Evaluating Probabilistic Queries over Uncertain Matching

Reynold Cheng†, Jian Gong†, David W. Cheung†, and Jiefeng Cheng‡

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

‡*Shenzhen Institute of Advanced Technology, China*
jf.cheng@siat.ac.cn

***Abstract*—A *matching* between two database schemas generated by machine learning techniques (e.g., COMA++) is often uncertain. Handling the uncertainty of schema matching has recently raised a lot of research interest, because the quality of applications relies on the matching result. We study query evaluation over an inexact schema matching which is represented as a set of "possible mappings", as well as the probabilities that they are correct. Since the number of possible mappings can be large, evaluating queries through these mappings can be expensive. By observing that the possible mappings between two schemas often exhibit a high degree of overlap, we develop two efficient solutions. We also present a fast algorithm to compute answers with the k highest probabilities. An extensive evaluation on real schemas shows that our approaches improve query performance by almost an order of magnitude.**

I. Introduction

Schema matching [1] is the process of finding the possible relationship between database schemas. It is the key for many techniques in data integration [2], such as mapping generation [3] and query reformulation [4]. To facilitate schema matching, tools like COMA++ [5] and LSD [6] have been developed.

A matching result, which captures relationships, or *correspondences*, between attributes across different schemas, is often *uncertain*. This is because a matching algorithm cannot guarantee that the correspondences returned are correct. Figure 1 illustrates a portion of the matching result between two relational schemas, which are about customers and purchase orders. It is not clear which attribute in the relation *Customer* should correspond to the *phone* attribute in *Person*: should it be *ophone*, *hphone*, or *mobile*? In this example, a similarity score, generated by a matching algorithm, is attached to each correspondence to indicate the confidence that the relationship between the attributes involved is valid.

To handle the uncertainty of a matching, one can hire a domain expert to select the correct set of correspondences. This may not work if an expert is not available, or if the scale of an application (e.g., web data integration) is large [7]. Another way is to choose the correspondences with the highest scores (as shown in bold lines in Figure 1), and ignore the rest of them. However, this may render some information missing in the query answers. Consider two schemas, called *source* and *target schemas* (or S and T respectively). A database D is associated with S. A *target query* is issued on T, which obtains information from D through the matching between

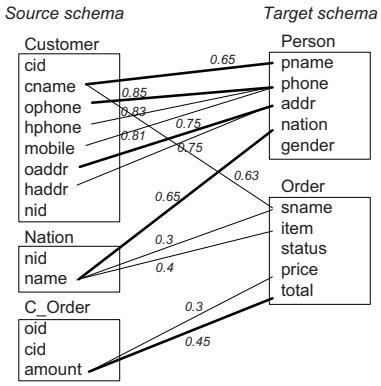


Fig. 1. A schema matching.

cid	cname	ophone	hphone	oaddr	haddr	...
t_1	*Alice*	*123*	*789*	*aaa*	*hk*	
t_2	*Bob*	*456*	*123*	*bbb*	*hk*	...
t_3	*Cindy*	*456*	*789*	*aaa*	*aaa*	

Fig. 2. A relation for *Customer*.

T and S. In Figure 1, *Person* is part of the *target schema*, while *Customer* is part of the *source schema*, with a database attached (Figure 2). Now consider the following *target query*:

$$q_0 : \pi_{addr}\sigma_{phone='123'}Person$$

Let us use a *mapping* between *Person* and *Customer*, which contains all the correspondences bolded in Figure 1. By using a query reformulation method (e.g., [4]) on this mapping, we translate q_0 to a *source query* defined on *Customer*. Since this mapping uses correspondence (*ophone, phone*), *aaa* is the answer to q_0. However, if we choose (*hphone, phone*), whose similarity score (0.83) is slightly lower than that of (*ophone, phone*) (0.85), then q_0's answer becomes *bbb*. This example shows that we may not ignore correspondences with lower similarity values, since they may yield a different query result.

Recently, researchers have considered an uncertain matching as a set of *possible mappings* [8], [7], [9], [10], [11]. Figure 3 shows five mappings, and their probabilities of being a correct one, for the matching in Figure 1. For example, to obtain the probability of m_1, the 5 mappings with the highest similarity scores (i.e., $m_1, \ldots, m_5$) are first found. The probability of m_1 is then equal to its score divided by the total score of these 5 mappings. Given these mappings, we examine a *probabilistic query* on T. This query returns a set of pairs (t_i, p_i), where

ID	correspondences	prob.
m_1	***(cname, pname)***, (ophone, phone), (oaddr, addr), (name, nation), ...	0.3
m_2	***(cname, pname)***, (ophone, phone), (oaddr, addr), (name, nation), ...	0.2
m_3	***(cname, pname)***, (ophone, phone), (haddr, addr), (name, nation), ...	0.2
m_4	***(cname, pname)***, (hphone, phone), (haddr, addr), (name, nation), ...	0.2
m_5	(cname, sname), (ophone, phone), (haddr, addr), (name, item), ...	0.1

Fig. 3. Illustrating possible mappings for Figure 1.

p_i is the probability that tuple t_i is correct. For instance, the answer for q_0 is $\{(aaa, 0.5), (hk, 0.5)\}$.

A simple way to evaluate this query is that for every mapping, the target query is reformulated to a source query. Then, we evaluate these source queries either sequentially, or by using some multiple query optimization algorithm (e.g., [12]). The results of the source queries are then aggregated to produce the target query answer. If the number of mappings is large, many source queries can be generated, yielding a poor query performance. A slightly better way is to produce a set of distinct source queries before evaluating them. When the number of mappings or the mapping size are large, it can still take a long time to translate a query. We have tested these methods on 500 mappings, each of which has 46 correspondences. In one case, it takes around 1.8 hours to complete a query.

In this paper, we study the efficient evaluation of probabilistic queries over possible mappings. We observe that the possible mappings between schemas are often very similar in terms of their correspondences. In an experiment, we found that a set of 500 possible mappings between two e-commerce schemas are highly similar. As another example, in Figure 3, $(cname, pname)$ and $(ophone, phone)$ (bolded and underlined respectively) are shared by four mappings. We thus develop two novel solutions based on this intuition:

1. Query-Level Sharing. If two source queries, generated by two different mappings, are identical, only one source query needs to be executed. We exploit this observation by developing the *query-level sharing* (or *q-sharing*) algorithm, which *partitions* the mappings according to the source queries they produce. Each group of mappings "share" a single source query, which only needs to be evaluated once for each group. This is faster than executing a source query for every mapping. A salient feature of this approach is that the source query groups are discovered during the mapping translation process, through the use of an efficient partitioning algorithm. Compared with the approaches mentioned earlier, which generates a source query through every mapping individually, *q-sharing* requires a lower translation effort. This solution is also flexible; it can be applied to any kind of queries.

2. Operator-Level Sharing. Query cost can also be reduced when the source queries produced by two mappings possess common operators. Suppose that q_0 is translated through m_2 and m_3 (Figure 3). Since m_2 and m_3 share $(ophone, phone)$, their respective source queries contain $\sigma_{ophone='123'}$*Customer*. The result of running this operator can then be shared by both queries. We thus develop the *operator-level sharing* (or *o-sharing*) solution. Particularly, we study two metrics for quantifying the amount of benefit that can be brought by executing a target query operator, in terms of the likelihood its query result can be used by other mappings. We also study how to use these metrics to arrange the evaluation order of unary (e.g., selection, projection), binary (e.g., join), and aggregate (e.g., SUM and COUNT) operators.

Top-k Queries. We examine a variant of the probabilistic queries, called the *top-k query*, which returns tuples whose probabilities are the k-highest among all the answer tuples. By specifying the value of k, a user can require a query to only return answers with a high confidence. Based on the *o-sharing* solution, we develop a query algorithm, which does not compute the exact probability of every answer tuple. Our experiments show that the performance of a top-k query can be significantly improved.

The rest of the paper is as follows. We discuss the related work in Section II. We present the problem definition and discuss some simple solutions in Section III. Section IV presents the *q-sharing* solution. In Sections V and VI, we describe the *o-sharing* solution. We present our top-k query algorithm in Section VII. Section VIII presents our experiment results. We conclude in Section IX.

II. Related Work

Schema matching is an important topic in data integration [2], [1], [13], [4]. However, most research does not consider uncertainty in a matching. The idea of modeling uncertainty of a schema matching as a set of *possible mappings* has been recently proposed [9], [8], [10]. They discuss how to obtain a set of h possible mappings, where h is user-specified. Particularly, a bipartite matching algorithm is evaluated on the matching, which returns h mappings with the highest similarity scores. The probability of each mapping is derived by normalizing the mapping's similarity score over the total scores of the h mappings. We study query evaluation on this model. Other uncertainty models include [7], which addresses uncertainty in the mediated schema; and [14], which handles uncertainty in the source database. In [15], the authors proposed methods to reduce human effort for analyzing the matching result, but they do not consider probabilistic mappings.

In [10], we found that the possible mappings derived from a XML schema matching share many correspondences. Interestingly, we also observe a similar phenomenon in relational matching. However, the methods in [10] are designed to evaluate a single twig query operator, which cannot be used to address a relational query that involves multiple operators. Moreover, our solution can remove query answer duplicates, which was not done in [10]. The work closest to ours is [8]. It mentioned that the cost of storing possible mappings can be reduced by grouping the mappings, but does not further explain how to group them. It also shows how to use these groups to answer a simple target query that involves a single

attribute. We develop a more systematic and comprehensive solution than [8]. Particularly, we devise efficient algorithms to cluster mappings. Our solutions can be applied to a complex query that contain multiple attributes and operators.

In uncertain and probabilistic databases, a few algorithms (e.g., [16], [17], [18], [19]) have been proposed to evaluate top-k queries efficiently. However, these solutions are not designed to handle schema matching uncertainty; instead, they address the uncertainty of tuples and attribute values in a relational database. It is not clear how these algorithms can handle matching uncertainty. We develop a query algorithm to evaluate top-k queries on possible mappings. By avoiding the computation of the exact probability of an answer tuple, our method significantly improves the query performance.

III. Problem Definition

We now describe the data and query models assumed in this paper, in Section III-A. Then we discuss three simple solutions for evaluating a probabilistic query, in Section III-B.

A. Data and Query Models

Data Model. Let S and T be the source and target schemas respectively. Let a_S (a_T) be an attribute of S (T), called *source (target) attribute*. A database D, called *source instance*, is associated with S. A matching between S and T is represented by a set M of h possible mappings [9], [8]. Each mapping m_i consists of a set of *correspondences* between source and target attributes. We assume that the correspondences between these attributes exhibit a one-to-one and partial relationship. Each m_i has a probability $Pr(m_i)$ to be correct. If e_i is the event that m_i is correct, then all e_is' are mutually exclusive. Thus, $\sum_{i=1}^{h} Pr(m_i) = 1$.

Query Model. We consider the evaluation of a *probabilistic query*, q_T, which is executed on the target schema T. Unless stated otherwise, q_T can be any kind of query. (In *o-sharing*, we study select, projection, join, and aggregate operators (e.g., SUM, COUNT).) Our solutions, which aggregate duplicate answers, can be easily changed if duplicate removal is not required. The answer of q_T is obtained through reformulating q_T to a query on S, as we will discuss later. Table I shows the symbols used in this paper.

TABLE I
Notations and meanings.

Notation	Meaning
S	Source schema
T	Target schema
D	Source instance of S
a_S	A source attribute
a_T	A target attribute
M	A set of h possible mappings between S and T
m_i	The i-th mapping of M, with $i \in [1, h]$
$Pr(m_i)$	Probability that m_i is correct
q_T	Target query, with l operators
q_{S_i}	Source query for mapping m_i
$(t, Pr(t))$	A tuple in the answer of q_T, with prob. t is correct

B. Simple Solutions

We now discuss three simple solutions for answering probabilistic queries, namely, *basic*, *e-basic*, and *e-MQO*.

1. basic. This algorithm requires three parameters: target query q_T, mapping set M, and source instance D. For every mapping m_i, *basic* reformulates q_T to a source query q_{S_i}. It then evaluates q_{S_i} on D. For each tuple obtained through m_i, its probability is equal to $Pr(m_i)$. Finally, the tuples returned from the M mappings are aggregated, by summing up probabilities of answers that are duplicates.

Example. Consider the following target query for Figure 1:

$$\pi_{phone}\sigma_{addr='aaa'}Person$$

using the possible mappings in Figure 3. For m_1, the target query q_T can be reformulated to:

$$q_{S_1} : \pi_{ophone}\sigma_{oaddr='aaa'}Customer$$

By evaluating q_{S_1} on the source instance in Figure 2, *123* and *456* are returned, each with probability $Pr(m_1) = 0.3$. For m_2, the same set of tuples are produced, each with probability 0.2; For m_3, only *456* is returned with probability 0.2; For m_4 and m_5, *789* and *456* are returned, with probability 0.2 and 0.1 respectively. After result aggregation, the final query answers are: (*123*, 0.5), (*456*, 0.8), and (*789*, 0.2).

In *basic*, a target query is executed on D for h times. If h and D are large, q_T can be costly to evaluate. We next discuss two better solutions.

2. e-basic. This is an enhanced version of *basic*. Different from *basic* that evaluates each of the h source queries once, *e-basic* clusters the identical source queries. Then, *e-basic* evaluates this set of distinct source queries. If there is only a small set of distinct source queries, *e-basic* can run much faster than *basic*.

3. e-MQO. This solution attempts to improve the performance of *e-basic*. Instead of evaluating each distinct source query independently, *e-MQO* first generates an optimal global query plan, by using some *multiple-query optimization* (MQO) method (e.g., [20], [12]). This guarantees that the number of operators used to evaluate the set of distinct source queries is minimal. This solution is also useful for us to compare with other methods experimentally, since they may not be optimal in terms of the number of source query operators executed.

A common problem of *e-basic* and *e-MQO* is that they do not save query rewriting effort – a set of h source queries need to obtained first. This cost can be high when q_T contains many attributes, or when h is large. Experimentally, *e-MQO* is slower than *e-basic*, as it often takes a long time to generate an optimal query plan. The next method, *q-sharing*, reduces rewriting effort by avoiding the generation of h source queries.

IV. Query-Level Sharing

The main idea of *q-sharing* is to identify the groups of mappings that lead to the same source queries. For each group of mappings, evaluating the source query once is sufficient. Algorithm 1 illustrates this method. Step 1 first *partitions* the

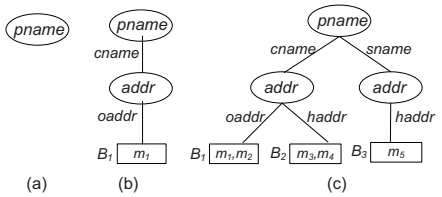


Fig. 4. Illustrating a partition tree.

mapping set M based on the target query q_T. This produces f groups of mappings, each of which leads to the same source query. Step 2 retrieves the set M' of f *representative mappings* from the f partitions, using the *represent* routine. This is the set of mappings that need to be used for query rewriting. Essentially, each mapping of M', which leads to different source queries, is obtained by selecting a mapping from each partition. The *represent* function also computes the "probability" of the representative mapping from the j-th partition, which is the sum of probabilities of all mappings in the j-th partition. This is also the probability of the answer tuples produced under all mappings of the j-th partition, since all mappings in the j-th partition yield the same query result. Step 3 invokes *basic* to compute the target query answer, based on M'.[1]

Algorithm 1 *q-sharing*

Input: mapping set M, target query q_T, source instance D
Output: answer of q_T
1: $P_1, \ldots, P_f = partition(q_T, M)$
2: $M' \leftarrow represent(P_1, \ldots, P_f)$
3: return $basic(q_T, M', D)$

Example. Consider the evaluation of the following query:

$$q_1 : \pi_{pname}\sigma_{addr='abc'}Person$$

over the mappings in Figure 3. In Algorithm 1, the mappings are classified into: $P_1 = \{m_1, m_2\}$, $P_2 = \{m_3, m_4\}$, and $P_3 = \{m_5\}$. The mapping(s) in each partition produce(s) the same source query. We can thus choose the representative mappings for m_1, m_3, and m_5, with respective probabilities 0.3+0.2=0.5, 0.2+0.2=0.4, and 0.1. Then we use *basic* to evaluate q_1 over these mappings.

Compared with *e-basic* and *e-MQO*, *q-sharing* does not obtain h source queries first before clustering them. Instead, it partitions the mappings and derives the distinct source queries from these mappings. As shown in our experiments, *q-sharing* performs better than *e-basic* and *e-MQO*. To further reduce the query rewriting time, we implement a fast *partition* routine, as described next.

A. Efficient Mapping Partitioning

Observe that a partition of mappings, which produce the same source query, must share the same correspondences for the attributes specified in the target query. We use this intuition to develop a *partition tree*, which supports efficient mapping partitioning. Given a target query q_T with l attributes, a *partition tree* has (l+1) levels. The nodes at the k-th level ($1 \leq k \leq l$) correspond to the k-th target attribute a_k, and each leaf node is a *bucket*, which contains a set of mappings that belong to the same partition. Each edge is labeled with some source attribute a'_k, which matches a_k according to a mapping's correspondence. After the partition tree has been completed, each leaf node contains a distinct partition of mappings, which can then be used by *q-sharing*.

Figure 4(a) shows the initial state of a partition tree for q_1, which contains a single root node for $pname$, the first attribute of q_1. We update the partition tree by using all mappings in M. When a mapping $m \in M$ is considered, the partition tree determines if m should be put into an existing bucket, or create a new bucket for m. This is done by traversing the partition tree in a top-down manner. At the k-th level, the target attribute a_k is examined: if there exists an out-going edge e from the current node, such that e is labeled a'_k and the correspondence (a'_k, a_k) is in m, it traverses the partition tree according to e. Otherwise, a new edge labeled a'_k is created, and is linked to a new child node for a_{k+1}. After all the target attributes are examined, m will be put into the leaf node, which is a bucket of partitions.

We illustrate this process with the mappings shown in Figure 3. First, for m_1, nodes $addr$ and B_1 are created, and m_1 is put into bucket B_1, as shown in Figure 4(b). Next, m_2 is also put into B_1, since both of them match $pname$ with $cname$, and $addr$ with $oaddr$. Notice that m_3 is assigned to another bucket B_2, since it matches $addr$ to a different attribute, $haddr$. Figure 4(c) shows the final state of the partition tree. As we can see, each bucket contains a distinct partition of the mappings.

Algorithm 3 (Appendix A) details the *partition* routine. It uses the mappings in M to update the partition tree. For every mapping, a recursive function is used to handle each attribute of q_T. The cost of *q-sharing* is dominated by the source query execution time, which depends on the number f of representative mappings. If f is large, *q-sharing* can still be slow. To alleviate this problem, we next study the *o-sharing*.

V. O-Sharing: Framework

A common problem of *e-basic*, *e-MQO*, and *q-sharing* is that they only save query costs only when two mappings have the same correspondences for the target query. This may not be true for a query with many attributes. However, *operator-level sharing*, or *o-sharing*, can save costs even if the correspondences of two mappings are not exactly the same. This is achieved by *interleaving* the processes of query rewriting and query execution. *o-sharing* can support a wide range of queries: SPJ operators (i.e., selection, projection, and join) and aggregate operators (e.g., COUNT and SUM). We present its framework below, and describe its algorithm details in Sections V-A and VI.

Framework. Given a target query, *o-sharing* chooses an operator based on the number of correspondences shared by the mappings. This operator is then executed. The process is repeated until all target operators are considered. We now

[1] Step 3 does not use an MQO algorithm, because it is slower than *basic*. More details can be found in Section VIII.

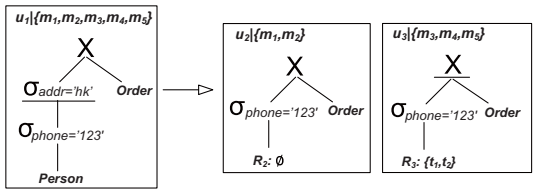


Fig. 5. Illustrating e-units.

introduce two data structures, namely, the *e-unit* and the *u-trace*.

An **e-unit** (or *execution unit*) captures the current state of the target query. The following constitutes an e-unit u:

• **query plan**, denoted by $u.T$, which organizes the target query operators not yet executed, and the intermediate results of the operators executed previously;

• **mapping set**, denoted by $u.M$, the mappings which share the correspondences for the target attributes associated with the operators evaluated before; and

• **next-op**, denoted by $u.o_{next}$, which is a query operator in $u.T$, and will be executed in the next step.

Essentially, u specifies a partially executed target query, $u.T$, where the operator, $u.o_{next}$, will be be handled next, under the mapping set $u.M$. Figure 5 shows three e-units, u_1, u_2, and u_3, generated based on the following target query:

$$q_2 : (\sigma_{addr='hk'}\sigma_{phone='123'}Person) \times Order$$

and the mappings shown in Figure 3. First, u_1 is produced by using q_2 as its query plan. The mapping set of u_1 contains mappings $m_1, \ldots, m_5$, while its next-op is chosen to be $\sigma_{addr='hk'}$ (underlined). Suppose now $\sigma_{addr='hk'}$ is executed. This is done by first rearranging it to be executed on *Person*, and then reformulated according to $u_1.M$ (We will explain the operator selection strategies in Section VI-B). Notice that the target attribute, *addr*, corresponds to the same source attribute *oaddr* for m_1 and m_2; and matches *haddr* for m_3, m_4, and m_5. Since *oaddr* and *haddr* are attributes of *Customer*, there are two ways of executing $u_1.T$:

• For m_1 and m_2, execute $\sigma_{oaddr='hk'}Customer$, and produce source relation R_2, which is an empty relation.

• For m_3, m_4, and m_5, execute $\sigma_{haddr='hk'}Customer$, and produce source relation R_3, which contains $\{t_1, t_2\}$.

The above execution produces two e-units, called u_2, and u_3. Figure 5 shows the components of these two e-units. Notice that $u_2.T$ ($u_3.T$) differs from $u_1.T$, in which $\sigma_{addr='hk'}Person$ is replaced by R_2 (R_3). Also, $u_2.M = \{m_1, m_2\}$, while $u_3.M = \{m_3, m_4, m_5\}$. Moreover, the number of times next-op ($\sigma_{addr='hk'}$) is evaluated is reduced from 5 (for 5 possible mappings) to 2 only. The next-op of u_2 and u_3 will be decided later.

A **u-trace** is a tree of e-units that have not yet been considered. We use Figure 5 to show how to use a u-trace to obtain answers for the mappings in Figure 3. Initially, a u-trace consists of u_1, where $u_1.T$ is created from q_2. We first evaluate $u_1.o_{next}$, through either m_1 and m_2. The result, R_2, is used to create another e-unit, u_2. Since R_2 is empty, the result of evaluating $u_2.T$ must be empty. Hence, $u_2.T$ yields

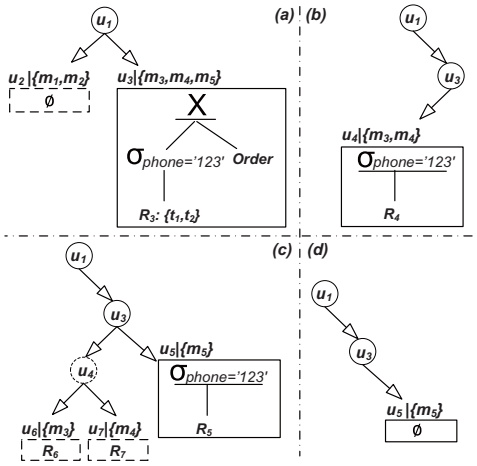


Fig. 6. Illustrating a u-trace.

an empty relation, for both m_1 and m_2. It is removed from the u-trace, as illustrated in Figure 6(a).

Next, u_3 is generated by using m_3, m_4, and m_5 (Figure 6(a)). The Cartesian product operator is used as u_3's next-op. The detail of selecting operators will be explained in Section VI-A. An e-unit u_4 is produced, with $u_4.M = \{m_3, m_4\}$ (Figure 6(b)). By considering $\sigma_{phone='123'}$ on $u_4.T$, we get two more e-units: u_6 and u_7, which correspond to executing the operator according to m_3 and m_4 respectively (Figure 6(c)). Since u_6 (u_7) does not contain any operator, R_6 (R_7) is the answer for m_3 (m_4). For u_5, since R_5 is empty, the query answer for m_5 is empty (Figure 6(d)).

Different from *e-basic*, *e-MQO* and *q-sharing*, which prepares a complete set of source queries before evaluating them, *o-sharing* interleaves the query rewriting and operator execution tasks by using the u-trace. This allows *o-sharing* to discover the opportunity of sharing operator evaluation effort during query reformulation. Moreover, since some intermediate relations are empty, *o-sharing* may not have to consider the whole target query for every mapping. We next study the algorithm for handling the u-trace.

A. The o-sharing algorithm

The *o-sharing* method (Algorithm 2) first finds the representative mappings M' (Steps 1-2). It initializes a u-trace by creating an e-unit, u_1, where $u_1.T$ is the query plan of q_T, and $u_1.M = M'$ (Step 3). It calls a recursive function *run_qt* to compute the answers $(R_1, \ldots, R_g)$ for the u-trace, in Step 4. These answers are aggregated and returned in Step 5.

We now explain *run_qt*, which evaluates the target query answer for a given e-unit u. Step 1 initializes two arrays, ans_S and ans_T, for holding temporary results returned by source and target queries respectively. Then, *run_qt* considers whether to return target query answers, or invoke another recursive call, based on three scenarios:

Case 1 (Steps 2-7): $u.T$ is a relation. Thus, all operators in q_T are used. All tuples in $u.T$ can then be returned as answers, each of which has a probability equal to the sum of probabilities of all mappings in $u.M$. In the e-unit u_6 (Figure 6(c)), all results of R_6 can be returned, each of which

Algorithm 2 *o-sharing*

Input: mapping set M, target query q_T, source instance D
Output: all the query answers

1: $P_1, \ldots, P_f \leftarrow partition(q_T, M)$
2: $M' \leftarrow represent(P_1, \ldots, P_f)$
3: $u_1 \leftarrow init\_u\_trace(q_T, M')$
4: $R_1, \ldots, R_g \leftarrow run\_qt(u_1, D)$
5: return $aggregate(R_1, \ldots, R_g)$

function *run_qt(e-Unit u, source instance D)*
Output: all the query answers for u from D

1: $ans_S \leftarrow \emptyset, ans_T \leftarrow \emptyset$
2: **if** $u.T$ is a relation **then**
3: **for all** $t \in u.T$ **do**
4: $ans_T \leftarrow ans_T \cup (t, \sum_{m \in u.M} Pr(m))$
5: **end for**
6: delete u
7: return ans_T
8: **else if** $u.T$ contains an empty relation **then**
9: $ans_T \leftarrow (\theta, \sum_{m \in u.M} Pr(m))$
10: delete u
11: return ans_T
12: **else**
13: $(P_1, \ldots, P_g) \leftarrow next(u)$
14: $reorder\_op(u, u.o_{next})$
15: **for all** $P_i \in P_1, \ldots, P_g$ **do**
16: $m \leftarrow$ an arbitrary mapping in P_i
17: $o' \leftarrow reformulate\_op(u.o_{next}, m)$
18: $ans_S \leftarrow run\_qs(o', D)$
19: $u_i.T \leftarrow create\_qtree(u.T, u.o_{next}, ans_S)$
20: $u_i.M \leftarrow P_i$
21: $R_i \leftarrow run\_qt(u_i, D)$
22: **end for**
23: delete u
24: return $R_1, \ldots, R_g$
25: **end if**

has probability $Pr(m_3)$. After Step 5, all answers in u are kept in ans_T. Then, u is deleted, and ans_T is returned.

Case 2 (Steps 8-11): $u.T$ contains an empty relation. Step 9 stores the result of executing $u.T$, which is a null tuple θ, and its probability in ans_T. Steps 10-11 delete u and return ans_T. In Figure 6(a), the probability of the empty result for u_2 is $Pr(m_1) + Pr(m_2)$.

Case 3 (Steps 12-24): The answer of $u.T$ is evaluated as follows:

(i) Call function *next* (Step 13). This function finds $u.o_{next}$ and returns the partition of mappings with respect to $u.o_{next}$.

(ii) Rearrange $u.o_{next}$ in the query tree of u, so that it is is ready to be computed (Step 14). In Figure 5, we push $\sigma_{addr='hk}$ down one level of $u_1.T$, so that it can be evaluated on *Person*.

(iii) Evaluate $u.o_{next}$, and create new e-units in the u-trace (Steps 15-22). In detail, for each mapping partition P_i, it first runs *reformulate_op*, which translates $u.o_{next}$ to a source operator o' (Steps 16-17). Using the *run_qs* routine, o' is executed on D, and the results are stored in ans_S (Step 18). A new e-unit u_i is created, whose query tree is generated by modifying $u.T$ with $u.o_{next}$ and ans_S (Step 19). Then, *run_qt* is invoked on u_i, using the mapping set P_i (Steps 20-21). In Figure 5, the execution of $\sigma_{addr='hk'}$ in $u_1.T$ produces u_2 and u_3. Note that the query plans of these e-units are created by replacing $\sigma_{addr='hk'}$ with query results R_2 and R_3, in $u_1.T$.

(iv) Delete u and return the results for u's partitions.

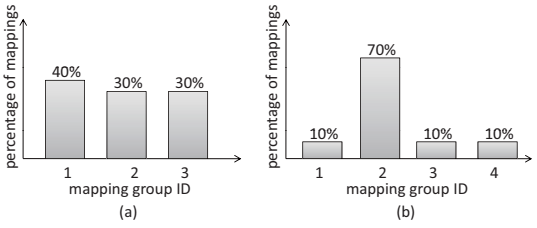


Fig. 7. Mapping distribution of operator o_1 (a) and o_2 (b).

We next study the details of *next* and *reformulate_op*.

VI. O-Sharing: Details

We now study how to select the next target operator for evaluation, in Section VI-A. Section VI-B then explains how to reformulate the chosen target operator.

A. Operator Selection Strategies

Recall that Algorithm 2 selects a target operator from q_T, using the function *next*, in Step 13. We first describe two criteria of choosing a target operator not yet executed: *correctness* and *effectiveness*. To understand *correctness*, notice that not all operators are allowed to be chosen. For example, if an operator o is not next to the leaf node of $u.T$, and it is a projection operator whose attributes do not contain all attributes contained in $u.T$, then o cannot be used. Hence, the first task of *next* is to select valid operators, in order to ensure correct query results.

Given a set of correct target operators, the next step is to find an *effective* one as the next operator to be evaluated. This is an important step; if we make a poor choice, an operator may be translated to many source operators and incur a high query cost. To accomplish this task, we consider three different methods, namely, *Random*, *SNF* and *SEF*.

1. Random arbitrarily selects the next operator. Although it is easy to implement, it does not consider any information about the possible mappings. For example, if a chosen operator happens to create a lot of mapping partitions, then many source operators will be generated and executed, resulting in a high computational cost. The next method, *SNF*, considers mapping information in making a choice.

2. SNF (or *Smallest Number of Partitions First*) chooses a target operator that leads to the fewest mapping partitions. Figure 7 illustrates the partition information of two target operators, o_1 and o_2, in some e-unit u's query plan. They are candidates for the next operator. Figure 7(a) shows that o_1 segments the mapping set of u into three partitions. That is, for every set of mappings in a partition, o_1 is translated to the same source operator. In Figure 7(b), o_2 provides four mapping partitions. Here, *SNF* chooses o_1, since it has fewer partitions than that of o_2. Intuitively, the smaller the number of partitions, the fewer source queries need to be translated. Hence, *SNF* prefers o_1 to o_2.

To implement *SNF*, we compute the number of mapping partitions of each correct operator o in $u.T$, by (1) running the *partition* routine on o, using mapping set $u.M$; and (2) counting the number of mapping partitions of o. We then

assign the operator that possesses the fewest partitions, to $u.o_{next}$. Finally, *next* returns the partition for $u.o_{next}$.

The main problem of *SNF* is that it does not use all the partition information. In Figure 7 we mark each partition with the fraction of the mappings that belong to that partition. If o_1 is chosen, it will be translated to 3 source operators. In other words, the query result of each of these operators can be shared by about 30% of the mapping set in u. For o_2, observe that partition 2 contains a large fraction (70%) of mappings. Thus, the result of executing o_2 on partition 2 can be shared by 70% of mappings. If a new e-unit v is produced as a result of executing o_2, 70% of mappings in partition 2 can appear in its mapping set. This can be beneficial to v; due to its large mapping set, the chance that the next operator can be shared among larger partitions is also high. Thus, it may better to execute o_2 than o_1. However, *SNF* does not consider the number of mappings in a partition, and it does not suggest o_2 as the next operator. Next, let us see how *SEF* takes the size of a partition into account.

3. SEF (or *Smallest Entropy First*), enhances *SNF* by considering the size of every mapping partition. This is done by using the *entropy* function [21]. To understand, given a mapping partition, we consider its fraction value to be the *probability* that the set of correspondences associated with that partition is used for reformulation. In Figure 7(b), the probability of the event that the set of attribute correspondences in partition 2 is used is then 70%. Since this mapping partitioning is disjoint, the sum of probabilities of all the events associated with the partitions must be one. Moreover, these events are mutually exclusive. We can then use *entropy*, which measures the spread of a probability distribution:

Definition 1: The **entropy** of a mapping set M for an operator o, denoted as E^o_M, is defined as:

$$E^o_M = -\sum_{j=1}^{g} \frac{|P_j|}{|M|} \log_2 \frac{|P_j|}{|M|}$$

where $P_1, \ldots, P_g$ are partitions of M with respect to o. We assume that $\frac{|P_j|}{|M|}$ is the probability that the correspondences in the j-th partition are used for query evaluation.

The *SEF* strategy chooses $u.o_{next}$ by finding an operator o in $u.T$, which has the lowest entropy (i.e., $E^o_{u.M}$). Intuitively, if the entropy is small, a large portion of the mappings in M may fall into the same partition. Hence, SEF prefers an operator whose mappings are concentrated in few partitions. For example, in Figure 7, $E^{o_1}_{u.M} = 1.53$, while $E^{o_2}_{u.M} = 1.36$. Different from *SNF*, o_2 is chosen ahead of o_1.

The implementation of *SEF* is similar to that of *SNF*, except that instead of counting the number of partitions of a candidate operator, we compute its entropy. The operator that has the lowest entropy is selected.

In Section VIII, we will evaluate the effectiveness of the above three operator selection strategies experimentally.

B. Operator Reformulation

The routine *reformulate_op*, invoked in Algorithm 2, translates a target operator o to a source operator o' through a mapping m. Since this function handles the intermediate results stored in the query plan of an e-unit, it has to handle both target schemas and source relations. We consider two classes of operators.

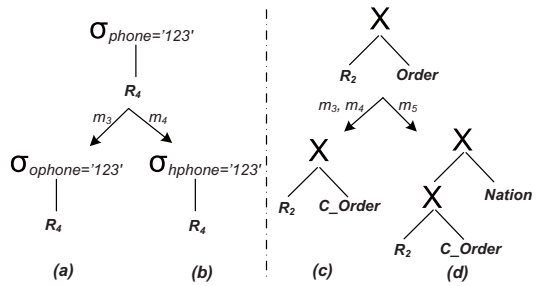


Fig. 8. Reformulation of operators.

1. o **is unary**. This means o operates on a relation R. Selection, projection, and aggregate operators belong to this class. Let A_o be a set of source attributes, which match o's attributes according to m.

[Case 1] R is a source relation (i.e., the result of the previous operators) that contains all the attributes in A_o. We obtain o' from o by replacing o's attributes with A_o. The input relation of o' is R.

[Case 2] R is a source relation that does not contain all the attributes in A_o. Then, o' is the same as Case 1. The input relation becomes $R \times R_1 \times \ldots \times R_f$ $(f \geq 1)$, where $R, R_1, \ldots, R_f$ is the minimal set of source relations that contain all attributes in A_o.

[Case 3] R is a target schema. Then o' is the same as Case 1. Its input relation is $R_1 \times \ldots \times R_f$ $(f \geq 1)$, where $R_1, \ldots, R_f$ form the minimal set of source relations that cover all attributes in A_o.

Example. Consider the reformulation of $u_4.o_{next}$ in Figure 6(a-b). The mappings in $u_4.M$, i.e., m_3 and m_4, match $phone$ to $ophone$ and $hphone$ respectively, both of which are contained in source relation R_4. Thus, $u_4.o_{next}$ is changed to $\sigma_{ophone='123'}R_4$ and $\sigma_{hphone='123'}R_4$, as shown in Figures 8(a) and (b).

2. o **is binary.** This means o operates on two relations, say, R and R'. A Cartesian product operator belongs to this class.

[Case 1] Both R and R' are source relations. Then, o' is a Cartesian product with input relations R and R'.

[Case 2] Only R (or R') is a source relation. Then o' is $R(R') \times R_1 \times \ldots \times R_f$ $(f \geq 1)$, where $R_1, \ldots, R_f$ form the minimal set of source relations that contain all source attributes for R' (R).

[Case 3] Both R and R' are target relations. All their attributes are matched to source relation(s) $R_1, \ldots, R_f$ $(f \geq 1)$ by m, and o' becomes $R_1 \times \ldots \times R_f$.

Example. Consider the reformulation of $u_3.o_{next}$ in Figure 6(c-d). After rearranging $u_3.T$, the input relations of $u_3.o_{next}$ are R_2 and $Order$, where R_2 is a source relation ($Customer$), and $Order$ is a target schema. For both m_3 and m_4, $u_3.o_{next}$ is reformulated to a Cartesian product, whose inputs are R_2 and C_Order (Figure 8(c)). For m_5, the attributes of $Order$

TABLE II
ILLUSTRATING THE TOP-k QUERY.

Node	prob.	heap	LB	UB
u_2	0.5	-	0	0.5
u_6	0.2	$t_a(0.2, 0.5)$	0.2	0.3
u_7	0.2	$t_a(0.4, 0.5), t_b(0.2, 0.3), t_c(0.2, 0.3)$	0.4	0.1
u_5	0.1	-	-	-

are contained in two source relations (C_Order and $Nation$). Therefore, $u_3.o_{next}$ is reformulated to $(R_2 \times Order) \times Nation$ (Figure 8(d)).

Analysis. The cost of SNF or SEF is polynomial to that of evaluating *partition* l times. Running *reformulate_op* once needs $O(|S|)$ times. The time cost of Algorithm 2 is polynomial to the size of the mappings. Its space complexity is linear to the size of l e-units.

VII. PROBABILISTIC TOP-k QUERIES

A probabilistic top-k query returns k tuples whose probabilities are the highest, among those with non-zero probabilities. This query is useful to a user who is only interested in the answers with sufficiently high confidence, but does not care about the exact probability values of the answer tuples. A simple way to evaluate this query is to find all potential answer tuples (e.g., by using *o-sharing*), sort these tuples according to their probabilities, and return the first k tuples. When there are many potential answer tuples, however, this approach is not efficient, since it has to compute and sort many probabilities.

Our new algorithm is able to prune non-answer tuples from computation. It also avoids evaluating the actual probabilities of the answer tuples. This is done by *partially* expanding the u-trace. Let us illustrate with the u-trace in Figure 6. Here, the total probabilities of the mappings in the leaf nodes, i.e., u_2, u_6, u_7, and u_5, are respectively 0.5, 0.2, 0.2, and 0.1 (Table II). To evaluate a top-1 query, we first traverse the u-trace to u_2, which returns no tuple. Then we go to u_6, which returns tuple t_a. The lowest probability of t_a is 0.2, which is the total probability of the mappings in u_6; the upper bound of t_a's probability is 0.5, since $u_2.M$, whose probability is 0.5, cannot contribute to t_a's probability. We continue to u_7, which returns t_a, t_b, and t_c. The minimum probability of t_a becomes 0.4, and the maximum probabilities of t_b and t_c are 0.3. We can now return t_a as the only top-1 answer, since: 1) t_b and t_c's maximum probabilities are both lower than t_a's minimum probability; and 2) any new tuple returned by the remaining e-units cannot have a probability larger than 0.1.

We now briefly describe our algorithm. (The details and cost analysis are in Algorithm 4 of Appendix B.) It first partitions the mapping set and finds the representative mappings M'. It initializes a u-trace, rooted at the e-unit u, whose query tree is q_T and mapping set is M'. It then calls the recursive function run_qt_topk on u, by considering three cases:

[Case 1] $u.T$ has no operator, i.e., $u.T$ is a set of tuples. It returns TRUE if all top-k tuples can be found, or FALSE otherwise.

TABLE III
TARGET QUERIES AND SCHEMAS USED IN THE EXPERIMENTS

ID	T	query expression
Q1	*Excel*	$\sigma_{telephone=335-1736}\ \sigma_{priority=2}$ $\sigma_{invoiceTo=Mary}$ PO
Q2	*Excel*	$\sigma_{quantity=10}\ \sigma_{itemNum=00001}$ PO $\times$ Item
Q3	*Excel*	$\sigma_{PO.orderNum=Item_1.orderNum}$ ($\sigma_{telephone=335-1736}\sigma_{itemNum_1=00001}$ PO) $\times$ ($\sigma_{Item_1.orderNum=Item_2.orderNum}$ ($Item_1 \times Item_2$))
*Q4	*Excel*	$\sigma_{itemNum_1=00001}((\sigma_{PO_1.orderNum=PO_2.orderNum}PO_1 \times PO_2) \times (\sigma_{Item_1.orderNum=Item_2.orderNum}(Item_1 \times Item_2)))$
Q5	*Excel*	COUNT($\sigma_{telephone=335-1736}\ \sigma_{company=ABC}$ $\sigma_{invoiceTo=Mary}\ \sigma_{deliverToStreet=Central}$ PO)
Q6	*Noris*	$\sigma_{telephone=335-1736}\ \sigma_{invoiceTo=Mary}$ $\sigma_{deliverToStreet=Central}$ PO
Q7	*Noris*	$\pi_{itemNum,unitPrice}\sigma_{orderNum=00001}\ \sigma_{deliverTo=Mary}\sigma_{deliverToStreet=Central}$ PO $\times$ Item
Q8	*Paragon*	$\sigma_{billTo=Mary}\ \sigma_{shipToAddress=ABC}$ $\sigma_{shipToPhone=335-1736}$ PO
Q9	*Paragon*	SUM ($\pi_{price}\sigma_{telephone=335-1736}\ \sigma_{billToAddress}$) $=ABC\sigma_{itemNum=00001}$ PO $\times$ Item
Q10	*Paragon*	COUNT ($\sigma_{invoiceTo=Mary}\ \sigma_{billToAddress}$ $=ABC$ PO $\times$ Item)

[Case 2] $u.T$ is an empty relation. It returns TRUE if all top-k tuples can be found, or FALSE otherwise.

[Case 3] Similar to Algorithm 2, it finds $u.o_{next}$; for each partition P_i of o_{next}, it executes o_{next} to obtain e-unit u_i, and recursively calls run_qt_topk on u_i. If the recursive call on any P_i returns TRUE, the function returns TRUE.

To implement the algorithm, for each tuple we store the lower (lb) and upper (ub) bounds of its probability. We use a *heap*, ordered by lb, to maintain the tuples that can be the answers. We also use two global variables: 1) LB, the lower bound probability of the tuple with the k-th highest probability in the heap; and 2) UB, the maximum probability of any tuple not in the heap.

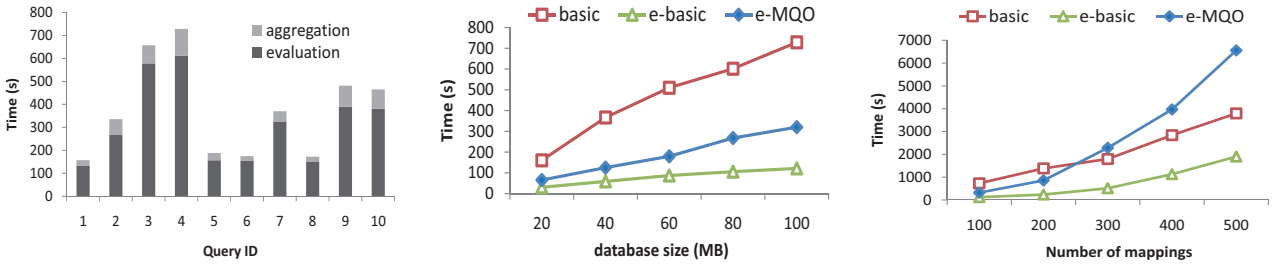
Table II shows the status of the heap, LB, and UB, after computing each e-unit. For each tuple in the heap, its lb and ub values are shown. After u_7 is computed, $UB < LB$, and the ub values of t_b and t_c are both 0.3, which is smaller than LB. Thus, the top-1 answer (t_a) can be returned without visiting u_5. In our experiments, this algorithm runs fast, especially for small values of k.

VIII. RESULTS

In Section VIII-A, we describe the experiment setup. Then we present the results in Section VIII-B.

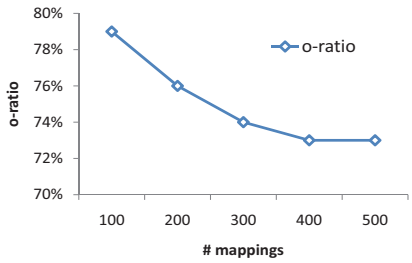
A. Setup

We use TPC-H (www.tpc.org/tpch) to generate a 100MB source instance, which contains 1M tuples about purchase orders. Its (source) schema, which we called *TPC-H*, contains 46 attributes and 8 relations. We consider three target schemas: *Excel*, *Noris*, and *Paragon*, with 48, 66, and 69 attributes respectively. They are related to purchase orders, and are provided by COMA++ (dbs.uni-leipzig.de/research/coma). The default target schema is *Excel*.



(a) Breakdown of t_q (b) t_q vs. database size (c) t_q vs. mappings

Fig. 10. Performance of simple solutions



(a) o-ratio vs. # mappings

Fig. 9. Overlap of mappings

We choose COMA++ as the schema matcher. It returns 34, 18, and 31 correspondences for *Excel*, *Noris*, and *Paragon* with *TPC-H* respectively. As COMA++ requires the input schema to be in XML format, we transform the relational *TPC-H* schema into XML by the method in [22]. Based on the similarity scores associated with the correspondences, we use a bipartite matching algorithm [10], [9] to generate $h = 100$ possible mappings.

The target schemas, in XML, are changed to a relational form by the method in [23]. Two relational schemas, namely $PurchaseOrder$ (PO) and $Item$, are yielded. We define 10 target queries: Q1-Q5 for *Excel*; Q6-Q7 for *Noris*; and Q8-Q10 for *Paragon* (Table III). Each query can contain selection, projection, Cartesian product, COUNT, and SUM, on more or more tables. By default, Q4 is the target query, and *SEF* is used for *o-sharing*. We implement *e-MQO* with the solution in [12]. We use t_q to denote the running time of target query T_q. Each data point is an average of 50 runs.

Our algorithms, implemented in C++, are run on a PC with Intel Core Duo 2.93GHz CPU and 3G RAM. Their source codes can be found in www.cs.hku.hk/~jgong/urm.

B. Results

1. Overlap of possible mappings. To measure the overlap among the possible mappings, 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 common correspondences over the number of all distinct correspondences for m_i and m_j. We also define the *o-ratio* of a mapping set M as the average of the *o-ratio* among all pairs of mappings in M. In our experiment, the o-ratio of the mappings between TPC-H and *Excel*, *Noris*, and *Paragon* are respectively 79%, 68%, and 72%. Figure 9(a) shows that the o-ratio for TPC-H and *Excel* is between 73%-79% for a wide range of number of mappings. Hence, these mappings are highly similar. As we show later, *q-sharing* and *o-sharing* exploit this property and yield a higher query performance.

2. Simple solutions. We next analyze the time distribution of the two phases in *basic*: query evaluation and tuple aggregation. Figure 10(a) shows that the computation time is dominated by query evaluation. This is because *basic* answers a target query with every possible mapping separately. For all the queries examined, the fraction of the query evaluation time is more than 80%. We next consider *e-basic* and *e-MQO*, which enhances the query evaluation phase of *basic*.

Figure 10(b) shows the performance of these solutions under different database sizes. Both *e-basic* and *e-MQO* outperform *basic*, because they evaluate distinct source queries, which are fewer than those evaluated by *basic*. Moreover, *e-basic* is faster than *e-MQO*. Note that *e-MQO* generates an optimal source query plan, which executes the smallest number of operators. However, the plan generation process is extremely expensive. Thus, *e-MQO* is slower than *e-basic*, which does not generate any query plan. In Figure 10(c), we test their performance under different number of mappings. The evaluation time of *e-MQO* rises sharply with the number of mappings; when $|M| > 300$, *e-MQO* is even worse than *basic*. With a larger number of mappings, more source queries are produced, which results in a lot of time for generating a query plan. Thus, *e-MQO* does not scale well with the mapping set size. Since *e-basic* is the best basic solution, we will compare it with *q-sharing* and *o-sharing* in the rest of this section.

3. Query performance. Figure 11(a) shows the running time of Q1 to Q10. We see that *q-sharing* is better than *e-basic*, with an average improvement of 16%. Recall that *e-basic* generates h source queries and finds the distinct ones among them. On the other hand, *q-sharing* identify the representative mappings, which are often a small portion of all mappings. For example, for Q1 that contains three operators, there are only 12 representative mappings. These mappings, which can be quickly obtained by the use of the *partition* routine, are then used to derive the distinct source queries. Since *q-sharing* does not involve the derivation of all source queries, it is faster than *e-basic*.

We also see that *o-sharing* performs better than *q-sharing*. While *q-sharing* can only combine identical mappings, *o-sharing* allows the sharing of query effort, even if the map-

TABLE IV
OPERATOR SELECTION STRATEGIES.

Strategy	time (s)	# source operators
Random	215	433
SNF	58	135
SEF	55	132
e-MQO	320	112

pings are not exactly the same. Thus, *o-sharing* is more flexible than *q-sharing*, and thus outperform it.

4. Effect of database size. Figure 11(b) shows t_q (query evaluation time) for different values of $|D|$ on the default query $Q4$. Again, *o-sharing* outperforms *q-sharing*, which is faster than *e-basic*. Moreover, while the performance times of all these methods increase the size of database, the increase rate of *o-sharing* is the slowest. Thus, *o-sharing* scales well with the database size.

5. Effect of the number of mappings. Figure 11(c) shows the performance of Q4 over 100-500 mappings. Notice that *e-basic* and *q-sharing* are quite sensitive to the number of mappings. When the number of mappings increases, the number of representative mappings, as well as that of distinct source queries, increases rapidly. Thus they do not scale well with the mapping set size. Although *o-sharing* also needs to compute more e-units when more mappings are considered, it is less sensitive to the number of mappings. This is because it enjoys a higher degree of sharing among query operators than *q-sharing*. Therefore, increasing the mapping set size brings less impact to *o-sharing*.

6. Effect of query size. Figure 11(d) shows the performance of queries with 1-5 selection operators on different attributes. Again, *q-sharing* and *o-sharing* perform better than *e-basic*, since they do not generate all the h source queries. When a query contains more operators ($\geq$2), *o-sharing* performs better, since it uses operator-level sharing on the mappings; by contrast, *q-sharing* obtains more representative mappings, and has to execute more distinct source queries.[2] Figure 11(e) shows the performance of queries with 1 to 3 self-joins on the PO schema. The results are similar to Figure 11(d). If a query contains more relations, more target attributes need to be handled, yielding more source queries and operators. Here, when the number of Cartesian products is two or more, *o-sharing*, which allows more sharing of query effort than that of *q-sharing* or *e-basic*, performs the best.

7. Operator selection strategies. Next, we study the operator selection strategies mentioned in Section VI-A: *Random*, *SNF*, and SEF. Figure 11(f) shows their performance for the queries on *Excel*. Observe that both *SNF* and *SEF* perform much better than *Random*. Also, *o-sharing* using *SEF* is faster than when *SNF* is used. This can be explained by Table IV, which shows the query evaluation time and the number of source operators executed for Q4. We also include the results of *e-MQO*, which yields an optimal query plan, or equivalently, the smallest number of source query operators. We see that *Random* executes more operators than both *SNF* and *SEF* do. This is because *Random* ignores mapping information; it makes a poor choice by choosing target operators that leads to many source operators. The number of operators for *SNF* and *SEF* are both close to the optimal (i.e., the number of operators required by *e-MQO*. However, *SEF* needs fewer operators than *SNF*. While *SNF* does not use the information about the number of mappings in each partition, *SEF* uses it effectively by computing the entropy function. Hence, *SEF* makes a better decision and helps the target query to run faster than when *SNF* is used. We conclude that o-sharing, when used with *SEF*, often performs the best in most of our experiments.

8. Top-k query. Figures 12(a)-12(c) compare the basic solution (which uses *o-sharing* to find probabilities of all tuples), and our new *top-k* algorithm. We vary k from 1 to 20, for Q4 (*Excel*), Q7 (*Noris*), and Q10 (*Paragon*). When a small value of k is used, *top-k* runs faster, since it can stop before completely exploring the u-trace. For Q10 (Figure 12(c)), at $k = 10$, top-k performs about the same as *o-sharing*. This is because Q10 returns no more than 10 distinct tuples; thus, *top-k* cannot stop evaluation earlier than *o-sharing* when $k \geq 10$.

[2]When there is only 1 operator, no operator sharing for *o-sharing* is allowed; it only enables query-level sharing. The extra overhead of maintaining the u-trace explains why *o-sharing* needs slightly more time than *q-sharing*.

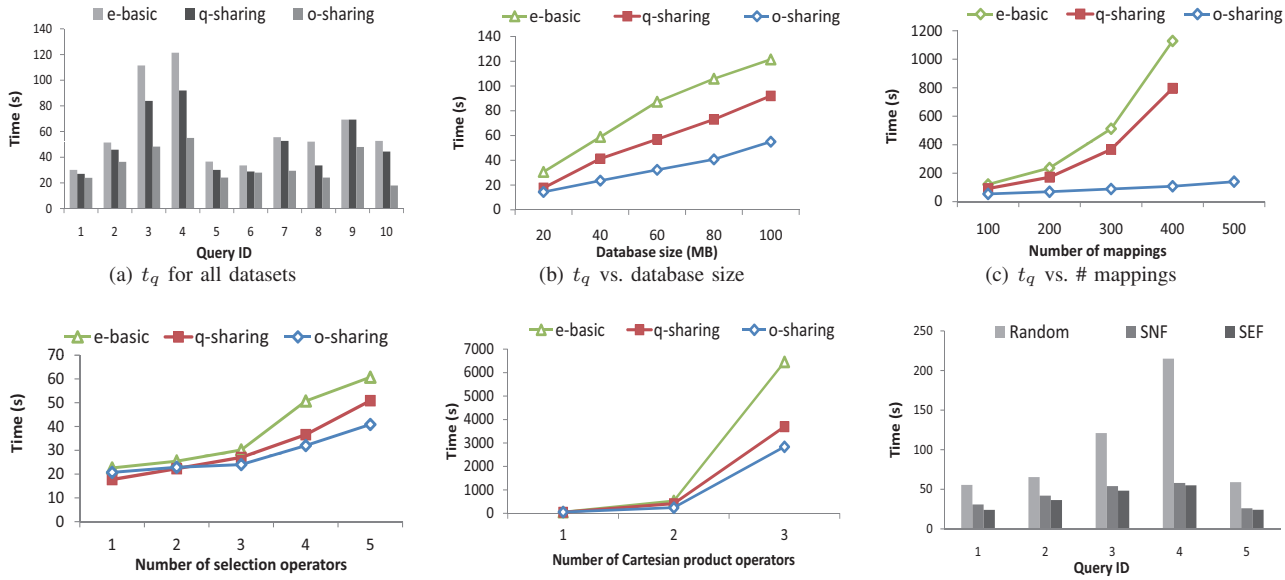
IX. CONCLUSIONS

We studied efficient query evaluation on uncertain schema matching. We developed *q-sharing* and *o-sharing*, which exploit the similarity among different mappings. We examined two metrics for choosing target operators in *o-sharing*. Our experiments show that *o-sharing* performs the best when used with SEF. We also effectively extended *o-sharing* to support probabilistic top-k queries. In the future, we will study the use of *o-sharing* to support other complex queries (e.g., set operators, subqueries, and recursive queries), and design data structures to facilitate o-sharing evaluation.

Acknowledgment. Reynold Cheng was supported by the Research Grants Council of Hong Kong (GRF Project 711309E). Jiefeng Cheng was supported by the Shenzhen New Industry Development Fund no. CXB201005250021A.

REFERENCES

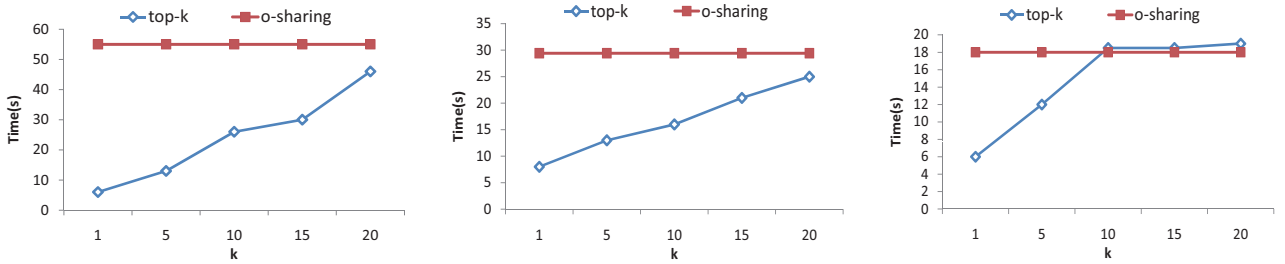
[1] E. Rahm and P. Bernstein, "A survey of approaches to automatic schema matching," *VLDB J.*, vol. 10, no. 4, pp. 334–350, 2001.
[2] M. Lenzerini, "Data integration: a theoretical perspective," in *PODS*, 2002.
[3] L. Popa, Y. Velegrakis, M. A. Hernández, R. J. Miller, and R. Fagin, "Translating web data," in *VLDB*, 2002.
[4] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.
[5] H. Do et al, "COMA: a system for flexible combination of schema matching approaches," in *VLDB*, 2002.
[6] A. Doan et al, "Reconciling schemas of disparate data sources: a machine-learning approach," in *SIGMOD*, 2001.
[7] A. Das Sarma, X. Dong, and A. Halevy, "Bootstrapping pay-as-you-go data integration systems," in *SIGMOD*, 2008.
[8] X. L. Dong, A. Halevy, and C. Yu, "Data integration with uncertainty," *The VLDB Journal*, vol. 18, no. 2, pp. 469–500, 2009.
[9] A. Gal, "Managing uncertainty in schema matching with top-k schema mappings," in *J. Data Semantics VI*, 2006.
[10] R. Cheng, J. Gong, and D. W. Cheung, "Managing uncertainty of XML schema matching," in *ICDE*, 2010.



(a) t_q for all datasets (b) t_q vs. database size (c) t_q vs. # mappings

(d) t_q vs. # selection operators (e) t_q vs. # Cartesian product operators (f) Operator selection strategies

Fig. 11. *e-basic*, *q-sharing*, and *o-sharing*



(a) t_q vs. top-k queries (Q4) (b) t_q vs. top-k queries (Q7) (c) t_q vs. top-k queries (Q10)

Fig. 12. Top-k Query Performance

[11] J. Gong, R. Cheng, and D. W. Cheung, "Efficient management of uncertainty in xml schema matching," *VLDB J.*, 2011.

[12] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *SIGMOD*, 2007.

[13] M. Friedman, A. Levy, and T. Millstein, "Navigational plans for data integration," in *AAAI*, 1999.

[14] P. Agrawal, A. Das Sarma, J. Ullman, and J. Widom, "Foundations of uncertain-data integration," in *VLDB*, 2010.

[15] P. A. Bernstein, S. Melnik, and J. E. Churchill, "Incremental schema matching," in *VLDB*, 2006.

[16] M. Soliman, I. Ilyas, and K. Chang, "Top-k query processing in uncertain databases," in *ICDE*, 2007.

[17] C. Re, N. Dalvi, and D. Suciu, "Efficient top-k query evaluation on probabilistic data," in *ICDE*, 2007.

[18] F. Li, K. Yi, and J. Jestes, "Ranking distributed probabilistic data," in *SIGMOD*, 2009.

[19] R. Cheng et al., "Evaluating probability threshold k-nearest-neighbor queries over uncertain data," in *EDBT*, 2009.

[20] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *SIGMOD*, 2000.

[21] E. S. Claude and W. Warren, *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[22] D. Lee et al, "NeT & CoT: translating relational schemas to XML schemas using semantic constraints," in *CIKM*, 2002.

[23] J. Shanmugasundaram et al, "Relational databases for querying XML documents: limitations and opportunities," in *VLDB*, 1999.

APPENDIX A
Q-SHARING (SECTION 3)

Algorithm 3 (*partition*) shows the details of constructing a partitioning tree, for a given set of mappings M and target query q_T. It first creates a new partition tree rooted at r (Step 1), then it calls the recursive function *put* to assign each mapping in M to an appropriate bucket in the partition tree (Steps 2-5).The recursive function *put* assigns the mapping m into the partition tree node n at the k-th level. The base case is when node n is a leaf node (Step 1), which implies that m is ready to be assigned to a partition. Then m is deposited to the bucket and the algorithm exists (Step 2). For the other cases, i.e., n is not a bucket, it checks if there exists a mapping m', such that m' and m match a_k, the k-th target attribute, with the same source attribute a'_k: 1) if m' exists, it can find a node n' of n's child, such that the edge (n, n') is labeled with a'_k. Then, it recursively calls the function *put* to assign m to the next level (Steps 6-9); and 2) if m' does not exist, then it creates a new node or bucket, depending on whether a_k is the last target attribute (Steps 12-18), and then recursively calls the function *put* to assign m to the next level (Step 18). Finally the algorithm returns all the buckets in the partition tree as partitions of mappings (Step 5).

Algorithm 3 The *partition* Algorithm

Input: mapping set M, target query q_T
Output: partitions of M

1: $ptree \leftarrow$create a new partition tree rooted at r
2: **for all** $m \in M$ **do**
3: $put(m, r, 1)$
4: **end for**
5: return all the buckets in $ptree$

procedure $put(mapping\ m, node\ n, integer\ k)$

1: **if** n is a bucket **then**
2: put m in n
3: **else**
4: $a_k \leftarrow$ the k-th attribute in q_T
5: **for all** out-going edge e of n **do**
6: **if** e is labeled as a'_k, $(a'_k, a_k) \in m$ **then**
7: let n' be the child of n connected by e
8: $put(m, n', k+1)$
9: **end if**
10: **end for**
11: **if** the above e does *not* exist **then**
12: create an edge e labeled with a'_k, $(a'_k, a_k) \in m$
13: **if** a_k is the last target attribute **then**
14: create a new bucket n' connected with e
15: **else**
16: create a new node n' for a_{k+1} connected with e
17: **end if**
18: $put(m, n', k+1)$
19: **end if**
20: **end if**

APPENDIX B
THE TOP-K ALGORITHM

Algorithm 4 first partitions the mapping set and finds out the representative mappings M' (Steps 1-2). Then it initializes a u-trace, rooted at the e-unit u_1, whose query tree is q_T and mapping set is M' (Step 3). Afterwards, it creates a heap, and set the initial values for LB and UB (Steps 4-5). Next it calls the recursive function run_qt_topk to find the candidate query answers and put them in the heap (Step 6), and finally returns the top-k tuples from the heap (Step 7).

The function run_qt_topk works as follows. It first examines u's query tree, and there are three cases.

Case 1: (Steps 1-2) $u.T$ has no operator, which means that $u.T$ is a single relation containing a set of tuples as possible query answers. Then it calls the function $decide\_result$ to process these tuples, and decides whether the algorithm can safely stop, i.e., all the top-k tuples are found. In detail, for each distinct tuple r, it first checks if r is already in the heap, then it increases the lower bound of r accordingly (Steps 3-4); else, if r is a potential top-k answer, i.e., $UB > LB$, then it inserts r into the heap, and set its lb and ub properly (Steps 6-7). After all the tuples are processed, it updates LB and UB based on definition (Steps 10-11). Then it decides whether all the top-k tuples are found, by checking the following two conditions: 1) for each tuple r' in the heap whose lower bound probability ranks larger than k, $r'.ub \leq LB$ (which implies that r' *cannot* be a top-k answer), and 2) whether $UB \leq LB$ (which implies that any new tuple returned by the remaining e-units *cannot* be a top-k answer). If the above two conditions satisfy, then the function returns TRUE, which implies that the all the top-k tuples are found (Steps 12-16).

Algorithm 4 *top-k*

Input: mapping set M, target query q_T, source instance D, k
Output: all the top-k query answers

1: $P_1, \ldots, P_n \leftarrow partition(q_T, M)$
2: $M' \leftarrow represent(P_1, \ldots, P_n)$
3: $u_1 \leftarrow init\_u\_trace(q_T, M')$
4: $heap \leftarrow$ new heap of tuples, sorted by tuple's lb
5: $LB \leftarrow 0$, $UB \leftarrow 1$
6: $run\_qt\_topk(u_1, D)$
7: return the top-k tuples in $heap$

function *run_qt_topk(E-Unit u, Instance D)*
Output: TRUE/FALSE (i.e., if all top-k results found, or not)

1: **if** $u.T$ has no operator **then**
2: return $decide\_result(u)$
3: **else if** $u.T$ contains empty relation **then**
4: $u.T \leftarrow \emptyset$
5: return $decide\_result(u)$
6: **else**
7: $(P_1, \ldots, P_g) \leftarrow next(u)$
8: $reorder\_op(u, u.o_{next})$
9: **for all** $P_i \in P_1, \ldots, P_g$ **do**
10: $m \leftarrow$ arbitrary mapping in P_i
11: $o' \leftarrow reformulate\_op(o_{next}, m)$,
12: $ans_S \leftarrow run\_qs(o', D)$
13: $u_i.T \leftarrow create\_qtree(u.T, u.o_{next}, ans_S)$
14: $u_i.M \leftarrow P_i$
15: **if** $run\_qt\_topk(u_i, D)$ = TRUE **then**
16: delete u, return TRUE
17: **end if**
18: **end for**
19: delete u, return FALSE
20: **end if**

function *decide_result(E-Unit u)*
Output: TRUE/FALSE (i.e., if all top-k results decided, or not)

1: remove duplicate tuples in $u.T$
2: **for all** $r \in u.T$ **do**
3: **if** $\exists r' \in heap$, $r' = r$ **then**
4: $r'.lb \leftarrow r'.lb + \sum_{m \in u.M} Pr(m)$
5: **else if** $UB > LB$ **then**
6: $r.ub \leftarrow UB$, $r.lb \leftarrow \sum_{m \in M'} Pr(m)$
7: $heap.push(r)$
8: **end if**
9: **end for**
10: $UB \leftarrow UB - \sum_{m \in u.M} Pr(m)$
11: $LB \leftarrow r'.lb$, r' is the k-th (or the last) tuple in $heap$
12: **if** $\forall r' \in heap[k+1, |heap|]$, $r'.ub \leq LB$ **and** $UB \leq LB$ **then**
13: delete u, return TRUE
14: **else**
15: delete u, return FALSE
16: **end if**

Case 2: (Steps 3-5) $u.T$ contains empty relation, then it also calls the function $decide\_result$, after replacing $u.T$ with an empty relation; in this case, $decide\_result$ will skip Steps 1-9, and only update UB.

Case 3: (Steps 7-20) Similar to Algorithm 2, it finds $u.o_{next}$ and reorder $u.T$; for each partition P_i of o_{next}, it reformulates o_{next} and computes it to obtain another e-unit u_i, and then recursively calls the function run_qt_topk on u_i. Notice that if the recursive call on any partition returns TRUE, which means the top-k tuples are found, the recursion stops immediately.

Complexity. In the worst case, all the e-units will be visited, so its complexity will be the same as the *o-sharing* algorithm. However, our experiments found that the performance of this algorithm is better, especially for small values of k. An example run of this algorithm can be found in Table II.