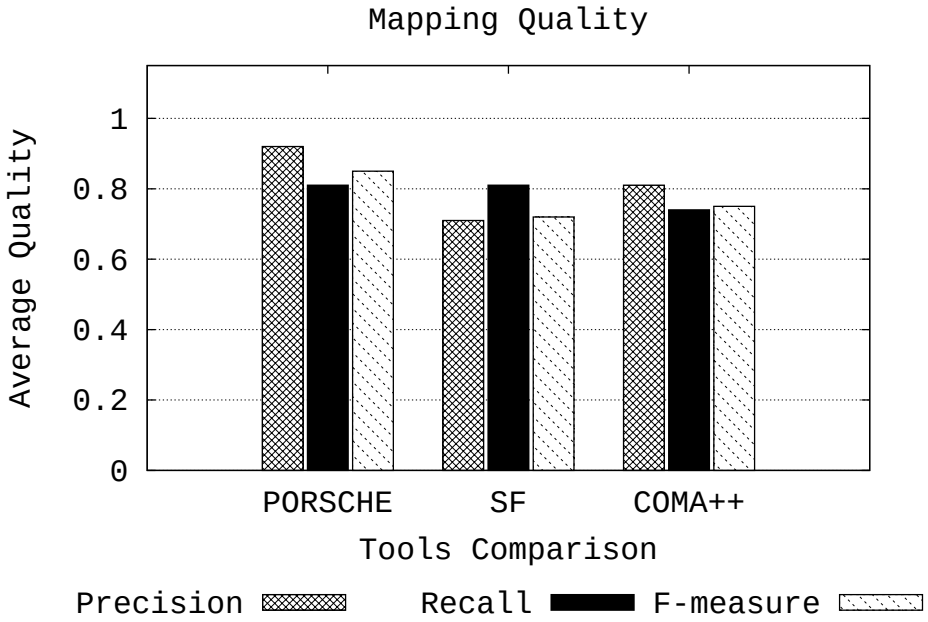

Fig. 23. Global comparison of quality metrics.

metrics of structural schema proximity. Structural proximity measures are based on differences in the number and size of sub-trees shared between input and mediated schemas, and schema proximity compares two schema trees which are being matched. PORSCHE demonstrated good results in comparison with COMA++ and RONDO (Similarity Flooding).

We gave a brief overview of the integrated schema quality evaluation in Sec. 2. Since our test domains are XML schema instances with only element label information, type consistency can not be evaluated. In the following, we consider only completeness and minimality. As discussed in Sec. 2.2 and algorithm implementation (Fig. 11, 12 and 13), when a match is not found in the mediated schema, a new concept node is added to it. Mapping from source schema tree to the new node is then established. This demonstrates that PORSCHE inherently fulfills the completeness criterion of mediated schema integrity.

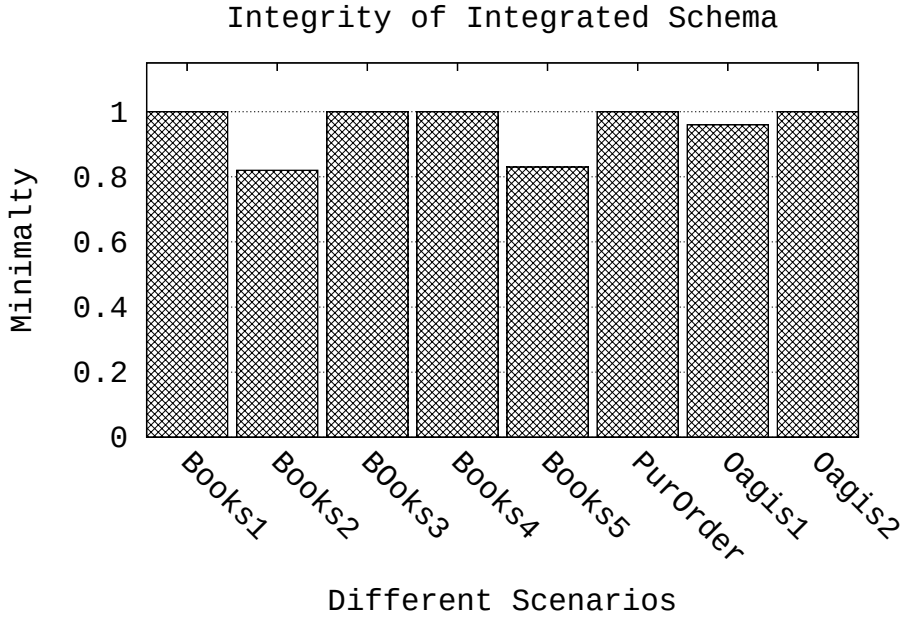

Fig. 24. Minimality of PORSCHE schema integration for the selected pairs of schemas.

To evaluate minimality, we take the pairs of schemas considered in mapping quality evaluation. For each pair of schemas we perform matching and merging. The resulting integrated schema is scrutinised manually for redundancies. The results ($minimalty = 1 - (redundantNodes/totalNodes)$) are shown in Fig. 24. An inspection of integrated schemas showed that minimality decreased where input schemas being matched had inverted paths. To further investigate the integrity of our approach, we calculated the minimalty of the integrated schema created by merging 176 books schemas (small in size and amenable to human inspection). We used an ideal integrated schema (manually created, containing 17 nodes) to assess redundancy. A batch of 176 experiments was performed, each selecting one schema as the seed mediated schema. Figure 25 shows that minimality fluctuates between 0.68 and 0.85.

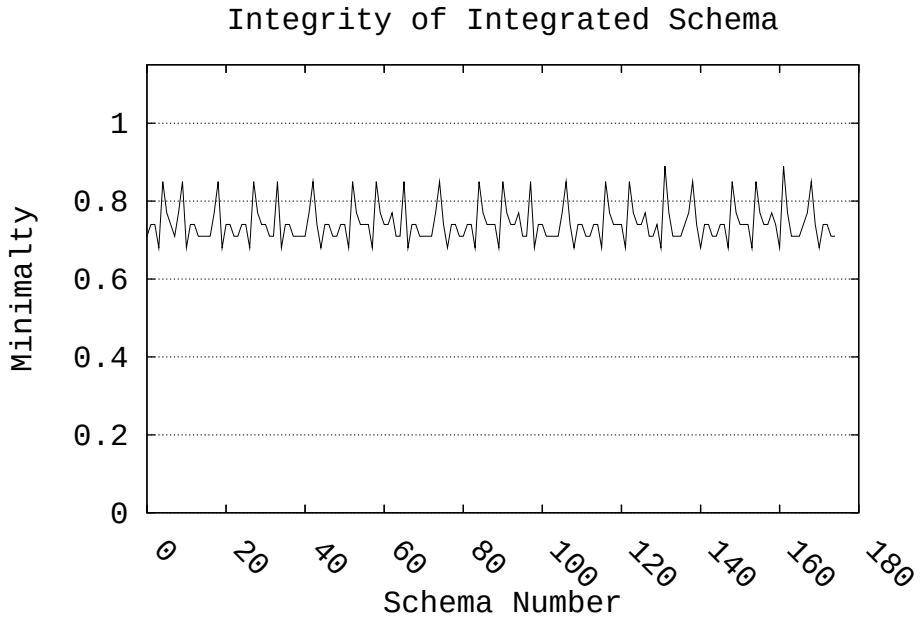

Fig. 25. Minimality in PORSCHE for 176 book schemas. 176 experiments are shown (along the x-axis), with each schema selected, in turn, as the initial mediated schema.

6 Related Work

Most schema-matching systems compare two schemas at a time and aim for quality matching but require significant human intervention. CUPID[16], COMA++[6], S-Match[10], QOM[9], GLUE[7] are some of these tools presented in this section. Several surveys[6,21,22] argue that extending the matching to data integration is time consuming and limited in scope. Matching of two large bio-medical taxonomies has been demonstrated by Do and Rahm using COMA++, and Mork and Bernstein [18] using CUPID and similarity flooding (RONDO[17]). Quick Ontology Matching (QOM) matches large ontologies with performance in view. Large scale schema matching has also been investigated in the web interface schema integration[12,24] using data mining. PORSCHE's goal is to match, merge and map in a hybrid manner, whereas most of the tools separate the two activities of matching and integrating, which makes them unsuitable for automated integration scenarios, as needed in e-commerce.

COMA++[6] is one of the most recent matching and merging tools. It is a composite matcher producing quality matches for a pair of schemas. It provides an extensible platform which can combine several matchers. It uses user defined synonym and abbreviation tables as a pre-mapping effort. It has a comprehensive interface for navigating through the matches produced by the software. The user verifies and selects or edits the match results to finalise the mapping between the two schemas. Based on these mappings, COMA++ can also generate a merged schema, which has to be validated by the user.

The authors of COMA++ give a comprehensive evaluation of match quality results but do not quantify the quality of the merged schema. In large scale schema matching, COMA++ requires a significant amount of human intervention. First, the user identifies fragments in the two schemas to be mapped. This aims to manage the namespace/ include characteristics of XML schemas. Then, individual fragments from the source schemas are mapped to fragments of the target schema, one by one, and match results are saved in a relational database for subsequent comparison. COMA++ uses a bottom up hierarchical approach: if leaves are similar, then parent nodes may also be similar. In PORSCHE we use a reverse approach: given similarity between parent or ancestor nodes, we hope to discover similarity among the descendants. We believe that our approach is more appropriate in single domain schemas with large tree depth, which requires matching of the structural context of descendant nodes in a robust manner.

S-Match[10] is a hybrid matcher which carries out semantic matching by using the WordNet dictionary. It tackles the matching as a propositional satisfiability problem. It demonstrates better mappings than COMA++ and CUPID[16] but has worse performance. S-Match employs 16 (13 element level and 3 structural level) match algorithms. It does not fulfill the requirement for merging and creating mappings from the source to the integrated schema. PORSCHE specifically addresses the data integration needs of large environments where efficiency is important.

Mork and Bernstein [18] present a case study of matching two large ontologies of human anatomy. They use lexical and hierarchical matching modules from CUPID[16] and a structural algorithm called Similarity Flooding[17] to find mappings. The hierarchical algorithm goes one step further than COMA++. The similarity of descendants is used to evaluate ancestor similarity, and the approach is not limited just to parent-child relationships. The authors argue that the hierarchical approach produced disappointing results because of differences in context. They report that a lot of customisation was required to get satisfactory results. Our approach, in comparison, considers only top-down similarity, and looks for matching ancestors only. It seems that further research is needed to properly define the role of node context in XML, in particular the scope of context, in terms of node distance and structure within a tree.

QOM[9] is a semi-automatic (RDF based) ontology mapping tool. It uses heuristics and ontological structures to classify candidate mappings as promising or less promising. It uses multiple iterations, where in each iteration the number of possible candidate mappings is reduced. It employs label string similarity (sorted label list) in the first iteration, and, afterwards, it focuses on mapping change propagation. The structural algorithm follows top down (level-wise) element similarity, which reduces time complexity. PORSCHE matching also uses label similarity using linguistic rather than lexical algo-

rithms. In QOM, in the second iteration, depth-first search is used to select the appropriate mappings from among the candidate mappings.

Another interesting schema matching domain under active research is matching across query interfaces of structured web databases. Web page layout forms a hierarchy backed by a database schema. For certain Web domains, such as travel, these interfaces are very numerous. [12,24] handle holistically the integration of these structured layouts as a mining problem. He and colleagues [12] observe that Web database query interfaces in the same domain are usually semantically similar, as a label is often unambiguous in a domain but it can have several meanings in a dictionary, and synonymous labels are rarely co-present in the same schema. However, grouping of elements such as *LastName* and *FirstName* under the same parent is common, as those together form a larger concept. He et al.[12] utilise data mining techniques on the input forms and data ranges, for elements available from the web pages. The technique is more biased towards the accuracy of integration than performance. Thus, it is ideal for scenarios where the schemas are small, as in web query/ data interfaces. Our method targets tree schema structures (or schemas which can be converted to trees) which come with minimum information (just element labels) and each schema can have thousands of elements. This gives PORSCHE a much broader application domain.

Clustering, both at the element and schema level, is often used in matching and merging. Smiljanic and co-authors[23] suggest element clustering within a schema, in a repository of schemas which are matched to a given person schema. The technique shows how the combinatorial nature of the matching problem can be reduced to polynomial, and, further, to linear complexity, by using element clustering. The method only caters for schema matching. Lee et al. [14] present an integration method based on the clustering of similar DTDs of XML sources. The method works incrementally, by creating clusters of similar schemas. Similar schema selection is based on element similarity. The solution uses external oracles defining element similarity. It mainly focuses on the integration process but lacks the mediation aspect.

Instance level schema matching tools work on a sample mapping set. They use data and example sets to learn a strategy to compute equal instances and concepts, and are characterised by a high time complexity. For example, GLUE[7] uses Relaxation Labelling to calculate schema mappings: mapping assigned to an entity is typically influenced by the features of the node's neighbourhood in the graph. Multiple iterations have to be performed to confirm a mapping. Overall, such methods are slow. An advantage of instance based learners is their capability to learn mapping expressions, as illustrated in iMAP[5], a variant implementation of GLUE.

7 Lessons Learned

Implementing PORSCHE gave us a good insight into the existing schemas and their properties. We learned that newer standard schemas available on the web can be used together with linguistic techniques, and that string similarity should not be the sole criterion for label matching. Element names are now becoming more structured and meaningful, and new matching techniques are increasingly based on external oracles like WordNet. Linguistic label similarity can help in minimizing the target search space for matching very significantly. In a large scale scenario, performing clustering just once will enhance performance. The application of tree mining further reduces the time to select the correct target for mapping.

For schema integration purposes, the selection of one of the schemas as the seed for the initial mediated schema follows no fixed rule. We tried random, the smallest and the largest schema from the input set of schemas as the initial selection. However, using such heuristics showed that the selection depending on schema size does not boost performance. Rather, it is the structure of the initial seed schema which influences performance: if the initial mediated schema contained more frequent sub-trees shared with the set of input schema trees, the speed of matching improved.

Overall, the architecture of PORSCHE is flexible, and can accommodate new syntactic and linguistic similarity algorithms. Most importantly, PORSCHE is scalable, as demonstrated, and can be used in large-scale data integration. At present, it uses a limited array of linguistic methods but its domain specific matching quality is approximately equivalent to other current tools.

In the future, we will investigate the application of this technique in information systems based on P2P architectures. Secondly, we want to enhance the linguistic matching part of the system. Our study of the tree mining technique reveals that it can be utilised to identify relationships between the elements and groups of elements within a single tree and in a forest of schema trees. This will help in identifying subsumptions and overlaps for $n : m$ complex mappings[19]. Another possible extension is the development of persistent indexes for incremental matching.

8 Conclusions

We presented a novel schema integration method, PORSCHE, which has shown very promising results for large scale schema integration. It uses a tree based depth-first traversal algorithm for matching, merging and mapping

a set of schema trees. To improve performance, we adapted a technique from tree mining used in the clustering of similar node labels. This minimises the target search space for a node match and improves performance. PORSCHE uses an optimistic top down depth-first match traversal (parents are mapped before children, and the left sub-tree is traversed before the right sub-tree), since our assumption is that we utilise it in a domain specific environment. This helps in using the structural contextual semantics of nodes for better quality matching.

The novelty of our method is fourfold. First, we support automated schema matching. Second, we not only generate matches, but also build an integrated schema at the same time. Third, our approach scales to hundreds of schemas. Fourth, the use of tree mining techniques for schema matching is also new in this field.

Acknowledgements: ZB is supported by ANR-05-MMSA-007 and EH is funded by an EU Marie Curie Fellowship.

References

[1] T. Asai, H. Arimura, and S. Nakano. Discovering Frequent Substructures in Large Unordered Trees. In *ICDS*, pages 47–61, 2003.

[2] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparitive Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[3] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-Strength Schema Matching. *SIGMOD Record*, 33(4):38–43, 2004.

[4] M. da Conceicao Moraes Batista and A. C. Salgado. Information Quality Measurement in Data Integration Schemas. In *Quality in Databases, VLDB workshop*, 2007.

[5] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Semantic Matches between Database Schemas. In *ACM SIGMOD*, pages 383–394, 2004.

[6] H.-H. Do and E. Rahm. Matching large schemas: Approaches and evaluation. *Information Systems*, 32(6):857–885, 2007.

[7] A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A. Y. Halevy. Learning to match ontologies on the Semantic Web. *VLDB J.*, 12(4):303–319, 2003.

[8] F. Duchateau, Z. Bellahsene, and E. Hunt. XBenchMatch: a Benchmark for XML Schema Matching Tools. In *VLDB*, pages 1318–1321, 2007.

[9] M. Ehrig and S. Staab. QOM – Quick Ontology Mapping. In *ISWC*, pages 683–697, 2004.

[10] F. Giunchiglia, P. Shvaiko, and M. Yatskevich. S-Match: an Algorithm and an Implementation of Semantic Matching. In *ESWS*, pages 61–75, 2004.

[11] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management Systems. In *ICDE*, pages 505–516, 2003.

[12] B. He, K. C.-C. Chang, and J. Han. Discovering complex matchings across web query interfaces: a correlation mining approach. In *KDD*, pages 148–157, 2004.

[13] A. Jhingran. Enterprise Information Mashups: Integrating Information, Simply – Keynote Address. In *VLDB*, pages 3–4, 2006.

[14] M.-L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: clustering XML schemas for effective integration. In *CIKM*, pages 292–299, 2002.

[15] J. Lu, S. Wang, and J. Wang. An Experiment on the Matching and Reuse of XML Schemas. In *ICWE*, pages 273–284, 2005.

[16] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *VLDB*, pages 49–58, 2001.

[17] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD*, pages 193–204, 2003.

[18] P. Mork and P. A. Bernstein. Adapting a Generic Match Algorithm to Align Ontologies of Human Anatomy. In *ICDE*, pages 787–790, 2004.

[19] T. Pankowski and E. Hunt. Data Merging in Life Science Data Integration Systems. In *Intelligent Information Systems*, pages 279–288, 2005.

[20] C. Pluempitiwiriyawej and J. Hammer. Element matching across data-oriented XML sources using a multi-strategy clustering model. *Data and Knowledge Engineering*, 48(3):297–333, 2003.

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

[22] P. Shvaiko and J. Euzenat. A Survey of Schema-Based Matching Approaches. *J. Data Semantics IV*, pages 146–171, 2005.

[23] M. Smiljanic, M. van Keulen, and W. Jonker. Using Element Clustering to Increase the Efficiency of XML Schema Matching. In *ICDE Workshops*, page 45, 2006.

[24] W. Su, J. Wang, and F. Lochovsky. Holistic Query Interface Matching using Parallel Schema Matching. In *ICDE*, pages 122–125, 2006.

[25] M. J. Zaki. Efficiently Mining Frequent Embedded Unordered Trees. *Fundamenta Informaticae*, 66(1-2):33–52, 2005.