Managing Uncertainty of XML Schema Matching

Reynold Cheng, Jian Gong, David W. Cheung

Department of Computer Science, The University of Hong Kong
Pokfulam Road, Hong Kong
{ckcheng, jgong, dcheung}@cs.hku.hk

***Abstract*— Despite of advances in machine learning technologies, a schema matching result between two database schemas (e.g., those derived from COMA++) is likely to be imprecise. In particular, numerous instances of "possible mappings" between the schemas may be derived from the matching result. In this paper, we study the problem of managing possible mappings between two heterogeneous XML schemas. We observe that for XML schemas, their possible mappings have a high degree of overlap. We hence propose a novel data structure, called the *block tree*, to capture the commonalities among possible mappings. The block tree is useful for representing the possible mappings in a compact manner, and can be generated efficiently. Moreover, it supports the evaluation of *probabilistic twig query* (PTQ), which returns the probability of portions of an XML document that match the query pattern. For users who are interested only in answers with k-highest probabilities, we also propose the top-k PTQ, and present an efficient solution for it.**

The second challenge we have tackled is to efficiently generate possible mappings for a given schema matching. While this problem can be solved by existing algorithms, we show how to improve the performance of the solution by using a divide-and-conquer approach. An extensive evaluation on realistic datasets show that our approaches significantly improve the efficiency of generating, storing, and querying possible mappings.

I. Introduction

Schema matching methods, which derive the possible relationship between database schemas, play a key role in data integration [1]. In B2B platforms (e.g., Alibaba and DIYTrade.com), each company involved has its own format of catalogs, as well as documents of different standards. The use of schema matching streamlines trading and document exchange processes among business partners. Moreover, important integration techniques like query rewriting (e.g., [2]) and data exchange (e.g., [3]) depend on the success of schema matching. Researchers have therefore developed a number of automatic tools for generating schema matchings (e.g., COMA++ [4], Clip [5], and Muse [6]).

Generally, a schema matching result consists of a set of edges, or *correspondences*, between pairs of elements in each of the schemas. A *similarity* score, augmented to a correspondence, indicates the likelihood that the pair of elements involved carries the same meaning. Figure 1 shows two simplified schemas used to represent a purchase order in two common standards: XCBL (www.xcbl.org) and OpenTrans (www.opentrans.org). A portion of the schema matching result between these two schemas, generated by COMA++, is also shown. For example, the element `CONTACT_NAME(ICN)` in the schema of Figure 1(b) corresponds to three `ContactName` elements (`BCN`,`RCN`, and `OCN`) in the schema of (a).

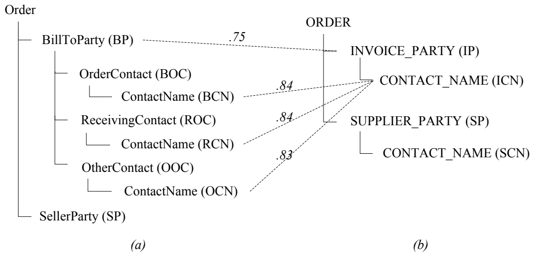


Fig. 1. (a) a source schema, (b) a target schema.

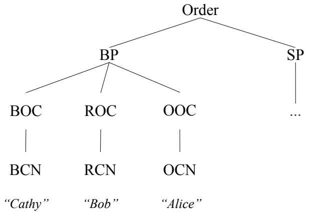


Fig. 2. A source document for Fig 1(a).

As we can see in Figure 1, the scores of the three correspondences shown are quite close. Intuitively, `ICN` has similar chances to correspond to each of the three elements in schema (a). Then, how should this "uncertainty" of the relationship among schema elements be handled? A possible way is to consult domain experts to point out which correspondence is a "true" one. In the absence of human advice, an alternative is to pick up the correspondence with the highest score (e.g., `RCN` and `BCN`), or use some aggregation algorithms (e.g., [7]). Unfortunately, this can lead to information loss. Consider an XML query:

$$Q = //IP//ICN$$

that is issued on the schema (b) (also called "target schema"). This query inquires the contact name information of the invoice party. Let us further suppose that a XML document, shown in Figure 2, conforms to schema (a) (also called "source schema"). Using the *query rewriting* approach [2], Q is first transformed to a query that can be answered on the source schema, using the correspondence provided by the schema matching. The query answer generated on the source schema is then translated back to the one that conforms to the target schema. Depending on the correspondence used, the query yields different answers. For example, if If `IP` is mapped to

BP, and ICN to RCN, then the query answer is “Bob”. If ICN is mapped to OCN instead, then the answer is “Alice”. Notice that the similarity scores in this example are very close, and so the query answers obtained by using any of these correspondences should not be ignored.

In fact, recent research efforts handle the above problem by viewing a schema matching as a set of *mappings* [8], [7]. For each mapping, an element either has no correspondence, or only matches to one single element in another schema. For example, a mapping for Figure 1 contains only one correspondence from ICN to RCN. Each mapping has a probability value, which indicates the chance that the mapping exists. An advantage of this approach is that it reduces the need of human advice. More importantly, it retains the information provided by different correspondences. For example, the correspondences shown in Figure 1 can yield three different mappings (with each mapping containing only one of these correspondences). Let us suppose the probabilities for the mappings that contain the correspondences (ICN, BCN), (ICN, RCN), (ICN, OCN) are respectively 0.3, 0.3, and 0.2. Then the answer for Q is {(“Cathy”, 0.3), (“Bob”, 0.3), (“Alice”, 0.2)}.

In this paper, we also consider a schema matching as a set of mappings with probabilities. We found that a mapping between typical XML schemas can contain hundreds of correspondences. Thus, a schema matching may have a large number of XML mappings, and a lot of space for storing the mappings can be required. Evaluating an XML query on these mappings can be inefficient, since every mapping may have to be visited. To tackle this problem, we develop the *block tree*, which is a compact representation of a given set of mappings. For example, suppose that both mappings A and B contain a correspondence (c, d) (where c and d are elements from the source and target schemas). Then, a *block*, which contains (c, d) for A and B, can be created. Hence, if many mappings share a large number of correspondences, then a “large” block can be created to store a single set of these correspondences, and this saves significant space costs. Our experimental investigations show that the block tree exploits the observation that many XML mappings have a high degree of overlapping in their correspondences, and is thus able to compress mappings effectively. In one of our experiments, there are 13 blocks, containing 20 correspondences each, between two XCBL and OpenTrans schemas. Each set of correspondences is shared by 20% of all possible mappings. Moreover, if a query is evaluated in the block tree, the part of the query relevant to the block needs only be translated to the source schema once, for all mappings that share the correspondences in the block. This reduces the time required to answer a query.

Two technical challenges remain to be resolved for the block tree. First, given a set of mappings, how do we find their common correspondences (for generating blocks)? This is not trivial. A mapping A may share a set x of correspondences with mapping B. Also, A may share another set y of correspondences with mapping C. Finding out all the common correspondences can be very expensive, since all sets of correspondences of varying sizes for each mapping have to be checked. Keeping these different blocks itself is space-inefficient. For storage and querying purposes, however, finding all blocks is *not* necessary. Instead, we aim at finding all *constrained blocks* (or *c-blocks* in short). A c-block has a minimum number of mappings that share the correspondences associated with it. Moreover, the target nodes of the correspondences associated with a c-block must form a complete subtree of the target schema. We present new pruning rules for detecting blocks that cannot be c-blocks. We also develop an efficient algorithm for creating a block tree that contains c-blocks. Notice that this block tree is a flexible structure, in which the user can determine the maximum number of c-blocks to be discovered, as well as their minimum sizes of shared mappings, depending on the construction time and the storage space allowed.

The second challenge is about how to use the block tree to answer XML queries. We perform a detailed case study on the *twig query*, which is a query that specifies a “path" on the target schema, inquiring documents defined on the source schema [9]. (The query Q that we just illustrated is an example of this query.) In view of numerous possible mappings that exist between source and target schemas, we propose a new definition of twig query, called *probabilistic twig query* (or *PTQ* in short). Conceptually, a PTQ returns a set of tuples (*pat*, *prob*), where *prob* is the probability that a pattern in a document (*pat*) satisfies the twig query. With the aid of the block tree, we develop an efficient algorithm to evaluate a PTQ. Our algorithm adopts the *query rewriting* approach, a common method used for answering twig queries under a single schema mapping [2]. Our algorithm recursively decomposes the given query into subqueries according to the correspondences specified by the blocks in the block tree. We further present a variant of PTQ, called *top-k PTQ*, which returns answers with k-highest probabilities. This query would be useful to users who are only interested in answers with high confidences. We demonstrate a simple and efficient method for evaluating top-k PTQ.

We also study the issues of generating possible mappings for a schema matching. There is a growing need in managing personal data scattered on computer desktops and mobile devices (e.g., *Dataspace* [10]), as well as retrieving information from user-defined databases in the Internet (e.g., GoogleBase, base.google.com). Due to the existence of numerous types of schemas, these systems have to handle the integration efforts of these schemas in a scalable manner. In this paper, we study the related problem of efficiently deriving possible mappings from a given schema matching. Since a schema matching may consist of an exponential number of possible mappings, a practical approach is to only extract from the schema matching the h mappings with the highest probabilities. This problem, as pointed out by Gal [7], [11], is essentially a h-maximum bipartite matching problem, and can be solved by algorithms like [12] and [13]. Adopting these algorithms to find the top-h mappings, however, suffer from the fact that a large-size bipartite has to be created. To tackle this, we propose a divide-and-conquer solution, where the bipartite graph is first decomposed into smaller, disconnected "sub-bipartites", before a bipartite-matching algorithm is used. Due to the sparse nature of XML schema matchings, we found from our experimental results that the speed of generating possible mappings can be improved by about an order of magnitude. It is worth noting that our solution is not limited to any specific bipartite matching algorithm. Also, although we address this problem in the context of XML schemas, our technique can be potentially applied to relational schemas.

To summarize, our contributions are:

- We develop the block tree to represent a set of mappings in a compact manner;
- We present efficient methods for generating a block tree;
- We propose the probabilistic twig query (PTQ) and use the block tree to evaluate it;
- We define top-k PTQ, and address its computation issues;
- We improve the possible mapping generation process; and
- We conduct experiments on real data to validate our methods.

The rest of the paper is as follows. In Section II we discuss the related work. Section III describes the details of the block tree structure and how it can be generated. We examine the evaluation of PTQ and top-k PTQ the probabilistic twig query in Section IV. We explain our approach in generating probabilistic mappings in Section V. In Section VI we present the experimental results. Section VII concludes the paper.

II. Related Work

Let us now examine the related work done in the management of schema matching uncertainty (Section II-A) and in XML integration (Section II-B).

A. Handling Uncertainty in Schema Matching

As surveyed in [14], the result of schema matching used in real-world applications is often uncertain. To handle these uncertainties automatically, recent works investigate the idea of representing a schema matching as a set of "probabilistic mappings", i.e., each mapping has a probability of being correct [8], [15], [7]. In [8], Halevy et al. study this model in the context of relational tables. They also propose the "by-table" and "by-tuple" semantics. A by-table semantic means that every tuple in the same source table follows the same mapping. In the by-tuple semantic, each tuple in the source table can have its own choice of the mapping. The model used in our paper is analogous to the by-table semantic, where we assume each XML document under the source schema uses the same mapping.

Based on the relational probabilistic mapping model, [8] studies the complexity of evaluating SPJ queries. Gal et al. [16] extend their algorithms to answer aggregate queries (e.g., `COUNT`). Our research differs from these works, since we consider the evaluation of queries on probabilistic mappings for XML schemas. We further propose a compact representation of probabilistic XML mappings called the block tree, and demonstrate how it can be used to answer XML queries.

A few works have investigated the derivation of multiple probabilistic mappings. In [15], Sarma et al. discuss the generation of the mediate schema, as well as the derivation of probabilities for the mappings between the mediate and the source schemas. In [7], [11], the authors point out that given a schema matching (with a set of correspondence of scores), finding the mappings with the k-highest probabilities is essentially a k-maximum bipartite matching problem. These "top-k mappings" can be used to represent the schema matching. The current fastest algorithms for deriving these mappings are based on Murty [12] and Pascoal [13]. In Section V, we explain how Pascoal's algorithm can be improved by employing a divide-and-conquer solution. Our experiments show that this enhancement can be an order of magnitude faster than that algorithm.

In [10], Salles et al. discussed another approach for managing uncertainty in data integration. They developed *trails*, which are essentially probabilistic and scored hints used for data integration. The *trails* can be gradually included in the *dataspace* system, in order to improve the query performance.

B. Data Integration in XML

In [2], Yu et al. present query rewriting approaches for XML schemas. In [17], the authors discuss the evaluation of queries using XML views. Berstein et al. [18] present a set of operators to create and manipulate XML schema mappings. [19] propose the "nested mapping" semantic for a XML schema mapping. [3] studies the XML data exchange problem. To our understanding, none of these work treats a schema matching as a distribution of mappings. We present an efficient method for evaluating a twig query over probabilistic XML mappings. Although the works in [20] discuss the evaluation of queries over "probabilistic XML documents", they address the representation of uncertainty in the elements of a XML document, rather than the imprecise relationship between source and target schemas as studied by us.

III. The Block Tree

The block tree is a compact representation of possible mappings. We explain the concepts of *blocks*, *c-blocks*, and *block tree* in Section III-A. Section III-B discusses how to efficiently create a block tree. We assume the schema matching in Figure 1 is represented by five possible mappings shown in Figure 3. Table I shows the symbols used in this paper.

A. Blocks, c-Blocks, and Block Tree

Let U be a schema matching, with S and T as its source and target schemas. Let $M = \{m_1, \ldots, m_{|M|}\}$ be a set of possible mappings between S and T. We use (x, y) to denote a correspondence of elements x and y, where x and y belong to S and T respectively. A block is a collection of correspondences shared by one or more mappings between S and T, as shown below:

Definition 1: A **block** b has two components:

- A set $b.C$ of correspondences in U; and

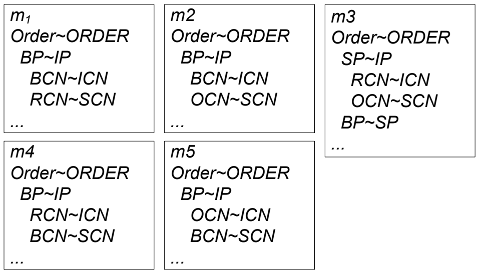
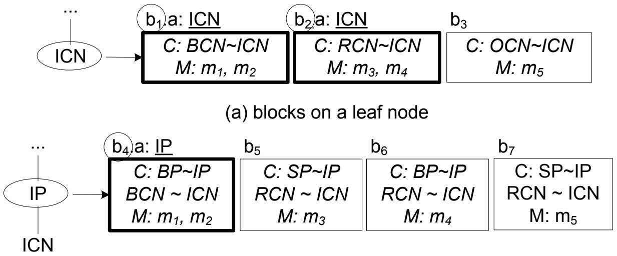


Fig. 3. Five possible mappings of Fig 1.



(a) blocks on a leaf node

(b) blocks on a non-leaf node

Fig. 4. Illustrating blocks and c-blocks (bolded).

TABLE I
NOTATIONS AND MEANINGS.

Notation	Meaning
	Schema Matching
S	Source schema
T	Target schema
U	Schema matching between S and T
M	Set of possible mappings between S and T
m_i	The i-th mapping of M, with $i \in [1, \|M\|]$
p_i	The probability of m_i
	Block Tree
$b.C$	Set of correspondences of block b
$b.M$	Set of mappings of block b
$b.a$	Anchor of c-block
τ	Confidence threshold of c-block
X	A block tree for M
H	Hash table associated with X
	Probabilistic Twig Query
q_T	A probabilistic twig query on T, with l nodes
d_S	An XML document which conforms to S
R	Answers to q_T
R_i	Matches of q_T on d_S using mapping m_i
$pr(R_i)$	Probability that R_i is correct

- A set $b.M$ of IDs of mappings, where $b.M \subseteq M$; for each $m_i \in b.M$, $b.C \subseteq m_i$.

Figure 4(a) shows three blocks (namely b_1, b_2, b_3). Each of these blocks contains a correspondence with element ICN in the target schema. For example, b_1 contains the correspondence (BCN,ICN), which appears in both m_1 and m_2 (see Figure 3). In Figure 4(b), b_4 contains two correspondences: (BP,IP) and (BCN,ICN), which are owned by both m_1 and m_2.

Ideally, if all blocks can be retrieved, then we can obtain a comprehensive view about how mappings overlap. This is prohibitively expensive, since a huge number of blocks can be produced. In fact, as we will discuss in Section IV, it is not necessary to generate all kinds of blocks; those with correspondences shared by *sufficient* mappings and *systematically* organized are already useful for providing low storage cost and high query performance. We formalize these "useful blocks" by the notion of *constrained blocks* (or *c-blocks* in short):

Definition 2: A **c-block**, b, is a block such that:

- b is associated with a target schema element $b.a$ (called *anchor*);
- For every element y of the subtree rooted at $b.a$, there exists source schema element x such that $(x, y) \in b.C$;
- $|b.C|$ is exactly the number of elements rooted at $b.a$; and
- The number of mappings in b, i.e., $|b.M|$, must not be less than $\tau \times |M|$, where τ is called the *confidence threshold*.

Suppose $\tau = 0.4$. Then, in Figure 4, $|M| = 5$. Block b_3 cannot be a c-block, because the number of mappings in b_3 is 1, which is less than 0.4×5=2. However, b_4 is a c-block (with anchor IP) because: (1) In $b_4.C$, there exists a correspondence for *every* descendant of IP (here ICN is the only descendant of IP); and (2) $|b_4.M| \geq 2$. The anchor of b_4 is IP. We circle all the constrained blocks in the figure.

Definition 3: Given a set of c-blocks defined for a schema matching U, a **block tree** X has the following properties:

- X is a tree with the same structure as that of T;
- For every node $e \in X$, e is associated with a linked list of zero or more c-blocks; and
- For every c-block b linked to e, $b.a = e$.

Figure 4 illustrates portions of the block tree for the schema matching of Figure 1. It has the structure of the target schema. In (a), ICN, a leaf node, contains a linked list of c-blocks (b_1 and b_2). In (b), IP is a non-leaf node and is linked to block b_4, with an anchor IP.

B. Constructing the Block Tree

To understand how the block tree can be efficiently generated, we first present two useful lemmas.

Lemma 1: Let t be a non-leaf node, with a c-block b_t. Let (s, t) be the correspondence with target node t in the correspondence set $b_t.C$, and s is some node in schema S. Suppose (s, t) is shared by a set M_t of mappings. Let $u_1, \ldots, u_f$ be child nodes of t. Then, for every u_i, there exists a c-block, b_{u_i}, with anchor u_i, such that:

$$b_t.C = \{(s,t)\} \cup (\bigcup_i^f b_{u_i}.C) \quad (1)$$

$$b_t.M = M_t \cap (\bigcap_i^f b_{u_i}.M) \quad (2)$$

Proof: We can express $b_t.C$ as $\{(s,t)\} \cup (\bigcup_i^f u_i.C)$, where $u_i.C$ is the set of correspondences with target nodes forming a complete subtree rooted at u_i ($i = 1, \ldots, f$), the i-th child node of t. Since b_t is a c-block, (s, t), as well as $u_i.C$, must be shared by the set $b_t.M$ of mappings, where

$|b_t.M| \geq \tau \times |M|$. Note that M_t (the set of mappings that share (s,t)) must be a superset of $b_t.M$. Moreover, since each $u_i.C$ is shared by $b_t.M$, a c-block b_{u_i} can be created with $b_{u_i}.a = u_i$, $b_{u_i}.M = b_t.M$ and $b_{u_i}.C = u_i.C$. Hence, Lemma 1 is correct. ■

Essentially, Lemma 1 states that a c-block of a non-leaf node t can be efficiently generated from the c-blocks at its children. We can further deduce that:

Lemma 2: Let t be a non-leaf node. If t has a c-block, then each of its child nodes must have at least one c-block.

Therefore, if a node does not have any c-block, we can immediately conclude that its parent must have no c-blocks. By visiting the block-tree nodes in a *bottom-up* manner, some high-level nodes may not need not to be examined.

Algorithm 1 *construct_block_tree*

```
Input: schema $T$, mapping set $M$, confidence threshold $\tau$
Output: block tree $X$, hash table $H$
 1: $X \leftarrow init\_block\_tree(T)$
 2: $count \leftarrow 0$
 3: Let $H$ be a hash table of block tree nodes
 4: $construct\_c\_block(X.root)$
 5: $remove\_duplicate\_corr(X, M)$
 6: return $X, H$
function $construct\_c\_block(node\ t)$
Return: no. of blocks created for $t$
 1: if $t$ is leaf then
 2:     $num\_blk\_count \leftarrow init\_block(t)$
 3:     if $num\_blk\_count > 0$ then
 4:         $insert\_hash\_entry(H, t)$
 5:     end if
 6:     return $num\_blk\_count$
 7: else
 8:     // $t$ is a non-leaf node
 9:     $mark \leftarrow$ TRUE
10:     for all $u_i$ in $t$'s child nodes do
11:         if $construct\_c\_block(u_i) = 0$ then
12:             $mark \leftarrow$ FALSE
13:         end if
14:     end for
15:     if $mark$ = FALSE then
16:         return 0
17:     else
18:         $num\_blk\_count \leftarrow gen\_non\_leaf(t)$
19:         if $num\_blk\_count > 0$ then
20:             $insert\_hash\_entry(H, t)$
21:         end if
22:         return $num\_blk\_count$
23:     end if
24: end if
```

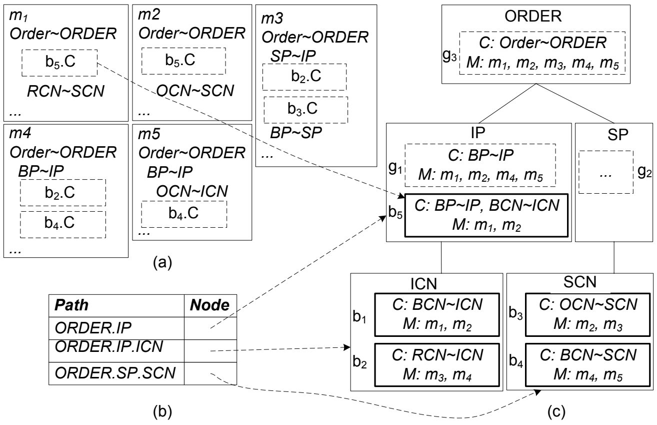


Fig. 5. (a) Mappings, (b) Hash table, and (c) Block tree

Algorithm 1 shows the block-tree construction process. Step 1 constructs a block tree, X, which has the same edges and nodes as that of the target schema T (Step 1). Step 2 uses a variable, *count*, to record the number of c-blocks generated so far. Then, Step 3 initializes a hash table, H, whose hash key is the path in T, and hash value is that node's location in X. We will explain how H is used to answer queries in Section IV. Step 4 calls *construct_c_block* to generate c-blocks for node t. We then perform "mapping compression" in Step 5. Observe that a c-block stores mappings that share correspondences, and so we only need to store a copy of these correspondences. The function $remove\_duplicate\_corr$ performs a pre-order traversal over X; for each mapping recorded in a c-block, we replace its correspondences with a pointer to the block in X. Finally, Step 6 returns X and H.

The recursive function *construct_c_block*, which is first invoked on X's root, performs a post-order traversal over X. It takes a node t as input, generates c-blocks for t, and returns the number of them created. We consider two cases:

CASE 1: t is a leaf node. $init\_block(t)$ is called (Step 2), whose job is to generate c-block(s) for t according to the mapping set M. Essentially, $init\_block(t)$ groups the mappings in M according to their correspondences, and creates c-blocks for groups that have enough mapping. Their details can be found in [21]. If the number of c-blocks is non-zero, we add t's path from root and its location in the block tree to H (Steps 3-5), and return the number of blocks created.

CASE 2: t is a non-leaf node. Step 9 initializes $mark$ to TRUE. Then, Steps 10-14 perform *construct_c_block* for each child node of t, and see if any one of them returns zero. If this happens, t cannot have any c-block (Lemma 2). So, $mark$ is set to FALSE (Step 12), and zero value is returned (Steps 15-16). Otherwise, *gen_non_leaf* is executed (Step 18), to generate all c-blocks for t. We will elaborate on this important function later. Steps 19-22 create a hash sentry for t, and return the number of blocks generated.

Figure 5 illustrates the block tree and the supporting data structure for the sample mappings in Figure 3. The block tree has the structure of a target schema, with each node containing a linked list of c-blocks. The dash-lined boxes in each mapping indicates the part of the correspondences that are stored in the block tree. For example, the correspondences {(BP,IP),(BCN, ICN)} of m_1 and m_2 are stored in block b_5. The hash table stores the name of a target element, and its link to the corresponding node in the block tree.

Next, we describe gen_non_leaf, which generates blocks for a non-leaf node t. This function (Algorithm 2) uses the result of Lemma 1. First, Step 1 executes $init\_block$ on t. Conceptually, we treat t as a "leaf node", and attempts to construct c-blocks for t based on its correspondences with the source schema. If none is found, no c-blocks can be created at t, and a zero value is returned (Step 2). Otherwise, we copy the block list created by $init\_block$ to a temporary list, $list_t$ (Step 4), and delete t's block list in Step 5. Steps 8 and 9 enumerate combinations of the blocks in $list_t$, as well as the block list at each of the child nodes of t. The conjunction of the mapping IDs of these blocks, namely M', are computed in Step 11. If $|M'| \geq \tau|M|$, a new c-block can be found based on the union of the correspondences of M' (Lemma 1). Steps 13-18 construct a new c-block. In Step 28, if a new c-block is found, a hash entry is inserted to H. Finally, $list_t$ is discarded in Step 30 and the number of new blocks constructed is returned.

To control the number of testings on the block combinations, we only allow the number of failed attempts (for generating a c-block) up to MAX_F. Since a user may not have enough time and space to generate and store all the blocks, we limit the number of c-blocks generated to MAX_B (Steps 22-24). The use of these two parameters implies fewer c-blocks may be found, which can affect query performance. We evaluate this effect experimentally in Section VI.

Figure 5 illustrates Algorithm 2. In the `IP` node, g_1 is the temporary block generated by Step 1, and b_5 is the c-block after considering the block list of `ICN`, IP's child node. For `ORDER`, after g_3 is created by Step 1, no c-block can be found after considering `IP` and `SP`, and so it is discarded.

Algorithm 2 Function $gen\_non\_leaf(node\ t)$

```
Return: no. of blocks created for $t$
 1: if $init\_block(t) = 0$ then
 2:     return 0
 3: end if
 4: Let $list_t \leftarrow$ c-block-list of $t$
 5: delete c-block-list of $t$
 6: $count\_new \leftarrow 0$ // no. of new c-blocks
 7: $num\_trial \leftarrow 0$ // no. of failed block-making attempts
 8: for all $b \in list_t$ do
 9:     for all tuple $\{b_{j_1}, b_{j_2}, \ldots, b_{j_f}\}$ do
10:         // $b_{j_k}$ is $j_k$-th c-block of $k$-th child of $t$, with a fanout of $f$
11:         $M' \leftarrow b.M \cap (\bigcap_{k=1}^{n} b_{j_k}.M)$
12:         if $(|M'| \geq \tau \times |M|)$ and $(count < \mathit{MAX\_B})$ then
13:             Let $new\_b$ be new c-block
14:             $new\_b.M \leftarrow M'$ // $new\_b$ is new block
15:             $new\_b.C \leftarrow b.C \cup (\bigcup_{k=1}^{n} b_{j_k}.C)$
16:             $attach\_to\_node(new\_b, t)$
17:             $count\_new \leftarrow count\_new + 1$
18:             $count \leftarrow count + 1$
19:         else
20:             $num\_trial \leftarrow num\_trial + 1$
21:         end if
22:         if $(count \geq \mathit{MAX\_B})$ or $(num\_trial \geq \mathit{MAX\_F})$ then
23:             break the for-loop of line 8
24:         end if
25:     end for
26: end for
27: if $count\_new > 0$ then
28:     $insert\_hash\_entry(H, t)$
29: end if
30: discard $list_t$
31: return $count\_new$
```

Spatial Complexity. Each block has $O(|T|)$ correspondences and $O(|M|)$ mappings, and a size of $O(|T| + |M|)$. Since the maximum number of blocks generated is *MAX_B*, and the block tree has $|T|$ nodes, the block tree size is $O(\mathit{MAX\_B}(|T| + |M|))$. The hash table size is $O(|T|)$.

Time Complexity. Generating c-blocks at a leaf node, using *init_block*, requires $C_L = O(|M||S| \log |M| + \tau^{-1})$. Producing c-blocks at a non-leaf node, using *gen_non_leaf*, has a cost of

$$C_N = O(\min(\mathit{MAX\_F}, \mathit{MAX\_B}) \cdot (f+1)|M| + \mathit{MAX\_B} \cdot |M|)$$

where f is the maximum fanout. Since *construct_c_block* uses a postorder traversal, each node is visited once. Let $h = \lfloor \log_f |T| \rfloor$ be the height of the block tree. Then, the total cost of *construct_block* is $O(f^h C_L + (|T| - f^h) C_N)$, which is less than $O(|T|(C_L + C_N))$. The cost of *remove_duplicate_corr* is $O(\mathit{MAX\_B}|M||T|)$. Hence, the spatial and the construction time complexities of the block tree are polynomial. The details can be found in [21].

IV. Evaluating Twig Query over Block Tree

We now study the Probabilistic Twig Query (PTQ), which provides query answers over possible mappings. We describe its definition in Section IV-A. Then, Section IV-B explains how the block tree can be used to answer PTQ efficiently. We discuss the top-k PTQ in Section IV-C.

A. The Probabilistic Twig Query

Let us briefly review a twig query. A *twig pattern*, q, is a tree, where each node has a label (eg., ICN) and an optional predicate (eg., ICN = "$Alice$"). Each node has an edge labeled either '/' (parent-child edge) or '//' (ancestor-descendant edge). For example, in Figure 1, a twig query $q = /ORDER//ICN$ asks for a contact name of the purchase order. Given a document d and a twig pattern q with l nodes, a *match* of q in d is a set of nodes $\{n_1, ..., n_l\}$ from d, such that for each node $n_i (1 \leq i \leq l)$, the label and the predicate (if any) of the i-th node in q is satisfied by n_i; in addition, the structural relationship (i.e., parent-child or ancestor-descendant) of the nodes in q is the same as that of $\{n_1, ..., n_l\}$. [1]

In schema matching, the twig pattern q_T (called *target query*) is posed against target schema T, but the XML document of interest, d_S, conforms to source schema S. An answer to q_T is then a "match" of q_T on d_S – which can be obtained by translating (or *rewriting*) q_T into a *source query* q_S according to a mapping m. Then, q_S is answered on d_S by finding all the matches of q_S on d_S. Each match to q_S is then translated through m, in order to become an answer of q_T.

As discussed before, the uncertainty of matching between source and target schemas can be modeled as a set of possible mappings with probabilities. If these possible mappings have similar probability values, for querying purposes it is better not to consider only one of them; instead, we consider the query

[1] We assume all nodes in q are *distinct*. If this does not hold, q is converted into multiple sub-queries, each of which having distinct nodes. We then combine the answer of each sub-query to form the answer of q.

result for each mapping. The result of a query then becomes a set of document patterns with probabilities.

Now, let the probability that a possible mapping $m_i \in M$ is true, be p_i, where $\sum_{1 \le i \le |M|} p_i = 1$. The following gives the semantics of a probabilistic twig query.

Definition 4: Given a set of possible mappings M, and a document d_S conforming to source schema S, a **Probabilistic Twig Query (PTQ)** over target schema T, denoted by q_T, is a twig pattern on T, which returns a set of pairs $R = \{(R_i, pr(R_i)\}(1 \le i \le |M|)$, where R_i is the set of matches of q on d_S through mapping m_i, and $pr(R_i)$ is the non-zero probability that R_i is correct.

Algorithm 3 (*query_basic*) Basic Query Evaluation

```
Input: PTQ $q_T$, mapping set $M$, document $d_S$
Output: Query answers to $q_T$
1: $M' \leftarrow filter\_mappings(M, q_T)$
2: return $twig\_query(q_T, M', d_S)$
function twig_query($query\ q, mapping\ set\ M', document\ d_S$)
Return: Answer $R$ to PTQ $q$
1: $R \leftarrow \emptyset$
2: for all $m_i \in M'$ do
3:     $q_S \leftarrow rewrite(q_T, m_i)$
4:     $R_i \leftarrow match(d, q_S)$
5:     $R \leftarrow R \cup \{(R_i, p_i)\}$
6: end for
7: return $R$
```

Basic Solution. Algorithm 3 illustrates *query_basic*, a straightforward solution to PTQ. Step 1 prunes all "irrelevant" mappings. A mapping m is irrelevant if it does not contain a correspondence for every query node in q_T. Hence, there will not be any match for q_T on d_S through m with a non-zero probability. The $filter\_mappings$ function scans each mapping, and removes all irrelevant ones. Step 2 then invokes *twig_query* on the mappings not filtered, i.e., M'.

In *twig_query*, Step 1 initializes the query result R. For each mapping $m_i \in M'$, we translate q, using m_i, to a source query q_S (Step 3), and match the query pattern on d_S (Step 4), in order to obtain a result R_i. Note that the probability that R_i is correct is exactly the probability that m_i is true, i.e., p_i. Hence, we can put (R_i, p_i) in R (Step 5), and return R in Step 7. The total cost of this algorithm is $O(|M||q|(|S|+|R|))$. The derivation of this cost can be found in our technical report.

The problem of *query_basic* is that the result of q_T for each mapping m_i has to be obtained independently. Note that this involves translating the query and results using m_i, and also retrieving the data from a source document. This process can be expensive if (1) not many mappings are filtered and (2) a mapping has many correspondences. Let us examine how the block tree can alleviate these problems.

B. Evaluating PTQ with the Block Tree

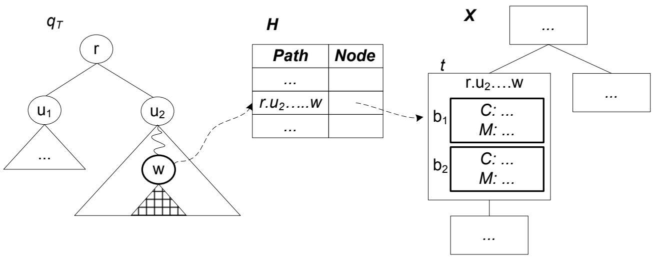


Fig. 6. Query evaluation with block tree

The process of supporting PTQ execution with a block tree is illustrated in Figure 6. Recall that each entry in the hash table H (generated together with the block tree) contains a path in the target schema, and a pointer to some block-tree node corresponding to that path. Suppose a query q_T has a root r. If r is found in H (and hashed to node t), then we can use the c-blocks stored at t to speed up the evaluation of q_T. Intuitively, we only have to execute q_T *once* for all the mappings indicated in each block of t, since these mappings have the same correspondences rooted at r. If r is not found in H, we decompose q into three subqueries:

- q_0, which has a single node r;
- q_1, which has the subtree rooted at u_1; and
- q_2, having the subtree rooted at u_2.

q_0 is simple and is evaluated directly; q_1 and q_2 are computed recursively. For example, q_2 is decomposed into subqueries until a node, w, is found in the hash table. Then, the subquery issued at w can use the two blocks stored in t to speed up evaluation. The answers to the subqueries are then joined to form the final query answer.

Let us now study the details of this method. First, we use *filter_mappings* to remove irrelevant mappings. Then, we invoke *twig_query_tree* (Algorithm 4). First, q's root is searched in the hash table H (Step 1). If a node t is found, then $query\_subtree$ is invoked in Step 3 to answer q. (We explain the details of this function later.) Otherwise, there are two cases:

1) q contains a single node: we answer q by calling *twig_query* (Steps 5-6);

2) q has one or more children: we call $split\_query(q)$ to decompose q into subquery q_0, which contains q's root node only, and a set of subqueries $q_1, \ldots, q_f$, each of which is rooted at q's i-th $(1 \le i \le f)$ child, where f is the fanout of q's tree (Step 9). The subquery q_0 is evaluated using *twig_query*, while other subqueries are evaluated by recursively calling *twig_query_tree* (Step 11-13). Next, we join the results from these subqueries. Let $R(q_j)$ be the query result for query q_j. Then, for each mapping m_i, we combine $R_i(q_0)$ with results at child nodes, i.e., $R_i(q_1), \ldots, R_i(q_f)$ (Steps 14-19). Note that a match f_0 in $R_i(q_0)$ can join with a match f_j in $R_i(q_j)$ if f_j's root is a child of f_0. Essentially, this is a binary structural join process, and can be supported efficiently with a stack-based join algorithm [22] (Step 16). The combined result is included in R (Step 18), which is returned in Step 20.

The *query_subtree* function uses c-blocks at node t to support efficient evaluation of query subtree q_t (which has t as the root node). For every c-block b associated with t, a twig query is issued on a single mapping that comprises only the correspondence set of b, i.e., $b.C$ (Step 4). The query result, y, is then replicated for all mappings that share these correspondences (i.e., $b.M$), in Steps 5-6. The set M_s is the union of all the mappings that appear in the c-blocks at t (Step 7). This set is used to answer q_t for mappings that are included in a c-block. For other mappings (i.e., $M' - M_s$), we invoke *twig_query* to evaluate them directly (Step 10). Finally, we return the answers generated by all mappings in M'. Notice that the query performance can be affected by the number of c-blocks generated, but query correctness will not be affected by using fewer c-blocks.

Algorithm 4 (*twig_query_tree*) PTQ evaluation with block tree

```
Input: PTQ $q_T$, relevant mappings $M'$, document $d_s$, block tree $X$
Output: query answers to $q_T$
 1: $t = find\_node(q_T.root, H)$ // $H$ is the hash table
 2: if $t \neq NULL$ then
 3:     return $query\_subtree(q_T, t, M', d_S, X)$
 4: else
 5:     if $q$ is a leaf then
 6:         return $twig\_query(q, M', d_s)$
 7:     else
 8:         $R \leftarrow \varnothing$
 9:         $(q_0, q_1, \ldots, q_f) \leftarrow split\_query(q)$ // $q_0$ is the root of $q$,
            and $q_1, \ldots, q_f$ are $q$'s children
10:         $R(q_0) \leftarrow twig\_query(q_0, M', d_S)$
11:         for all $j \in [1, f]$ do
12:             $R(q_j) \leftarrow twig\_query\_tree(q_j, M', d_S, X)$
13:         end for
14:         for all $i \in [1, |M'|]$ do
15:             for all $j \in [1, f]$ do
16:                 $R_i(q_0) \leftarrow stack\_join(R_i(q_0), R_i(q_j))$
17:             end for
18:             $R \leftarrow R \cup \{(R_i(q_0), p_i)\}$
19:         end for
20:         return $R$
21:     end if
22: end if
function $query\_subtree$(query tree $q_t$, node $t$, mapping set $M'$,
    document $d_S$, block tree $X$)
Return: Answer to $q_t$
 1: Let $M_s \leftarrow \varnothing$ // all mappings involved at $t$
 2: Let $Y \leftarrow \varnothing$ // query result for blocks at $t$
 3: for all $b \in$ blocks at $t$ do
 4:     $y \leftarrow twig\_query(q_t, \{b.C\}, d_S)$
 5:     for all $m_i \in b.M$ do
 6:         $Y \leftarrow Y \cup \{(y, p_i)\}$
 7:         $M_s \leftarrow M_s \cup \{m_i\}$
 8:     end for
 9: end for
10: $Z \leftarrow twig\_query(q_t, M' - M_s, d_S)$
11: return $Y \cup Z$
```

Compared with *query_basic*, which treats each mapping independently, our new approach can achieve faster performance for mappings that share correspondences. The price for this is the cost of decomposing/merging subquery results. In the worst case, no block is found in the block tree, and *twig_query* needs to be evaluated for every node of q. The most expensive function in the decomposition/merge process, *stack_join* combines the result for each edge in q in $O(\max_{i=1,\ldots,|M|}(|R_i|))$ [22] times. If q has E edges, the cost of decomposition-and-join is $O(|E| \max_{i=1,\ldots,|M|}(|R_i|))$. Our experiments show that this worst case is rare, and the additional overhead does not override the benefit of using the block tree for query evaluation.

C. Top-k Probabilistic Twig Query

A query user may only be concerned about answers with high probabilities. To facilitate a user for expressing this preference, we propose a variant of PTQ, called top-k PTQ, as follows:

Definition 5: A **top-k Probabilistic Twig Query**, or **top-k PTQ**, is a PTQ, where only k answer tuples $\{(R_i, pr(R_i)\}$ $(1 \leq i \leq |M|)$, whose probabilities are among the highest ones, are returned.

Essentially, this query allows a user to obtain query answers with the k-highest probabilities. If there are more than k answers with the k highest probabilities, we assume that any one subset of these answers need to be returned.

A top-k PTQ can be evaluated by first computing its PTQ counterpart, and then return the answer tuples with the k highest probabilities. This is not the fastest method, however. Instead, we insert the following two steps at the end of the *filter_mappings* function (which prunes mappings before the *twig_query_tree* is evaluated):

1) Sort the mapping set M' in ascending order of the probability of each mapping in M'.
2) Return the first k mappings in M'.

This change must be correct, since the answers to a top-k PTQ must be derived from k distinct mappings with the highest probabilities. By using this method, the number of mappings considered by *twig_query_tree* can be reduced, thereby achieving a higher query performance.

V. Efficient Possible Mappings Generation

We now discuss an efficient method for constructing possible mappings. Section V-A reviews existing methods for producing these mappings. Section V-B presents an enhancement.

A. Finding top-h Mappings

Let us first explain how possible mappings can be created. Recall that a schema matching between source and target schemas (S and T) is a set of correspondences with similarity scores. To obtain possible mappings, an element in S can choose to match an element in T (based on the correspondence information), or not match any element at all. By enumerating these choices, all legal mappings between S and T can be derived. To generate the probability of a possible mapping m_i, the "score" of m_i can be used. This score, which is usually a function of scores of correspondences that appear in m_i (e.g., the sum of correspondence scores of m_i) [7], reflects the confidence that m_i is correct. One simple way to obtain m_i's probability is to normalize m_i's scores over the total scores of all mappings between S and T.

However, the number of mappings obtained in this way can be exponentially large. A more practical method is to represent

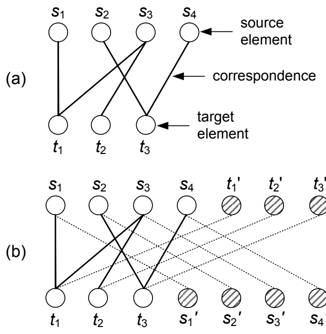


Fig. 7. Finding top-h mappings: (a) schema matching; (b) bipartite.

a schema matching with h mappings, which have the highest scores among the possible mappings [7], [16]. These "top-h mappings" can be obtained with polynomial-time algorithms. The probability of each top-h mapping is then yielded by normalizing its score over the total scores of the h mappings.

The retrieval of top-h mappings can be viewed as the h-maximum bipartite matching [7], which extracts h matchings with the highest costs. If the score of a mapping is the sum of its correspondence scores, then a polynomial-time algorithm (e.g., Murty [12], [13]) can be used. To use these algorithms, the schema matching is preprocessed; to model the fact that a schema element may not correspond to any other elements, an "image" of each element in S (respectively T) needs to be added to T (respectively S). A correspondence between an element and its image is added, which has a score of zero or a value specified by the matching semantics). Figure 7 illustrates a schema matching, and the resulting bipartite. The image nodes are shaded, and their correspondences are drawn in dotted lines. If a mapping returned by a bipartite matching algorithm contains an element e that corresponds to its image, this means e does not correspond to any other element. Let $S.N$ and $T.N$ be the set of elements of S and T respectively. Then, the size of the bipartite is the sum of the number of source and target elements, i.e., $|S.N| + |T.N|$ (e.g., in Figure 7, the bipartite has a size of 4+3=7). Hence, the complexity of finding top-h matching, using Murty's algorithm, is $O(k(|S.N| + |T.N|)^3)$.

B. Partitioning a Schema Matching

The real XML schemas used in our study contain up to hundreds of elements. Thus, $|S.N|$ and $|T.N|$ can be large, and the speed of top-h-mappings retrieval can be affected. This can be a burden for systems like Dataspace and GoogleBase, which maintain mappings for many user- and application-defined schemas. We observe that a schema mapping can be viewed as a set of *partitions*, which are "sub-matchings" of a given schema matching. Figure 8 illustrates two partitions derived from Figure 7(a). Notice that these partitions are *disjoint*, i.e., they do not have the same elements or correspondences.

Since these partitions are disjoint, deriving a top-h mapping from a schema matching U can be done in two steps:

1) Obtain the top-h-mappings from each partition.
2) Generate the top-h-mappings by combining the results in Step 1.

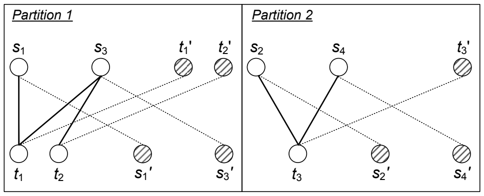


Fig. 8. Partitions of Figure 7, with image elements shown.

The advantage of this approach is that if partitions are small, finding top-h-mappings on each partition is faster than on U. In Figure 8, for instance, the sizes of partitions 1 and 2 are respectively 4 and 3, which are smaller than the size-7 bipartite in Figure 7. As we show next, deriving and merging the mappings in these partitions is easy.

Deriving Partitions. We now explain how to generate partitions. Let e_s and e_t be elements in $S.N$ and $T.N$ respectively. We say e_s and e_t are *connected* (denoted by $e_s \leftrightarrow e_t$), if $\exists e_1, \ldots, e_j \in S.N \cup T.N$, such that $e_1 = e_s$, (e_l, e_{l+1}) exists (where $1 \leq l \leq j-1$), and $e_j = e_s$.

Definition 6: A *partition* is a pair $\langle S', T' \rangle$, such that:

- (subset) $S'.N \subseteq S.N$, $T'.N \subseteq T.N$,
- (connected) $\forall e_i, e_j \in S'.N \cup T'.N$, $e_i \leftrightarrow e_j$, and
- (maximum) $\forall e_i \in S.N \cup T.N$, if $\exists e_j \in S'.N \cup T'.N$, such that $e_i \leftrightarrow e_j$, then $e_i \in S'.N \cup T'.N$.

Essentially, a partition is a maximal connected subgraph of the bipartite. In Figure 8, for example, the two partitions are maximal. Definition 6 implies that two partitions cannot share any elements or correspondences (else they become a single partition), Moreover, for a given schema matching, there can only be one single set of partitions. To find a partition, we randomly pick an element (called *seed*) in S. Then, the partition that contains the seed is generated by inserting to the partition all elements connected to the seed, and their correspondences with the seed. This "seed expansion" process is repeated for the newly discovered elements in the partition, until no more new elements can be found.

Algorithm 5 describes the detailed process. First, U is partitioned in Step 1. Then, the top-h mappings are computed from each partition using a standard algorithm (e.g., [12], [13]), and are *merged* to obtain the top-h mappings for U (Steps 2 to 6). The top-h mappings are returned in Step 7.

The *partition* function produces a set of partitions, using the seed expansion process that we have discussed (Step 4).This is repeated for every element in S not yet visited, so that all partitions can be found (Steps 3-7). The complexity of *partition* is $O(|U|)$, or $O(|S.N| \cdot |T.N|)$.

The *merge* function derives top-h-mappings Z from the combination of two partitions. Since these two partitions are disjoint, Z can be found by considering only the top-h-mappings obtained from the two partitions, i.e., *top_h_mappings* and *current*. If the number of partitions for U is l, the average size of a partition is $|U|/l$. Then, the average

Algorithm 5 Partitioning algorithm

```
Input: source schema $S$, target schema $T$, schema matching $U$, no. of mappings $h$
Output: top-$h$ mappings
$\{P_1, \ldots, P_l\} \leftarrow partition(U)$
$top\_h\_mappings \leftarrow bipartite\_match(P_1, U)$
for $i$ = 2 to $l$ do
    $current \leftarrow bipartite\_match(P_i, U)$
    $merge(top\_h\_mappings, current)$
end for
return $top\_h\_mappings$
function $partition(schema\ matching\ U)$
Output: Set of partitions $R$
$R \leftarrow \varnothing$
$flag[e] \leftarrow false$, $\forall e \in S.N$
while $\exists seed \in S.N$, $flag[seed] = false$ do
    $P \leftarrow expand(seed, U)$ // $P$ is a new partition
    $R \leftarrow R \cup \{P\}$
    $flag[e] \leftarrow false$, $\forall e \in$ source node of $P$
end while
return $R$
```

complexity of *merge* is $O((\frac{|U|}{l})^2)$.

On average, a partition has $\frac{|S.N|+|T.N|}{l}$ elements. Assuming a fast algorithm like Murty [13] is used, the average complexity of generating top-h mappings from a partition is O($k(\frac{|S.N|+|T.N|}{l})^3$). Since there are l partitions, the average complexity of Algorithm 5 is O($\frac{k(|S.N|+|T.N|)^3}{l^2} + (\frac{|U|^2}{l} + |U|)$), the former and the latter term being the bipartite-matching and merging-partitioning costs respectively. Generally, the larger number the partitions (and thus a smaller average partition size), the higher is the performance. Next, we examine this algorithm experimentally.

VI. Experimental Results

We now describe the experimental results. Section VI-A describes the experimental setup. We present our results in Section VI-B.

A. Setup

We used a variety of real XML schemas commonly used in E-Commerce. These include the OpenTrans (OT) and XCBL schemas (which can be downloaded from their websites), as well as schemas provided by COMA++[2]. Based on these schemas, we generate ten matching results, namely, $D1, \ldots, D10$ (Table II). Each matching contains a source schema S, a target schema T, and an option (opt) of the matching method used in COMA++ (f means *fragment* and c means *context*). The capacity (Cap.) is the number of element correspondences of the matching.

A document `Order.xml`, chosen from XCBL sample $autogen\_full$ and contains 3473 nodes, is used as our source document. For the block tree, the default values are: $|M| = 100$, $\tau = 0.2$, $MAX\_B = 500$, $MAX\_F = 500$. Unless stated otherwise, $D7$ is used for analysis. We also tested ten queries, about purchase orders, on $D7$, as shown in Table III [3]. These queries cover different portions of the target schema and have a wide range of sizes. We implemented the advanced version of Murty's algorithm [13] in order to efficiently generate top-h mappings. Our experiments are run on a PC with Intel Core Duo 2.66GHz CPU and 2G RAM. The algorithms are implemented in C++. Each data point is an average of 50 runs.

TABLE II
Schema Matching Datasets

ID	S	$\|S\|$	T	$\|T\|$	opt	Cap.	*o-ratio*
$D1$	Excel	48	Noris	66	f	30	0.79
$D2$	Excel	48	Paragon	69	c	47	0.63
$D3$	Excel	48	Paragon	69	f	31	0.57
$D4$	Noris	66	Paragon	69	c	41	0.64
$D5$	Noris	66	Paragon	69	f	21	0.53
$D6$	OT	247	Apertum	166	c	77	0.87
***D7**	XCBL	1076	Apertum	166	c	226	0.84
$D8$	XCBL	1076	CIDX	39	c	127	0.82
$D9$	XCBL	1076	OT	247	c	619	0.91
$D10$	OT	247	XCBL	1076	c	619	0.91

TABLE III
Queries used in the experiment

ID	PTQ on dataset $D7$
$Q1$	Order/DeliverTo/Address[./City][./Country]/Street
$Q2$	Order/DeliverTo/Contact/EMail
$Q3$	Order/DeliverTo[./Address/City]/Contact/EMail
$Q4$	Order/POLine[./LineNo]//UP
$Q5$	Order/POLine[./LineNo][.//UP]/Quantity
$Q6$	Order/POLine[./BPID][./LineNO][//UP]/Quantity
***Q7**	Order[./DeliverTo//Street]/POLine[.//BPID][.//UP]/Quantity
$Q8$	Order[./DeliverTo[.//EMail]//Street]/POLine[.//UP]/Quantity
$Q9$	Order[./Buyer/Contact]/POLine[.//BPID]/Quantity
Q10	Order[./Buyer/Contact][./DeliverTo//City]//BPID

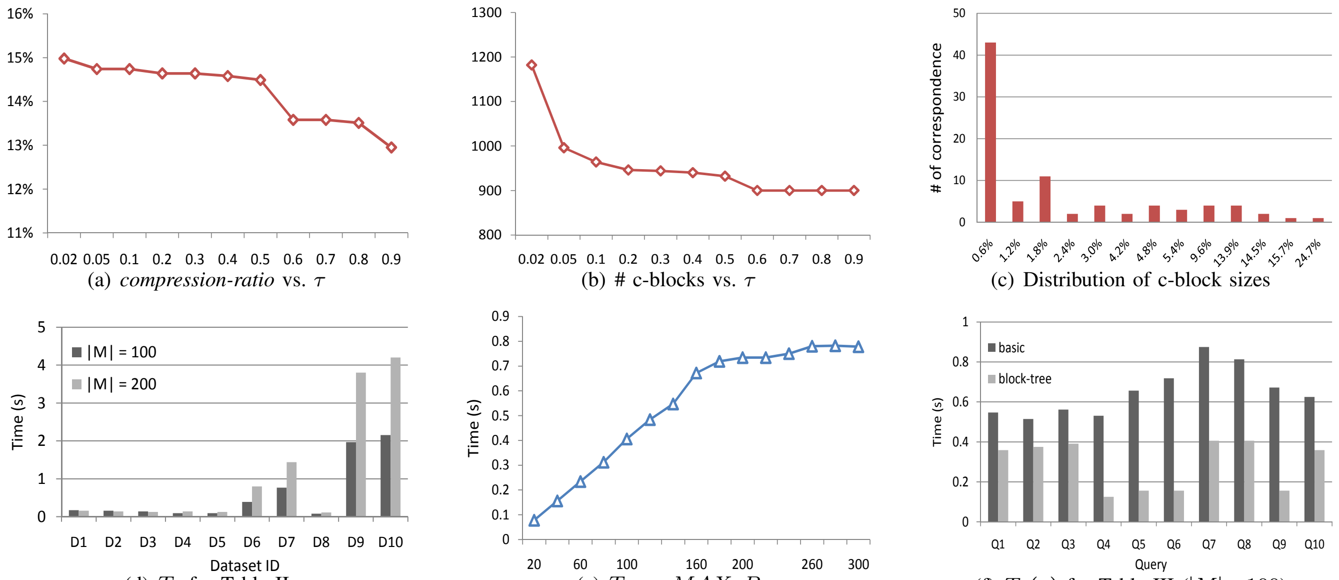
B. Results

1. Mapping Overlap. First, we examine the degree of overlap among the possible mappings generated from a schema matching. For this purpose, we define the *o-ratio* of two mappings m_i and m_j as $\frac{|m_i \cap m_j|}{|m_i \cup m_j|}$. We also define the *o-ratio* of M as the average of the *o-ratio* between all pairs of mappings in M. Table II shows that the *o-ratio* values for the mapping sets are between 0.53 and 0.91. Hence, there exists a high overlap among the mappings. Next, we study how well the block tree exploits this property.

2. Spatial Efficiency of Block Tree. Given a mapping set M, let B be the total number of bytes required to store the block tree and the hash table for M, as well as the mappings of M (with correspondences removed). Then, we define the compression ratio as $1 - \frac{B}{|M|}$. This metric captures the amount of space saved by representing M with a block tree. Figure 9(a) shows the result under different values of τ. At $\tau = 0.2$, the block tree saves 14.64%. When τ increases, fewer c-blocks are created (Figure 9(b)). Therefore, the compression ratio drops.

[2] http://dbs.uni-leipzig.de/Research/coma_index.html

[3] BPID and UP represent BuyerPartID and UnitPrice respectively



(a) *compression-ratio* vs. τ (b) # c-blocks vs. τ (c) Distribution of c-block sizes

(d) T_c for Table II (e) T_c vs. MAX_B (f) $T_q(s)$ for Table III ($|M|$ = 100)

Fig. 9. Experiment results (1)

3. Effectiveness of c-blocks. From Figure 9(b), we can see that the number of c-blocks drop much slower after around τ = 0.1. This means that the number of mappings contained in many c-blocks is much larger than $\tau \times |M|$. Next, Figure 9(c) shows the distribution of c-block sizes, in terms of the number of correspondences contained in the c-blocks. The x-axis is the fraction of target nodes that are contained in the correspondence set of the c-block, and the y-axis is the number of c-blocks of that size. We observe that there is a large proportion (50%) of c-blocks whose sizes are larger than one. The largest c-block contains 41 correspondences. This covers 24.7% of all target schema nodes, and is shared by more than $\tau = 20\%$ of all possible mappings. The average size of all c-blocks is 5.33. These reflect that the c-blocks can effectively exploit the high overlap among mappings.

4. Block Tree Construction. Figure 9(d) shows the time for creating a block tree (T_c) for each dataset in Table II, under different values of $|M|$. We can see that the block tree can be created within a few seconds. Hence, our block tree construction algorithm is efficient. In Figure 9(e), we study the effect of MAX_B on T_c. As expected, T_c increases with MAX_B. When MAX_B is larger than 180, T_c ceases to increase, since the number of c-blocks that can be created is less than MAX_B.

5. Query Performance. Figure 9(f) shows the running time (T_q) for each PTQ shown in Table III. We denote Algorithms 3 and 4 as *basic* and *block-tree*. We observe that *block-tree* outperforms *basic* for all queries we tested. For example, the query time of *block-tree* is 27.18% faster than that of *basic* for Q2; and is 78.27% faster for Q5. On average, *block-tree* is 54.60% faster than *basic*. Similar results can also be observed for a larger set of mappings (i.e., $|M|$ = 500), as shown in Figure 10(a). Hence, our techniques can use the block tree effectively to improve query performance.

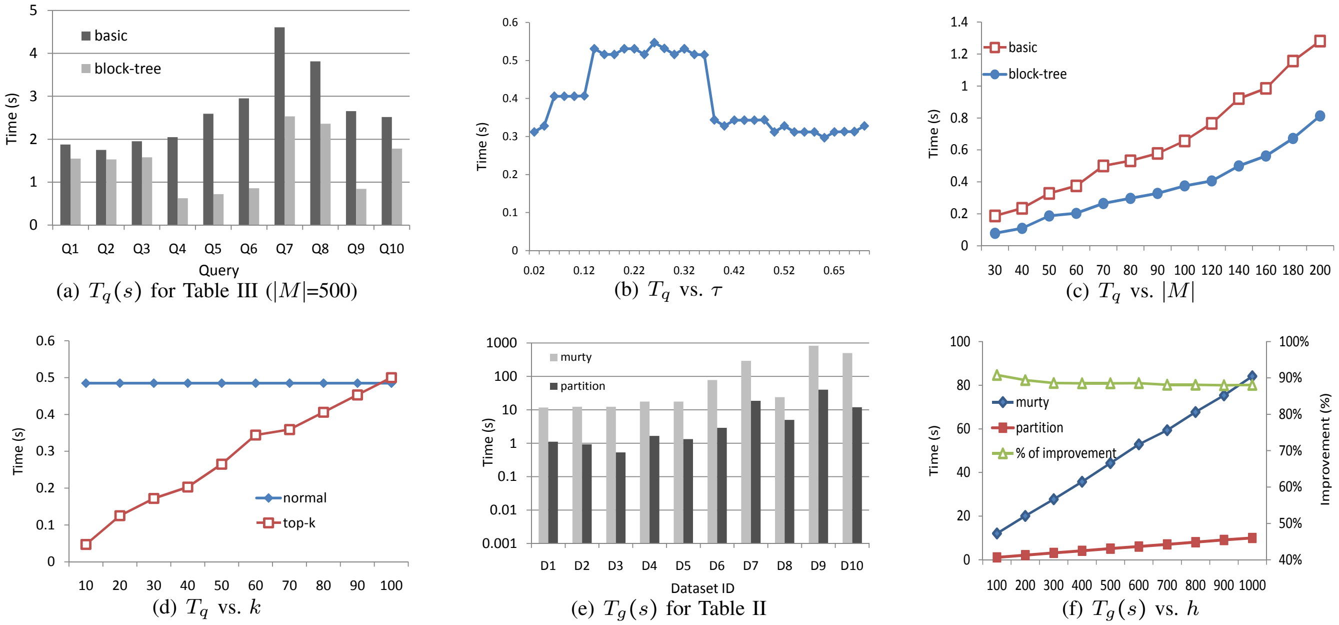
Next, we focus on the query $Q10$, and consider only the block-tree query algorithm.

Figure 10(b) studies the effect of τ on query performance. When τ increases from 0.02 to 0.2, T_q also increases. This is due to the significant drop in the number of c-blocks (c.f. Figure 9(b)), and so there are far less c-blocks in the block tree for facilitating query evaluation. Interestingly, when $\tau \geq 0.4$, T_q becomes smaller. In these situations, although fewer c-blocks can be generated, these c-blocks tend to be shared by many mappings. Also, due to the fewer number of c-blocks, the overhead of decomposing and merging query results is lower. Hence the PTQ performs well when τ is large.

We also test the effect of the size of M on T_q. As shown in Figure 10(c), *block-tree* consistently outperforms *basic* for a wide range of possible mapping sizes; an average improvement 47.05% is registered.

6. Top-k PTQ. We then investigate top-k PTQ. Figure 10(d) shows the performance of top-k PTQ under different values of k. As k increases, more mappings need to be considered, and so T_q increases. The *normal* curve refers to a PTQ without using the top-k constraint. We can see that by placing the top-k constraint on the query, its performance can be significantly improved (e.g., 90.31% when k = 10).

7. Top-h Mapping Generation. We then compare the performance of top-h mapping generation algorithms: $murty$, and our partitioning-based approach $partition$. Figure 10(e) shows the time needed (T_g) on each matching in Table II. We observe that $partition$ consistently outperforms $basic$. This is because the bipartite of the schema matching is sparse, and the number of partitions is large (it ranges from 23 (for $D3$) to 966 (for $D7$)). Finally, Figure 10(f) compares the scalability of $murty$ and $partition$ in terms of h, on dataset $D1$. The left y-axis shows that the amount of time needed by $partition$ is much less than $murty$. We also show the fraction of time



(a) $T_q(s)$ for Table III ($|M|$=500)

(b) T_q vs. τ

(c) T_q vs. $|M|$

(d) T_q vs. k

(e) $T_g(s)$ for Table II

(f) $T_g(s)$ vs. h

Fig. 10. Experiment results (2)

improvement of $partition$ over $murty$ on the right y-axis. We observe that the improvement is always larger than 87.97%. Our approach can therefore improve T_g significantly.

VII. Conclusions

The need of managing uncertainty in data integration has been growing in recent years. In this paper, we studied the problem of handling uncertainty in XML schema matching. We exploited the observation that XML mappings have high degree of overlap, and proposed the block tree to store common parts of mappings. A fast method for constructing the block tree was proposed. We also studied how to efficiently evaluate PTQ and top-k PTQ with the aid of the block-tree. By noticing that XML schema matchings are often sparse, we proposed to partition the matchings in order to improve the performance of the mapping generation process.

In the future, we would consider how the block tree can facilitate the evaluation of other types of XML queries (e.g., XQuery and keyword query). We would consider the querying of probabilistic XML documents [20], under an uncertain schema matching. We would also study the effectiveness of our mapping generation method in relational schemas.

Acknowledgment

This work was supported by the Research Grants Council of Hong Kong (GRF Project 711309E). We would like to thank the anonymous reviewers for their insightful comments.

References

[1] M. Lenzerini, "Data integration: a theoretical perspective," in *PODS*, 2002.

[2] C. Yu and L. Popa, "Constraint-based XML query rewriting for data integration," in *SIGMOD*, 2004.

[3] M. Arenas and L. Libkin, "XML data exchange: consistency and query answering," *J. ACM*, vol. 55, no. 2, pp. 1–72, 2008.

[4] H.-H. Do and E. Rahm, "COMA: a system for flexible combination of schema matching approaches," in *VLDB*, 2002.

[5] A. Raffio, D. Braga, S. Ceri, P. Papotti, and M. A. Hernández, "Clip: a tool for mapping hierarchical schemas," in *SIGMOD*, 2008.

[6] B. Alexe, L. Chiticariu, R. J. Miller, D. Pepper, and W. C. Tan, "Muse: a system for understanding and designing mappings," in *SIGMOD*, 2008.

[7] A. Gal, "Managing uncertainty in schema matching with top-k schema mappings," in *J. Data Semantics VI*, 2006, pp. 90–114.

[8] X. Dong, A. Y. Halevy, and C. Yu, "Data integration with uncertainty," in *VLDB*, 2007.

[9] L. Qin, J. X. Yu, and B. Ding, "TwigList: make twig pattern matching fast," in *DASFFA*, 2007.

[10] M. A. Vaz Salles, J.-P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi, "iTrails: pay-as-you-go information integration in dataspaces," in *VLDB*, 2007.

[11] H. Roitman, A. Gal, and C. Domshlak, "Providing top-k alternative schema matchings with ontomatcher," in *Proc. Intl. Conf. Conceptual Modeling*, 2008.

[12] K. G. Murty, "An algorithm for ranking all the assignment in increasing order of cost," *Operations Research*, vol. 16, pp. 682–687, 1986.

[13] M. M. B. Pascoal, M. E. V. Captivo, and J. C. N. Clímaco, "A note on a new variant of murty's ranking assignments algorithm," *4OR*, vol. 1, no. 3, pp. 243–255, 2003.

[14] E. Rahm and P. Bernstein, "A survey of approaches to automatic schema matching," *VLDB J.*, vol. 10, no. 4, pp. 334–350, 2001.

[15] A. Das Sarma, X. Dong, and A. Halevy, "Bootstrapping pay-as-you-go data integration systems," in *SIGMOD*, 2008.

[16] A. Gal, M. Martinez, G. Simari, and V. Subrahmanian, "Aggregate query answering under uncertain schema mappings," in *ICDE*, 2009.

[17] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou, "Structured materialized views for XML queries," in *VLDB*, 2007.

[18] P. A. Bernstein and S. Melnik, "Model management 2.0: manipulating richer mappings," in *SIGMOD*, 2007.

[19] A. Fuxman, M. A. Hernandez, H. Ho, R. J. Miller, P. Papotti, and L. Popa, "Nested mappings: schema mapping reloaded," in *VLDB*, 2006.

[20] B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv, "Query efficiency in probabilistic xml models," in *SIGMOD*, 2008.

[21] R. Cheng, J. Gong, and D. W. Cheung, "Managing uncertainty of XML schema matching," The University of Hong Kong, Tech. Rep. TR-2009-12, 2009.

[22] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava, "Structural joins: A primitive for efficient XML query pattern matching," in *ICDE*, 2002.