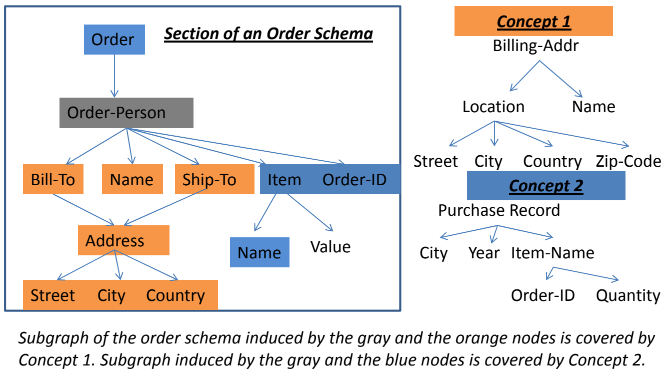
Schema Covering: a Step Towards Enabling Reuse in Information Integration

Barna Saha [+1] Ioana Stanoi [*2] Kenneth L. Clarkson [*3]

[+]Computer Science Department, University of Maryland College Park
AV Williams Building, College Park, US 20742
[1]barna@cs.umd.edu

[*]IBM Almaden Research Center
San Jose, CA
[2]irs@us.ibm.com [3]klclarks@us.ibm.com

***Abstract*—We introduce *schema covering*, the problem of identifying easily understandable common objects for describing large and complex schemas. Defining transformations between schemas is a key objective in information integration. However, this process often becomes cumbersome when the schemas are large and structurally complex. If such complex schemas can be broken into smaller and simpler objects, then simple transformations defined over these smaller objects can be reused to define suitable transformations among the complex schemas. Schema covering performs this vital task by identifying a collection of common concepts from a repository and creating a *cover* of the complex schema by these concepts. In this paper, we formulate the problem of schema covering, show that it is NP-Complete, and give efficient approximation algorithms for it. A performance evaluation with real business schemas confirms the effectiveness of our approach.**



Subgraph of the order schema induced by the gray and the orange nodes is covered by Concept 1. Subgraph induced by the gray and the blue nodes is covered by Concept 2.

Fig. 1. Schema Covering with Concepts

I. Introduction

Data exchange and *integration* are two central problems of data management. A major step in *data exchange* and *integration* is defining transformations among schemas. The schemas might come from different applications, be written in different formats, or even have different data models, such as relational or XML. The ability to do data exchange and integration is fundamental to many tasks: Extract-Transform-Load (ETL) workflows, that populate data warehouses from sets of data sources; web applications like mash-ups, that bring data from one or more sources into a single integrated tool; XML messaging; schema evolution; and database restructuring. However, with the advent of flexible XML formats, large and structurally complex schemas are being written frequently, making traditional methods for data transformation a formidable task.

The two main drawbacks of traditional tools for designing transformations between schemas are the *level of abstraction* and *monolithic mapping*. Traditionally, while designing mappings, users think in terms of the actual source and the target schema. But since schemas are low-level structures that may be large and do not usually have a one-to-one correspondence with real-life objects, they are difficult to visualize and understand. Thus working at such a lower level of abstraction makes the process both difficult and error-prone. Ideally, users should be able to express the mappings in terms of the objects at higher levels of abstraction, such as business objects, that are easier to understand and do not couple implementation with semantics.

Designing a single monolithic mapping between a source and a target schema lacks modularity and reusability. Complex mappings are difficult to build and debug in one step [13][14]. Mapping designers only have access to the final result of the complex transformation, and can miss some potentially useful results of intermediate steps in the computation. Moreover, when the source or the target schema evolves, the designers will typically have to modify a considerable part of the mapping, although most of the represented concepts remain unchanged. Schema covering can be used to eliminate both of these shortcomings. Given a source and a target schema, schema covering first identifies a collection of commonly used *concepts* (business objects) from a repository, that are represented within the two schemas. Next, for each of the source and the target schemas, it computes an appropriate layout of these concepts, that is, how these concepts can be placed together to suitably cover the two respective schemas. Schema covering thus provides a higher level of abstraction, by describing a schema in terms of the common objects it represents. Simple transformations defined individually over these objects can be reused to come up with complex transformations. When schemas evolve, only the objects covering the changed parts need to be modified. Schema covering also aids in visualization: a schema can be visualized at the object level, and then each object can be elaborated as needed to see the precise details.

An example of a schema covering is given in Figure 1. The schema on the left is a section of a large *order* schema. There are two schemas Billing-Addr and Purchase Record on the right corresponding to two business objects. Figure 1 also shows the two parts of the Order schema that are covered by Billing-Addr and Purchase Record respectively. The node with label City is present in both the business objects. However for the coverage to be semantically correct, only Billing-Addr must cover it. If both the business objects are allowed to cover the node City, then an ambiguity will occur in deciding whether it corresponds to the city where a customer stays, or the city from where an item is purchased. There may be multiple business objects competing to cover the same or overlapping parts of the source/target schema. Only the best alternatives can be chosen in such a scenario.

The framework for applying schema covering in an end-to-end transformation involves three major steps, the second one being schema covering itself, the topic of this paper.

a) Building the Concept Repository: Employees, products, articles, and so on, that represent single business objects are examples of concepts. They are higher-level abstractions than schemas, and as such, are closer to the understanding of the mapping designer. These concepts can be either defined manually, or automatically extracted from a source that provides standardized schemas, such as Freebase or OAGI. Once created, they are stored in the metadata repository of the system, ready to be used in mappings. The metadata repository can also hold mappings among the concepts themselves.

b) Schema Covering: Given the repository, schema covering finds the relevant concepts and the mappings from the source and the target schema separately to these concepts.

c) Flow of Mappings: Using the existing mappings among the concepts in the repository, a mapping path from the concepts that cover the source to the concepts that cover the target can be computed. From this mapping path, an end to end source to target mapping can be composed.

These three steps together complete the picture of a simple, reusable, and modular approach to defining transformations. They will be incorporated in the new extension of Clio, a schema mapping tool designed at IBM Almaden [13][14]. In this paper our focus is only on *schema covering*. To learn more details about the other two steps and the current progress on them in the Clio context, the reader is encouraged to see [1].

Our specific contributions are as follows:

- We formulate the problem of schema covering, fundamental for enabling reuse.
- We show the schema covering problem is NP complete.
- We develop efficient algorithms for schema covering, and analyze and establish their theoretical performance guarantees. We give experimental evidence that algorithms should work very well in practice.

To the best of our knowledge, schema covering has not been studied before. We start with an overview of the different steps involved in schema covering, which will give a road map for the rest of the paper.

II. Preliminaries and an Overview of Schema Covering

We consider schemas as rooted directed graphs $G = (V, E)$, where V is a set of nodes and E is a set of directed edges (see Figure 1). For XML schemas, nodes correspond to complex elements or attributes. For relational schemas, nodes represent relational tables and attribute columns. Edges represent structural relationships (such as parent-child links) and value links (such as foreign key constraints). Each vertex v has a label, denoted $\mathrm{label}(v)$, with the name of the element, table or attribute it represents. While relational schemas have a simple tree-structure, with no undirected cycle, XML schemas might exhibit complex graphical structure due to edge sharing. If a schema graph does not contain any directed cycle, it is called a *Directed Acyclic Graph* (DAG) or *nonrecursive* schema. Otherwise the schema graph is *recursive*. We assume that in all schema graphs, there is a unique root node r_G with indegree 0, from which every vertex in the graph is reachable.

There are several challenges that need to be addressed for schema covering. We discuss them in detail now.

1) Filtering: First, the repository might contain a large number of concepts, many of which are not relevant to a given schema. Trying to match each of them individually is prohibitively time consuming. Therefore as a very first step, we need to design efficient filtering methods that choose a subcollection of the concepts for further processing. The chosen concepts should be the potential candidates for final coverage. Ideally, the running time of filtering should only depend on the size of the chosen subcollection, independent of the repository size.

2) Alignment Score Computation: Next, for each concept chosen by the filtering step, we need to identify all the subgraphs of the schema that can be covered by the concept. The number of different possible subgraphs of a schema is exponential in the number of nodes of the corresponding schema graph. So we cannot use *schema matching* as a black box and try to match each of the possible subgraphs with a concept. The schemas can have arbitrary structural complexity. Most schema matching algorithms do not handle recursive schemas [3]. COMA [4] and CUPID [10] can handle cycles to some extent, but in the worst case COMA might require exponential time, since it explores every possible path to a node from the root. CUPID uses a bottom-up approach, building the match result from the leaves. In the context of schema covering, a concept might match a schema at any place and even can match multiple number of times. It may not always start matching from the leaves, and thus the technique of CUPID cannot be applied here. Similar difficulties apply to using COMA, since COMA considers paths from the root, whereas in schema covering any sub-path of the schema can be matched. We need a fast algorithm that, given a concept, identifies a small number of subgraphs matchable to the concept. It should also compute a score, which we call *alignment score*, for each computed subgraph, showing how well the concept can be aligned to that subgraph. We require the subgraphs computed be connected, to ensure that a concept covers a coherent region in the schema. More details about the requirements can be found in Section IV.

3) Schema Covering: The final step is the schema covering. Given a collection of concepts, the subgraphs they cover and the respective alignment scores, schema covering computes the *cover*, which comprises a chosen set of concepts and the subgraphs they match. The subgraphs covered by the concepts might overlap. To control ambiguity, nodes should not be covered more than a given number of times. The optimum cover *maximizes the total alignment score* of the selected concept-subgraph pairs, maintaining the overlap constraint.

In the rest of the paper, we design and analyze algorithms for each of these three steps. For clarity we start the presentation with the schema covering problem (Section III), where we assume that all the alignment scores are known. Next, in Section IV, we describe the alignment score computation step, followed by the filtering step in Section V. We conclude with performance analysis on several real world schemas using a repository of objects created with relatively smaller schemas from different business domains.

III. Schema Covering

In this section, we begin with a description of a specific constraint, called the *ambiguity constraint*, that is essential for schema covering. We then formally define the schema covering problem and establish its NP-Hardness. We next develop a greedy approximation algorithm that we prove will always give results close to optimum. We show that for a special case, where the schema is a tree and with certain ambiguity constraints, there is an exact algorithm using dynamic programming.

A. Ambiguity Constraint

Consider the schema of Figure 1. There may be different concepts similar to the subtree rooted at Address. The objective of schema covering is to choose concepts such that the total alignment score is maximized. Without additional constraints, suppose all the different concepts related to the subtree rooted at Address, are allowed to cover it. Then each of these concepts contribute a positive alignment score to the objective function. Thus the total alignment score grows while the quality of the coverage does not improve. Ideally there should not be any ambiguity in deciding which concepts are used to cover a particular portion of the input schema. Note that, if ambiguity is not controlled, in the worst case, coverage might retrieve the entire set of concepts.

To control overlaps, we introduce the *ambiguity constraint*. A solution of coverage has a *node ambiguity constraint* of λ, if each node of the schema can be covered at most λ times by the chosen concepts. In some cases, having $\lambda = 1$ (strict ambiguity constraint) may be too restrictive. Consider again the same example. There may be a Billing-Address concept that covers Address. Thus even though there is a Shipping-Address concept, which contains exactly the same subschema for Address, that will be dropped from consideration. If we have allowed Address to be covered twice or the edge (Ship-To, Address) once, this effect might have been removed. A more expressive constraint is then the *edge ambiguity*, where instead of having an ambiguity constraint on nodes, we require an edge to be covered at most λ times.

The subtree rooted at Address is commonly seen in many schemas and likely to have many coverage options. Therefore for portions in schema like Address, it is useful to have a stricter ambiguity constraint (with λ smaller). However it may be useful to relax the ambiguity constraint for the infrequent nodes and edges and explore the different matching possibilities. Our proposed greedy algorithm can handle variable ambiguity.

B. Schema Covering Problem

Now we are ready to provide formal definitions for schema covering.

Definition 1: We are given a schema graph $G = (V, E)$, an ambiguity requirement λ_v for all $v \in V$ (for edge ambiguity, each edge $e \in E$ will have edge-ambiguity constraint λ_e), a repository $\mathcal{C}$ of s concepts, a collection $\mathcal{H}$ of t subgraphs, and an $s \times t$ array $\mathcal{A}$ of alignment scores, with $\mathcal{A}(H, C)$ computed for each $H \in \mathcal{H}$ and $C \in \mathcal{C}$. A schema covering is a subset $\mathcal{S} = \{(H_i, C_i)\}_{i=1\ldots r}$, such that the node (edge, for edge-ambiguity constraint) $v \in V$ appears at most λ_v times in $\{H_1, H_2, \ldots, H_r\}$ and the total sum of the alignment scores, that is, $\sum_{1\leq \ell \leq r} \mathcal{A}(H_\ell, C_\ell)$ is maximized.

The next theorem proves that schema covering is NP-complete, using an easy reduction from a variation of hypergraph matching.

Theorem 3.1: Schema covering is NP-Complete, if all $H \in \mathcal{H}$ have at least three nodes.

Proof: First of all, given a collection of concepts and schema subgraph pairs along with the respective alignment scores, it is easy to check whether they form a valid instance of schema covering and whether the total alignment score sums up to the specified value. So the problem clearly is in NP.

Now to show it is NP-hard, consider the following version of hypergraph matching problem, *hypergraph variable b matching*. We have a set of nodes V and a collection of hyperedges defined on the subsets of those nodes. The hyperedges have size at least three. Each hyperedge has an associated weight and each node v has a value $b_v \in \mathbb{N}$. The goal is to obtain a collection of hyperedges such that the total weight of the chosen edges is maximized and each node v is covered at most b_v times. The problem is NP-complete, since a special case of this problem is hypergraph b-matching, [8], which is known to be NP-complete.

Given an instance of hypergraph variable b matching, create a schema graph $G = (V, E)$ on the same set of nodes. For every hyperedge, add all the edges among the vertices of the hyperedge. Create a repository with only one concept C. The collection of subgraphs of schemas corresponds to all the hyperedges and hence each of them are connected. An alignment score between C and a subgraph (a hyperedge) is simply its weight. Set $b_v = \lambda_v$. Now it is straight-forward to see that there is a solution of schema covering of weight x if and only if there is a solution for this hypergraph variable b-matching problem with weight x. ■

Known hypergraph b-matching algorithms [8][17] can be used to design an algorithm for schema covering. However the approximation guarantees of all these algorithms are poor. For example, when $b = 1$, the best known algorithm only guarantees that the solution found is within a factor of $\sqrt{|V|}$ of the optimum.

Instead we propose a new greedy algorithm, called *C-Greedy*, which is fast and gets a constant factor approximation, which is much better than the $\sqrt{|V|}$ approximation guarantee of hypergraph matching. The greedy algorithm relaxes the ambiguity constraint by bounding the average ambiguity of the elements.

C. *C-Greedy Algorithm*

For every pair (H, C) with $H \in \mathcal{H}$ and $C \in \mathcal{C}$, we construct a set S, where S comprises either all the nodes of H, if a node ambiguity constraint is to be satisfied, or all the edges of H, if an edge-ambiguity constraint is to be satisfied. We define a weight $W(S)$ for S with $W(S) := \mathcal{A}(H, C)$, and a concept $C(S) := C$. Define also a "normalized weight" $\hat{W}(S) := W(S)/|S|$. The input to our *C-Greedy* algorithm consists of these sets with their weights and concepts. There may be multiple sets having different concepts but containing identical elements. Their weights may be same or different. For every element e in all these sets, an integer λ_e is specified, indicating the number of times that element can be covered. An element e is said to be saturated if in the existing solution, it is covered at least λ_e times. The output of the *C-Greedy* algorithm is a collection of sets $\mathcal{T}$, and the corresponding schema covering solution is $\{S, C(S)\}_{S\in\mathcal{T}}$.

C-Greedy consists of the following steps:

- Arrange the sets in non-increasing order of $\hat{W}(S)$.
- Consider the sets S_j in that ordering, and choose one if and only if at least half of its elements are not already saturated by the previously chosen sets.
- Return the collection $\mathcal{T}$ of chosen sets.

Now we analyze the approximation guarantee of *C-Greedy*. We say that an algorithm is an (x, y) *bi-criteria approximation algorithm* for the schema covering problem if the total alignment score is at least $1/x$ of the optimum total alignment score, and the average number of times each node (or edge) is covered is at most y. When $\lambda_e = 1$ for all e, our *C-Greedy* is a $(2, 2)$ bi-criteria approximation algorithm for the schema covering problem, that is, its total alignment score is at least one half of the optimum aggregated alignment score, and on average each element is covered at most twice. For arbitrary λ_e values, if $\lambda_{\max} = \max_e \lambda_e$, then *C-Greedy* guarantees a bi-criteria approximation of $(2\lambda_{\max}, 2\lambda_{\max})$.

1) *Bi-Criteria Approximation for C-Greedy:*

We prove the result when $\lambda_e = 1$ for all e. The proof for arbitrary values of λ_e is similar and omitted.

Lemma 3.2: The solution returned by *C-Greedy* when $\lambda_e = 1$ for all e has an average ambiguity less than two.

Proof: Each of the sets S in the returned collection $\mathcal{T}$ contributes at least $\lceil |S|/2 \rceil$ new elements, since otherwise the set S wouldn't have been chosen. Letting N denote the total number of elements covered by the solution $\mathcal{T}$, we have

$$N > \sum_{S\in\mathcal{T}} \lceil |S|/2 \rceil \geq \frac{1}{2} \sum_{S\in\mathcal{T}} |S|. \tag{1}$$

Now let f_e denote the frequency of the element e in $\mathcal{T}$, that is, f_e denotes how many times e is covered by the sets in $\mathcal{T}$. Let $\hat{f}$ denote the average frequency of the elements in $\mathcal{T}$. Then $\hat{f} = \sum_e f_e/N = \sum_{S\in\mathcal{T}} |S|/N < 2$, by (1). ■

The following lemma establishes the bound on x. If the optimum solution is denoted by OPT and $\mathcal{T}$ is the obtained solution, we try to charge or distribute the scores of the sets in OPT to the scores of the sets in $\mathcal{T}$. In our charging mechanism, if we can show that the scores of all the sets in OPT can be distributed by raising the scores of each set in $\mathcal{T}$ at most twice, this will imply that the total score of the sets in $\mathcal{T}$ is at least one half of the optimum score.

Lemma 3.3: The solution returned by *C-Greedy* has a total alignment score which is at least one half of the optimum total alignment score.

Proof: List the members o_i of OPT in non-increasing order of $\hat{W}(o_i)$. We charge OPT to $\mathcal{T}$ via the following charging scheme:

- If o_i is included in $\mathcal{T}$, charge $o_i \in \text{OPT}$ to itself.
- If o_i is not included, let $\mathcal{T}(o_i) \subset \mathcal{T}$ be the sets before o_i in the greedy order that are also included in $\mathcal{T}$. Charge o_i to $S \in \mathcal{T}(o_i)$ by $|S \cap o_i|\hat{W}(o_i)$.

First we bound the total charge received by $\mathcal{T}$, by bounding the charge received by each set $S \in \mathcal{T}$. For a set $o_i \in \text{OPT}$ and $S \in \mathcal{T}(o_i)$, we have $\hat{W}(o_i) \leq \hat{W}(S)$, since S comes before o_i in the greedy order. Hence the charge on $S \in \mathcal{T}$ is

$$\sum_{i\in\text{OPT}} |S \cap o_i|\hat{W}(o_i) \leq \sum_{i\in\text{OPT}} |S \cap o_i|\hat{W}(S) \leq |S|\hat{W}(S) = W(S),$$

where the last inequality follows from the fact that each element of S is covered at most once by the sets in OPT. Therefore the total charge received by $\mathcal{T}$ is at most $W(\mathcal{T})$.

We next calculate how much of the total score of OPT is charged by this scheme. The total charge from o_i is

$$\sum_{S\in\mathcal{T}(o_i)} |S \cap o_i|\hat{W}(o_i) = \frac{W(o_i)}{|o_i|} \sum_{S\in\mathcal{T}(o_i)} |S \cap o_i| > W(o_i)/2,$$

where the last inequality follows from the condition that at least half of the elements of o_i are covered by the $S \in \mathcal{T}(o_i)$. Hence the total charge transferred from OPT is greater than $W(\text{OPT})/2$. Since the total charge received by $\mathcal{T}$ is no more than $W(\mathcal{T})$, we have $W(\mathcal{T}) > W(\text{OPT})/2$. ■

These two lemmas imply the following theorem.

Theorem 3.4: *C-Greedy* is a $(2, 2)$ bi-criteria approximation algorithm for the schema covering problem.

Proof: From Lemma 3.2 the average ambiguity is at most 2, and from Lemma 3.3, the total score of the solution returned is at least one half of the total score of the optimum solution. ∎

This completes the analysis of *C-Greedy*.

D. Exact Coverage for Schema Tree with Strict Ambiguity Constraint

C-Greedy guarantees an approximate solution that is close to optimum. However, when the schema graph is a tree and we require strict node or edge ambiguity, the schema covering problem is no longer NP-hard. In this subsection, we develop an exact polynomial-time solution for this special case. We first consider the case of node ambiguity, where each node is allowed to be covered at most once, and then show how the result can be generalized for edge ambiguity, where an edge can be covered at most once. Under the edge ambiguity constraint, the number of times a node can be covered is equal to one more than its outdegree, for non-root nodes, and is equal to its outdegree, for the root node. In both of these algorithms, we follow a bottom-up dynamic programming approach, starting from the leaves. Each node's level is its height in the tree, where leaves have level 0.

1) Coverage with Strict Node Ambiguity: The alignment score requires that the subgraphs $H \in \mathcal{H}$ are each connected (Section IV). Therefore, if $H \subseteq G$ is covered by some $C \in \mathcal{C}$, then H is a subtree, and except for one node, which we call the root r of H, every node in $V(H)$ has indegree 1. Define $\mathrm{OUT}(H)$ as all the nodes outside H, reachable from $V(H)$ through a directed edge. The level of each node in $\mathrm{OUT}(H)$ is less than the level of $r \in H$. One possible solution for schema covering, for the subtree rooted at r, might consist of concepts to cover H and the subgraphs rooted at each node in $\mathrm{OUT}(H)$. Let $D(r)$ denote

$$\{(H, C) \mid H \in \mathcal{H}, C \in \mathcal{C}, H \text{ rooted at } r\}.$$

For any node v, let $\mathrm{Cover}(v)$ denote the best coverage solution for the subtree rooted at v, and let $\mathrm{Score}(v)$ denote the corresponding alignment score. We have the following theorem.

Theorem 3.5: The Score() function satisfies

$$\mathrm{Score}(r) = \max_{(H,C)\in D(r)} [\mathcal{A}(H, C) + \sum_{u\in \mathrm{OUT}(H)} \mathrm{Score}(u)].$$

If the pair H, C achieves the maximum in this expression, then $\mathrm{Cover}(r) = H \cup \bigcup_{u\in\mathrm{OUT}(H)} \mathrm{Cover}(u)$. If r_G denotes the root of the entire tree G, then $\mathrm{Cover}(r_G)$, along with the corresponding concepts, gives the optimum schema covering solution, and $\mathrm{Score}(r_G)$ is the optimum total alignment score.

Proof: Since each node can be covered only once, $\mathrm{Cover}(r)$ is a feasible solution for schema covering that covers the subtree rooted at r. If the optimum solution for the subtree rooted at r is not $\mathrm{Cover}(r)$, then we can improve the total alignment score and arrive at a contradiction, since we consider all the feasible solutions for the subtree rooted at r when computing $\mathrm{Score}(r)$. Hence $\mathrm{Cover}(r_G)$, with the corresponding concepts, is the optimum solution, and $\mathrm{Score}(r_G)$ is the total alignment score. ∎

The optimum solution now can be computed easily using dynamic programming. If we create a hash table to access the entries in each $D(r)$ in constant time, dynamic programming will run in time $\Theta(|V(G)|\mathcal{H}_{\max})$, where $\mathcal{H}_{\max}$ is the maximum size of any $H \in \mathcal{H}$.

2) Coverage with Strict Edge Ambiguity: When ambiguity constraints are placed on edges rather than vertices, the recurrence given by Theorem 3.5 does not hold. To obtain an exact solution by dynamic programming, we need a trick, which we call *set splitting*.

a) Set splitting: We know that each H is a subtree and has a root r of indegree 0. Let d denote the outdegree of r, and write the d children of r as $c_1, c_2, \ldots, c_d$. Create d sets from H, namely $H_1, H_2, \ldots, H_d$, such that each H_k contains exactly one child of r, namely c_k, for $k \in [d]$. (Here $[d]$ denotes the set of integers $\{1, 2, \ldots, d\}$.) We call the edge (r, c_k) the *root-edge* of H_k. If (H, C) is a feasible subgraph-concept pair, we also decompose C into $C_1, C_2, \ldots, C_d$, such that C_ℓ contains those nodes of C that are matched to H_ℓ, for $\ell \in [d]$ and, we update the alignment score for each (H_ℓ, C_ℓ) pair accordingly. This is called *set splitting*. Set splitting and score computation for each split set can be done in time linear in $|V(H)|$, and can be calculated during the computation of the alignment score. Once the set splitting is done, each H_k contains a root-edge.

b) Dynamic Programming: Call an edge (u, v) a *leaf edge* if v is a leaf vertex. We define $\mathrm{Cover}(x, y)$ as the optimum cover of a subtree with (x, y) as a root-edge, and $\mathrm{Score}(x, y)$ denotes the corresponding alignment score. Let $D(x, y)$ denote all the pairs (H_ℓ, C_ℓ) after set-splitting, such that H_ℓ contains the edge (x, y) as the root edge and the concept C_ℓ can cover it. The list of such subtrees can be accessed in $O(1)$ time using a hash-table. When (x, y) is a leaf edge, $\mathrm{Cover}(x, y)$ and $\mathrm{Score}(x, y)$ can be found by simply checking $D(x, y)$ and picking the one with maximum alignment value. If $D(x, y)$ is empty, $\mathrm{Cover}(x, y)$ is set to empty and $\mathrm{Score}(x, y)$ becomes 0. We traverse the tree bottom up. Assuming we have computed all the $\mathrm{Cover}(x, y)$ and $\mathrm{Score}(x, y)$ values, where x is at level at most $h - 1$ for some h, we next compute $\mathrm{Cover}(x, y)$ and $\mathrm{Score}(x, y)$ values for those x which are at level h, that is, higher up in the tree. We define $\mathrm{OUT}(H_\ell)$ as all the edges having their higher endpoint in H_ℓ and their lower endpoint outside of H_ℓ, that is, $\mathrm{OUT}(H_\ell) := \{(u, v) | u \in H_\ell, v \notin H_\ell, (u, v) \in E(G)\}$. We get the following theorem for the dynamic programming recurrence, proof of which is similar to Theorem 3.5

Theorem 3.6: The Score() function satisfies

$$\mathrm{Score}(r, c_\ell) = \max_{(H_\ell,C_\ell)\in D(r,c_l)} [\mathcal{A}(H_\ell, C_\ell) + \sum_{(u,v)\in\mathrm{OUT}(H_\ell)} \mathrm{Score}(u, v)].$$

If the pair (H_ℓ, C_ℓ) maximizes $\mathrm{Score}(r, c_\ell)$ then $\mathrm{Cover}(r, c_\ell) = (H_\ell, C_\ell) \bigcup_{(u,v)\in OUT_{H_\ell}} \mathrm{Cover}(u, v)$. If r_G denotes the root of the entire tree G and $c^r_1, c^r_2, \ldots, c^r_k$ are the k children of r_G, then $\bigcup_{s=1}^{k} \mathrm{Cover}(r_G, c^r_s)$, along with the corresponding concepts, gives the optimum schema covering solution, and $\sum_{s=1}^{k} \mathrm{Score}(r_G, c^r_s)$ is the optimum total alignment score.

IV. Alignment Score Computation

In this section, we describe how alignment scores for all the required subgraph-concept pairs can be computed efficiently. We first illustrate with an example, the requirements for alignment score. We then define it formally and give algorithms to compute it.

A. *Requirements and Definitions*

We use the examples of schema and concepts from Figure 1, to show the necessary requirements of alignment score. Ideally the nodes labeled Bill-To, Name (child of Order-Person), Address, Street, City, Country in the schema graph should match the nodes labeled Billing-Addr, Name, Location, Street, City, Country in Concept 1. To achieve this, first we need *approximate semantic matching* that allows Bill-To to match Billing-Addr, Address to match Location, and so on. We denote the semantic matching score between two nodes v_1, v_2 by $\text{sim}(v_1, v_2)$. The semantic matching score indicates how well the label of node v_1 matches v_2. Second, we should allow some *structural flexibility* in the matching. In Concept 1, Name appears as a child of Billing-Addr, but in the Order schema, Name is not a child of Bill-To but a sibling. Third, we should also ensure that all the nodes of a subgraph which is matched to a concept are *structurally close*. The node Name that is a child of Item is farther away from (Bill-To, Address, Street, City, Country) than the node Name that is a child of Order-Person. Hence alignment score computation should favor the node Name that is a child of Order-Person. Fourth, we need each subgraph that is matched to a concept to be *maximal*, that is there is no bigger subgraph containing the computed one and matching the given concept. Thus the subgraph {Address, Street, City, Country} of the input schema is not a maximal subgraph with respect to Concept 1. This property ensures that the number of subgraphs to which a concept can be matched is polynomially bounded. Finally, we want each subgraph to be *connected*, to make the relationship among the nodes in the subgraph clear. The number of extra nodes that cannot be matched with the concept but are added to the subgraph to ensure connectivity must be minimal. The subgraph {Order-Person, Bill-To, Name (child of Order-Person), Address, Street, City, Country} has all the above properties and can be matched with Concept 1.

Definition 2: (Path Length) Given a directed graph $G = (V, E)$ and $u, v \in V$, there is a path of length t from u to v if either v can be reached from u using t directed edges, or there is a common node a from which u and v can be reached through directed edges, and the total number of edges used is t. We will write $\text{len}(u, v) = t$.

Definition 3: (Alignment Score) We are given a schema graph $G = (V, E)$, an "flexibility" integer parameter $\text{F} \in [|V(G)| - 1]$, and a similarity threshold θ. An alignment score is nonzero between a subgraph $H \subseteq G$ and a concept C if the following conditions hold:

- (**PI**) (**Flexible Structural Matching**) There exists a set $U \subseteq V(H)$ such that all the vertices of U are connected in H by paths of length at most F, each node $u \in U$ is matched to some node $v \in C$ with $\text{sim}(u, v) \geq \theta$, and there are nodes $u' \in U, v' \in C$, with $\text{sim}(u', v') \geq \theta$, such that $\text{len}(u, u') \leq \text{F}$, and $\text{len}(v, v') \leq 1$. That is, a path of length at most F can be matched to an edge in the concept. We consider the node pairs (u, v) to be contributing to the alignment score, and denote such pairs by $(u, v) \to \mathcal{A}$.
- (**PII**) (**Maximal Subgraph**) There does not exist any $u \in V(G) \setminus U$ such that there exists $u' \in V(U)$ with $\text{len}(u, u') \leq \text{F}$ and there are $v, v' \in C$ with $\text{len}(v, v') \leq 1$, such that $\text{sim}(u, v), \text{sim}(u', v') \geq \theta$.
- (**PIII**) (**Minimally Connected**) H is connected, and removing any vertex in $V(H) \setminus V(U)$ disconnects H.

Under these conditions, the alignment score between H and C is

$$\mathcal{A}(H, C) = \sum_{\substack{u \in V(H), v \in V(C) \\ (u,v) \to \mathcal{A}}} \text{sim}(u, v)$$

Our goal is to efficiently compute, for every C, all the different subgraphs H of G for which an alignment score can be defined, and the corresponding score.

B. *Data Structures*

Now we describe the data-structures needed for alignment score computation. Some of these are built offline as a preprocessing step. The rest are built online, when the schema graph is presented for covering.

1) Offline Preprocessing. Inverted Index and Edge List: Offline preprocessing involves building an inverted index data structure for the repository and an edge-list for each concept. This is done once when the concept repository is built, and maintained incrementally with changes (additions and deletions of concepts) in the repository. The *edge-list* contains the directed edges in the concept graphs, indexed by a unique concept-ID. The *inverted index*, on the other hand is indexed by the node-labels of the concepts. For each node-label, it stores the concept-IDs in which this node label occurs, and its respective structural positions in the concept graphs. The structural positions of a node include all the different possible lengths, or depths, of the directed paths from the root to that node.

Example 1: Making up some additional concepts that are not shown, the partial inverted-index data structure for the node labels in Concept 1 of Figure 1 might look like the following:

$$\begin{aligned}
&\text{Billing-Addr} \to (1,0),(2,4),(10,1),(10,5)\\
&\text{Location} \to (1,1),(2,5),(3,8)\\
&\text{Name} \to (1,1),(1,2),(7,0),(9,1)\\
&\text{Street} \to (1,2),(2,6),(10,6)\\
&\text{City} \to (1,2),(2,6),(10,2)\\
&\text{Country} \to (1,2),(2,6),(10,2)\\
&\text{Zip-Code} \to (1,2),(2,6),(9,2)
\end{aligned}$$

The first line of the index indicates that node with label Billing-Addr appears in concepts with ID $1, 2, 10$ at depths $\{0\}, \{4\}, \{1, 5\}$ respectively.

a) Tree and DAG: If the concept is a tree, there is a unique directed path from the root to any node, and thus each node has a single depth value, which can be computed in $O(|V|)$ time by tree traversal. However, when the concept is a DAG, there may be multiple directed paths of different lengths from the root to a particular node. We call a node u as parent of v, $u = \mathrm{parent}(v)$, if there is a directed edge from u to v in the graph. Since a DAG does not contain any directed cycle, if $u = \mathrm{parent}(v)$, $v \neq \mathrm{parent}(u)$. Using this, we can define the set of depths, $\mathrm{Depth}(u)$ of a node u as follows: the root r of the concept DAG has $\mathrm{Depth}(r) = \{0\}$, and for other u, $\mathrm{Depth}(u) = \bigcup_{v=\mathrm{parent}(u)} \mathrm{Depth}(v) + 1$, where for a set S, $S + 1$ denotes $\{w + 1 \mid w \in S\}$.

The depth values can be computed by performing a topological sort of the DAG and then using the above recursion on nodes in a topologically sorted order. Recall that a topological sorting of a directed graph $G = (V, E)$ is a linear ordering of all vertices in V such that if G contains an edge (u, v), then u appears before v in the ordering. Such an ordering exists if and only if the graph does not contain any directed cycle. Computing the depth values in a topologically sorted order ensures that all the depth values of all the parents of a node are computed before that node. The running time for computing depth values is $\Theta(|V|\delta+|E|)$, where δ is the maximum number of different depth values of a vertex.

b) Recursive Schema: However if the schema graph has directed cycles, we cannot apply this procedure. Instead, we maintain a separate depth list for each node in the concept graph. We scan the edge list of the concept and whenever we encounter a directed edge (u, v), update $\mathrm{Depth}(v)$ by adding the elements $\mathrm{Depth}(u) + 1$ that it doesn't contain. We repeat this procedure n times, and use the resulting $\mathrm{Depth}()$ sets. Since the distance can be at most n, all the depth values are correctly computed. The time complexity of the algorithm is $O(|V||E|\delta)$.

Once the depth values for each concept node are calculated, the inverted-index can be created easily as in Example 1.

2) Online Preprocessing. M^{F} *Computation:* To compute alignment scores, we build a collection of arrays holding tabulations of similarity scores; this collection is called M^{F}, since it depends on the structural flexibility parameter F,

Let r_G denote the unique root of G and $\mathrm{Depth}(G)$ denote the maximum depth value of any vertex $v \in V(G)$. Clearly $\mathrm{Depth}(G) \leq |V(G)| - 1$.

The structure M^{F} includes an array $M_C[][]$ for each concept $C \in \mathcal{C}$ in the repository that has at least one matching node in the input schema graph. (The concepts that have no matching node are thereafter ignored for this input graph.) The array $M_C[][]$ for C has $\mathrm{Depth}(C)$ rows and $\mathrm{Depth}(G)+\mathrm{F}$ columns; a cell $M_C[t][r]$ includes a numerical similarity entry, a node pair, and a subarray of F entries, as described next.

Let a node $v \in V(G)$ be at depth s in G. Using the inverted index INV, finding the nodes in all the concepts that match v and retrieving their depth information can be done easily as follows: for the node v, we consider its label, and any prefix, suffix and n-grams (for approximate match) and consult a thesaurus to obtain the related labels. For each of these labels, we retrieve all the concept nodes having those labels using INV, together with their depth values. If a node $u \in C$ at depth t is one such node, then we update $M_C[t][s] \mathrel{+}= \mathrm{sim}(u, v)$. (Initially all array entries are zero.) However, since we want to allow structural flexibility, we also update with $M_C[t][r] \mathrel{+}= \mathrm{sim}(u, v)$ for $r = s, s+1, \ldots, s+\mathrm{F}-1$. The cell $M_C[t][r]$ is modified, because we allow v to shift from level s to r, or $r - s$ levels. To keep track of the shifted level of v at which $\mathrm{sim}(u, v)$ contributes to the aggregated similarity in a cell, the $M_C[t][r]$ data structure also includes a *shift* subarray with F entries, updated by $M_C[t][r][r-s] \mathrel{+}= \mathrm{sim}(u, v)$, for $r = s, s+1, \ldots, s+\mathrm{F}-1$. The cell $M_C[t][r]$ also stores the pair (u, v). The same processing is carried over for each visited node in G. The M^{F} structure thus stores in a condensed way all the matching node-pairs between a source schema and a concept, and the different structural positions of these nodes where they can be matched.

Source Node Labels	Concept 1 Node Labels	Normalized Similarity Score
Bill-To	Billing-Addr	0.9
Name	Name	1.0
Address	Location	0.9
Street	Street	1.0
City	City	1.0
Country	Country	1.0

Source Node Labels	Concept 2 Node Labels	Normalized Similarity Score
Order	Purchase-Record	0.3
Order-Person	Purchase-Record	0.1
City	City	1.0
Name	Item-Name	0.5
Item	Item-Name	0.5
OrderID	Order-ID	1.0

Fig. 2. For each source schema node, table lists the nodes of concept 1 and 2 which are retrieved during matrix computation and the corresponding similarity value.

		0.9	0.9		
		1.0	2.9	1.9	
				3.0	3.0

0.3[x]	0.4	0.1			
		1.0	1.0	1.0[x]	1.0[x]
		1.0[x]	1.0		

Fig. 3. The two matrices for Concept 1 and 2 with F = 2

Claim 4.1: If for $u, u' \in V(G)$ and $v, v' \in C$, it holds that $\mathrm{sim}(u, v) > 0$, $\mathrm{sim}(u', v') > 0$, $\mathrm{len}(u, u') \leq \mathrm{F}$, and $\mathrm{len}(v, v') \leq 1$, then there are cells $M_C[i][j]$ and $M_C[i+1][j+1]$ of M_C that contain the pairs (u, v) and (u', v').

The proof is easy and is left to the reader. This is an essential property, which helps in alignment score computation. Not all of the features of the M^{F} data structure are required for the alignment score computation. The shift subarray, for example, is maintained to facilitate filtering, which we describe in the next section.

Example 2: Consider the source schema and Concept 1 of Figure 1. The depth of the concept nodes that can be matched are retrieved from the inverted index data-structure shown in Example 1. The created matrix for F = 2 is shown in Figure 3. Similarity values for various node pairs are shown in Figure fig:matrix1. Starting the count from 0, consider row=1 of the above matrix. Name appears at depth $2, 3$ in the source schema and at depth 1 at the concept. Therefore the similarity value $\mathrm{sim}(\mathsf{Name}, \mathsf{Name}) = 1$ is added to matrix cells $M_1[1][2]$ and $M_1[1][3]$, and once again to $M_1[1][3]$ and $M_1[1][4]$. At cell $M_1[1][2]$, the similarity value of (Name,Name) is added to $M_1[1][2][0]$, at $M_1[1][3]$ it is added to both $M_1[1][3][0]$ and $M_1[1][3][1]$, and at $M_1[1][4]$ it is added to $M_1[1][4][1]$. Similarly, the similarity value for (Address, Location) is added at $M_1[1][3]$ and $M_1[1][4]$, and the shift arrays are updated as well. The F value of 2 allows Name and Address, which are at different depths in the source schema, to match Name and Location respectively in Concept 1 at $M_1[1][3]$. The cell $M_1[1][3]$ will also store (Name,Name), (Address,Location) and a pointer to the corresponding nodes in the source and the concept.

C. Algorithm and Analysis

Now, for a given concept C, we describe the algorithm to compute all the maximal subgraphs that can have nonzero alignment score with C. Along with each step of the algorithm, the corresponding correctness analysis is described briefly. Details are omitted, due to space constraints.

Initialization: create an empty graph X on the same set of nodes as G. For each node $u \in V(G)$, initialize $m[u] = 0, \text{list}[u] = \emptyset$. Here $m[u]$ will contain the similarity score involving u, and $\text{list}[u]$ will contain the nodes of C matched to it.

Step 1: *Obtain pairs that contribute to alignment score.* Check each pair of entries $M_C[i][j]$ and $M_C[i+1][j+1]$ that are nonzero, to identify vertices u, u', v, v' such that $\text{sim}(u,v), \text{sim}(u',v') > \theta$, $\text{len}(u,u') \leq \text{F}$, and $\text{len}(v,v') \leq 1$. By Claim 4.1, we can identify all such nodes correctly in this step. Update $m[u], m[u'], \text{list}[u], \text{list}[u']$ accordingly, that is, $m[u] \mathrel{+}= \text{sim}(u,v)$, $\text{list}[u] = \text{list}[u] \bigcup \{v\}$, and similarly for u'. Add the edge (u,u') in $E(X)$. After this step, we have correctly identified all the vertices that contribute to the alignment score.

Step 2: *Find maximal subgraphs.* Find the maximal connected components $Y_1, \ldots, Y_\ell$ of X with more than a single vertex. Each Y_i corresponds to a maximal subgraph to which C can be matched. If some Y_i is not a maximal subgraph, then there exists a node u outside Y_i that has a path of length at most F to some node $u' \in V(Y_i)$, and u, u' matches with $v, v' \in C$, which are distance 1 apart. But then the edge (u,u') will be added to $E(X)$ and thus Y_i is not a maximal subgraph, giving a contradiction.

Step 3: *Obtain alignment score.* For each Y_i, for $i \in [\ell]$, the alignment score is $A = \sum_{u \in Y_i} m[u]$. This is correct, since all the nodes in Y_i contribute towards the alignment score and $m[u]$ stores for each contributing node its similarity score with the nodes in C. The set $\bigcup_{u \in Y_i} \text{list}[u]$ gives the nodes in C to which Y_i is matched.

Step 4: *Find the connected subgraphs.* The nodes in Y_i may not be connected in G. To satisfy the connectedness property, by adding a minimum number of extra vertices, we do the following: Consider each Y_i separately, obtain the subgraph induced by Y_i in G and find the connected components $R_1, \ldots, R_k$ of it. Shrink each component R_i, $i \in [k]$, in G and obtain a minimum spanning tree on these shrunk vertices. Since a tree spanning R_i's is obtained, removing any node from the tree will disconnect at least one R_i. The vertices inside each R_i are connected, and so the overall subgraph H obtained by expanding each R_i in the computed tree is connected. Thus we get $\mathcal{A}(H,C) = A$.

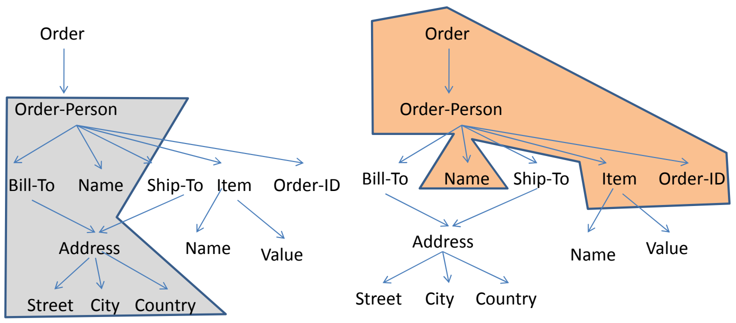


Fig. 4. Two subgraphs of the source schema for which an alignment score is defined for concept 1 and 2 respectively.

Example 3: Figure 4 shows two subgraphs of the source (Figure 1) over which alignment scores are defined for Concept 1 and Concept 2 (Figure 1). Note that, the subgraph covered by Concept 2 contains the wrong node labeled Name. The alignment scores are respectively 5.8 and 2.4.

It is important to note that as with schema matching algorithms, our alignment score computation might not always return the right transformation automatically. However, because of its modular approach, users can easily view the concept that covers a particular subgraph, and expand and correct the matching as required. Except step 1, all the steps run in $O(|V(G)| + |E(G)|)$ time. If the maximum number of elements stored in any cell of $M_C[][]$ is k, then the time required for step 1 is $O(\text{Depth}(C)(\text{Depth}(G) + \text{F})k^2)$. Generally both k and $\text{Depth}(C)$ are quite small and thus the computation is fast.

V. Filtering

In this section we describe the semantic and structural filtering used to reduce the search space of possible matches between concepts and subgraphs of the input schema. We use the M^F data-structure defined in the previous section, for filtering as well. A filtering method is correct, if it does not eliminate any concept that could have been useful later. We also establish the correctness of our filtering strategy in this section.

A. Semantic Filtering

The semantic score for a concept C is simply the sum of the sim values over all its nodes. This score can be readily computed using $M_C[][]$ as

$$\text{semantic}[C] = \frac{1}{\text{F}} \sum_{i,j} M_C[i][j]$$

If $\text{semantic}[C] < \delta$, where δ is the threshold on the alignment score, C is not considered further, since (as it is easy to show) if $\text{semantic}[C] < \delta$, there cannot exist any subgraph H of G such that $\mathcal{A}(H,C) \geq \delta$. The reason for dividing by F is that the $\text{sim}()$ value for each pair of nodes is added in F different cells in M_C. Computing the semantic score for each matrix M_C requires $O(\text{Depth}(C) \times (\text{Depth}(G) + \text{F}))$ time.

Example 4: The semantic score for Concept 1 computed from the matrix in Figure 3 is 6.8.

B. Structural Filtering

A concept might have nodes that are semantically similar with the input schema, but they may be structurally scattered in the input schema graph. Such a concept is not useful for schema covering and has low alignment score with any subgraph of the schema graph. But semantic filtering alone cannot remove it and we need a filtering strategy that considers structural features.

Structural filtering first partitions the rows of matrix $M_C[][]$ to form submatrices based on the position of the nonzero values in the matrix cells. Then, for each submatrix, it computes a structural score and returns the maximum.

1) Matrix Partitioning: Suppose in two consecutive rows, i and $i+1$, there is no index j such that the cells $M_C[i][j]$ and $M_C[i+1][j+1]$ (2-diagonal) both have nonzero entries. Then from Claim 4.1 of the previous section, no subgraph H of G, for which we have $\mathcal{A}(H,C) \geq \delta$, can contain vertices mapped to both the rows. Therefore, we initially start with the first row in partition P_0, and whenever we encounter rows $i, i+1$ with the above characteristic, we end the current partition at row i and initiate a new partition from row $i+1$. Each partition corresponds to the submatrix over which the structural score is computed.

Example 5: Consider the matrix computed for concept 2 in Figure 3. The cells which are empty or have a [x] mark, have zero 2-diagonal value. Every consecutive rows have some non-zero 2-diagonals and thus the entire matrix forms a single partition P_0.

2) Partition Processing: For each partition of rows, the algorithm computes a structural score, and if the maximum of all these structural scores is below δ, then C is discarded. For correctness, we want to show that if the maximum of the structural scores of all the partitions is below δ, then for every subgraph H of G, $\mathcal{A}(H,C) < \delta$. Since we have already shown that any such subgraph H cannot contain nodes from multiple partitions, we can concentrate on finding the structural score for each partition separately. Suppose we are considering the partition P_0 containing rows 0 to $\ell - 1$. The structural score for P_0, denoted by $\mathrm{structural}[P_0]$, is initialized to 0. While traversing the rows left to right, if we are at cell $M_C[i][j]$, then the following two cases can happen:

Case 1: $M_C[i][j] = 0$ **or the value of both the cells** $M_C[i-1][j-1]$ **and** $M_C[i+1][j+1]$ **are** 0**.**

Here it follows from Claim 4.1 that $M_C[i][j]$ does not contain any node that contributes to some alignment score. Hence, we can ignore $M_C[i][j]$ and this step is correct.

Case 2: $M_C[i][j] > 0$ **and the value of at least one of** $M_C[i-1][j-1]$ **and** $M_C[i+1][j+1]$ **is greater than zero.**

Here $M_C[i][j]$ has a *nonzero 2-diagonal* value. Let $M_C[i][j']$, with $j' < j$, be the last cell before $M_C[i][j]$ whose 2-diagonal value is non-zero. That is, $M_C[i][j']$ has a nonzero 2-diagonal value. For every $M_C[i][j'']$ with $j' < j'' < j$, the 2-diagonal value is 0, and $M_C[i][j]$ has a nonzero 2-diagonal value. We update with

$$\mathrm{structural}[P_0] \mathrel{+}= \sum_{y=0}^{\min(j-j'-1, \mathrm{F}-1)} M_C[i][j][y].$$

The reasoning is as follows. If $u \in V(G), v \in C$ contributes $\mathrm{sim}(u,v)$ to $M_C[i][j][r]$, for some r with $0 \leq r \leq \mathrm{F}-1$, then if $r \geq j - j'$, $\mathrm{sim}(u,v)$ also contributes to $M_C[i][j']$. Thus its contribution to $\mathrm{structural}[P_0]$ will come from $M_C[i][j']$, or from a cell considered before that. We therefore do not need to consider its value while processing $M_C[i][j]$. Hence, if $u \in P_0$ and $\mathrm{sim}(u,v)$ contributes to some alignment score, then $\mathrm{sim}(u,v)$ is added to $\mathrm{structural}[P_0]$. So, for all subgraphs H which have vertices mapped to P_0, their alignment scores with C is no more than $\mathrm{structural}[P_0]$.

Thus if the final structural score of M_C,

$$\mathrm{structural}[C] = \max_{0 \leq i \leq l} \mathrm{structural}[P_i],$$

is below δ, then C cannot be matched with any subgraph of G with alignment score at least δ. Hence C can be eliminated for further consideration. The time for computing the structural score for each matrix M_C is $O(\mathrm{Depth}(C)(\mathrm{Depth}(G)+\mathrm{F})\,\mathrm{F})$.

Example 6: Consider the matrix of Concept 2 in Figure 3. Below is the expanded matrix with each entry of the shift subarray separated by ",".

0.3,0	0.1,0.3	0,0.1	0,0	0,0	0,0
0,0	0,0	1.0,0	0,1.0	1.0,0	0,1.0
0,0	0,0	1.0,0	0,1.0	0,0	0,0

After putting 0s in the cells which have 2-diagonal value 0, we have,

0,0	0.1,0.3	0,0.1	0,0	0,0	0,0
0,0	0,0	1.0,0	0,1.0	0,0	0,0
0,0	0,0	0,0	0,1.0	0,0	0,0

Contribution to $\mathrm{structural}[P_0]$ from each cell is tabulated below:

0	0.4	0	0	0	0
0	0	1.0	0	0	0
0	0	0	1.0	0	0

Hence the structural score is 2.4, whereas the semantic score is 3.4.

VI. Performance Evaluation

In this section we evaluate the performance of our schema covering algorithm on real business schemas and schemas generated synthetically from them following several modifications. The goal is to investigate the impact of different components and constraints of the schema covering algorithm on the quality of the final result. We study the effectiveness of the filtering step, the impact of structural flexibility on matching, the effect of different ambiguity constraints, and how the structure of schema graph affects the result quality.

A. Experimental Set-up

1) Input Schemas: For our evaluations, we used five different real world schemas, two from SAP and one each from ORACLE, PeopleSoft and BSVN. For short, these are referred as S1, S2, O1, P1 and B1 respectively. The following Table I summarizes the structural characteristics of these schemas.

Here the average indegree and outdegree are computed with respect to internal nodes only. An average indegree of at least

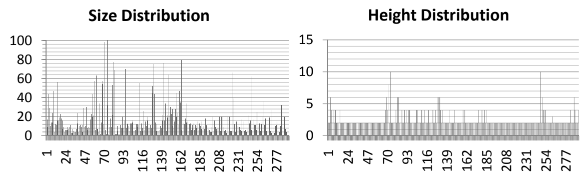


Fig. 5. Distribution of size and height of concepts in the repository.

Schema	S1	S2	O1	P1	B1
# Nodes	175	144	456	252	154
# Internal Nodes/Leaves	25/150	16/128	38/418	28/224	14/140
Average Indegree	2.9	2	5.9	2.6	0.93
Average Outdegree	8.9	10	16.9	10.6	10.9
Depth	3	3	12	10	4
Directed Cycle	No	No	Yes	No	No

TABLE I
CHARACTERISTICS OF TEST-SCHEMA

one indicates shared components. Schema B1 is a tree; S1, S2 and P1 are DAG, whereas O1 is recursive.

From each of these 5 schemas, we generated several synthetic schemas by following one or more of the following modifying operations:

- Edge Insert (EI): Two non-adjacent nodes are selected uniformly at random and a directed edge is added between them. The edge addition might create directed or undirected cycles, and can increase the number of depth values in which a node occurs.
- Move Node (MN): A node that is not a root and has at least one sibling is chosen uniformly at random. The edge connecting the node to one of its parents is deleted, and an edge is inserted to make the sibling the node's new parent. This operation increases degree and depth.
- Delete Subtree (DS): If there exists a subtree rooted at r, then select any node of the subtree uniformly at random and delete the entire subtree below it.
- Change Level (CL): An edge is selected uniformly at random. Let the edge be between u and v. Then a dummy node w is created with a label formed by the concatenation of the labels of u and v. The edge (u, v) is deleted and edges (u, w) and (w, v) are added. As a result, the depth of v increases.

2) Concept Repository: We created a concept repository of various business objects, with 292 concepts from 30 different public sources. Examples are: Account, Billing-Details, Item-Delivery-Schedule, Employee-Job, Customer-BankData, Order-Sales-Credit-Interface, and so on. Among these 292 concepts, there are 56 SAP schemas, 22 Oracle schemas, 14 People-Soft schemas and 15 BSVN schemas. Each of these concepts are then standardized by removing any domain-specific prefix or recurring prefix inherited from the root. For example, an initial SAP object with root label SAP-OrderLineItem is converted into OrderLineItem. The elements with label SAP-OrderLineID and OrderLineCustomerGroup occurring under the root element SAP-OrderLine is converted into ID and CustomerGroup respectively. The standardization is done automatically at the time of inverted index creation and helps to remove any domain specific bias in the labeling. The size and height distribution of all the concepts in the repository are shown in Figure 5. As can be seen from the figure, a majority of the concepts have small size (no more than fifteen) and height. There are few larger concepts as well. There are some concepts like Address which is very popular among different domains (BVSN-MR-BILL-ADDR, BVSN-MR-SHIP-ADDR, Jdbctest-Address, MetaSolv-Address, RMA-Address, SAP-OrderLineContacts etc.) and have large number of shared elements. On the other hand, there are some concepts like Tax Exempt (MetaSolv-TaxExempt) which is specific to a domain. However overall most of these concepts are on related business concepts and exhibit a good number of common attributes among themselves.

Since the repository is quite big, to properly evaluate the covering quality, we explicitly broke each of the test-schemas into several smaller components. For example, schema S1 that has 175 nodes and contains 25 complex elements is broken into 13 concepts. The concepts created from each of the test-schema are named Concept-S1, Concept-S2, Concept-O1, Concept-P1, and Concept-B1, respectively. The number of common elements of the concepts obtained from each test schema depends on the number of shared elements of the original test schemas. We inserted these created concepts in the repository. To distinguish the new repository from the original one, it is referred as the *modified repository*. The hope is that our covering algorithm will detect the pieces of the test-schemas in the modified repository, even when the test-schemas have evolved through the modifications specified earlier.

3) Measure for Decomposition Quality: To evaluate the quality of our decomposition in the modified repository, we compared the real cover R (found manually) between the created concepts and the several variations of the test-schemas, and the cover P found by the decomposition algorithm. Let I denote the correct matches in the cover, that is, $I = R \cap P$, and let F denote the false matches, or the elements which are covered but have no matching elements in the concept ($F = P \setminus R$). Let M denote the elements that have matches but are not covered, that is $M = R \setminus P$. We used the quality measures previously employed by many other researchers for match studies [9][12][5], namely, *precision*, which is $|I|/|P|$, and *recall*, which is $|I|/|R|$. It is difficult to determine R manually for the original repository due to its size. Therefore, in this case, we used precision and recall to detect the quality of the chosen concepts. That is, for each detected concept, we estimated the precision and recall value and computed the average.

B. Results on the Modified Repository

1) Effect of graph structure and modifying operations: Figure 6 (i),(ii),(iii),(iv),(v) show the values of precision and recall for the different variations of S1, S2, O1, P1 and B1 respectively. The greedy algorithm for coverage with strict ambiguity constraint and F = 2 is used to obtain the decomposition. Consider Figure 6(i). The first graph indicates that when

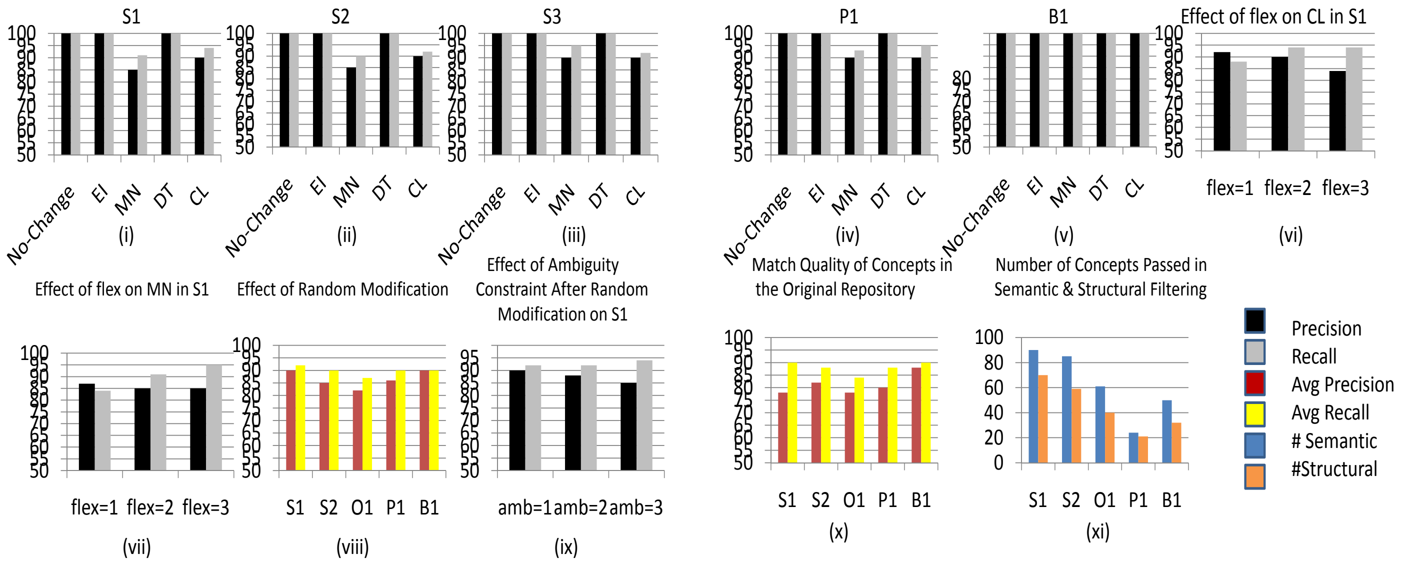


Fig. 6. Results of Schema Decomposition

S1 is presented for covering in the modified repository, all the concepts in Concept-S1 are detected with correct matches among the elements. In this case precision and recall are 100%. The same value is observed across all five schemas in this case. The next four plots are obtained by modifying S1 by 10 random EI, 10 random MN, 5 random DT, and 10 random CL operations respectively. In the case of MN and CL, precision and recall are mostly above 90% for all five schemas. For DT and EI, precision and recall are both 100% for all of them. For B1, which is a tree, none of the operations has any effect on its precision and recall values. Figure 6(viii) shows the effect of applying these operations at random 20 times, for $\mathrm{F} = 2$. We ran the experiment five times and computed the average precision and recall over the runs. They are both close to 90% in all the schemas.

2) Effect of structural flexibility: Figure 6(vi) and (vii) show the effect of varying $\mathrm{F} = 1, 2, 3$, when S1 is modified by 10 random CL and MN operations respectively. As F increases, recall improves for CL at the cost of precision. For MN, precision remains unaffected but recall improves. As we observed, the value of precision is affected by CL, due to the insertion of dummy nodes with similar labels.

3) Effect of ambiguity constraint: Figure 6(ix) shows the effect of varying the ambiguity values to $1, 2, 3$ after S1 has undergone 20 random modifications. As can be seen, recall starts improving with looser ambiguity constraint. There is a little drop of precision, which can be recovered, once the ambiguity is removed manually. This is the trend we observed for the rest of the test schemas as well.

Note that a schema can evolve in many different ways over time, semantically and structurally. We selected to implement modifications that we believe are general enough to capture a large number of these structural variations. However since semantic matching is not the focus of our work, we have not tried to change the semantic labels. High value of precision and recall in all the cases indicate, that even if there are semantic variations, our schema decomposition algorithm coupled with good semantic matcher (see [15] for a survey) will perform substantially well. Our current implementation of schema decomposition only supports some basic semantic matching based on n-grams, prefix, suffix, and so on.

C. Results on the Original Repository

We ran experiments on the original repository. We maintained the semantic and structural threshold with the low value of 3, since most of the concepts have small size.

1) Effectiveness of filtering and bias towards same domain: The number of concepts that passed the semantic and structural filtering are given in Figure 6(xi). As can be seen, the semantic filter eliminates a large portion of the concepts in the repository. However more than a tenth of the concepts passed by semantic filtering are removed by structural filtering. This shows the effectiveness of using structural filtering on top of semantic filtering. Even after semantic filtering, for any given test schema, concepts from other domains persist along with the concepts from the same domain. Most of these other domain concepts are removed by the structural filtering. For S2, after the final filtering step, all the remaining concepts are SAP concepts. For S1, there are few concepts from other domains containing some very popular attributes like ADDRESS, NAME, CONTACT, and so on. The remaining concepts are mostly from the same domain. For example, four BSVN schemas named BVSN-BV-USER-PROFILE, BVSN-MR-DESTINATIONS, BVSN-MR-BILL-ADDR and BVSN-MR-ACCT-PROFILE containing attributes including NAME, ADDRESS, CITY, ZIPCODE, and COUNTRY pass the structural filtering step. They match with the address portion of S1 containing a complex element SAP4-OrderSoldToInfo with attributes AddressLine, CustomerName, City, Country, and PostalCode. However none of these BSVN schemas are retrieved in the final decomposition. In the final decomposition, the complex element SAP4-OrderSoldToInfo is covered by SAP-OrderLineContacts, which although is an approximate match, is qualitatively a better one than any of the BSVN schemas. In another fragment of S1, it contains information about the date an item is sold in attributes DATEQUALIFIER, DATE and TIME. Although the filtering step qualifies schemas like DeliverySchedule from a different domain, the final result matches a SAP concept called SAP-OrderLineDateData to it. In fact this same trend is observed for the other three schemas. Though most of the concepts in the repository are on related concepts using similar attribute names, the result indicates that their structural layout must be quite different. Thus when overall coverage is concerned, schema covering always favors concepts from the same domain.

2) Goodness of the retrieved concepts: The average precision and recall values of the concepts discovered are shown in Figure 6 (x). The value of recall in all the cases are around 85-90%, precision is between 78-88%. The somewhat low values of precision and recall are mainly due to our simple semantic matching strategy. When the retrieved concepts are small enough, the automated schema covering can be used as an advisor. The wrong matchings can be manually corrected, improving the result. For all the 5 schemas, around 40% of the nodes are covered by the obtained concepts in the original repository.

VII. Related Work

In Information Integration, the role of automatic schema matching is to suggest candidate matches/correspondences between the elements of two schemas. For a comprehensive compilation of approaches see the survey of Rahm and Bernstein [15]. Some of the different systems that have been developed for matching include SemInt, CUPID, SF, LSD, GLUE, DIKE, COMA, TranScm, Momis etc. It has also been recognized that robustness can be further improved if outcomes from several schema matchers are combined. Marie et al. [11] for example investigates ways to estimate the uncertainty of matching from schema matcher *ensembles*. There is also a large body of work that explores semantic matchings between ontologies, such as [5] and [6].

Related to our goal of better reuse, Madhavan et al. [9], leverage a *corpus* of schemas and mappings to improve the results of the schema matching task. In particular, they augment the evidence about elements in the schemas being matched, and use statistics about schemas and their elements to infer domain constraints.

However, the reuse of matching has been limited to element to element match and not on larger matching concepts. Most of these works can only handle small and structurally simple schema and define direct schema matching from source to target. Finally recent works by Rahm et al. [16] introduces an interesting fragment based matching approach for handling large schemas. In their work, source and target are divided into several fragments. Each pair of source and target fragments are then compared to detect the best matching pairs. However their proposed fragments are either disjoint sub-schemas, which can be very large or leaf elements (complex and simple), and simple shared types, which can be very small. Also comparing each source fragment with all the target fragments is time-consuming. In addition to this, Hu et al. [7] consider a similar fragment based approach for ontology matching, where they break each ontology into blocks of RDF sentences. However in both these cases, there is no notion of a fragment that matches a well understood concept or covering an ontology with such concepts.

Finally An et al. [2] utilizes an underlying ER model to obtain the concepts related to a schema. They further assume that schema matching between source and target are known, and use the known matchings along with the concepts to improve schema mapping. In contrast, our goal has been to discover a small collection of related concepts and the schema matchings from each concept to appropriate subgraphs of the source and the target schema.

VIII. Conclusion

In this paper, we introduced the problem of schema covering, the method of covering a schema with select objects from a repository and proposed efficient algorithms for solving it. In this regard, we defined new measures of coverage and similarities which can be of independent interest. Schema covering is a building block in the large framework of reuse. In this area there are other challenges that need to be addressed properly. One example is the creation of concept repository: it involves selection, cleansing and unification of input objects. Another example is the automatic identifications of map flows among the concepts of the decomposed source and the target schema. An extension of Clio project at IBM Almaden [1] has started investigating these issues and incorporating schema covering to progress towards the goal of a *simple, modular* and *reusable* system for data integration.

References

[1] Bogdan Alexe, Michael Gubanov, Mauricio A. Hernández, Howard Ho, Jen-Wei Huang, Yannis Katsis, Lucian Popa, Barna Saha, and Ioana Stanoi. Simplifying information integration: Object-based flow-of-mappings framework for integration. In *BIRTE '08, Invited Paper.*

[2] Yuan An, Alexander Borgida, Renée J. Miller, and John Mylopoulos. A semantic approach to discovering schema mapping expressions. In *ICDE*, pages 206–215, 2007.

[3] Hong Hai Do, Sergey Melnik, and Erhard Rahm. Comparison of schema matching evaluations. In *Revised Papers from NODe 2002.*

[4] Hong-Hai Do and Erhard Rahm. Coma: a system for flexible combination of schema matching approaches. In *VLDB '02.*

[5] Anhai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Learning to map between ontologies on the semantic web. In *WWW '02.*

[6] Avigdor Gal, Giovanni Modica, Hasan Jamil, and Ami Eyal. Automatic ontology matching using application semantics. *AI Mag.*, 2005.

[7] Wei Hu, Yuzhong Qu, and Gong Cheng. F. *Data Knowl. Eng.'08.*

[8] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics 1975.*

[9] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Halevy. Corpus-based schema matching. In *ICDE '05.*

[10] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *VLDB '01.*

[11] Anan Marie and Avigdor Gal. Managing uncertainty in schema matcher ensembles. In *SUM '07.*

[12] Sergey Melnik, Hector Garcia-molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm. In *ICDE '02.*

[13] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *VLDB '00.*

[14] Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *VLDB '02.*

[15] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal '01.*

[16] Erhard Rahm, Hong-Hai Do, and Sabine Massmann. Matching large xml schemas. *SIGMOD Rec. '04.*

[17] Aravind Srinivasan. Improved approximations of packing and covering problems. In *STOC '95.*