Using Element Clustering to Increase the Efficiency of XML Schema Matching

Marko Smiljanić
University of Twente
The Netherlands
m.smiljanic@utwente.nl

Maurice van Keulen
University of Twente
The Netherlands
m.vankeulen@utwente.nl

Willem Jonker
Philips Research and
University of Twente
The Netherlands
willem.jonker@philips.com

Abstract

Schema matching attempts to discover semantic mappings between elements of two schemas. Elements are cross compared using various heuristics (e.g., name, data-type, and structure similarity). Seen from a broader perspective, the schema matching problem is a combinatorial problem with an exponential complexity. This makes the naive matching algorithms for large schemas prohibitively inefficient. In this paper we propose a clustering based technique for improving the efficiency of large scale schema matching. The technique inserts clustering as an intermediate step into existing schema matching algorithms. Clustering partitions schemas and reduces the overall matching load, and creates a possibility to trade between the efficiency and effectiveness. The technique can be used in addition to other optimization techniques. In the paper we describe the technique, validate the performance of one implementation of the technique, and open directions for future research.

1. Introduction

Schema matching attempts to discover semantic correspondence between the elements of two schemas. Schemas are designed by humans – they are a product of human creativity. As such, two schemas, even if they have an identical meaning, can be quite different on the syntactic level. This makes schema matching a very difficult problem, even for humans.

Schema matching is a crucial activity in the design of many interoperable applications. Driven by high demand, many schema matching systems have been developed [5, 6, 15, 16, 18]. In these systems, the similarity between the elements of two schemas is computed by exploiting various heuristics, or, hints. Hints exploit properties of schemas, such as node names, data-types, or schema structure. Hints can also use external data sources such as data instances, dictionaries of synonyms, and results of previous matchings. Schema matching research has shown that the more hints are used, the greater the effectiveness of the matching system.

We place our research in the context of *personal schema querying* – a technique which could be the next approach for querying XML data on the Internet. In this approach a user, unfamiliar with to the structure of the XML data on the Internet, wants to query that data. The user first provides a *personal schema* – his own virtual view on the unknown data. A simple personal schema s is shown if Fig. 1. A schema matching system then matches the personal schema against the schemas of the Internet, which are, for example, all stored in a large XML schema repository. The user is presented with a ranked list of mapping choices generated by the schema matching system. The user asserts the choices and picks one to be used to retrieve the actual data. The user can then provide a query defined in terms of his personal schema, say, an XPath query */book[title="Iliad"]/author*. A query evaluation system rewrites the query into queries over the real data sources, and evaluates the real-data queries.

Personal schema querying technique is envisioned as an on-line interactive querying technique. As such, it must not only be effective but also very efficient. In this paper, we are investigating ways to improve the efficiency of the schema matcher, which is the core component in the personal schema querying technique. Throughout the paper, we use the terms *personal schema* and *repository schema* to denote two schemas which are being matched.

Seen from a broader perspective, the schema matching problem is a combinatorial optimization problem [19]. The number of all possible mappings between two schemas grows exponentially with sizes of schemas being matched. To find the best mappings for two schemas, the schema matcher has to search through all the possible mappings. In large scale applications, the number of possible mappings explodes, and the naive schema matching algorithms become prohibitively inefficient.

In this paper we propose and investigate the *clustered schema matching* technique which is designed to improve the efficiently of the existing schema matching systems. Clustering is used to quickly identify regions, i.e., clusters, in the large schema repository which are likely to produce good mappings for a personal schema. The schema matcher then needs to look for mappings only within clusters. This reduces the matching workload and improves the efficiency. Improved efficiency comes with a penalty: due to clustering not all mappings can be discovered. Still, the clustering technique is designed in a way that preserves schema mappings which rank high, and only loose mappings which rank low.

In the paper we make the following contributions.

- We propose clustered schema matching technique for improving the efficiency of schema matching.
- We propose and validate the k-means algorithm as one specific implementation of the clustering algorithm used in clustered schema matching.

This paper has the following structure. Sec. 2 starts by an overview of the schema matching problem and techniques used by schema matchers to solve the problem. The section then introduces the clustered schema matching technique. Sec. 3 describes our experimental schema matching system Bellflower. Sec. 4 describes the k-means clustering algorithm – a concrete implementation of the clustered schema matching technique. In Sec. 5, experimental results are analyzed in order to validate the expected properties of the clustered schema matching technique. Sec. 6 discusses related work.

2. Overview

This section introduces the *clustered schema matching* approach in three steps: • step one formalizes the *schema matching problem* [19]. • Step two describes how current schema matching systems solve the schema matching problem. The architecture shared by these systems is presented. • Step three adds clustering to this architecture and discusses how clustering is expected to improve the efficiency of schema matching.

2.1. Schema matching problem

The definition of the *schema matching problem* is preceded by two other definitions. First the *schema graph* is defined as a data model for representing XML schemas. Second, the *schema mapping* is defined; the schema matching problem is solved by discovering schema mappings.

We use the schema matching problem illustrated in Fig. 1 as a running example: personal schema s, created by a user interested in books, is to be matched against a schema repository R. The figure shows a small fragment of the repository. Arrows pointing to the gray subtree t depict a schema mapping – one possible solution for this matching problem.

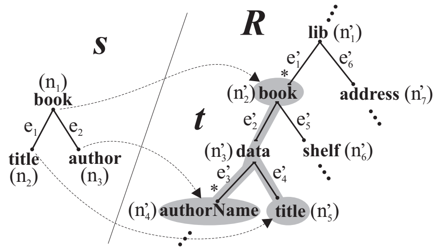

Figure 1: personal schema s, schema repository R, and one possible schema mapping $s \mapsto t$ (t is the gray subtree of R)

Definition 1 *Schema graph* $PS = (N, E, I, H)$ is a data structure used to represent an XML schema, where: • N is a set of nodes (elements), e.g., $N_s = \{n_1, n_2, n_3\}$ (see Fig. 1), • E is the set of edges, e.g., $E_s = \{e_1, e_2\}$, • I is the incidence function associating each edge to its source and target nodes, e.g., $I_s(e_1) = (n_1, n_2)$, and • H is a function which assigns *(property, value)* pairs to nodes and edges in the tree. We write *name*(n_1) = *"book"* to specify that particle n_1 has the property *name* with the value *"book"*. • A *path* is an alternating sequence of nodes and edges where consecutive edges have exactly one common node, e.g., $p' = n'_3 - e'_2 - n'_2 - e'_5 - n'_6$. We overload the incidence function I to support paths; e.g., $I_R(p') = (n'_3, n'_6)$. In this paper, terms *node* and *element* are used interchangeably.

Definition 2 (notation: $\diamond$ reads "*such that*", $x \mapsto y$ reads "*x maps to y*") When matching schema graphs s and R, $s \mapsto t$ is called *schema mapping* if t is a subgraph of R and if the following holds (Fig. 1 illustrates one mapping):

- $\forall n \in N_s,\ \exists_1 n' \in N_t \ \diamond\ n \mapsto n'$
  Each node n in s must be mapped to exactly one node n' in t.
- $\forall e \in E_s,\ I_s(e) = (u, v),\ u \mapsto u',\ v \mapsto v'$
  $\exists_1 p' \in$ *paths(t)*, $I_t(p') = (u', v') \ \diamond\ e \mapsto p'$
  where *paths(t)* is a set of all paths in t.
  Each *edge* e in s must be mapped to exactly one *path* p' in t (this *edge-to-path* mapping rule is a practical simplification of a more general *path-to-path* rule)

The first point in Def. 2 restricts a mapping to what is commonly known as "1 to 1" element mapping [18]. Most of the existing schema matching systems are designed to discover "1 to 1" mappings [7], and we have chosen to first investigate clustered schema matching in the context of such systems. When mapping two elements n and n', we use the following naming convention: $n \mapsto n'$ is called *element mapping*, n is the *mapped element* and n' is the *mapping element*. In the paper, it is always clear from the context which of the tree concepts is being discussed. The second point in Def. 2 is rarely considered in related schema matching systems. This is because most of the systems model schemas as trees, and this point is needed only when modeling schemas as graphs; in graphs there can exist more than one path between two nodes, and to define a mapping it must be clear which of the paths is used.

Def. 2 specifies a schema mapping on a syntactic level, but does not say anything about the semantic correctness of a mapping. To determine the semantic correctness of a schema mapping $s \mapsto t$, a schema matching system uses an *objective function* $\Delta(s, t) \rightarrow [0, 1]$. The objective function computes the *similarity index* for two schemas. The similarity index is an indication of the semantic similarity of the two schemas; the larger the index, more likely that the mapping is correct. Note that only humans can judge if a mapping is correct or not. The objective function only approximates this human judgment [19].

In practice, schema matching systems are built to deliver *top-N* mappings, or mappings with the similarity index above certain numerical threshold δ. It is expected that most of the correct mappings are comprised within the set of highly ranked mappings. Based on this, we define the schema matching problem as follows.

Definition 3 *Schema matching problem* is a quadruple $P = (s, R, \Delta(s, t), \delta)$ where s is a personal schema schema graph, t is a subgraph of the repository schema graph R such that $s \mapsto t$ is a schema mapping, $\Delta(s, t)$ is the objective function, and δ is the objective function threshold. The *solution of the schema matching problem* P is a sorted list of all possible schema mappings $s \mapsto t$ for which $\Delta(s, t) \geq \delta$. The list is sorted on the value of similarity index. A more detailed formalization of the XML schema matching problem can be found in [19].

In the remainder of the paper, we consider only XML schemas which can be represented as *schema trees*, as opposed to *schema graphs*. This is a common simplification in the schema matching research. Consequently, personal schema s is a tree, and the repository schema R is a collection of a large number of trees, i.e., a forest. For brevity, in the expressions we shall treat the repository schema R as being a single large tree. In our experiments, however, the repository is a forest. Work with *schema graphs* is future research.

2.2. Existing approaches for solving the schema matching problem

A number of existing schema matching approaches such as Cupid [16], COMA [5], and LSD [6], share, with a degree of distinction, a common schema matching architecture. This common architecture is shown in Fig. 2.

The main input into a schema matching system are two schemas being matched, e.g., a personal schema and a repository schema ①. Matching starts by comparing every element of the personal schema with every element of the

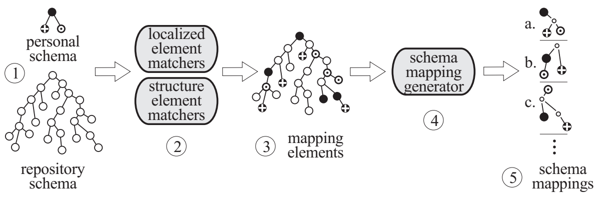

Figure 2: Basic architecture of schema matching systems

repository. Elements are compared using different *element matchers* ②. Each element matcher uses different heuristics to compute a *similarity index*. Matchers can be divided into two groups depending on the type of information they use to compute the similarity index: *localized matchers* and *structure matchers*. Localized matchers compute the similarity index by using only local properties of schema elements, such as element names or element data types. For example, COMA compares element names, name synonyms, and data types to compute the similarity index. Structure matchers take into account the structural properties of elements such as relations with other elements in a schema graph. For example, Cupid uses a *TreeMatch* operator which computes the similarity of structural contexts of elements being compared.

For every element pair being compared, each matcher produces a different similarity index. These indexes are combined into a single similarity index by means of weighed average or other combining techniques [5, 6] (this step is not illustrated in Fig. 2). Element pairs n, n' with non-zero similarity index become *element mappings* $n \mapsto n'$. Each personal schema node can be mapped to multiple repository elements. For example, the black element of the personal schema ① is mapped to four different mapping elements in the repository ③ (also depicted as black nodes). The number of mapping elements is also expected to be proportional to the size of the repository.

With the *mapping elements* known, in the next step the *schema mapping generator* ④ is used to generate *schema mappings* ⑤. Schema mappings are formed by making various combinations of mapping elements. Mapping generator uses an objective function to compute the similarity index for different schema mappings, and to rank the mappings accordingly.

The presented architecture has two sources of computational complexity: the *complexity of element matchers* and the *complexity of the schema mapping generator*. The complexity of element matchers is individual for each matcher. Various techniques, such as approximate string joins [10] or quick computations of structural relations through node labeling [11, 12], are used to implement element matchers efficiently. The mapping generator has to look for mappings within an exponentially growing search space. This search space is a set of all possible schema mappings the number of which can be expressed as $O(|ME_n|^{|N_s|})$, where s is the personal schema and ME_n is the set of mapping elements for each node in s. To handle such large search space, schema matchers use efficient search algorithms, e.g., *beam search* which is used in iMap system [4] or $A*$ algorithm used in LSD [6].

In the next section we propose a clustering based technique for further improving the efficiency of schema matching. The technique is orthogonal to techniques used in existing systems and can be used in addition.

2.3. Clustered schema matching technique

In a nutshell, clustered schema matching works as follows. Clustered schema matching identifies regions in the repository schema R which are likely to comprise good schema mappings t for a certain personal schema s. The schema matcher then looks for mappings only within these regions, instead of searching through the repository as whole. This reduces the workload and improves the efficiency of schema matching. The personal schema is not affected by clustering.

In a search for an algorithm which can quickly generate such regions we have resorted to clustering. Hence, we call such regions *clusters* and we call the technique *clustered schema matching*. The clustered schema matching technique is built by adding clustering to an existing (*non-clustered*) schema matching system.

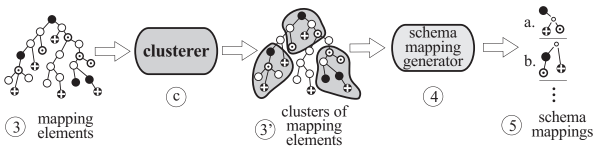

Figure 3: Clustered schema matching

We present the simplest, and the most generic way to add clustering to an existing schema matching system. Fig. 3 illustrates the approach. The components ①, ②, and ③ remain the same as in the non-clustered matching system (see Fig. 2). The difference lies in the addition of the clusterer ⓒ which groups mapping elements ③ into *clusters of mapping elements* ③'. The rest of the matching process continues by sending each cluster individually to the schema mapping generator ④ which delivers schema mappings. Schema mappings coming from individual clusters are all placed together in a single ordered list ⑤.

This approach reduces the workload of the schema mapping generator by reducing the size of the search space within which mappings are to be looked for. In the non-clustered case the mapping generator had to traverse the search space with the size $O(|ME_n|^{|N_s|})$, explained above. In the clustered case, the nodes in the repository are partitioned into c clusters, with each cluster having approximately $\frac{|ME_n|}{c}$ elements. Mapping generator considers each cluster independently. Consequently, the search space size in the clustered matching approach is $O(c \cdot \left(\frac{|ME_n|}{c}\right)^{|N_s|})$. The clustering algorithm can be tuned to keep the $\frac{|ME_n|}{c}$ ratio constant with the varying size of repository, e.g., by creating more clusters in larger schemas. In such a case, the problem complexity changes from polynomial to linear in respect to the size of the repository schema. When compared to the non-clustered matching technique the clustered schema matching technique reduces the search space $c^{(|N_S|-1)}$ times; the more the clusters (i.e., c) the larger the search space reduction.

This reduction, however, comes with a cost. In an ideal hypothetical case, clusters comprise all the good mappings t, i.e., mapping for which for which $\Delta(s,t) \geq \delta$. In practice, clusters are not ideal, and they cut-out some good mappings by shredding mappings over several clusters. The result of this side effect is the loss of effectiveness. Clustered schema matching is therefore a non-exhaustive matching technique which offers a trade-off between the efficiency and the effectiveness: the more clusters the more efficient schema matching, but the higher the chances of loosing some valuable schema mappings. Such a trade-off is acceptable in many applications. Nevertheless, clustering must try to preserve as much mappings as possible, and in particular the mappings which are ranked high by the objective function. It is desirable to loose only the mappings which rank low. Sec. 4 discusses a concrete clustering algorithm for the use in clustered schema matching.

We have described the simplest and the most generic way to use clustering in schema matching. There exist another, similar but non-generic clustered schema matching technique. This technique heavily depends on the heuristics and the implementation details of individual element matchers ②. In the alternative technique, element matchers are split in two groups. For example, group one comprises localized element matchers, group two comprises structure element matchers. First group is used before the clustering step to produce a set of *preliminary mapping elements*. Clustering groups the preliminary mapping elements into clusters. The second group of matchers is used after the clustering step by considering each cluster individually. We expect that some structure element matchers would have less work, and consequently an improved efficiency, if being applied on clusters, rather than on the whole repository. In this paper we only discuss the generic technique.

When strictly following Def. 2, a cluster can produce a schema mapping, only if it has all the necessary mapping elements: at least one mapping element for each personal schema element. Such clusters are called *useful clusters*. The chances are that some clusters will not have all the needed elements. These clusters do not produce any schema mappings. To overcome this limitation, the definition of a schema mapping should be extended with a notion of *partial schema mapping*. This would enable the discovery of partial mappings in *non-useful* clusters. Such partial mappings might, nevertheless, be valuable to the user. This is future research.

3. Experimental system for clustered schema matching

To investigate the effects of the clustered schema matching, we have developed an experimental schema matching system called *Bellflower*. Bellflower implements clustered schema matching as described in the previous section (see Fig. 3). In this section we describe Bellflower's components in more detail.

Schema repository: Bellflower uses a schema repository built by randomly selecting XML schemas available on the Internet. Google^{TM} search engine was used to discover 1700 non-recursive DTDs and XML schemas with a total number of 178252 element (attribute) nodes distributed over 3889 trees (note, that one schema can have multiple roots, each represented with one tree). A repository of such size, proved to be too big for our experimental framework, and we built several smaller repositories with sizes from 2500 to 10200 elements, by randomly selecting schemas from the collection.

Element matcher: Bellflower uses one element matcher $sim(n, n') \rightarrow [0, 1]$ which compares names of elements n and n'. The matcher is implemented using the *CompareStringFuzzy* [1] function. The *CompareStringFuzzy* function computes a normalized string similarity based on character substitution, insertion, exclusion, and transposition.

Objective function for schema mappings: Bellflower computes the objective function for schema mappings by combining *name similarity indexes* (sim) and the *path length similarity index (path)*. The value of the objective function $\Delta(s, t)$ is computed as follows. Eq. 1 computes the name similarity index for a schema mapping by averaging the name similarities of individual element mappings. Eq. 2 computes the difference in total path lengths of the personal schema s and the mapping schema t. The difference is normalized to $[0, 1]$ by means of a normalization constant K. The value of K is determined using other constraints in the system (e.g., the maximum length of a path).

$$\Delta_{sim}(s,t) = \frac{1}{|N_s|} \cdot \sum_{n \in N_s} sim(n, n') \qquad (1)$$

$$\textit{where } n' \in N_t \textit{ and } n \mapsto n'$$

$$\Delta_{path}(s,t) = 1 - \frac{|E_t| - |E_s|}{|E_s| \cdot K}, \qquad (2)$$

$$\textit{where } K \textit{ is a normalization constant}$$

The two hints are combined using a weighted sum (Eq.3) with parameter α determining the relative importance of the two.

$$\Delta(s,t) = \alpha \cdot \Delta_{sim}(s,t) + (1 - \alpha) \cdot \Delta_{path}(s,t) \qquad (3)$$

This objective function, though simple, simulates the two most important types of heuristics used in schema matching systems. First, Δ_{sim} simulates the heuristic based on localized properties of elements, and two, Δ_{path} simulates the heuristic based on structural properties of schemas. Please note, that our research does not try to develop a new, or better, objective function; the simple objective function used in Bellflower probably has an inferior effectiveness when compared to that of the other schema matching system. Nevertheless, we believe that Bellflower has enough behavioral similarity to the other schema matching system, to present a good platform for investigating the clustered schema matching approach, and even for making conclusions about a broader applicability of the clustered schema matching.

Schema mapping generator: The generator uses an adaptation of the Branch and Bound algorithm (B&B) (see [13], pg. 141). The generator, produces all schema mappings for which $\Delta(s,t) \geq \delta$, where δ is a manually selected threshold. The generator gains efficiency by using a bounding function for an early detection of mappings for which $\Delta(s,t) < \delta$. We omit the implementation details due to the lack of space. In the experimental section, the effects of the *B&B* algorithm will, however, be indicated.

4. Choosing the clustering algorithm

Central part of the clustered schema matching is the clustering algorithm. The algorithm must be efficient to ensure that the efficiency gains induced by clustering in the mapping generation step ($\textcircled{4}$ Fig. 2) are much larger than the overhead generated by the clustering itself ($\textcircled{C}$ Fig. 3). At the same time, the clustering must try to preserve schema mappings which would be produced in the non-clustered matching.

Our choice for the initial implementation is an adaptation of the k-means clustering algorithm ([8], pg. 140). The choice is based on the simplicity and the non-exponential complexity of the algorithm. In the sequel we first introduce the general k-means algorithm, and then propose a concrete implementation in the context of the Bellflower experimental system. Along the way, we discuss general problems that clustering faces in the clustered schema matching.

K-means clustering algorithm: Algorithm 1 shows the k-means algorithm as used in clustered schema matching. Note that the *reclustering* step (line 10) does not exist in the "standard" k-means algorithm.

The k-means algorithm uses the concepts of *element*, *cluster*, *centroid*, and *distance measure*. In this work, cluster elements are in fact *mapping elements* ($\textcircled{3}$ Fig. 3) created in the element matching step. *Clusters* are thus collections of mapping elements. Each cluster is represented with a *centroid*, and the *distance measure* calculates the distance between a mapping element and a centroid.

The algorithm starts with the initialization of centroids (line 1). The initialization is used to "seed" the centroids around which the clusters will be formed. The initialization determines two things: first, the number of clusters to be created, and second, the (approximate) locations around which the clusters will be formed.

Algorithm 1 K-means clustering algorithm

```
1: initialize centroids
2: repeat
3:     for each mapping element do
4:         for each centroid do
5:             compute distance(mapping element,centroid)
6:         end for
7:         assign mapping element to nearest centroid
8:     end for
9:     compute new centroids for all clusters
10:    perform reclustering
11: until convergence criterion is met
```

The iterative part of the algorithm, starts with a nested loop (lines 3 to 8) in which the nearest centroid is determined for each mapping element. A mapping element becomes a member of the nearest centroid's cluster (line 7). After the clusters have been formed, new centroids are calculated (line 9). These new centroids represent clusters in the next iteration. For reasons discussed later, we have introduced a reclustering step (line 10). This step is used to perform additional modifications of clusters. The iterations continue until a *convergence criterion* is met (line 11).

The complexity of the k-means clustering algorithm is $O(c \cdot i \cdot |ME|)$, where c is the number of clusters being formed, i is the number of iterations, and *ME* is the set of all mapping elements in the repository schema.

We continue by describing an implementation of the k-means clustering algorithm. The implementation represents our initial efforts in building the clustered schema matching system.

Initialization of centroids: In clustered schema matching, it is hard to determine beforehand the number of clusters; depending on the matching problem, the number of regions in the repository, which comprise good mappings, varies. For this reason, we let the initialization seed large number of centroids, which we then bring to a desirable number by means of *reclustering* (described below).

We have explored various heuristics for initializing the centroids, one of which we describe in detail. This heuristic tries to place initial centroids in repository areas which have the highest capacity to deliver useful clusters, i.e., clusters which produce mappings. In Bellflower we implement this idea as follows. Let ME_n denote a set of all mapping elements for personal schema node $n \in N_s$, and ME_{min} the smallest one. For example, in Fig. 3 $\textcircled{3}$, ME_{min} is the set of black nodes. There are only four black nodes (mapping elements) compared to five mapping elements in other two sets. Since each cluster needs at least one mapping element for each personal schema node, Bellflower initializes centroids by declaring all the elements of ME_{min} as centroids.

An inherent problems of the k-means algorithm is its sensitivity to the initial choice of centroids. With our approach which combines large number of initial centroids and reclustering, this problem is reduced.

Distance measure: In Bellflower, the distance measure $distance(n', m')$ (line 5 in Alg. 1) is the actual tree distance (i.e., path length) between the centroid node n' and the mapping element m'. In principle, the distance measure must be designed to support a specific objective function. Thus, each schema matching system must have an accustomed distance measure. Path length is important in Bellflower's objective function, therefore the distance measure uses this value. We are investigating ways to extend the distance measure with other heuristics, both localized and structural.

In k-means clustering, distances are computed very often, and the efficiency of the distance computation is important. Bellflower uses node labeling techniques [12] to provide low-cost computation of path lengths.

Computation of centroid: The centroid is an entity used to represent a cluster. In Bellflower, the centroid for a cluster is selected from the mapping elements which belong to the cluster (such centroids are also known as medoids [8]). More specifically, the mapping element which is the *center of weight* for the cluster is used as a centroid. We are investigating other kinds of centroids, such as using two elements to represent a single cluster.

Reclustering: Reclustering is used to directly act upon the clusters in each clustering iteration. In Bellflower reclustering dynamically changes the number of clusters by joining clusters or by removing clusters. Reclustering makes it possible to bring the number and the size of clusters to a desired state.

We illustrate the effects of reclustering by observing the sizes and the number of clusters which are formed by using three different reclustering strategies: *no reclustering*, *join reclustering*, and *join & remove reclustering*. Fig. 4 shows the size distribution of the resulting clusters.

The *no reclustering* algorithm created a total of 579 clusters. Dark bars show the size distribution of these clusters. For example, out of these 579 clusters, 134 clusters have a size in the [8,15] range. The majority of clusters is very small. We have observed that tiny clusters appear in areas in

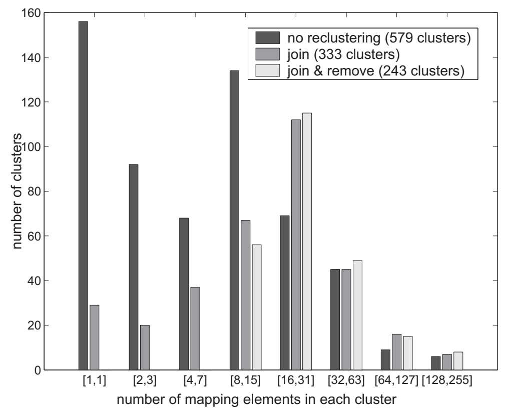

Figure 4: Cluster size distribution for different reclustering techniques

which initial centroids are close to each other, and thus compete to attract the same mapping elements. Consequently, some centroids "starve" and form tiny clusters. This problem can be solved through *join* reclustering. The join technique unites clusters if the centroids of these clusters are near each other. Fig. 4 shows the effects of using *join reclustering*. Clustering with join reclustering creates 333 clusters, which is 256 clusters less than in the no reclustering case. These 256 clusters have joined neighboring clusters. With join reclustering the tiny cluster problem is significantly reduced.

The remaining tiny clusters can be either ignored or removed with the *remove* reclustering technique. The technique removes all clusters with less then a certain number of mapping elements. The mapping elements belonging to these clusters are free to join other clusters in the neighborhood. Fig. 4 shows that the combination of the *join and remove* reclustering completely eliminates the tiny clusters.

In some cases clustering can create *huge* clusters. This occurs in areas with a large number of mapping elements, in which only one centroid is initiated. This centroid attracts all the mapping elements, creating a cluster with possibly several hundred nodes. The problem of huge clusters can be solved in several ways: by improving the initialization algorithm, by splitting huge clusters, or by discarding a large number of the huge cluster's elements, namely the ones that are far from the centroid. In our experiments, huge clusters rarely occur, and are removed "manually" if necessary. For now, Bellflower does not implement procedures for handling huge clusters.

Convergence criteria: If in two successive iterations of the k-means algorithm clusters do not change, clustering has reached the total stability criterion, and the algorithm terminates. In practice, however, the total stability criterion can be relaxed. Bellflower monitors, in each iteration, the number of mapping elements which switched form one cluster to another, and the change in the number of clusters. When these numbers drop below a certain threshold, e.g., 5 percent of the total number of mapping elements/clusters, algorithm terminates. The selection of termination criteria is not trivial. So far, we found no direct correspondence between the termination criteria and the resulting performance of the clustered schema matching system. Finding a convergence criterion which minimizes iteration and yet delivers the desired clusters is largely open research question.

5. Experiments

We have conducted numerous experiments over different schema matching problems, i.e., different repositories and different personal schemas. Although results differ for different matching problems, the clustered schema matching has shown the same general behavior and gave rise to consistent conclusions. Experiments were conducted on a standard PC. This is sufficient because we study efficiency improvements, not absolute times. In this section, we report the results of a typical experiment: the personal schema has nodes "name", "address", and "email", and a structure similar to schema s in Fig. 1. The personal schema is matched against the repository with 9759 elements, distributed over 262 trees. Bellflower is asked to discover all the schema mappings, $s \mapsto t$ for which $\Delta(s,t) \geq 0.75$. In this experiment, Bellflower's element matcher produces 4520 mapping elements.

In the experiment, we use three different variants of the clustering algorithms, plus the case with no clustering. The three clustering variants differ only in the value of the distance threshold which is used in *join reclustering*. For the distance threshold 4, join reclustering joins clusters whose centroids are at the distance not bigger than 4. Due to joining, resulting clusters are larger and hence the name for this clustering variant: "large clusters" variant. The other two clustering variants use distance thresholds 3 and 2 and create smaller clusters, hence the names "medium clusters" and "small clusters". In the non-clustered case, each tree in the repository is treated as one cluster. These are called "tree clusters".

Properties of clusters: Tab. 1a shows the properties of resulting clusters. The table reports only the properties of the *useful clusters*, i.e., clusters which can deliver schema mappings.

The "tree clusters" row in the table is interpreted as follows. For the given matching problem, there exist 95 useful trees in the repository. Trees contain on average 45.5 mapping elements. The maximum number of mappings which can be generated in all these trees, i.e., the search space for a schema matcher, is 11.9 million schema mappings. We shall see later, that amongst all these mappings, there exist only 4271 schema mapping for which $\Delta(s,t) \geq 0.75$. The other rows illustrate how clustering reduces the search

clustering algorithm	a) properties of clusters			b) mapping generator performance		
	# of useful clusters	avg. # of mapping elements	total # of schema mappings	# of partial mappings	# of schema mappings $\Delta \geq 0.75$	time (s)
small	251	16.1	153311 (1.28%)	51491	620	16.0
medium	235	17.4	168877 (1.41%)	56965	1009	23.8
large	172	24.1	486383 (4.07%)	109341	2444	56.3
tree	95	45.5	11962741 (100%)	386817	4271	106.3

Table 1: Experimental results: **a)** properties of clusters, **b)** mapping generator performance

space. For example, the "medium clusters" row, shows that 235 clusters are formed with an average of 17.4 mapping elements. The size of the search space is reduced 70 times, from 11.9 million, in the case of "tree clusters" to 168877 (1.41 percent) in the "medium clusters" case. Note that different clustering variants deliver different reductions of the search space which means that parameters of the clustering algorithm can be used to control the size of the search space.

Efficiency of clustering: Each of the three clustering variations needed almost equal times to complete the clustering: 12.2 sec., 12.0 sec., and 11.9 sec. Time differences occur only due to the different number of resulting clusters; the other parameters which influence the complexity of the k-means algorithm, i.e., the number of trees (95), the number of mapping elements (4250), and the number of iterations (4), were the same in all three cases. In other experiments, we have observed that large time savings can be acquired by fine tuning the convergence criterion. Each unnecessary iteration is a waste of time. How to do this automatically is a open question.

Efficiency of the schema mapping generator: Bellflower, is a proof of concept experimental system and the time measurements are not accurate. To assess the performance of the schema mapping generator we measure not only time by also use more reliable performance indicators - counters. The Branch and Bound algorithm, which drives the generator, saves time by not generating all possible schema mappings. Instead it generates and tests a much smaller number of partial schema mappings. In Bellflower we count the number of partial schema mapping generated by the B&B algorithm. The performance of a well-tuned implementation is expected to be proportional to this number. Tab. 1b shows the measured efficiency indicators.

The "tree clusters" row, i.e., the non-clustering case, is interpreted as follows. To find the solutions for a given matching problem, in the search space of 11962741 mappings (see "tree clusters" in Tab. 1a), the B&B algorithm generated 386817 partial mappings and discovered 4271 schema mappings having the similarity index larger than 0.75. It took 106.3 seconds to finish the whole process. We see here the benefits of using the B&B algorithm. Instead of generating and testing all 11962741 mappings, B&B algorithm tested 30 times less partial mappings.

Other rows show the efficiency improvements introduced with the use of clustering. For example, with the "medium clusters" clustering, the search space contains 168877 mappings (see "medium clusters" in Tab. 1a), and the B&B algorithms tests 56965 partial mappings, which is one third of the total search space. This is less improvement than in the non-clustered case in which B&B brought much more benefits; by using the "medium clusters" clustering, the search space is already condensed by clustering, and B&B cannot bring that much improvement. We can also compare the efficiency of the mapping generator when no clustering was used, i.e., for"tree clusters", and when the "medium clusters" clustering was used. In the clustered case the number of partial mappings generated by B&B is reduced by factor 6.8 which is approximately confirmed by the time measurements. The effects that clustering has on the number of discovered mappings are also visible. "Medium clusters" approach did not discover all the 4271 mappings. Instead, it produced 1009 schema mappings. This issue will be discussed later in more detail.

If we add together the *clustering time* and the *schema mapping generation time*, the non-clustered approach consumes $0sec + 106.3sec$, while the "medium" clustering consumes $12.0sec + 23.8sec = 35.8sec$. In this particular case, an efficiency improvement of factor 3 is achieved. A creation of an elaborate cost model for the whole clustered schema matching technique is future research.

On how clustering affects the results of schema matching: As discussed, clustering improves efficiency, but with a penalty of losing mappings. Tab. 1b "tree clusters" row, shows that the non-clustered matching delivers 4271 schema mappings, while the three clustered schema matching techniques deliver 620, 1009, and 2444 schema mappings. Obviously, clustering cuts-off many mappings that are otherwise generated in the non-clustered case. We have, however, already argued that this loss can be accepted if it occurs among the solutions which rank low. To see where the loss of mappings actually occurs Fig. 5 shows the *percentage of preserved mappings* for different threshold levels. The non-clustered schema matching exhaustively traverses the search space, and finds all the schema mappings with $\Delta(s, t) \geq 0.75$. Hence the constant 100 per-

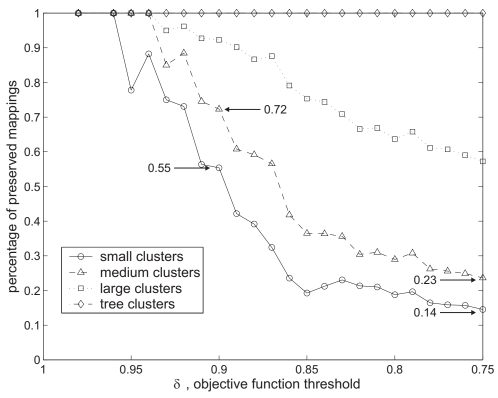

Figure 5: Percentage of preserved mappings for different variants of the clustering algorithm

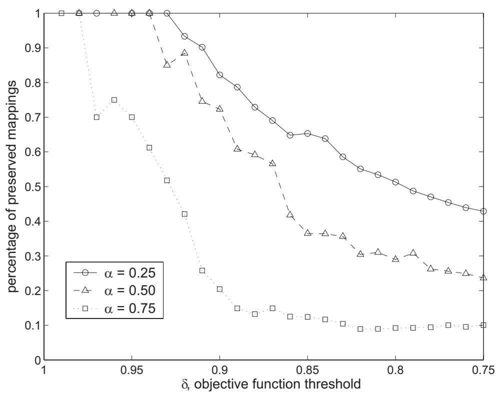

Figure 6: The effectiveness of clustered schema matching for three variations of the objective functions

cent line for non-clustered matching, i.e., the "tree clusters" line. Clustered schema matching loses mappings. For example, the "small clusters" clustering preserves 55 percents of schema mappings which have the similarity index gereater of equal 0.9 (see arrows in Fig. 5), and 14 percent of mappings for $\delta = 0.75$.

The figure demonstrates that clustering provides the desired behavior: in the areas with high value of objective function, large proportion of mappings are preserved. The loss of mappings is greater among the lower ranked mappings.

Fig. 5 also demonstrates that different clustering variants exhibit different percentages of preservation. The larger reductions of the search space result in the larger losses of mappings. Still, the numbers are promising. For example, the "medium clusters" clustering reduces the search space to only 1.41 percent of its original size (see Tab. 1a), and yet it preserves 23 percent (i.e., 1009 / 4271) of schema mappings for $\delta = 0.75$, or even 72 percent for $\delta = 0.9$.

On the correlation of clustering and the objective function: Clustering in clustered schema matching cannot be designed in a generic way. It has to be designed with a specific objective function in mind. To show how a single clustering approach performs with different objective functions we use the following experiment. The given schema matching problem is solved with three different objective functions. These functions differ only in the value of the α parameter (see Eq. 3) which varies over 0.25, 0.50, and 0.75. Large α favors name similarity heuristics Δ_{sim}. Small α favors the path length heuristics Δ_{path}. The experiment uses the "medium clusters" clustering variant for all three objective functions.

Fig. 6 shows the preservation percentage for the three objective functions. Large differences can be observed. As show in Sec. 4, the distance measure used in clustering is based on path length only. It is designed to preserve mappings in systems which treat the path length feature as an important heuristics in matching. Consequently, clustering delivers the best results if used with the $\alpha = 0.25$, i.e., with the objective function which favors the path length hint. As the α increases, the number of preserved mappings decreases. This demonstrates the importance of adapting the clustering algorithm to a specific objective function.

6. Related work

Schema matching attracts significant attention as it finds application in many areas dealing with highly heterogeneous data. A comprehensive survey by Rahm and Bernstein [18] identifies main schema matching techniques. More specific schema matching techniques and system include COMA [5], Cupid [16], LSD [6], and Corpus-based schema matching [15], to name a few. These systems focus only on the effectiveness of schema matching. Various hints and the techniques to combine these hints are investigated in order to improve the schema matching effectiveness. On the other side, there exist very little work which addresses the efficiency of schema matching.

Bernstain et. al, describe a design of an industrial-strength schema matcher called PROTOPLASM [3]. While analyzing the techniques for building a customizable schema matching system, authors underline the importance of developing the efficient schema matchers – the same motive drives our research.

Recently, Rahm et. al [9, 2], suggested and demonstrated the use of fragmentation to make schema matching more efficient. While our clustered matching forms clusters on-line and by taking into account the personal schema, they propose that fragments can be formed off-line by exploiting syntactic substructures, such as complex types or groups, of individual XML schemas. They further perform schema matching in two steps. First, fragments as a whole are compared to other fragments to find the most similar ones. Likewise, we plan, as a future research, to devise a metrics which identifies clusters with high potential for delivering good mappings. In the second step, in both our an their approach, detailed matching is performed per cluster, that is, per fragment.

Clustering has already been used to support schema matching in two ways. First, clustering is used to detect corresponding schema elements [17, 21]. It has been shown that different clustering techniques can be used to form the groups of semantically similar elements [22]. Second, clustering is used to identify groups of semantically related schemas and to confine the matching efforts within these schema groups [14]. The way we use clustering in clustered schema matching has common points with both approaches. An overview of other Web data clustering practices is given in [20].

7. Conclusion and future research

We have proposed the *clustered schema matching* as a technique for efficient matching of one smaller schema against the large schema repository. Clustering is used to quickly identify regions in the schema repository which are likely to comprise good mappings for the smaller schema. The schema matcher then looks for mappings only within these regions, i.e., clusters. This reduces the matching workload and improves the efficiency. The improved efficiency, however, comes at the cost of the loss of some mappings. The loss mostly occurs among the mappings which rank low which is an acceptable trade off. For validation we have built an experimental clustered schema matching system called Bellflower. Experimentation confirmed two main abilities of clustered schema matching: (1) the ability to improve efficiency of schema matching, and (2) the ability to preserve highly ranked mappings, in doing so.

Future research includes: (1) establishing a tighter control over clustering – more insight into the effects of certain clustering parameters on the efficiency/effectiveness trade off allows for better tuning, (2) ordering the clusters – a measure of cluster's quality can be used to decide which clusters have better chances to produce good mappings. In this way, the time-to-first good mapping can be improved, (3) design of a other distance measures for clustering.

Other challenging issues include, the generation of partial mappings, matching with larger personal schemas, and clustered schema matching in graph based repositories.

References

[1] FuzzySearch library. http://www.textolution.com.

[2] D. Aumueller, H. H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with COMA++. In *SIGMOD Conference*, pages 906–908, 2005.

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

[4] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: discovering complex semantic matches between database schemas. In *Proc. SIGMOD'04*. ACM Press, 2004.

[5] H. H. Do and E. Rahm. COMA — A system for flexible combination of schema matching approaches. In P. A. Bernstein et al., editors, *Proc. Intl. Conf. VLDB 2002*. Morgan Kaufmann Publishers.

[6] A. Doan, P. Domingos, and A. Halevy. Learning to Match the Schemas of Data Sources: A Multistrategy Approach. *Machine Learning*, 50(3):279–301, 2003.

[7] A. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AAAI AI Magazine, Special Issue on Semantic Integration*, 26(1):83–94, 2005.

[8] M. Dunham. *Data Mining: Introductory and Advanced Topics*. Prentice Hall, 2003.

[9] Erhard Rahm and Hong-Hai Do and Sabine Mamann. Matching Large XML Schemas. *SIGMOD Rec.*, 33(4):26–31, 2004.

[10] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB 2001*, 2001.

[11] H. Kaplan and T. Milo. Short and simple labels for small distances and other functions. *Lecture Notes in Computer Science*, 2125, 2000.

[12] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proc. SODA'02*, pages 954–963. Society for Industrial and Applied Mathematics, 2002.

[13] D. L. Kreher and D. R. Stinson. *Combinatorial algorithms : generation, enumeration, and search*. CRC Press LLC, Boca Raton, Florida, 1999.

[14] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: clustering XML schemas for effective integration. In *Proc. CIKM'02*, pages 292–299. ACM Press, 2002.

[15] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. Corpus-based schema matching. In *ICDE*, pages 57–68. IEEE Computer Society, 2005.

[16] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *Proc. of VLDB'01*, pages 49–58, Orlando, Sept. 2001. Morgan Kaufmann.

[17] C. Pluempitiwiriyawej and J. Hammer. Element matching across data-oriented XML sources using a multi-strategy clustering model. *Data Knowl. Eng.*, 48(3):297–333, 2004.

[18] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, Dec. 2001.

[19] M. Smiljanic, M. van Keulen, and W. Jonker. Formalizing the XML Schema Matching Problem as a Constraint Optimization Problem. In *Proc. DEXA'05*, Aug. 2005.

[20] A. Vakali, J. Pokorn, and T. Dalamagas. An overview of web data clustering practices. In A. Vakali and et al, editors, *Proc. EDBT'04 Workshops*, 2004.

[21] W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proc. SIGMOD'04*, 2004.

[22] H. Zhao and S. Ram. Clustering schema elements for semantic integration of heterogeneous data sources. *Journal of Database Management*, 15(4):88–106, 2004.