The VLDB Journal (2012) 21:385–409
DOI 10.1007/s00778-011-0248-4

REGULAR PAPER

Efficient management of uncertainty in XML schema matching

Jian Gong · Reynold Cheng · David W. Cheung

Received: 14 October 2010 / Revised: 3 July 2011 / Accepted: 25 July 2011 / Published online: 25 August 2011

Abstract Despite 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 problems related to managing possible mappings between two heterogeneous XML schemas. First, we study how to efficiently generate possible mappings for a given schema matching task. 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. Second, storing and querying a large set of possible mappings can incur large storage and evaluation overhead. For XML schemas, we observe that their possible mappings often exhibit 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 efficiently generated. Moreover, it facilitates the evaluation of a *probabilistic twig query* (PTQ), which returns the non-zero probability that a fragment of an XML document matches a given query. 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. An extensive evaluation on real-world data sets shows that our approaches significantly improve the efficiency of generating, storing, and querying possible mappings.

Keywords Data integration · Schema matching · Uncertainty · XML

J. Gong (✉) · R. Cheng · D. W. Cheung
Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong, People's Republic of China
e-mail: jgong@cs.hku.hk

R. Cheng
e-mail: ckcheng@cs.hku.hk

D. W. Cheung
e-mail: dcheung@cs.hku.hk

1 Introduction

Schema matching methods, which derive the possible relationship between database schemas, play a key role in data integration [15]. In B2B platforms (e.g., Alibaba and DIYTrade.com),[1] 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., [23]) and data exchange (e.g., [4]) depend on the success of schema matching. Researchers have therefore developed a number of automatic tools for generating schema matchings (e.g., COMA++ [9], Clip [19], and Muse [3]).

In general, 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 and OpenTrans.[2] A portion of the matching result between these two schemas, generated by COMA++, is also shown. For example, the element `CONTACT_NAME(ICN)` in the schema of Fig. 1b can correspond

[1] Alibaba: http://www.alibaba.com, DIYTrade: http://www.diytrade.com.

[2] XCBL: http://www.xcbl.org, OpenTrans: http://www.opentrans.org.

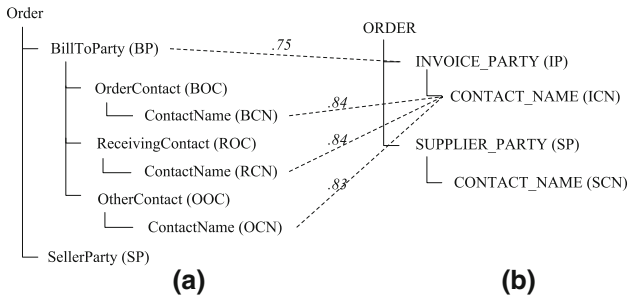


Fig. 1 **a** A source schema and **b** a target schema used in e-commerce. A line between a pair of schema elements represent a correspondence, whose similarity scores is shown

to three ContactName elements (BCN, RCN, and OCN) in the schema of (a).

As we can see in Fig. 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 the "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., [12]). Unfortunately, this can lead to information loss. Consider an XML twig query:

$$q_T = //IP//ICN$$

that is issued on schema (b) (also called "target schema"). This query inquires the contact name information of the invoice party. Consider an XML document, shown in Fig. 2, which conforms to schema (a) (or "source schema"). Using the *query rewriting* approach [23], q_T 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 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 becomes "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.

Recent research efforts handle the uncertainty in a matching by viewing it as a set of *mappings* [10,13]. For each mapping, an element either has no correspondence or only matches to one single element in another schema. Figure 3 shows five mappings derived from the matching in Fig. 1. Notice that m_3 contains only one correspondence from ICN to RCN. The probability that each mapping exists is also known. An advantage of this approach is that it reduces the

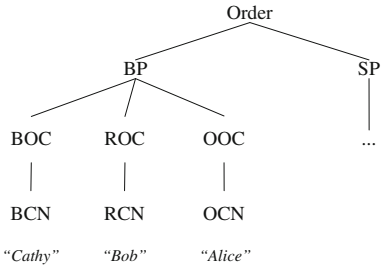


Fig. 2 An XML source document that conforms to the source schema in Fig. 1a (*left*)

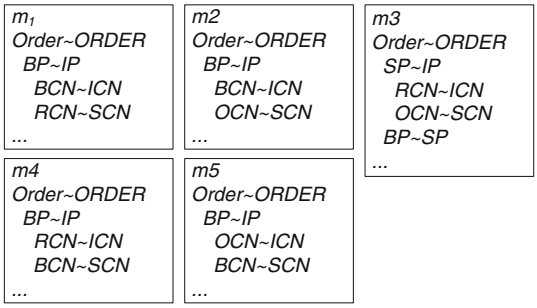


Fig. 3 Illustrating 5 mappings derived from the matching in Fig. 1. For each mapping, an element either has no correspondence, or only matches to one single element in another schema

need of human advice. More importantly, it retains the information provided by different correspondences. Suppose that the total probabilities for the mappings in Fig. 3 that contain (ICN, BCN), (ICN, RCN), and (ICN, OCN) are, respectively, 0.3, 0.3, and 0.2. Then, the answer for q_T is {("Cathy", 0.3), ("Bob", 0.3), ("Alice", 0.2)}. Based on this model, we study three interesting problems:

Problem 1 (*Efficient mapping generation*) We first study the issues of generating possible mappings for a schema matching. There is a growing need for managing personal data scattered on computer desktops and mobile devices (e.g., *Dataspace* [22]), as well as retrieving information from user-defined databases in the Internet (e.g., GoogleBase).[3] Due to the existence of numerous types of schemas, these systems have to handle the integration efforts of these schemas in a scalable manner. We examine the problem of efficiently deriving mappings from a given matching task. Since a matching may derive an exponential number of mappings, a practical approach is to only extract from the matching the h mappings with the highest probabilities. This is essentially an h-maximum bipartite matching problem [12,21] and can be solved by algorithms like [16,17]. Adopting these algorithms to find the top-h mappings, however, suffers from the fact that a large-size bipartite has to be created. We thus propose a divide-and-conquer solution, where the bipartite graph is first

[3] GoogleBase: http://base.google.com

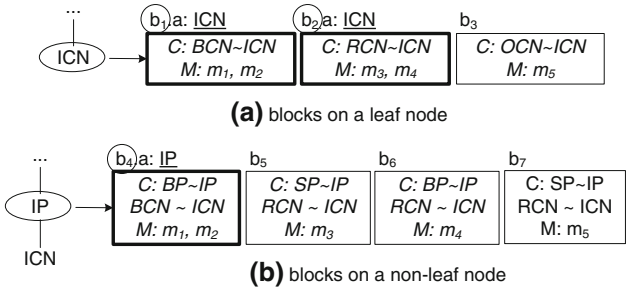


Fig. 4 Blocks and c-blocks: a block consists of a set of correspondences and the mappings that share them. A c-block (*bolded*) requires that the number of mappings that share the correspondences are larger than some threshold

decomposed into smaller and disconnected "sub-bipartites". A bipartite-matching algorithm is then applied to each of the sub-bipartites. The top-h mappings can then be found by merging the results obtained from the sub-bipartites. Due to the sparse nature of XML schema matchings, the speed of generating possible mappings can be improved by an order of magnitude in our experiments. Our solution is not limited to any specific bipartite matching algorithm. Although we address this problem in the context of XML schemas, our technique can potentially be applied to relational schemas also.

Problem 2 (*Mapping compression*) A matching between XML schemas can have hundreds of correspondences. Thus, a large number of mappings that contain many correspondences can be obtained. Evaluating a query on these mappings can also be inefficient, since each of them has to be examined. We thus propose an efficient representation of mappings, based on the concept of *blocks*. Figure 4a shows three blocks (b_1, b_2, b_3) for the mappings in Fig. 3. Each of them contains a correspondence with `ICN`, an element in the target schema. For example, b_1 contains `(BCN,ICN)`, which appears in both m_1 and m_2. In Fig. 4b, b_4 contains two correspondences: `(BP,IP)` and `(BCN,ICN)`, which are owned by both m_1 and m_2. A mapping can now be represented as a set of pointers to the blocks. For example, two correspondences of m_3 (`(SP,IP)` and `(RCN,ICN)`) can be replaced by a pointer to block b_5. As another example, correspondences `(BP,IP)` and `(BCN,ICN)`, which are common to both m_1 and m_2, can be replaced by their respective pointers to b_4. In fact, if many mappings share a large number of correspondences, a "large" block can be created to store a single set of these correspondences, thereby saving significant space costs. In 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 mappings. We use this intuition to develop the *block tree*, a compact representation of mappings. This structure is simply a target schema, whose elements are attached with blocks. Figure 4 shows part of a block tree, where two lists of blocks are attached to elements `ICN` and `IP` of the target schema in Fig. 1. Our experimental study shows that the block tree can compress mappings effectively. 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 significantly reduces the query evaluation time.

How do we find the blocks described above? This question is not easy to answer. As shown in Fig. 4, m_3 forms block b_2 with m_4, since they share (`RCN,ICN`); it also forms a block itself, using (`SP,IP`) and (`RCN,ICN`). Finding all these blocks can be costly, since every subset of each mapping's correspondences needs to be checked. Keeping these different blocks itself is space inefficient. Hence, our solution does not derive all blocks; instead, we compute *constrained blocks* or *c-blocks* in short. A c-block is essentially a block that is deemed useful for mapping compression and query evaluation. Its correspondences are guaranteed to be shared by a sufficient number of mappings. In Fig. 4, the (circled) c-blocks are b_1, b_2, and b_4. Observe that the correspondences stored in each of these blocks are shared by at least 2 mappings. Also, the target elements of the correspondences associated with these blocks form a complete subtree of the target schema. We study pruning rules for detecting blocks that cannot be c-blocks. We also develop a novel algorithm for creating a block tree, which only contains c-blocks. In this algorithm, each mapping is assigned a *signature*, a succinct representation of how correspondences are shared by the mapping. We demonstrate how to use signatures to construct the block tree efficiently.

Problem 3 (*Query evaluation*) We also study how to use the block tree to answer XML queries. We examine the *twig query*, which specifies a "path" on the target schema, inquiring documents defined on the source schema [18]. The query q_T that we illustrated in the proceeding text is an example of this query. In view of numerous possible mappings that exist between source and target schemas, we propose a *probabilistic twig query* (or *PTQ* in short). 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. We develop an efficient algorithm that uses the block tree to evaluate a PTQ. Our algorithm adopts the *query rewriting* approach, a common method used for answering twig queries under a single schema mapping [23]. 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 the k highest probabilities. This query can be useful to users who are only interested in answers with high confidences. We demonstrate a simple and efficient method for evaluating a top-k PTQ.

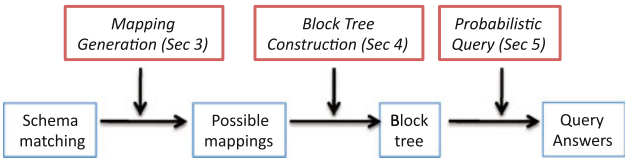


Fig. 5 Mappings, block tree, and probabilistic queries

Figure 5 summarizes the process of handling uncertainty in schema matchings. Given a schema matching, an efficient mapping generation algorithm is applied to derive a set of possible mappings. Then, a block tree is derived for these mappings through an efficient construction algorithm. Probabilistic queries (e.g., PTQ and top-k PTQ) are then evaluated on the block tree.

The rest of the paper is structured as follows. In Sect. 2, we discuss the related work. We explain our approach in generating probabilistic mappings in Sect. 3. In Sect. 4, we describe the details of the block tree structure and how it can be generated. We examine the evaluation of PTQ and top-k PTQ in Sect. 5. In Sect. 6, we present the experimental results. Section 7 concludes the paper with directions of the future work.

2 Related work

Let us now examine the related work done in the field of management of schema matching uncertainty, in Sect. 2.1. We then briefly address the work done in the field of XML integration, in Sect. 2.2.

2.1 Handling uncertainty in schema matching

As surveyed in Rahm and Bernstein [20], the result of schema matching used in real-world applications is often uncertain. To handle these uncertainties automatically, recent works have investigated the representation of a schema matching as a set of "probabilistic mappings", i.e., each mapping has a probability of being correct [8,10,12]. In [10], Halevy et al. studied this model for relational tables.

Based on the relational probabilistic mapping model, [10] studied the complexity of evaluating Selection-Porjection-Join (SPJ) queries. Gal et al. [13] extended 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 use it to evaluate probabilistic twig queries.

A few works have investigated the derivation of multiple probabilistic mappings. In [8], Sarma et al. discussed the generation of the mediate schema, as well as the derivation of probabilities for the mappings between the mediate and the source schemas. In [12,21], the authors pointed 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 [16] and Pascoal [17]. In Sect. 3, 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.

The work closest to us is Cheng et al. [7]. In that paper, an algorithm was proposed to generate the block tree. That method finds all c-blocks in exponential time. Our algorithm finds all c-blocks in polynomial time and uses fewer input parameters. We also develop a top-h mapping generation algorithm, which is faster than the one discussed in Cheng et al. [7].

We now summarize other approaches for managing schema matching uncertainty. In [22], Salles et al. developed the concept of *trails*, which are probabilistic and scored hints. The *trails* can be gradually included in the *dataspace* system, in order to improve its query performance. Recently, Agrawal et al. studied the problem of handling the uncertainty in the source database [1]. As discussed in Sect. 1, a simple way to remove matching uncertainty is to choose the correspondence with the highest score, among the correspondences attached to each target element. We will carry out a qualitative study on this approach in Sect. 6. In [20], the problem of generating different types of possible mappings (e.g., one-to-many) is studied. However, it is not clear how they can be used to answer probabilistic queries. In this paper, we assume that a possible mapping between target and source elements is one-to-one. An interesting future work is to consider other forms of possible mappings.

2.2 Data integration in XML

In [23], Yu et al. presented query rewriting approaches for XML schemas. In [5], the authors discussed the evaluation of queries using XML views. Berstein et al. [6] used a set of operators to create and manipulate XML schema mappings. Fuxman et al. [11] proposed the "nested mapping" semantic for an XML schema mapping. Arenas and Libkin [4] studied 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 [14] discussed the evaluation of queries over "probabilistic XML documents", they address the representation of uncertainty in the elements of an XML document, rather than the

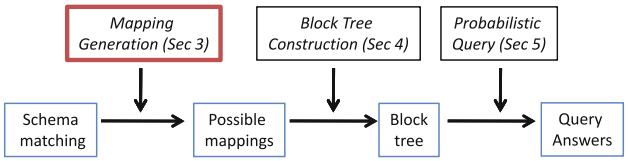


Fig. 6 Possible mapping generation

Table 1 Notations and meanings used in this paper

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 ith 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

imprecise relationship between source and target schemas as studied by us.

3 Efficient possible mappings generation

We now discuss an efficient method for constructing possible mappings, as shown in Fig. 6. Section 3.1 reviews existing methods for producing these mappings. Section 3.2 presents an enhancement, based on partitioning the schema matching. We further discuss an improvement to this algorithm, in Sect. 3.3. Table 1 summarizes the symbols used in this paper.

3.1 Finding top-h mappings

Let U be a given schema matching and S and T be the source and target schemas of U. Let M be a set of mappings derived from U, and $m_i \in M$ (where $i \in [1, |M|]$) be a mapping of

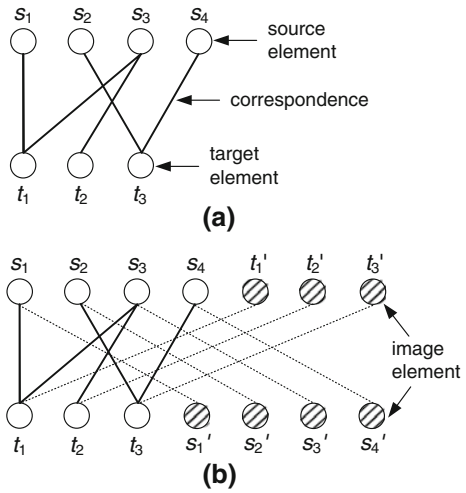


Fig. 7 Finding top-h mappings: **a** schema matching; **b** bipartite. If a mapping returned by a bipartite matching algorithm contains an element e that corresponds to its image, this means e does not correspond to any other element

M. Let (x, y) be a correspondence of elements x and y, where x and y belong to S and T, respectively. To find M, 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. The *score* of a possible mapping m_i is a function of scores of correspondences that appear in m_i (e.g., the sum of correspondence scores of m_i [12]). This score reflects the confidence that m_i is correct. Since the number of mappings can be exponentially large, in practice a matching is represented as a set of h mappings, which have the highest scores among the mappings [12]. The probability values of the top-h mappings are then the normalized scores of these mappings.

The retrieval of top-h mappings can be viewed as the h-maximum bipartite matching problem [12]. If the score of a mapping is the sum of its correspondence scores, a polynomial time algorithm (e.g., *Murty* [16] and *Pascoal* [17]) can be used. To model the fact that an element may not correspond to any other elements, an "image" of each element in S (respectively T) is added to T (respectively S). A correspondence between an element and its image is then added. Figure 7 illustrates a matching and its bipartite graph. The image nodes are shaded, and their correspondences are drawn in dotted lines. Let $S \cdot N$ and $T \cdot N$ be the set of elements of S and T, respectively. The complexity of finding top-h matching, using *Murty* or *Pascoal*, is $O(h(|S \cdot N| + |T \cdot N|)^3)$.

3.2 Partitioning a schema matching

The real XML schemas used in our study contain up to hundreds of elements. Thus, $|S \cdot N|$ and $|T \cdot N|$ can be large, and the speed of top-h-mappings retrieval can be affected. This

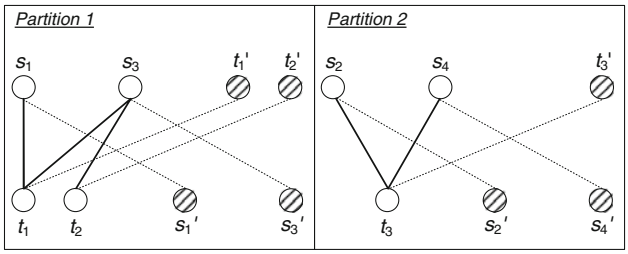


Fig. 8 Partitions of Fig. 7, with image elements shown. Observe that partitions 1 and 2 are *disjoint*, i.e., they do not share the same source or target elements

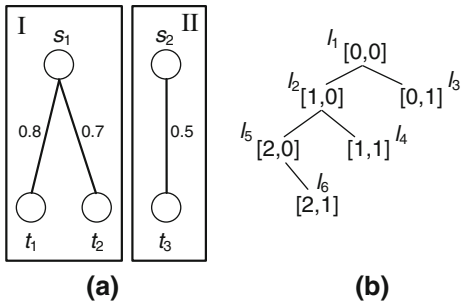


Fig. 9 Heap algorithm: **a** a remainder partition of two non-conflicting partitions; **b** enumeration of vectors

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 "submatchings" of a given schema matching. Figure 8 illustrates two partitions derived from Fig. 7a. Notice that these partitions are *disjoint*, i.e., they do not have the same elements or correspondences. Deriving a top-h mapping from a matching U can be performed as:

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

The advantage of this approach is that if partitions are small, finding top-h-mappings on each partition is faster than on U. In Fig. 8, for instance, the sizes of partitions 1 and 2 are respectively 4 and 3, which are smaller than the size-7 bipartite in Fig. 7. Next, we explain how to derive and merge mappings in these partitions.

Deriving partitions

A *partition* is essentially a *maximal connected subgraph* of the bipartite. Figure 8 shows two example partitions. Notice that any two distinct partitions cannot share elements or correspondences, or else they become a single partition. Moreover, for a given 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 element can be found. The related algorithm (Algorithm 8) and its analysis can be found in Appendix A.

3.3 Handling non-conflicting partitions

A detailed study of the partitions produced by Algorithm 8 reveals that many of these partitions only contain a few correspondences. In our experiments, a matching can contain hundreds of correspondences. Thus, many "small" partitions can be obtained, and a lot of effort is needed to merge these partitions.

To alleviate this problem, we observe that many partitions handled by Algorithm 8 contain target elements that are not shared by more than one source element. We call this kind of partition a *non-conflicting partition*. Figure 9a illustrates two non-conflicting partitions (I and II). Observe that in each partition, none of the target elements are shared by more than one source element. An example conflicting partitions is $\{(s_1, t_1), (s_3, t_1)\}$, where target element t_1 corresponds to source elements s_1 and s_3.

An important observation for non-conflicting partitions is that deriving a mapping for this kind of partitions is rather simple—a sophisticated mapping algorithm, such as *Murty*, is not necessary. For example, the top-2 mappings of partition I are simply $\{(s_1, t_1)\}$ and $\{(s_1, t_2)\}$; for partition II, there is only 1 mapping, i.e., $\{(s_2, t_3)\}$.

Based on this intuition, we develop an efficient algorithm optimized for generating top-h mappings from non-conflicting partitions. Let us define a *remainder partition*, which is essentially a maximal collection of non-conflicting partitions derived from the same matching. Figure 9a shows a remainder partition, which contains two non-conflicting partitions. An experiment on our schema matching data sets shown in Table 2 shows that the size of the remainder partition of a schema matching is on average 77.57% of the size of the target schema. Moreover, the remainder partition contains 68 non-conflicting partitions on average. Hence, a remainder partition can contain a large number of non-conflicting partitions.

We now present a top-h mapping generating algorithm that uses the remainder partition and other partitions. The details are shown in Algorithm 1. First, the function *partition_rmd* finds out all the conflicting partitions and the remainder partition from the schema matching (Step 1). This function is modified from the original algorithm (function *partition* in Algorithm 8): if the current partition, obtained by expanding a *seed*, is non-conflicting, then it is merged into the

Algorithm 1 Partitioning Algorithm (with Remainder Partitions)

Input: source schema S, target schema T, schema matching U, number of mappings h
Output: top-h mappings

1: $\{P_1, \ldots, P_l, P_{rmd}\} \leftarrow partition\_rmd(U)$
2: $top\_h\_mappings \leftarrow bipartite\_match(P_1)$
3: **for** $i = 2$ to l **do**
4: $\quad current\_mappings \leftarrow bipartite\_match(P_i, U)$
5: $\quad merge(top\_h\_mappings, current\_mappings)$
6: **end for**
7: $top\_h\_mapping\_rmd \leftarrow remainder\_match(P_{rmd})$
8: $merge(top\_h\_mappings, top\_h\_mapping\_rmd)$
9: return $top\_h\_mappings$

function $partition\_rmd$
Return: remainder partition, conflicting partitions

1: $R \leftarrow \emptyset$, $P_{rmd} \leftarrow \emptyset$
2: $flag[e] \leftarrow$ FALSE, $\forall e \in S \cdot N$
3: **while** $\exists seed \in S \cdot N$, $flag[seed] =$ FALSE **do**
4: $\quad P \leftarrow expand(seed, U)$ // P is a new partition
5: $\quad$ **if** $is\_conflicting(P)$ **then**
6: $\quad\quad R \leftarrow R \cup \{P\}$
7: $\quad$ **else**
8: $\quad\quad P_{rmd} \leftarrow P_{rmd} \cup \{P\}$
9: $\quad$ **end if**
10: $\quad flag[e] \leftarrow$ TRUE, $\forall e \in$ *source node of* P
11: **end while**
12: return R, P_{rmd}

Algorithm 2 Function $remainder\_match$

Return: top-h mappings of the remainder partition

1: $sort\_correspondence(U)$
2: $l \leftarrow [0, \ldots, 0]$
3: $heap.push(l)$
4: $c \leftarrow find\_score(l)$
5: $rst \leftarrow \{\}$
6: **while** $|rst| < h$ **and** $heap$ is not empty **do**
7: $\quad l \leftarrow heap.pop()$
8: $\quad rst \leftarrow rst \cup \{new\ mapping(l)\}$
9: $\quad L \leftarrow grow(l)$
10: $\quad$ **for all** $l' \in L$ **do**
11: $\quad\quad$ **if** $(|heap| < h$ **or** $find\_score(l') \geq c)$ **then**
12: $\quad\quad\quad heap.push(l')$
13: $\quad\quad\quad c = min(c, find\_score(l'))$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: **end while**
17: return rst

function $grow(int[]\ l)$
Return: a set of vectors

1: $L \leftarrow \{\}$
2: **for** $i := |l| - 1$ to 0 **do**
3: $\quad$ **if** $l[i] + 1$ is valid **then**
4: $\quad\quad l' = [l[0], \ldots, l[i-1], l[i]+1, l[i+1], \ldots, l[|l|]]$
5: $\quad\quad L = L \cup \{l'\}$
6: $\quad$ **end if**
7: $\quad$ **if** $l[i] > 0$ **then**
8: $\quad\quad$ break
9: $\quad$ **end if**
10: **end for**
11: return L

remainder partition P_{rmd}, instead of being created as a separate partition (Steps 5–9). In the end, P_{rmd}, as well as the conflicting partitions, are returned. For the partitions not in the remainder partition, it generates the top-h mappings from them using the same $bipartite\_matching$ function used in Algorithm 8 (Steps 2–6); for the remainder partition, a new function *remainder_match* is used (Step 7), which is then merged with the top-h mappings in the other partitions, in order to obtain the complete top-h mappings (Step 8).

The function *remainder_match*, which finds the top-h mappings from a remainder partition, is detailed in Algorithm 2. It first sorts the correspondences for each source schema element according to the descending order of similarity (Step 1). A vector of integer values, l, is used to uniquely represent a mapping m, where the ith source element chooses the correspondence ranked the $(l[i] + 1)$th among all of its correspondences. The vector l is initialized and pushed to a heap at the beginning (Step 2–3). We use $|l|$ to denote the size of the vector, where $|l|$ equals to the number of source elements in the remainder partition. The variable c maintains the *minimal* score of all the mappings that can be identified by the vectors in the heap, where the function *find_score* returns the score of a mapping identified by a vector l (Step 4). At each iteration of the while loop (Steps 6–16), a vector l is popped from the heap (Step 7) and is used to create a mapping in the result (Step 8). Afterward, the function *grow* generates a set of vectors L from l, which contains a set of vectors that can potentially be used to create a mapping in the result (Step 9); the vectors in L that can be a top-h mapping is then pushed to the heap (Steps 10–15). The above iteration ends when h mappings are returned, or no more mappings can be created (i.e., the heap is empty).

Given a vector l, the function *grow* returns L, which is a set of vectors that: 1) have not been enumerated before and 2) may be used to create a mapping whose score is among the top-h ones. In detail, it iterates forward from $l[|l|-1]$, the last component of l: if the current component, say, $l[i]$, can be extended to get a new valid vector (which implies that $l[i]+1$ is smaller than the total number of correspondences of the ith source element), then a new vector is created and added to L (Steps 3–6). The above iteration ends when it encounters a non-zero component, or all the components are visited.

Now, we use an example to illustrate how Algorithm 1 generates the top-h mappings from the remainder partition shown in Fig. 9a. First, the vector $l_1 : [0, 0]$ is pushed to the heap and then popped up to create the top-1 mapping. In the mapping created from l_1, since $l_1[0] = l_1[1] = 0$, the source element s_1 and s_2 choose their correspondences ranked the $(0+1)$th among all of their correspondences, i.e., the correspondence with the highest similarity, which are t_1 and t_3, respectively. Afterward, l_1 is used to produce two vectors $l_2 : [1, 0]$ and $l_3 : [0, 1]$, which are pushed to the heap. In the next iteration, l_2 is popped and the top-2 mapping is created

from l_2, where s_1 and s_2 matches t_2 and t_3, respectively; afterward, $l_4 : [1, 1]$ and $l_5 : [2, 0]$ is produced from l_2 and are pushed to the heap. Notice that when l_3 is popped, no new vector is produced: the last component of l_3 is non-zero, and so [1, 1] will not be computed; [0, 2] is invalid, since it means that s_2 chooses its third correspondence, which does not exist. Finally, six mappings can be created on this remainder partition. The above process is illustrated in Fig. 9b. Notice that no redundant mapping is created during the process.

Correctness of the algorithm

Let us use m_l to denote the mapping associated with a vector l. The function $score(m)$ returns the score of a mapping m. We first show the following lemma.

Lemma 1 *Let* $l_1 : [c_1, \ldots, c_{|S|}]$ *and* $l_2 : [c'_1, \ldots, c'_{|S|}]$ *be two different vectors, if for all* $1 \leq i \leq |S|$, $c_i \leq c'_i$, *then* $score(m_{l_1}) \geq score(m_{l_2})$.

Proof By definition of a mapping vector, all correspondences are sorted and named in descending order of their similarities. In addition, the score of mapping is a monotonic function of correspondence similarities, which implies that $score(m_{l_1}) \geq score(m_{l_2})$. □

We now prove that Algorithm 2 is correct, i.e., it generates all valid top-h mappings. Suppose that this is not true. This means that there exists some valid top-h mapping m' which cannot be yielded by Algorithm 2, i.e., there exists at least one correspondence in m' not generated by Algorithm 2. Let us suppose that this correspondence originates from the jth element of the source schema, and suppose that the first non-zero value of the vector of m' is c_k, where k is the index of the vector l'. We can then write the vector l' of m' as follows: $l' = [\underbrace{0 \cdots 0}_{(k-1)\ 0's}, c_k, \ldots, c_j, \ldots, c_{|S|}]$.

To generate l', the function $grow()$ can start by the vector $[\underbrace{0 \cdots 0}_{(k-1)\ 0s}, 1, \underbrace{0 \cdots 0}_{(|S|-k)\ 0s}]$. Since m' cannot be found, there exists another mapping, m'', which must be generated by the function $grow()$ before getting m'. Since m'' is not placed in the heap, m' cannot be generated. This implies that : (1) all values of the vector of m'' are less than or equal to all values of the vector of m' and (2) m'' is not in the top-h, and $score(m'') < c$.

By Lemma 1, we know that $score(m') < score(m'')$. Thus, m' must not be in the top-h mapping as well. Hence, m' is not a valid mapping, which contradicts our assumption. Algorithm 2 is thus correct.

Complexity analysis

Let U be a schema matching, the total sizes of all conflicting partitions of U be $|U_1|$, and the size of the remainder partition be $|U_2|$, where $|U_1| + |U_2| = |U|$. We first analyze the conflicting partitions. Based on the analysis of Algorithm 8, the complexity of generating the top-h mappings from all conflicting partitions is

$$C_F = O\left(\frac{h|U_1|^3}{l^2} + \left(\frac{|U_1|^2}{l} + |U_1|\right)\right)$$

Next, we analyze the remainder partition. Suppose the remainder partition contains $|S'|$ source elements and $|T'|$ target elements. According to the definition of a non-conflicting partition, the total number of correspondences of the remainder partition is at most $|T'|$. Therefore, sorting all the correspondences needs $O(|T'| \log |T'|)$. The while loop in Algorithm 2 iterates at most h times. Heap pop and push need $O(\log h)$. Function *grow* needs $O(|l| \times (|l| + \log h))$, where $|l|$ is $O(|S'|)$. Function *find_score* and *create_mapping* is linear with the size of source schema S. Therefore, the overall time complexity of Algorithm 2 is: $O(|T'| \log |T'| + h|S'|(|S'| + \log h))$, which is less than:

$$C_R = O(|U_2| \log |U_2| + h|U_2|(|U_2| + \log h))$$

Hence, the complexity of Algorithm 1 is: $O(C_F + C_R)$, which is polynomial with h and the schema matching size.

The space complexity of Algorithm 2 is dominated by *heap*, which is linear with h. Each entry in *heap* is linear with $|U_2|$. Therefore, the total space cost is $O(h|U_2|)$, which is polynomial with h and the remainder partition size.

Next, we discuss how to represent the probabilistic mappings that are generated with the partitioning algorithms. A simple way is to store them as a set of possible mappings with probabilities, where each possible mapping is a set of correspondences. However, the space cost of this method can be high, especially when there are a large number of possible mappings. In the next section, we introduce a novel data structure which can represent the possible mappings in a compact manner, and more importantly, can support efficient query evaluation over the possible mappings.

4 The block tree

The block tree is a compact representation of possible mappings. In this section, we discuss how to construct a block tree (Fig. 10). In particular, we explain the concepts of *blocks*, *c-blocks*, and the *block tree*, in Sect. 4.1. Sections 4.2 and 4.3 discuss how to efficiently create a block tree. We discuss how to combine mapping generation and block tree construction in Sect. 4.4. We use the five possible mappings, illustrated in Fig. 3, as a running example.

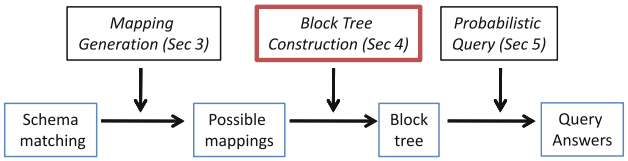


Fig. 10 Constructing a block tree

4.1 Blocks, c-blocks, and block tree

A *block* is a collection of correspondences shared by one or more mappings between schema S and schema T. Formally,

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

- A set $b.C$ of correspondences in U and
- A set $b.M$ of IDs of mappings, where $b.M \subseteq M$; for each $m_i \in b.M (1 \leq i \leq |M|)$, $b.C \subseteq m_i$.

Example Figure 4a shows three blocks, each of which contains a correspondence with element `ICN` in the target schema. For instance, $b_1.C = \{$`(BCN, ICN)`$\}$ and $b_1.M = \{m_1, m_2\}$. This means that `(BCN, ICN)` appears in m_1 and m_2 (Fig. 3). In Fig. 4b, $b_4.C = \{$`(BP, IP), (BCN, ICN)`$\}$, and $b_4.M = \{m_1, m_2\}$.

Ideally, if all blocks can be retrieved, 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 Sect. 5, it is not necessary to generate every block; 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*):

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 $\lfloor \tau \times |M| \rfloor$, where $\tau \in [|M|^{-1}, 1]$ is called the *confidence threshold*.

In this definition, when $\tau = |M|^{-1}$, the correspondences of the block are shared by at least one possible mapping; when $\tau = 1$, its correspondences are shared by all mappings.

Example In Fig. 4, $|M| = 5$. Let $\tau = 0.4$. Then, b_3 cannot be a c-block, because the number of mappings in b_3 is 1, which is less than $0.4 \times 5 = 2$. However, b_4 is a c-block (with $b_4.C = \{$`(BP, IP), (BCN, ICN)`$\}$, $b_4.M = \{m_1, m_2\}$, and $b_4.a =$ `IP`). This is 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) $|b_4.M| \geq 2$. Thus there are "enough" mappings that share $b_4.C$. 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, $b.a = e$.

Example Figure 4 illustrates two nodes of X (`ICN` and `IP`) for the matching in Fig. 1. 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 anchor $b_4.a$ equal to `IP`.

4.2 Constructing the block tree

To understand how the block tree can be generated, we first present some useful observations, which can be used to derive efficient algorithms for constructing the block tree.

Lemma 2 *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 $d_1, \ldots, d_f$ be child nodes of t. Then, for every d_i, there exists a c-block, b_{d_i}, with anchor d_i, such that*:

$$b_t.C = \{(s, t)\} \cup \left(\bigcup_i^f b_{d_i}.C \right) \tag{1}$$

$$b_t.M = M_t \cap \left(\bigcap_i^f b_{d_i}.M \right) \tag{2}$$

Proof We can express $b_t.C$ as $\{(s, t)\} \cup (\bigcup_i^f d_i.C)$, where $d_i.C$ is the set of correspondences with target nodes forming a complete subtree rooted at d_i $(i = 1, \ldots, f)$, the ith child node of t. Since b_t is a c-block, (s, t), as well as $d_i.C$, 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 $d_i.C$ is shared by $b_t.M$, a c-block b_{d_i} can be created with $b_{d_i}.a = d_i$, $b_{d_i}.M = b_t.M$ and $b_{d_i}.C = d_i.C$. Hence, Lemma 2 is correct. □

Essentially, Lemma 2 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:

Corollary 1 *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.*

Proof Let b be a c-block of t. By definition of a c-block, all correspondences $b.C$ originate from the nodes under the subtree of t. Also, the number of mappings that share $b.C$ must not be less than $\tau \times |M|$. Let d_i be any one of the child nodes of t. Then, we can construct a block b' with anchor d_i and with the subset of correspondences in $b.C$ that have target nodes rooted at d_i. The correspondences of b' must be shared by not less than $\tau \times |M|$ mappings. Hence, b' must also be a c-block. □

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

Algorithm 3 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 2 uses a global 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 Sect. 5. 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 preorder 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 postorder 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. 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.

Algorithm 4 describes the function $init\_block$, which conceptually assigns all mappings in M to a set of groups (which are blocks), such that each group contains a distinct source node s which matches t. Step 1 creates a block at t for mapping m_1. Then, for every mapping m_i (where $i = 2, \ldots, |M|$), it calls $find\_node$, which examines whether m_i contains the correspondence specified in $b \cdot C$ (Step 4). If this is true, we insert m_i to b (Step 6). Otherwise, a new block is created for m_i (Step 8). Steps 13–20 filter the blocks that contain the number of shared mappings less than τ and update the total number of blocks created so far, *count*. A parameter *MAX_B* is used to control the maximal number of c-blocks generated. The number of newly created blocks is returned in Step 21.

Algorithm 3 *construct_block_tree*

Input: schema T, mapping set M, confidence threshold τ
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: number of blocks created
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** d_i in t's child nodes **do**
11: **if** $construct\_c\_block(d_i) = 0$ **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**

Algorithm 4 Function $init\_block(node\ t)$

Return: number of blocks created for t
1: $create\_block(t, m_1)$
2: **for** $i = 2, \ldots, |M|$ **do**
3: **for all** block b at t **do**
4: $s \leftarrow find\_node(b, m_i)$
5: **if** s is found **then**
6: $insert(b, m_i)$
7: **else**
8: $create\_block(t, m_i)$
9: **end if**
10: **end for**
11: **end for**
12: $count\_new \leftarrow 0$
13: **for all** block b at t **do**
14: **if** $|b \cdot M| \geq \tau \times |M|$ **and** $count < MAX\_B$ **then**
15: $count\_new \leftarrow count\_new + 1$
16: $count \leftarrow count + 1$
17: **else**
18: delete b
19: **end if**
20: **end for**
21: return $count\_new$

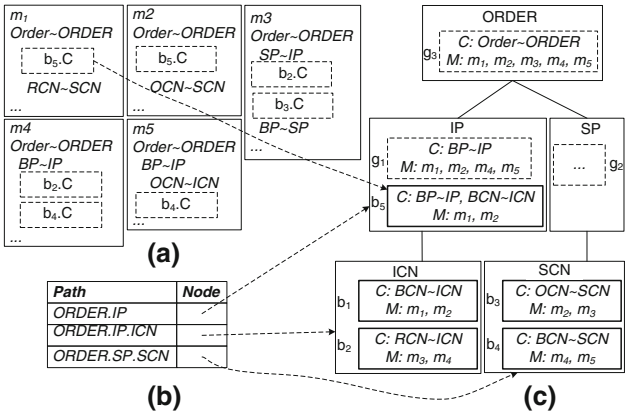


Fig. 11 Examples of **a** mappings, **b** hash table, and **c** block tree. The set of correspondences of a mapping that can be found in a c-block is replaced by a pointer to that c-block. The hash table is used to support query evaluation

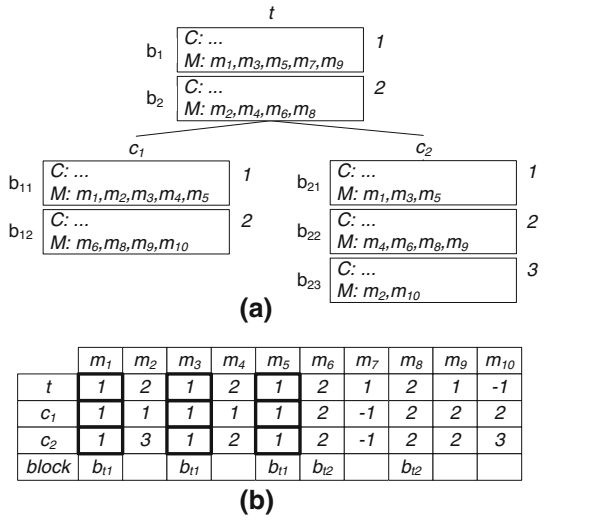


Fig. 12 Examples of **a** nodes of the block tree, **b** signatures of the mappings. A mapping signature facilitates the construction of a non-leaf node in a block tree

CASE 2: t is a non-leaf node. In Algorithm 3, Step 9 initializes $mark$ to TRUE. Then, Steps 10–14 perform *construct_c_block* for each child node of t and see whether any one of them returns zero. If this happens, t cannot have any c-block (Corollary 1). 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 entry for t and return the number of blocks generated.

Figure 11 illustrates the block tree and the supporting data structure for the sample mappings in Fig. 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 indicate 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.

4.3 Generating a non-leaf node in a block tree

We now discuss the important function gen_non_leaf, which generates c-block(s) for a non-leaf node. In fact, a simple way of implementing it has been proposed in Cheng et al. [7]. To find out the c-blocks of a non-leaf node t, [7] simply considers all combinations of c-blocks in the child nodes of t. More specifically, it enumerates all combinations of t's initial blocks (returned by the function $init\_block$), as well as the c-blocks of each child node of t. By using Lemma 2, each combination of blocks becomes a candidate c-block of t, which can then be tested to see if they are real c-blocks or not. Since the number of candidate c-blocks generated is exponential to the number of possible mappings, a parameter *MAX_F* is used in Cheng et al. [7] to restrict the number of failed cases (i.e., a candidate tested not a real c-block) to be no more than *MAX_F*.

The problem with the procedure discussed above is that it is hard to determine *MAX_F*. We found from our experimental results that if *MAX_F* is too large, gen_non_leaf involves a large number of candidate c-block tests and increases the running time significantly. On the other hand, if *MAX_F* is too small, some valid c-blocks may not have a chance to be generated. Hence, the query effectiveness of the block tree can be reduced. Our new algorithm presented here avoids using *MAX_F* and still finds out *all* c-blocks for a non-leaf node, in polynomial time.

Our observation is that finding all c-blocks of a non-leaf node t does not necessitate the enumeration of all c-blocks in its child nodes. Notice that the number of the c-blocks of t cannot be larger than $|M|$; in addition, the only case for the number of c-blocks of t equals to $|M|$ is when: (1) each mapping in M contains a distinct set of correspondences for the subtree rooted at t, and (2) $\tau = |M|^{-1}$. We create a *signature* for each mapping m, which is a vector of integers with length $(t.fanout + 1)$, where the *fanout* of t equals to the number of t's children. Each initial block of t is assigned a distinct integer ID (starting from 1); the IDs of the c-blocks of each of t's children are assigned similarly, as shown in Fig. 12a. The first component of the mapping m_i's signature stores the ID of the initial block of t containing m_i; the ith $(1 < i \leq t.fanout + 1)$ component of the signature stores the ID of the c-block of the ith child of t containing m_i. Notice that the signatures of all the mappings can be found by scanning the blocks of t and t's children once. Mappings with identical signatures may yield a c-block, since they must contain the same set of correspondences for the nodes in the subtree of t.

We now illustrate the algorithm with an example. In Fig. 12a, the non-leaf node t has two children, namely c_1 and c_2. In addition, t has two initial blocks, namely b_1 and b_2, while c_1 and c_2 have two and three c-blocks, respectively. The ID of each block is illustrated on the right of the block. Figure 12b shows the signatures of the mappings after gen_non_leaf is completed. For example, mappings m_1, m_3 and m_5 share the signature [1, 1, 1] (shown in bold lines), while m_6 and m_8 share the signature [2, 2, 2]. If $\tau = 0.2$ and $|M| = 10$, two blocks will be created for t: b_{t1}, which contains m_1, m_3, and m_5; and b_{t2}, which contains m_6 and m_8.

Algorithm 5 shows the details of gen_non_leaf. First, Step 1 executes $init\_block$ on t. Conceptually, it treats t as a "leaf node" and attempts to construct potential c-blocks for t based on its correspondences. If none is found, no c-blocks can be created at t, and zero is returned (Step 2). Otherwise, it detaches the block list created by $init\_block$ and moves it to a temporary list, $list_t$ (Step 4). In Step 6, it initiates a new signature for each mapping m in M, which is a vector $[-1, \ldots, -1]$ with length $(t.fanout + 1)$. Then, in Steps 7–11, it updates the signatures with node t and its children. Procedure sig_update modifies the posth component of the signatures according to the blocks associated with a node n: if the mapping m_j is found in the ith block of n, then the posth component of $sig(m_j)$, the signature of m_j, is set to be i.

Finally, the algorithm finds out all subsets of mappings M_i which contains at least $\tau \times |M|$ mappings with identical signatures (Steps 14–24). In order to find out all such subsets, M is sorted first according to the signatures of mappings, and then scanned once to find out all such subsets of mappings. Notice that those mappings with signatures containing "-1" would not be considered for generating c-blocks (Step 12); they must not be included in any c-block. A c-block for t is created from each of these subsets based on Lemma 2: the mapping IDs consist of each mapping's ID in the subset, and the correspondence set consists of the union of the correspondences for t and t's children contained arbitrarily in the subset. A parameter MAX_B is used to control the maximal number of c-blocks generated (Step 21): when the algorithm finds MAX_B c-blocks, it stops finding new c-blocks.

Algorithm 5 Function $gen\_non\_leaf(node\ t)$

Return: number of blocks created for t
1: **if** $init\_block(t) = 0$ **then**
2: return 0
3: **end if**
4: Let $list_t \leftarrow$ detached block list of t
5: $count\_new \leftarrow 0$ // number of new c-blocks
6: create a new signature $sig(m_i)$, for each $m_i \in M$
7: $pos \leftarrow 1$
8: $sig\_update(t, pos)$
9: **for all** $c \in t.children$ **do**
10: $sig\_update(c, pos)$
11: **end for**
12: discard any $m_i \in M$, if m_i's signature contains -1
13: sort M in ascending order of the signature's components
14: **for all** $M_i \subseteq M$ such that M_i contains maximal mappings with identical signatures **do**
15: **if** $|M_i| \geq \tau \times |M|$ **then**
16: $b \leftarrow create\_block(M_i)$
17: $attach\_to\_node(b, t)$
18: $count\_new \leftarrow count\_new + 1$
19: $count \leftarrow count + 1$
20: **end if**
21: **if** $count \geq MAX\_B$ **then**
22: break
23: **end if**
24: **end for**
25: **if** $count\_new > 0$ **then**
26: $insert\_hash\_entry(H, t)$
27: **end if**
28: discard $list_t$, and $sig[m_i]$ for all $m_i \in M$
29: return $count\_new$

procedure $sig\_update(node\ n, int\ pos)$
1: **for all** b_i of n's block **do**
2: **for all** $m_j \in b_i.M$ **do**
3: $sig(m_j)[pos] \leftarrow i$
4: **end for**
5: **end for**
6: $pos \leftarrow pos + 1$

Space costs

Each block has at most $|T|$ correspondences and $|M|$ mappings, and a size of at most $|T| + |M|$ units. Notice that the maximum number of c-blocks generated is MAX_B. On the other hand, each node in T can have at most τ^{-1} c-blocks. Since a block tree has $|T|$ nodes, the total number of c-blocks in the block tree cannot be larger than $\tau^{-1}|T|$. Therefore, the total space occupied by c-blocks is min $(MAX\_B, \tau^{-1}|T| \times (|T| + |M|))$. The hash table size is $O(|T|)$. The total size used for storing all the signatures is $(f + 1)|M|$, where f is the maximum fanout.

Time costs

We first analyze function *init_block*, which generates c-blocks at a leaf node. A leaf node t can have $O(|S|)$ blocks, and each block can contain $O(|M|)$ mappings. In Step 3, *find_node*, which looks for a given mapping in a block with binary search, is $O(\log |M|)$. Therefore, *init_block* requires a time complexity of

$$C_L = O(|M||S| \log |M|) \tag{3}$$

Next, we analyze function $gen\_non\_leaf\_sig$, which generates c-blocks at a non-leaf node. Signatures initialization needs $O(f|M|)$ to clear the existing components in the current signature of each mapping. Updating of signatures needs $O(f|M|)$, since the total number of mappings contained in the blocks of each node (either t or t's child) is at most $|M|$. Sorting the mappings and finding all subsets of mappings with identical signatures needs $O(f|M| \log |M|)$,

since there are total $|M|$ vectors, and the length of each vector is $f + 1$. The cost of inserting all new blocks into the hash table (Steps 25–27) is $O(|T|)$, since the hash table has size $O(|T|)$. Therefore, producing c-blocks at a non-leaf node, using Algorithm 5, has a cost of

$$C_N = O(f|M| \log |M| + |T|) \quad (4)$$

Function *construct_c_block* uses a postorder traversal, where each node is visited once. Let $h = \lfloor \log_f |T| \rfloor$ be the height of the block tree, then the total number of leaf nodes is $O(f^h)$, and the total number of non-leaf nodes is $O(|T| - f^h)$. Therefore, the total cost of *construct_c_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 $O(|M||T|)$. Hence, the space and the construction time complexities of the block tree are polynomial with the number of mappings and the size of the schema matching.

4.4 An integrated approach of block tree construction

We have so far assumed that all the h mappings are created by a mapping generation algorithm, before they are used to construct a block tree. If the number of mappings is large, a significant amount of space can be consumed. We now describe an "integrated approach", which generates the block tree during the mapping derivation process. This method consists of four steps:

Step 1. Divide the bipartite corresponding to a schema matching (Sect. 3.1) into a *parent bipartite* and a set of *child bipartites*. For any two bipartites, their correspondences do not share the same source or target element. The partitioning has to "preserve" the structure of the target schema. For example, in Fig. 13, a target element p, which is also a leaf node of the parent bipartite, is connected to two child bipartites that contain the child subtrees of p. This process can be done by using a postorder traversal of the target schema.

Step 2. For each child bipartite, execute the mapping generation and the block tree construction algorithms discussed before. We then obtain a set of mappings and a block tree for every child bipartite. Notice that the set of mappings associated with each child bipartite is independent of each other, because their correspondences do not share the same source or target elements. The block tree of every child bipartite is also independent of each other.

Step 3. The mappings of the child bipartites that belong to the same parent target node p in the parent bipartite are merged to find the top-h mapping in the subtree under p, using the merging algorithm discussed in Sect. 3.2. Then, the c-blocks attached to the root of the block trees of the child bipartites are used to generate the c-blocks for p. For each mapping found, the references of their correspondences to the new c-blocks are updated accordingly.

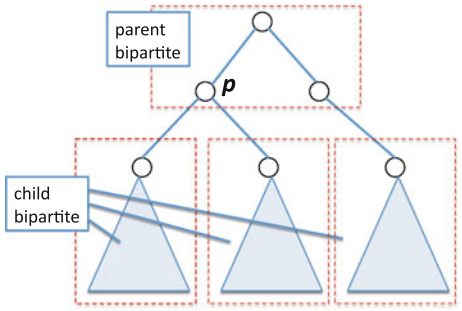


Fig. 13 Partitioning of the target schema in the integrated approach

Step 4. Repeat Step 3 until the root of the target schema is reached. If no more c-blocks are generated at a node, we just merge the mappings from the child subtrees, in order to obtain the required mappings. Finally, the hash table used by the block tree is created.

The above steps generate a "partial" block tree while discovering mappings, in a bottom-up approach. It does not require all mappings to be known before the block tree is constructed. If a schema matching changes, this approach may also allow the block tree to be updated. Suppose that a correspondence is inserted between a source and a target element. If this correspondence happens to be inserted to the same child bipartite e, then in Steps 1 and 2 above, we only have to generate the mappings and the block tree for e. The costs of deriving the block tree for the new schema matching can thus be reduced. Next, we study how the block tree supports query evaluation.

5 Evaluating twig queries over block trees

We now study the Probabilistic Twig Query (PTQ), which provides query answers over possible mappings. We describe its definition in Sect. 5.1. As shown in Fig. 14, a PTQ can be evaluated on a block tree. Section 5.2 explains how this can be done efficiently. We discuss the top-k PTQ in Sect. 5.3.

5.1 The probabilistic twig query

Let us briefly review a twig query. A *twig pattern*, q, is a tree, where each node has a label (e.g., ICN) and an optional predicate (e.g., ICN = "$Alice$"). Each node has

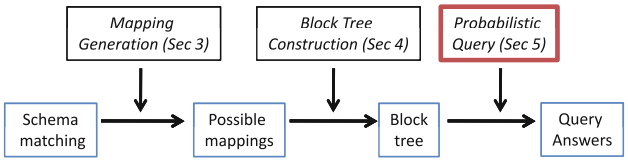


Fig. 14 Evaluating probabilistic twig queries

an edge labeled either '/' (parent–child edge) or '//' (ancestor-descendant edge). For example, in Fig. 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 $\{n_1, \ldots, n_l\}$ from d, such that for each node $n_i (1 \leq i \leq l)$, the label and the predicate (if any) of the ith 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 $\{n_1, \ldots, n_l\}$.[4]

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

Now, let a possible mapping $m_i \in M$ is true, having probability p_i, where $\sum_{1 \leq i \leq |M|} p_i = 1$. 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 $R = \{(R_i, pr(R_i)\}(1 \leq i \leq |M|)$, 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.

Basic solution

Algorithm 6 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'.

Algorithm 6 (*query_basic*) Basic Query Evaluation

Input: PTQ q_T, mapping set M, document d_S
Output: Answer R to PTQ 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: $\quad q_S \leftarrow rewrite(q_T, m_i)$
4: $\quad R_i \leftarrow match(ds, q_S)$
5: $\quad R \leftarrow R \cup \{(R_i, p_i)\}$
6: **end for**
7: return R

In *twig_query*, Step 1 initializes the query result R. For each mapping $m_i \in M'$, we translate q_T, 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.

Complexity

Function $filter\_mappings$ is $O(|M||q||S|)$, as for each mapping $m_i \in M$ and for each node n in q, it checks whether m_i contains a correspondence for n. Function $rewrite$ is $O(|q|h)$, where h is the maximal depth of a node in q; the function $match$ is $O(|q|^2|d_s|)$ [18]. The total cost is $O(|M||q|(|S| + h + |q||d_s|))$.

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.

5.2 Evaluating PTQ with the block tree

The process of supporting PTQ execution with a block tree is illustrated in Fig. 15. Let the root node of a given query q_T be r. We first traverse the block tree X to get the node t that matches r. The c-blocks attached to t can then be used to improve the performance of evaluating q_T. Intuitively, we only execute q_T *once* for all the mappings indicated in each c-block of t, since these mappings have the same correspondences rooted at r. If t does not contain any c-block, we decompose q into three subqueries:

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

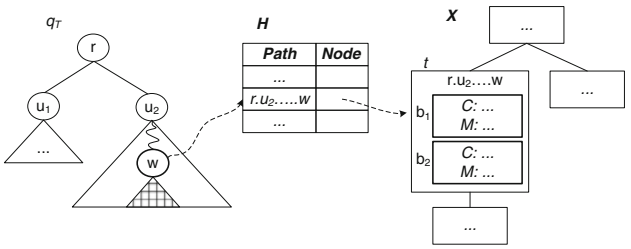


Fig. 15 Query evaluation using a block tree: given a subtree rooted at w, its respective position in the node t in the block tree X is located. A hash table can be used to facilitate this process. The c-blocks associated with t are then used to evaluate the twig query rooted at w

- q_0, with a single node r;
- q_1, with the subtree of q_T rooted at u_1; and
- q_2, with the subtree of q_T rooted at u_2.

Query q_0 is simple and is evaluated directly with the existing twig query method; q_1 and q_2 are computed recursively. For example, q_2 is decomposed into subqueries until a node, w, contains a c-block. Then, the subquery issued at w use the c-blocks stored in t to facilitate query evaluation. The answers to the subqueries are then joined to form the final query answer. To speed up the retrieval of t, we can use the hash table H obtained during block tree construction. Recall that each entry in H contains a path in the target schema and a pointer to some block tree node corresponding to that path. If r is found in H, we use H to obtain t. In Fig. 15, for instance, H contains the block tree node t that corresponds to query node w.

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 7). First, q_T's root is searched from block tree X (Step 1). As discussed before, the hash table associated with the block tree can be used to speed up this step. If a node t is found, then $query\_subtree$ is invoked in Step 3 to answer q_T. (We explain the details of this function later.) Otherwise, there are two cases:

(1) q_T contains a single node: we answer q_T by calling *twig_query* (Steps 5–6);
(2) q_T has one or more children: we call $split\_query(q_T)$ to decompose q_T into subquery q_0, which contains q_T's root node only, and a set of subqueries $q_1, \ldots, q_f$, each of which is rooted at q_T's ith ($1 \leq i \leq f$) child, where f is the fanout of q_T's tree (Step 9). The subquery q_0 is evaluated using *twig_query*, while other subqueries are evaluated by recursively calling *twig_query_tree*, at the subtree of X rooted at t (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 [2] (Step 16). The combined result is included in R (Step 18), which is returned in Step 20.

Algorithm 7 (*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: $t = find\_node(q_T.root, X)$
2: **if** t has a c-block **then**
3: return $query\_subtree(q_T, t, M', d_S)$
4: **else**
5: **if** q_T is a leaf **then**
6: return $twig\_query(q_T, M', d_s)$
7: **else**
8: $R \leftarrow \emptyset$
9: $(q_0, q_1, \ldots, q_f) \leftarrow split\_query(q_T)$ // q_0 is the root of q_T, and $q_1, \ldots, q_f$ are q_T's children
10: $R(q_0) \leftarrow twig\_query(q_0, M', d_S)$
11: **for all** $j \in [1, f]$ **do**
12: $R(q_j) \leftarrow twig\_query\_tree(q_j, M', d_S, t)$
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 $query\_subtree(PTQ\ q_T, node\ t, mappings\ M', document\ d_S)$
Return: Answer to q_t

1: Let $M_s \leftarrow \emptyset$ // all mappings involved at t
2: Let $Y \leftarrow \emptyset$ // 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 $Y \cup Z$

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. For example, if we use a small value of *MAX_B* during block tree construction, then fewer c-blocks can be generated. This makes the size of M_s small, so that Step 10 involves visiting a larger number of mappings ($|M' - M_s|$). However, the query correctness will not be affected by using fewer c-blocks. This is because in the end of the function $query\_subtree$, it considers all the possible mappings that are not contained in the block tree and uses them to return a set of query answers Z. Therefore, for the worst case where there is no c-block, Z would contain all the query answers.

Compared with *query_basic*, which treats each mapping independently, our method 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_T. The most expensive function in the decomposition/merge process, *stack_join*, combines the result for each edge in q_T in $O(|q_T| \cdot |d_s|)$ times [2]. If q_T has E edges, the worst-case cost of decomposition-and-join is $O(|E| \cdot |q_T| \cdot |d_s|)$. Our experiments show that this is rare, and the additional overhead does not override the benefit of using the block tree for query evaluation.

5.3 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 probabilities of query answers, 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):

- Sort the mapping set M' in ascending order of the probability of each mapping in M'.
- 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.

6 Experimental results

We now present our results. Section 6.1 discusses the experiment setup. In Sect. 6.2, we discuss the experiment results on the data sets commonly used in the geospatial and e-commerce applications.

6.1 Setup

We have used two data sets, namely, Geospatial data set and E-Commerce data set, in our experiment.

Geospatial data sets

We used a geospatial data set called CityGML,[5] which is an implementation of the application schema for the Geography Markup Language, an international standard for spatial data exchange issued by the Open Geospatial Consortium (OGC).[6] CityGML is a popular model used for representing urban objects, and multiple versions have been released. We have downloaded several XML schemas and documents from the Resources section[7] of the website (the subsections of Schema download and CityGML data sets). In the 0.4.0 version, it uses a large schema to represent all kinds of urban objects ($building$, $waterbody$, $bridge$, etc), while in the 1.0.0 version, it uses separate schemas to represent each category of objects. We downloaded the version 0.4.0 schema, as well as the schemas for the *building* and *waterbody* objects in the 1.0.0 version of the schemas. The part related to *building* and *waterbody* of the version 0.4.0 schema is extracted. Then, we used COMA++[8] to find the matchings between two versions of the schemas. We then obtain two matching results, namely, $MS1$ and $MS2$ (Table 2, where CG.4 is the version 0.4.0 version of the schema, and CG1B and CG1W refer to the $building$ and $waterbody$ schema in the 1.0.0 version of the schema respectively).

In Table 2, each matching, named "ID", has a source schema S and a target schema T, which contain $|S|$ and $|T|$ elements respectively. We have used the default matching option of COMA++, namely, *context*, to perform the schema

[5] CityGML: http://www.citygml.org.

[6] OGC: http://www.opengeospatial.org.

[7] CityGML Resources: http://www.citygml.org/index.php?id=1522.

[8] COMA: http://dbs.uni-leipzig.de/Research/coma.

Table 2 Schema matchings

ID	S	$\|S\|$	S.depth	S.width	T	$\|T\|$	T.depth	T.width	Cap.	*o-ratio*
Geospatial datasets										
$MS1$	CG.4	86	26	30	CG1B	378	26	30	323	0.92
$MS2$	CG.4	86	26	30	CG1W	84	26	30	119	0.85
E-commerce datasets										
$MC1$	Excel	48	5	12	Paragon	69	6	12	47	0.63
$MC2$	Noris	66	4	10	Paragon	69	6	12	41	0.64
$MC3$	OT	247	9	16	Apertum	166	7	7	77	0.87
$MC4$	XCBL	1076	10	25	Apertum	166	7	7	226	0.84
$MC5$	XCBL	1076	10	25	CIDX	39	8	12	127	0.82
$MC6$	XCBL	1076	10	25	OT	247	9	16	619	0.91
$MC7$	OT	247	9	16	XCBL	1076	10	25	619	0.91

matching tasks. Using this option, the matching method returns context-dependent correspondences. The capacity (*Cap.*) is the number of element correspondences of the matching. The *o-ratio*, which we will detail later, measures the degree of overlap among the set of possible mappings used in the matching.

E-commerce data sets

We also used a variety of real XML schemas commonly used in e-commerce. These include the OpenTrans (OT) and XCBL schemas, as well as schemas provided by COMA++. Based on these schemas, we generate seven more matching results, namely, $MC1, \ldots, MC7$ (Table 2).

Source documents

We used a set of geospatial XML files as the source documents. Those files are downloaded from the website of CityGML,[9] which contain the geospatial information of urban objects in different cities. The details of the source documents $DS1$-$DS6$ are shown in Table 3. The largest source document, $DS6$, whose size is about 36MB, contains 301,509 elements. We also examined queries on the data with the e-commerce schemas. An XML document `Order.xml`, chosen from the XCBL sample file *autogen_full* containing 3473 nodes, is used as the source document for the e-commerce data set.

Target queries

We tested four queries, namely, $QS1, \ldots, QS4$, on the matching geospatial data set $MS1$, as shown in Table 4. We also tested ten queries, namely, $QC1, \ldots, QC10$ in Table 4, on the e-commerce data set $MC4$. These queries cover different portions of the target schema and have a variety of sizes.

Table 3 Geospatial XML documents

ID	File name	Size (MB)
$DS1$	Koenigswinter_Drachenfelsstrasse_v0.4.0	3
$DS2$	Berlin_Pariser_Platz_v0.4.0	4
$DS3$	CityGML_British_Ordnance_Survey_v0.4.0	4
$DS4$	waldbruecke_v0.4.0	7
$DS5$	080305SIG3D_Building_Levkreuz	28
$DS6$	Berlin_Alexanderplatz_v0.4.0	36

We implemented the block tree, with $|M| = 500$, $\tau = 0.2$, and $MAX\_B = 500$ by default. We also implemented the Pascoal's algorithm [17], an advanced version of Murty's algorithm [16], in order to efficiently generate top-h mappings. Our experiments are run on a PC with Intel Core Duo 2.66GHz CPU and 2 Gigabyte RAM. The algorithms are implemented in C++. Each data point is an average of 50 runs.

In this paper, we focus on the geospatial data set. For the e-commerce data set, the observation is similar, and we only present their representative results. A more comprehensive discussion of the e-commerce data set experiments can be found in Cheng et al. [7].

6.2 Results

1. Top-h mapping generation. We first compare the performance of the three top-h mapping generation algorithms mentioned in Sect. 3: (1) the *murty* algorithm (Pascoal's improved method), (2) Algorithm 8 in Appendix A (called *partition* here), and (3) Algorithm 1 (called *heap* here). Figure 16a shows the time needed (T_g) on each match-

[9] CityGML data set: http://www.citygml.org/index.php?id=1539.

Table 4 Queries used in the experiment

ID	Matching	PTQ	Meaning
Geospatial datasets			
$QS1$	MS1	//Building//lod1Solid//posList	Return the coordinates of a building's vertices (1st level-of-detail)
$QS2$	MS1	//Building//lod2Solid//posList	Return the coordinates of a building's vertices (2nd level-of-detail)
$QS3$	MS1	root/Building/lod3Solid//posList	Return the coordinates of a building's vertices (3rd level-of-detail)
$QS4$	MS1	root[./Building/lod3Solid//LinearRing]	Return the shapes of all buildings (3rd level-of-detail, in the form of a linear ring)
E-commerce datasets			
$QC1$	MC4	Order/DeliverTo/Address[./City][./Country]/Street	Return the street name of the delivery address for a given order
$QC2$	MC4	Order/DeliverTo/Contact/EMail	Return the email address of the recipient of a given order
$QC3$	MC4	Order/DeliverTo[./Address/City]/Contact/EMail	Return the email address of the recipient with a given city name
$QC4$	MC4	Order/POLine[./LineNo]//UnitPrice	Return the unit prices for all items in a given order
$QC5$	MC4	Order/POLine[./LineNo][.//UnitPrice]/Quantity	Return the unit prices and quantities of all items in a given order
$QC6$	MC4	Order/POLine[./BuyerPartID][./LineNO] [//UnitPrice]/Quantity	Return the unit prices and quantities of items with have buyer IDs and line numbers
$QC7$	MC4	Order[./DeliverTo//Street]/POLine[.//BuyerPartID] [.//UnitPrice]/Quantity	Return the quantities of items with specified buyer IDs, unit prices, and street names
$QC8$	MC4	Order[./DeliverTo[.//EMail]//Street]/POLine [.//UnitPrice]/Quantity	Return the quantities of items with specified unit prices, for a recipient with a specified email address and street
$QC9$	MC4	Order[./Buyer/Contact]/POLine[.//BuyerPartID] /Quantity	Return the quantities of items of an order with a specified buyer ID
$QC10$	MC4	Order[./Buyer/Contact][./DeliverTo//City] //BuyerPartID	Return the buyer ID of an order, with the city address of the recipient specified

ing in Table 2. Notice that the y-axis in both Fig. 16a and b is logarithmic. We observe that *partition* consistently outperforms *murty* (96% on average). This is because the bipartite of the schema matching is sparse, and the number of partitions is large (it ranges from 22 for $MC2$ to 966 for $MC4$). We also see that *heap* addresses significant improvement over *partition* (69% on average, maximal 93% on $MC4$). The reason behind this improvement is that the size of the remainder partition is generally large on all the schema matchings we tested—the average size of the remainder partition is 77.57% of the target schema size. Hence, developing a bipartite matching algorithm tailored for the remainder partition is justified.

Next, Fig. 16b compares the scalability of *murty*, *partition*, and *heap* in terms of h, on the schema matching $MS1$. We measure the time used for generating 100–500 possible mappings with different methods. We observe that although these three methods are all polynomial with the number of mappings, *partition* outperforms *murty* 98% on average. Moreover, the average running time of *heap* is 35% faster than *partition*. Our approaches can therefore improve T_g significantly.

2. Mapping overlap. Now, 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 $\frac{|m_i \cap m_j|}{|m_i \cup m_j|}$, which is the fraction of the number of shared correspondences between m_i and m_j over the number of the total distinct correspondences of them. We also define the *o-ratio* of M as the average of the *o-ratio* between all pairs of mappings in M. Table 2 shows

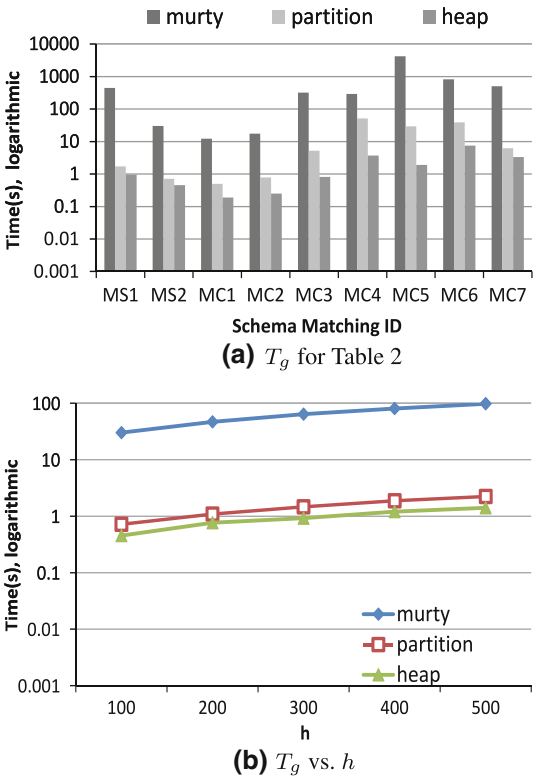


Fig. 16 Top-h mapping generation performance

Table 5 No. of c-blocks and block tree size (in KB)

Matching	# c-blocks	block tree size	B_M	B
$MS1$	376	116	4,150,613	128,669
$MS2$	91	20	128,723	24,586
$MC1$	78	16	4,080,857	199,962
$MC2$	80	28	594,745	30,332
$MC3$	180	39	127,822	45,121
$MC4$	174	55	78,350	66,911
$MC5$	45	10	19,051	18,365
$MC6$	232	75	148,889	112,411
$MC7$	500	161	13,466,000	215,456

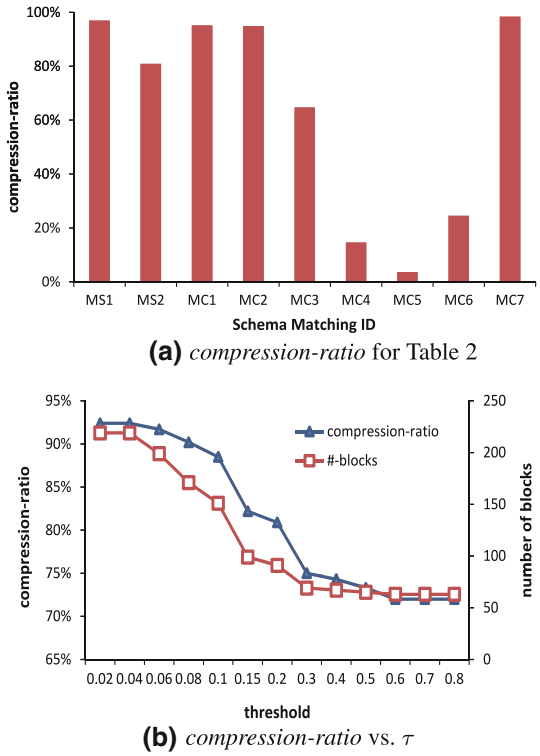


Fig. 17 Space efficiency of the block tree

that the *o-ratio* values for the mapping sets are between 0.53 and 0.92. Hence, there exists a high overlap among the mappings. Next, we study how well the block tree exploits this property.

3. Space 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). Let B_M be the total number of bytes required to store the original mappings of M. Table 5 shows the number of c-blocks generated and the block tree size information, for each matching in Table 2. We define the compression ratio as $max(1 - \frac{B}{B_M}, 0)$. This metric captures the amount of space saved by representing M with a block tree, which can be affected by the number of c-blocks generated.

Figure 17a shows the compression ratio on all the schema matchings under the default setting. We observe that the compression ratios for most schema matchings are above 50%. For example, for MS1, the compression ratio is 97%. This is because a large number of c-blocks (376) can be generated for this matching (Table 5). For $MC5$, the compression ratio is the lowest, since the number of c-blocks generated is the fewest (45) among all matchings that we have considered. Although the o-ratio for $MC5$ is high, the overlap of mappings happens mainly at the higher levels of the target schema. However, since the degree of overlap at the lower levels (e.g., leaf nodes) of the target schema is low, few c-blocks are generated. In $MC5$, we found forty c-blocks at the leaf level and five c-blocks at the second-last non-leaf nodes. This results in a low compression ratio.

To study how the number of c-blocks affects the compression ratio, we examine $MS2$, by decreasing the number of blocks generated through increasing the threshold. The number of blocks generated and the compression ratio is shown in Fig. 17b. We observe that when the number of blocks decreases, the compression ratio drops accordingly.

4. Effectiveness of c-blocks. From Fig. 17b, we observe that the number of c-blocks drop much slower after around $\tau = 0.1$. This means that the number of mappings contained in many c-blocks is larger than $\tau \times |M|$. Next, Fig. 18

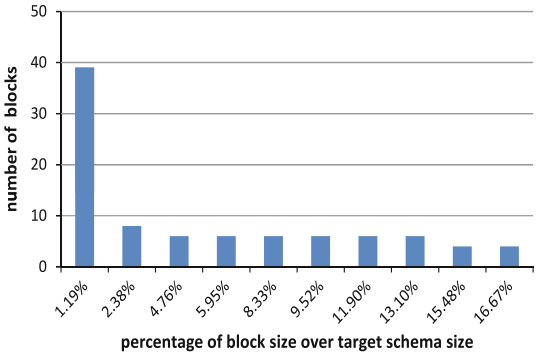


Fig. 18 Effectiveness of c-blocks: distribution of different sizes of c-blocks

shows the distribution of c-block sizes, in terms of the number of correspondences contained in the c-blocks in $MS2$. 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 (57%) of c-blocks whose sizes are larger than one. The largest c-block contains 14 correspondences. This covers about 17% of all target schema elements and is shared by more than $\tau = 20\%$ of all possible mappings. The average size of all c-blocks is 4.8. Hence, c-blocks can effectively capture the high overlap among mappings.

5. Block tree construction. Next, we study the block tree creation algorithm presented in Cheng et al. [7]. For convenience, we call this algorithm *combination*. As discussed in Sect. 4.3, *combination* uses parameter *MAX_F* to control the maximal number of false trials. By default, *MAX_F* is 500. Let us analyze its performance on the schema matching $MS2$.

Figure 19a shows the block tree generation time (T_c) and the number of c-blocks generated, by varying the value of *MAX_F*. We notice that the number of c-blocks generated is sensitive to *MAX_F*: in order to improve the query performance by generating more c-blocks, we need to increase *MAX_F*; however, the running time also increases accordingly. Figure 19b shows the maximal number of valid c-blocks that can be generated and the minimal number of false trials needed for generating those c-blocks, for different τ. We observe that in order to generate more c-blocks by decreasing τ, we need to increase the minimal *MAX_F* accordingly. As we can see, the minimum value of *MAX_F* has a large fluctuation with respect to τ. Therefore, it is hard to set a proper value for *MAX_F*.

Next, we compare *combination* with the new block tree construction algorithm proposed in Sect. 4.3. Let us call this algorithm *signature*. Figure 19c shows the block tree construction time of *combination* and *signature*, under different *MAX_F* values. Observe that *signature* consistently outperforms *combination* and is *not* sensitive to *MAX_F*.

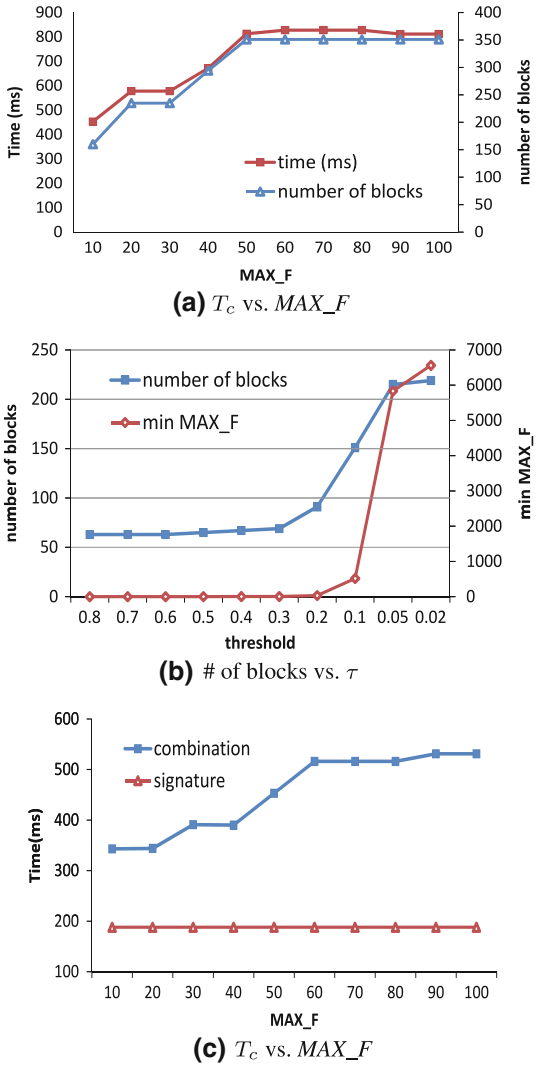


Fig. 19 Effect of MAX_F on block tree generation time

Figure 20a shows the block tree generation time of *combination* and *signature* on all the schema matchings shown in Table 2. We observe that *signature* outperforms *combination* by 33.27% on average. On $MS1$, the improvement is the largest (73.04%), since a large combination of child c-blocks needs to be enumerated in *combination*. On $MC7$, the improvement is small, since the target schema is large, and most target elements have identical correspondences in the possible mappings. Therefore, few trials are needed to generate all the c-blocks. To conclude, *signature* generates exact number of c-blocks faster, without the need of specifying *MAX_F*. Moreover, *signature* is easier to use than *combination*, since it involves fewer parameters.

Figure 20b shows the block tree construction time for different MAX_B values for *combination* and *signature*. We

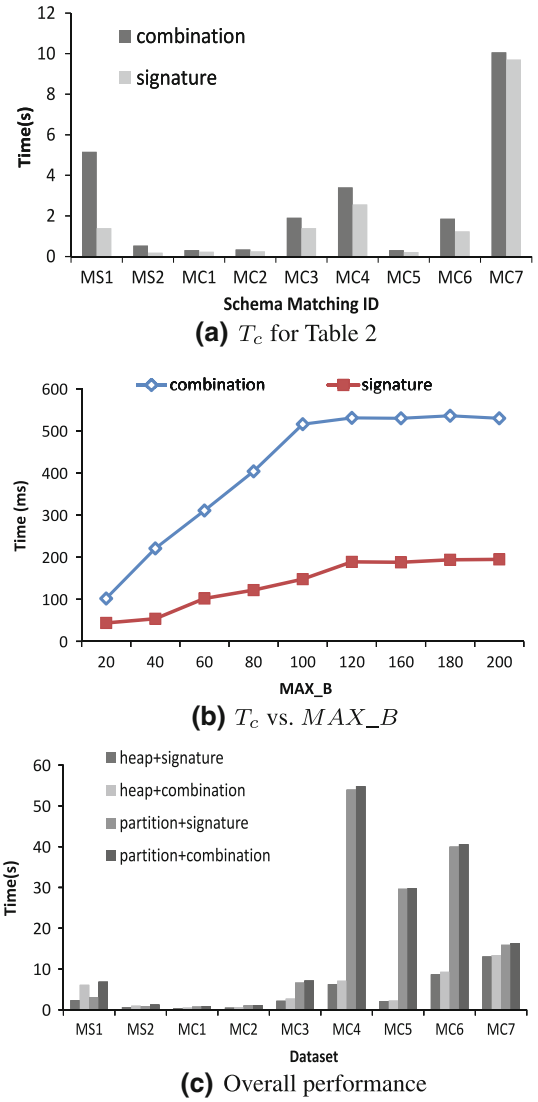


Fig. 20 Block tree construction methods: *combination* versus *signature*

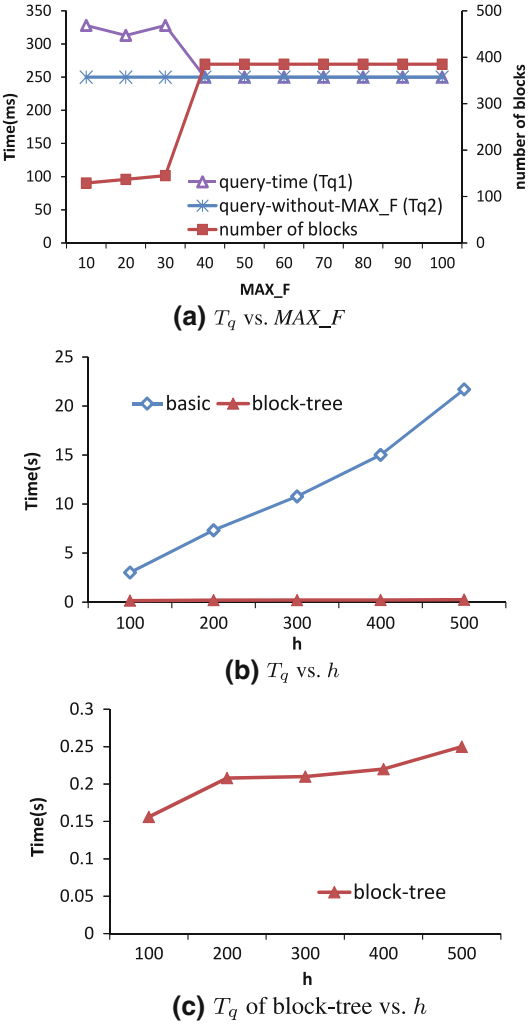


Fig. 21 Query performance of block tree (Geospatial datasets)

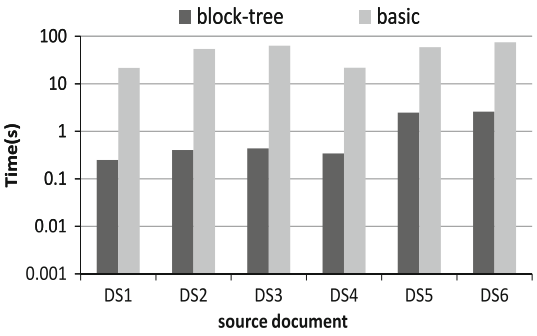
observe that *signature* always performs faster. Notice that both lines flatten when $MAX\_B = 120$, since the number of c-blocks is smaller than this value.

Figure 20c shows the overall time for creating the block tree, which is equal to the sum of mapping generation time (using *heap*/*partition*) and the block tree construction time (using *signature*/*combination*). We observe the use of *heap* and *signature* always yields the best performance.

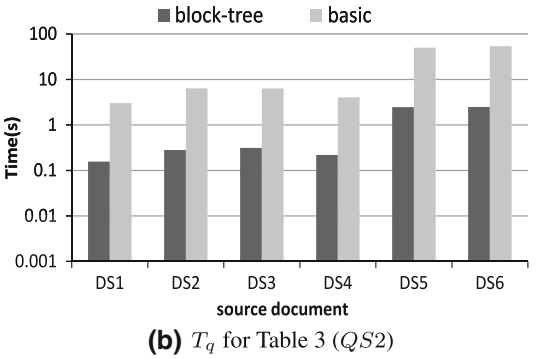
6. Query performance. Next, we study the query performance. We use the schema matching $MS1$ and the source document $DS1$ for analysis. We denote Algorithms 6 and 7 as *basic* and *block tree*, respectively. Unless stated otherwise, we assume that the version of the block tree that employs the hash table is used.

Figure 21a shows the number of c-blocks generated and the query evaluation time (T_{q1}) of $QS1$ using *block tree*, where the block tree is derived by the *combination* algorithm. The figure also shows the query evaluation time (T_{q2}) with the blocks generated by the new algorithm, *signature*. We observe that when *MAX_F* equals to 10, T_{q1} is 31.2% worse than T_{q2}. When *MAX_F* increases, more c-blocks are generated, which can support query evaluation. Therefore, T_{q1} decreases accordingly. When *MAX_F* becomes large enough to generate all the 385 c-blocks, T_{q1} converges to T_{q2}. From this experiment, we conclude that the *signature* algorithm offers better query support than *combination*.

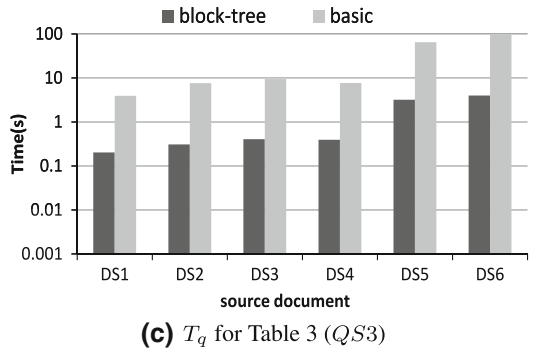
Figure 21b shows the query evaluation time of *basic* and *block tree* for the query $QS1$, by varying the number of mappings. We notice that *block tree* consistently outperforms *basic* for a wide range of possible mapping sizes. In addition, *block tree* is less sensitive to the number of mappings. From Fig. 21c, we observe that block tree-based query evaluation time increases slightly as the h becomes larger.



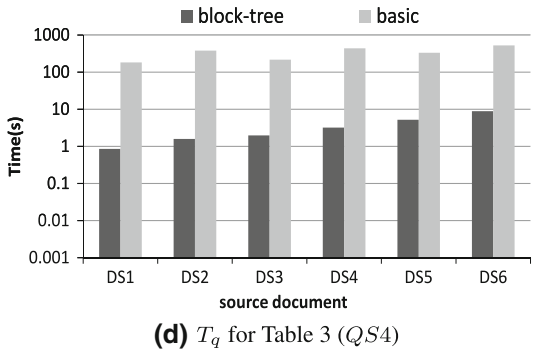
(a) T_q for Table 3 ($QS1$)



(b) T_q for Table 3 ($QS2$)

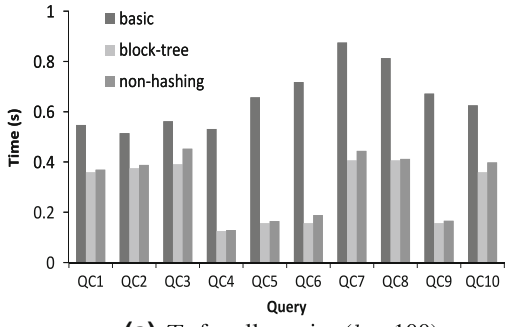


(c) T_q for Table 3 ($QS3$)

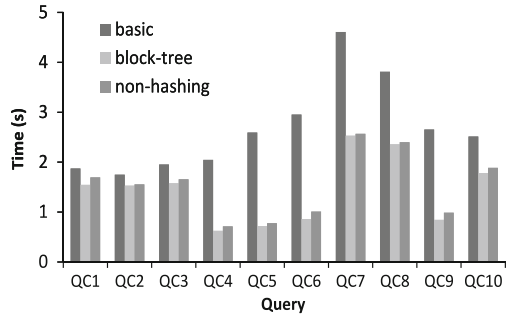


(d) T_q for Table 3 ($QS4$)

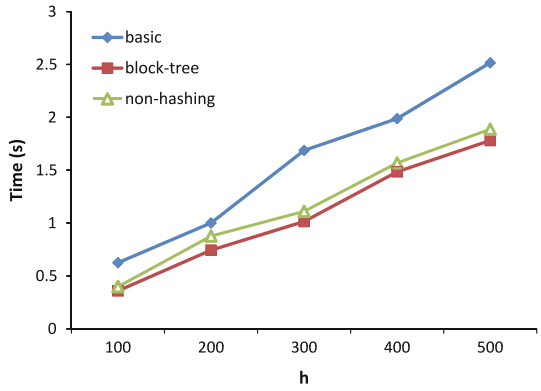
Fig. 22 Query performance on source documents (Geospatial data sets)



(a) T_q for all queries (h = 100)



(b) T_q for all queries (h = 500)



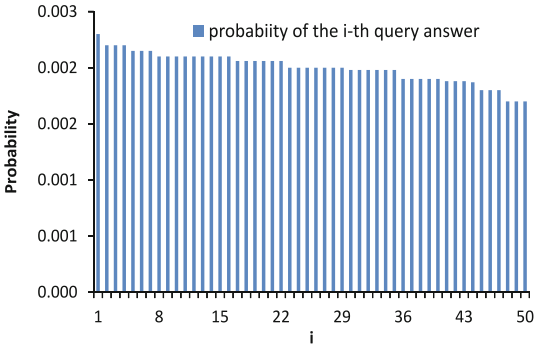
(c) T_q vs. h

Fig. 23 Query performance of block tree (e-commerce data sets)

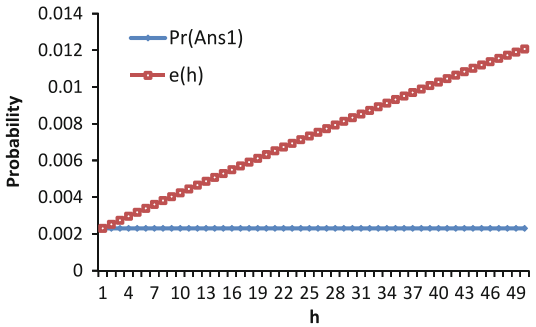
Figure 22a shows the query evaluation time of $QS1$ for all the source documents shown in Table 3, under the default setting. The y-axis is in logarithm scale. We notice that *block tree* significantly outperforms *basic* on all source documents and improves *basic* at 98% on average.

For other queries ($QS2$, $QS3$, $QS4$), we observe similar results (Fig. 22b, c, d).

Figure 23a shows the running time of $QC1$-$QC10$ on the schema matching $MC4$, using the block tree generated by the *signature* approach, where h is 100. 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*. When h is 500, a similar trend can be observed in Fig. 23b. In these two figures, *non-hashing* shows the running time of the block tree-based query evaluation method without using the hash table. We notice that when hash table is used in



(a) Probability distribution of query answers



(b) Query answer quality vs. h

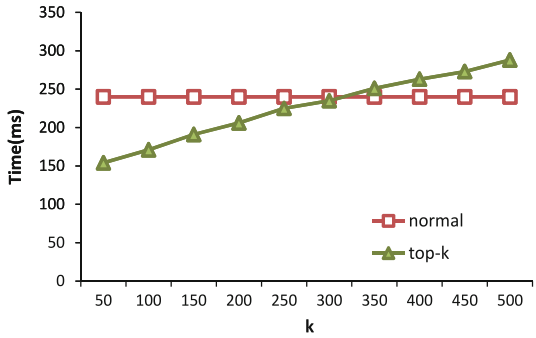
Fig. 24 Results on query answer quality

the querying process, the query performance improves by 8% on average.

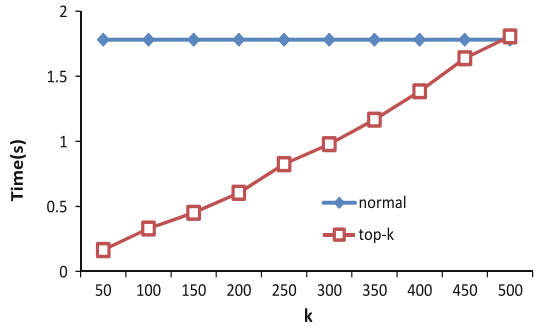
Figure 23c presents the evaluation time of query QC10, by running *basic*, *block tree*, and *non-hashing* on the matching MC4, for different values of h. For all methods, the time required increases with h, since more mappings need to be handled. Both *block tree* and *non-hashing* perform better than *basic*. Also, *block tree* runs faster than *non-hashing*, due to the use of the hash table.

Next, we examine the evaluation of query QS1 on matching MS1. Figure 24a shows the 50 highest answer probabilities, for $h = 500$. Since the probabilities of the mappings are close to each other, their corresponding answer probabilities are also similar. Figure 24b illustrates: (1) the probability of the top-1 query answer (i.e., $Pr(Ans1)$), and (2) the sum of probabilities of top h query answers. We observe that $Pr(Ans1)$ is not dominant, especially when h is large. Thus, it is better to consider a set of possible mappings than just the mapping with the highest probability.

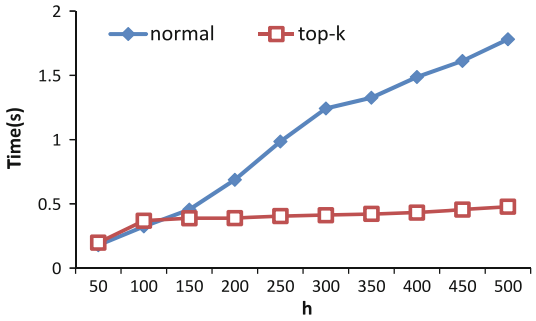
7. Top-k PTQ. We then investigate top-k PTQ, using the schema matching $MS1$. Figure 25a shows the performance of top-k PTQ under different values of k for the query $QS1$. As k increases, more mappings need to be considered, so T_q increases accordingly. The *normal* curve refers to a PTQ without using the top-k constraint. We can see that by placing the top-k constraint on the query, its performance can be significantly improved when k is small (e.g., 35.83% when $k = 50$). When k is larger than the number of results, the method top-k needs more time than *normal*, due to the sorting overhead of finding the top-k mappings.



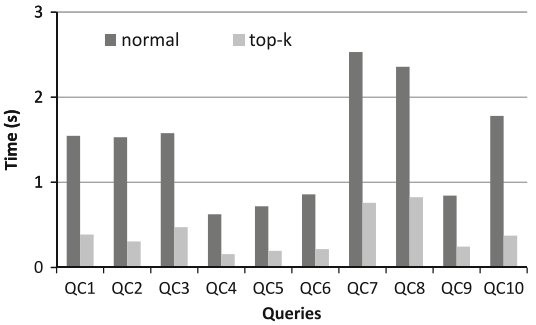
(a) T_q vs. k $(MS1)$



(b) T_q vs. k $(MC4)$



(c) T_q vs. h



(d) T_q for all queries $(MC4)$

Fig. 25 Top-k query performance

Figure 25b shows the performance of top-k PTQ under different values of k for query $QC10$ on the e-commerce data set $MC4$. We observe that by placing the top-k constraint on the query, its performance can also be significantly improved (e.g., 91.75% when $k = 50$). Figure 25c shows the performance of *normal* and *top-k* for different number of mappings h, on the data set $MC4$ and query $QC10$, with $k = 100$. For the *basic* method, its time increases sharply as h becomes larger, since it needs to process every mapping; for the *top-k* method, its time increases when h varies from 50 to 100, since the number of mappings to be processed is more (from 50 to 100). When $h > 100$, the query evaluation time then increases slightly, since the block tree size increases with the number of mappings. Finally, Fig. 25d compares the performance of *normal* and *top-k* for different queries, with $k = 100$. We observe that *top-k* is on average 73% faster than *normal*.

7 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 a 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 based on the signatures of mappings. 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. We further propose a bipartite matching algorithm customized for the remainder partition, 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 will study how to perform incremental update of the block tree due to insertion and deletion of correspondences. We will also consider the querying of probabilistic XML documents [14], under an uncertain schema matching later.

Acknowledgments "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".

Appendix

A The partitioning algorithm (Sect. 3.2)

Algorithm 8 describes the details of the partitioning process. First, U is partitioned in Step 1. Then, the top-h mappings are computed from each partition using a standard algorithm (e.g., [16,17]), and are *merged* to obtain the top-h mappings for U (Steps 2–6). The top-h mappings are returned in Step 7.

Algorithm 8 Partitioning algorithm

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

Complexity analysis

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

On average, a partition has $\frac{|S|+|T|}{l}$ elements. Assuming a fast algorithm like Pascoal [17] is used, the average complexity of generating top-h mappings from a partition is $O(h(\frac{|S|+|T|}{l})^3)$. Since there are l partitions, the average complexity of Algorithm 8 is $O(\frac{h(|S|+|T|)^3}{l^2} + (\frac{|U|^2}{l} + |U|))$, the former and the latter term being the bipartite-matching and merging-partitioning costs, respectively.

References

1. Agrawal, P., Sarma, A.D., Ullman, J., Widom, J.: Foundations of uncertain-data integration. Proc. VLDB Endow. **3**(1–2), 1080–1090 (2010)

2. Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the 18th International Conference on Data Engineering, ICDE'02, pp. 141–152. IEEE Computer Society, Washington (2002)
3. Alexe, B., Chiticariu, L., Miller, R.J., Pepper, D., Tan, W.C.: Muse: a system for understanding and designing mappings. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, Vancouver, Canada, pp. 1281–1284. ACM, New York. ISBN: 978-1-60558-102-6 (2008)
4. Arenas, M., Libkin, L.: XML data exchange: consistency and query answering. J. ACM **55**(2), 1–72 (2008). http://doi.acm.org/10.1145/1346330.1346332
5. Arion, A., Benzaken, V., Manolescu, I., Papakonstantinou, Y.: Structured materialized views for XML queries. In: Proceedings of the 33rd international conference on Very Large Data Bases, VLDB'07, Vienna, Austria, pp. 87–98. VLDB Endowment. ISBN: 978-1-59593-649-3 (2007)
6. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, Beijing, China, pp. 1–12. ACM, New York. ISBN: 978-1-59593-686-8 (2007)
7. Cheng, R., Gong, J., Cheung, D.W.: Managing uncertainty of XML schema matching. In: ICDE, pp. 297–308 (2010)
8. Das Sarma, A., Dong, X., Halevy, A.: Bootstrapping pay-as-you-go data integration systems. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, Vancouver, Canada, pp. 861– 874. ACM, New York. ISBN: 978-1-60558-102-6 (2008)
9. Do, H.H., Rahm, E.: COMA: a system for flexible combination of schema matching approaches. In: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB '02, Hong Kong, China, pp. 610–621. VLDB Endowment (2002)
10. Dong, X.L., Halevy, A., Yu, C.: Data integration with uncertainty. VLDB J. **18**(2), 469–500 (2009)
11. Fuxman, A., Hernandez, M.A., Ho, H., Miller, R.J., Papotti, P., Popa, L.: Nested mappings: schema mapping reloaded. In: Proceedings of the 32nd International Conference on Very Large Data Bases,VLDB '06, Seoul, Korea, pp. 67–78. VLDB Endowment (2006)
12. Gal, A.: Managing uncertainty in schema matching with top-k schema mappings. J. Data Semant. **VI**, 90–114 (2006)
13. Gal, A., Martinez, M.V., Simari, G.I., Subrahmanian, V.S.: Aggregate query answering under uncertain schema mappings. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, pp. 940–951. IEEE Computer Society, Washington. ISBN: 978-0-7695-3545-6 (2009)
14. Kimelfeld, B., Kosharovsky, Y., Sagiv, Y.: Query efficiency in probabilistic XML models. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, Vancouver, Canada, pp. 701–714. ACM, New York. ISBN: 978-1-60558-102-6 (2008)
15. Lenzerini, M.: Data integration: a theoretical perspective. In: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, Madison, Wisconsin, pp. 233–246. ACM, New York. ISBN: 1-58113-507-6 (2002)
16. Murty, K.G.: An algorithm for ranking all the assignment in increasing order of cost. Oper. Res. **16**, 682–687 (1986)
17. Pascoal, M., Captivo, M., Clímaco, J.: A note on a new variant of Murty's ranking assignments algorithm. 4OR **1**(3), 243–255 (2003)
18. Qin, L., Yu, J.X., Ding, B.: Twiglist: make twig pattern matching fast. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications, DASFAA '07, Bangkok, Thailand, pp. 850–862. Springer, Berlin. ISBN: 978-3-540-71702-7 (2007)
19. Raffio, A., Braga, D., Ceri, S., Papotti, P., Hernández, M.A.: Clip: a tool for mapping hierarchical schemas. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, Vancouver, Canada, pp. 1271–1274. ACM, New York. ISBN: 978-1-60558-102-6 (2008)
20. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. VLDB J. **10**(4), 334–350 (2001)
21. Roitman, H., Gal, A., Domshlak, C.: Providing top-k alternative schema matchings with ontomatcher. In: Proceedings of the International Conference on Conceptual Modeling (2008)
22. Vaz Salles, M.A., Dittrich, J.P., Karakashian, S.K., Girard, O.R., Blunschi, L.: iTrails: pay-as-you-go information integration in dataspaces. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, Vienna, Austria, pp. 663–674. VLDB Endowment. ISBN: 978-1-59593-649-3 (2007)
23. Yu, C., Popa, L.: Constraint-based XML query rewriting for data integration. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04, Paris, France. pp. 371–382. ACM, New York. ISBN: 1-58113-859-8 (2004)