| Title | Evaluating probabilistic queries over uncertain matching |
| :---: | :--- |
| Author（s） | Cheng，R；Gong，J；Cheung，DWL；Cheng，J |
| Citation | The IEEE 28th International Conference on Data Engineering <br> （ICDE 2012），Washington，D．C．，1－5 April 2012．In International <br> Conference on Data Engineering Proceedings，2012，p．1096－－ <br> 1107 |
| Issued Date | 2012 |
| URL | http：／／hdl．handle．net／10722／164909 |
| Rights | International Conference on Data Engineering Proceedings． <br> Copyright © IEEE Computer Society． |

# Evaluating Probabilistic Queries over Uncertain Matching 

Reynold Cheng ${ }^{\dagger}$, Jian Gong ${ }^{\dagger}$, David W. Cheung ${ }^{\dagger}$, and Jiefeng Cheng ${ }^{\ddagger}$<br>${ }^{\dagger}$ Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong<br>\{ckcheng, jgong, dcheung\}@cs.hku.hk<br>${ }^{\ddagger}$ Shenzhen Institute of Advanced Technology, China<br>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 | $\ldots$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $t_{1}$ | Alice | 123 | 789 | aaa | $h k$ |  |
| $t_{2}$ | Bob | 456 | 123 | $b b b$ | $h k$ | $\ldots$ |
| $t_{3}$ | 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:

$$
q_{0}: \pi_{a d d r} \sigma_{p h o n e=^{\prime} 123^{\prime}} \text { Person }
$$

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 $b b b$. 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 $\left(t_{i}, p_{i}\right)$, where

| ID | correspondences | prob. |
| :---: | :--- | :---: |
| $m_{1}$ | (cname, pname), (ophone, phone), <br> (oaddr, addr), (name, nation), ... | 0.3 |
| $m_{2}$ | (cname, pname), (ophone, phone), <br> (oaddr, addr), (name, nation), .. | 0.2 |
| $m_{3}$ | (cname, pname), (ophone, phone), <br> (haddr, addr), (name, nation), .. | 0.2 |
| $m_{4}$ | (cname, pname), (hphone, phone), <br> (haddr, addr), (name, nation), .. | 0.2 |
| $m_{5}$ | (cname, sname), (ophone, phone), <br> (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 $\{(a a a, 0.5),(h k, 0.5)\}$.

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 ecommerce 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 $\sigma_{\text {ophone }={ }^{\prime} 123^{\prime}}$ 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 userspecified. 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 $a_{S}\left(a_{T}\right)$ 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 $\operatorname{Pr}\left(m_{i}\right)$ to be correct. If $e_{i}$ is the event that $m_{i}$ is correct, then all $e_{i} \mathrm{~s}^{\prime}$ are mutually exclusive. Thus, $\sum_{i=1}^{h} \operatorname{Pr}\left(m_{i}\right)=1$.

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 $i \in[1, h]$ |
| $\operatorname{Pr}\left(m_{i}\right)$ | Probability that $m_{i}$ is correct |
| $q_{T}$ | Target query, with $l$ operators |
| $q_{S_{i}}$ | Source query for mapping $m_{i}$ |
| $(t, \operatorname{Pr}(t))$ | 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 $\operatorname{Pr}\left(m_{i}\right)$. 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:

$$
\pi_{p h o n e} \sigma_{a d d r=^{\prime} a a a^{\prime}} \text { Person }
$$

using the possible mappings in Figure 3. For $m_{1}$, the target query $q_{T}$ can be reformulated to:

$$
q_{S_{1}}: \pi_{o p h o n e} \sigma_{o a d d r=^{\prime} a a a^{\prime}} \text { Customer }
$$

By evaluating $q_{S_{1}}$ on the source instance in Figure 2, 123 and 456 are returned, each with probability $\operatorname{Pr}\left(m_{1}\right)=0.3$. 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, $e-M Q O$ 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 $e-M Q O$ 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. Querr-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 $M^{\prime}$ 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 $M^{\prime}$, 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 $M^{\prime} .{ }^{1}$

```
Algorithm 1 q-sharing
Input: mapping set \(M\), target query \(q_{T}\), source instance \(D\)
Output: answer of \(q_{T}\)
    \(P_{1}, \ldots, P_{f}=\operatorname{partition}\left(q_{T}, M\right)\)
    \(M^{\prime} \leftarrow \operatorname{represent}\left(P_{1}, \ldots, P_{f}\right)\)
    return basic \(\left(q_{T}, M^{\prime}, D\right)\)
```

Example. Consider the evaluation of the following query:

$$
q_{1}: \pi_{p n a m e} \sigma_{a d d r={ }^{\prime} a b c^{\prime}} \text { Person }
$$

over the mappings in Figure 3. In Algorithm 1, the mappings are classified into: $P_{1}=\left\{m_{1}, m_{2}\right\}, P_{2}=\left\{m_{3}, m_{4}\right\}$, and $P_{3}=\left\{m_{5}\right\}$. 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 $e-M Q O$. 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

[^0]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 $a_{k}^{\prime}$, 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 $a_{k}^{\prime}$ and the correspondence ( $a_{k}^{\prime}, a_{k}$ ) is in $m$, it traverses the partition tree according to $e$. Otherwise, a new edge labeled $a_{k}^{\prime}$ 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 $a d d r$ 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 $a d d r$ with oaddr. Notice that $m_{3}$ is assigned to another bucket $B_{2}$, since it matches $a d d r$ 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, $e-M Q O$, 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


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 $u . o_{n e x t}$, 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:

$$
q_{2}:\left(\sigma_{\text {addr }=^{\prime} h k^{\prime}} \sigma_{\text {phone }=^{\prime} 123^{\prime}} \text { Person }\right) \times \text { Order }
$$

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 $\sigma_{a d d r=^{\prime} h k^{\prime}}$ (underlined). Suppose now $\sigma_{a d d r=^{\prime} h k^{\prime}}$ is executed. This is done by first rearranging it to be executed on Person, and then reformulated according to $u_{1} \cdot M$ (We will explain the operator selection strategies in Section VI-B). Notice that the target attribute, $a d d r$, corresponds to the same source attribute $o a d d r$ 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 $u_{1} \cdot T$ :

- For $m_{1}$ and $m_{2}$, execute $\sigma_{o a d d r=^{\prime} h k^{\prime}}$ Customer, and produce source relation $R_{2}$, which is an empty relation.
- For $m_{3}, m_{4}$, and $m_{5}$, execute $\sigma_{h a d d r='}{ }^{\prime} h k^{\prime} C u s t o m e r$, and produce source relation $R_{3}$, which contains $\left\{t_{1}, t_{2}\right\}$.

The above execution produces two e-units, called $u_{2}$, and $u_{3}$. Figure 5 shows the components of these two eunits. Notice that $u_{2} \cdot T\left(u_{3} \cdot T\right)$ differs from $u_{1} \cdot T$, in which $\sigma_{a d d r={ }^{\prime} h k^{\prime}}$ Person is replaced by $R_{2}\left(R_{3}\right)$. Also, $u_{2} \cdot M=$ $\left\{m_{1}, m_{2}\right\}$, while $u_{3} \cdot M=\left\{m_{3}, m_{4}, m_{5}\right\}$. Moreover, the number of times next-op ( $\sigma_{\text {addr }}=^{\prime} h k^{\prime}$ ) 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 utrace consists of $u_{1}$, where $u_{1} \cdot T$ is created from $q_{2}$. We first evaluate $u_{1} . o_{\text {next }}$, 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 $u_{2} \cdot T$ must be empty. Hence, $u_{2} \cdot T$ 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 nextop. The detail of selecting operators will be explained in Section VI-A. An e-unit $u_{4}$ is produced, with $u_{4} \cdot M=\left\{m_{3}, m_{4}\right\}$ (Figure 6(b)). By considering $\sigma_{\text {phone }=^{\prime} 123^{\prime}}$ on $u_{4} \cdot T$, 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 $6(\mathrm{c})$ ). Since $u_{6}\left(u_{7}\right)$ does not contain any operator, $R_{6}\left(R_{7}\right)$ is the answer for $m_{3}\left(m_{4}\right)$. 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 $M^{\prime}$ (Steps 1-2). It initializes a u-trace by creating an e-unit, $u_{1}$, where $u_{1} \cdot T$ is the query plan of $q_{T}$, and $u_{1} \cdot M=M^{\prime}$ (Step 3). It calls a recursive function run_qt to compute the answers $\left(R_{1}, \ldots, R_{g}\right)$ 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, $a n s_{S}$ and $a n s_{T}$, 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
    \(P_{1}, \ldots, P_{f} \leftarrow \operatorname{partition}\left(q_{T}, M\right)\)
    \(M^{\prime} \leftarrow \operatorname{represent}\left(P_{1}, \ldots, P_{f}\right)\)
    \(u_{1} \leftarrow i n i t \_u \_t r a c e\left(q_{T}, M^{\prime}\right)\)
    \(R_{1}, \ldots, R_{g} \leftarrow\) run_qt \(\left(u_{1}, D\right)\)
    return aggregate \(\left(R_{1}, \ldots, R_{g}\right)\)
function run_qt(e-Unit \(u\), source instance \(D\) )
Output: all the query answers for \(u\) from \(D\)
    \(a n s_{S} \leftarrow \emptyset, a n s_{T} \leftarrow \emptyset\)
    if \(u . T\) is a relation then
        for all \(t \in u . T\) do
            \(a n s_{T} \leftarrow a n s_{T} \cup\left(t, \sum_{m \in u . M} \operatorname{Pr}(m)\right)\)
        end for
        delete \(u\)
        return \(a n s_{T}\)
    else if \(u . T\) contains an empty relation then
        \(a n s_{T} \leftarrow\left(\theta, \sum_{m \in u . M} \operatorname{Pr}(m)\right)\)
        delete \(u\)
        return \(\mathrm{ans}_{T}\)
    else
        \(\left(P_{1}, \ldots, P_{g}\right) \leftarrow \operatorname{next}(u)\)
        reorder_op \(\left(u, u . o_{n e x t}\right)\)
        for all \(\overline{P_{i}} \in P_{1}, \ldots, P_{g}\) do
            \(m \leftarrow\) an arbitrary mapping in \(P_{i}\)
            \(o^{\prime} \leftarrow\) reformulate_op \(\left(u . o_{n e x t}, m\right)\)
            \(a n s_{S} \leftarrow r u n \_q s\left(o^{\prime}, D\right)\)
            \(u_{i} \cdot T \leftarrow\) create_qtree \(\left(u . T\right.\), u.onext, ans \(\left._{S}\right)\)
            \(u_{i} \cdot M \leftarrow P_{i}\)
            \(R_{i} \leftarrow\) run_qt \(\left(u_{i}, D\right)\)
        end for
        delete \(u\)
        return \(R_{1}, \ldots, R_{g}\)
    end if
```

has probability $\operatorname{Pr}\left(m_{3}\right)$. After Step 5, all answers in $u$ are kept in $a n s_{T}$. Then, $u$ is deleted, and $a n s_{T}$ 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 $\theta$, and its probability in $a n s_{T}$. Steps 10-11 delete $u$ and return $a n s_{T}$. In Figure 6(a), the probability of the empty result for $u_{2}$ is $\operatorname{Pr}\left(m_{1}\right)+\operatorname{Pr}\left(m_{2}\right)$.
Case 3 (Steps 12-24): The answer of $u . T$ is evaluated as follows:
(i) Call function next (Step 13). This function finds $u . o_{\text {next }}$ and returns the partition of mappings with respect to $u . o_{\text {next }}$. (ii) Rearrange $u . o_{\text {next }}$ in the query tree of $u$, so that it is is ready to be computed (Step 14). In Figure 5, we push $\sigma_{a d d r={ }^{\prime} h k}$ down one level of $u_{1} \cdot T$, so that it can be evaluated on Person.
(iii) Evaluate $u . o_{n e x t}$, 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 $u . o_{\text {next }}$ to a source operator $o^{\prime}$ (Steps 16-17). Using the run_qs routine, $o^{\prime}$ 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 $u . o_{n e x t}$ 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 $\sigma_{a d d r={ }^{\prime} h k^{\prime}}$ in $u_{1} \cdot T$ produces $u_{2}$ and $u_{3}$. Note that the query plans of these e-units are created by replacing $\sigma_{a d d r=^{\prime} h k^{\prime}}$ with query results $R_{2}$ and $R_{3}$, in $u_{1} \cdot T$. (iv) Delete $u$ and return the results for $u$ 's partitions.

(a)

(b)

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 $u \cdot T$, 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, $S N F$, 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, $S N F$ 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 $S N F$, 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 $u . o_{n e x t}$. Finally, next returns the partition for $u . o_{n e x t}$.

The main problem of $S N F$ 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, $S N F$ 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 $E_{M}^{o}$, is defined as:

$$
E_{M}^{o}=-\sum_{j=1}^{g} \frac{\left|P_{j}\right|}{|M|} \log _{2} \frac{\left|P_{j}\right|}{|M|}
$$

where $P_{1}, \ldots, P_{g}$ are partitions of $M$ with respect to $o$. We assume that $\frac{\left|P_{j}\right|}{|M|}$ is the probability that the correspondences in the $j$-th partition are used for query evaluation.

The SEF strategy chooses $u . o_{n e x t}$ by finding an operator $o$ in $u . T$, which has the lowest entropy (i.e., $E_{u . M}^{o}$ ). 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, $E_{u . M}^{o_{1}}=1.53$, while $E_{u . M}^{o_{2}}=1.36$. Different from $S N F, o_{2}$ is chosen ahead of $o_{1}$.

The implementation of $S E F$ is similar to that of $S N F$, 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 $o^{\prime}$ through

(a)

(c)
(d)

Fig. 8. Reformulation of operators.
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.

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 $o^{\prime}$ from $o$ by replacing $o$ 's attributes with $A_{o}$. The input relation of $o^{\prime}$ is $R$.
[Case 2] $R$ is a source relation that does not contain all the attributes in $A_{o}$. Then, $o^{\prime}$ 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 $o^{\prime}$ 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 $u_{4} \cdot o_{\text {next }}$ in Figure $6(\mathrm{a}-\mathrm{b})$. The mappings in $u_{4} \cdot M$, 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, $u_{4}$. onext is changed to $\sigma_{\text {ophone }=^{\prime} 123^{\prime}} R_{4}$ and $\sigma_{\text {hphone }=^{\prime} 123^{\prime}} R_{4}$, as shown in Figures 8(a) and (b).
2. $o$ is binary. This means $o$ operates on two relations, say, $R$ and $R^{\prime}$. A Cartesian product operator belongs to this class.
[Case 1] Both $R$ and $R^{\prime}$ are source relations. Then, $o^{\prime}$ is a Cartesian product with input relations $R$ and $R^{\prime}$.
[Case 2] Only $R$ (or $R^{\prime}$ ) is a source relation. Then $o^{\prime}$ is $R\left(R^{\prime}\right) \times R_{1} \times \ldots \times R_{f}(f \geq 1)$, where $R_{1}, \ldots, R_{f}$ form the minimal set of source relations that contain all source attributes for $R^{\prime}(R)$.
[Case 3] Both $R$ and $R^{\prime}$ are target relations. All their attributes are matched to source relation(s) $R_{1}, \ldots, R_{f}(f \geq 1)$ by $m$, and $o^{\prime}$ becomes $R_{1} \times \ldots \times R_{f}$.
Example. Consider the reformulation of $u_{3} . o_{\text {next }}$ in Figure $6(\mathrm{c}-\mathrm{d})$. After rearranging $u_{3} \cdot T$, the input relations of $u_{3} . o_{\text {next }}$ are $R_{2}$ and Order, where $R_{2}$ is a source relation (Customer), and Order is a target schema. For both $m_{3}$ and $m_{4}, u_{3} . o_{\text {next }}$ 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, $u_{3} . o_{\text {next }}$ is reformulated to $\left(R_{2} \times O r d e r\right) \times$ 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 utrace 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 $u_{2} \cdot M$, 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 $M^{\prime}$. It initializes a u-trace, rooted at the e-unit $u$, whose query tree is $q_{T}$ and mapping set is $M^{\prime}$. 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 | $\begin{aligned} & \sigma_{\text {telephone }}=335-1736 \sigma_{\text {priority }}=2 \\ & \sigma_{\text {invoiceT }}=\text { Mary } P \text { PO } \end{aligned}$ |
| Q2 | Excel | $\sigma_{\text {quantity }=10} \sigma_{\text {item }}$ Num $=00001 \mathrm{PO} \times$ Item |
| Q3 | Excel | ```\mp@subsup{\sigma}{PO.orderNum=Item}{1} 335-1736 泣emNum ( }\mp@subsup{\sigma}{\mp@subsup{Item}{1}{}}{}\cdot\mathrm{ order Num= Item}2..orderNum (Item}1\times\mp@subsup{Item}{2}{\prime})``` |
| *Q4 | Excel | $\begin{aligned} & \sigma_{\text {item } N u m_{1}}=00001\left(\left(\sigma_{P O_{1}} . \text { order } N u m=P O_{2} .\right.\right. \\ & \text { order } \left.N u m P O_{1} \times P O_{2}\right) \times\left(\sigma_{\text {Item }}^{1} . \text { order } N u m\right. \\ & \left.\left.=\text { Item }_{2} . \text { order } N u m\left(\text { Item }_{1} \times \text { Item }_{2}\right)\right)\right) \end{aligned}$ |
| Q5 | Excel | COUNT ( $\sigma_{\text {telephone }=335-1736} \sigma_{\text {company }=A B C}$ $\sigma_{\text {invoiceTo }}=$ Mary $\sigma_{\text {deliverToStreet }}=$ Central PO$)$ |
| Q6 | Noris | $\sigma_{\text {telephone }}=335-1736 \sigma_{\text {invoiceTo }}=$ Mary <br> $\sigma_{\text {deliverToStreet }}=$ Central PO |
| Q7 | Noris | $\begin{aligned} & \pi_{\text {item }} N \text { um, unitPrice } \sigma_{\text {order }} N u m=00001 \sigma_{\text {deliverTo }} \\ & =\text { Mary } \sigma_{\text {deliverToStreet }}=\text { Central } \mathrm{PO} \times \mathrm{Item} \end{aligned}$ |
| Q8 | Paragon | $\begin{aligned} & \sigma_{\text {billTo }=\text { Mary }} \sigma_{\text {shipToAddress }}=A B C \\ & \sigma_{\text {shipToPhone }}=335-1736 \mathrm{PO} \end{aligned}$ |
| Q9 | Paragon | $\begin{aligned} & \text { SUM }\left(\pi_{\text {price }} \sigma_{\text {telephone }}=335-1736 \sigma_{\text {billToAddress }}\right) \\ & =A B C \sigma_{\text {item }} N u m=00001 \mathrm{PO} \times \text { Item } \end{aligned}$ |
| Q10 | Paragon | $\begin{aligned} & \text { COUNT }\left(\sigma_{\text {invoiceTo }}=\text { Mary } \sigma_{\text {billToAddress }}\right. \\ & =A B C \text { PO } \times \text { Item }) \end{aligned}$ |

[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 $u . o_{n e x t}$; for each partition $P_{i}$ of $o_{\text {next }}$, it executes $o_{\text {next }}$ 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 ( $l b$ ) and upper ( $u b$ ) bounds of its probability. We use a heap, ordered by $l b$, to maintain the tuples that can be the answers. We also use two global variables: 1) $L B$, the lower bound probability of the tuple with the $k$-th highest probability in the heap; and 2) $U B$, the maximum probability of any tuple not in the heap.

Table II shows the status of the heap, $L B$, and $U B$, after computing each e-unit. For each tuple in the heap, its $l b$ and $u b$ values are shown. After $u_{7}$ is computed, $U B<L B$, and the $u b$ values of $t_{b}$ and $t_{c}$ are both 0.3 , which is smaller than $L B$. Thus, the top- 1 answer $\left(t_{a}\right)$ 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 100 MB source instance, which contains 1 M 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) 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 $T P C-H$ 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 Q8Q10 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 $e-M Q O$ 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 2.93 GHz 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 $\frac{\left|m_{i} \cap m_{j}\right|}{\left|m_{i} \cup m_{j}\right|}$, 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 $e-M Q O$, 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 $e-M Q O$. Note that $e-M Q O$ generates an optimal source query plan, which executes the smallest number of operators. However, the plan generation process is extremely expensive. Thus, $e-M Q O$ 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 $e$ $M Q O$ rises sharply with the number of mappings; when $|M|>$ $300, e-M Q O$ 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 |
| $e-M Q O$ | 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 $11(\mathrm{~b})$ shows $t_{q}$ (query evaluation time) for different values of $|D|$ on the default query $Q 4$. 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 $P O$ 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, $S N F$, 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 $S E F$ is faster than when $S N F$ is used. This can be explained by Table IV, which shows the query evaluation time and the number of source operators executed for Q 4 . We also include the results of e-MQO, which yields an optimal query plan, or equiva-

[^1]lently, 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 $S E F$ are both close to the optimal (i.e., the number of operators required by $e-M Q O$. However, $S E F$ needs fewer operators than $S N F$. While $S N F$ does not use the information about the number of mappings in each partition, SEF uses it effectively by computing the entropy function. Hence, $S E F$ makes a better decision and helps the target query to run faster than when $S N F$ is used. We conclude that o-sharing, when used with $S E F$, 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 $k \geq 10$.

## 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 $V L D B, 2002$.
[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.

[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 $V L D B, 2010$.
[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 $E D B T, 2009$.
[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 $m^{\prime}$, such that $m^{\prime}$ and $m$ match $a_{k}$, the $k$-th target attribute, with the same source attribute $a_{k}^{\prime}$ :1) if $m^{\prime}$ exists, it can find a node $n^{\prime}$ of $n$ 's child, such that the edge ( $n, n^{\prime}$ ) is labeled with $a_{k}^{\prime}$. Then, it recursively calls the function put to assign $m$ to the next level (Steps 6-9); and 2) if $m^{\prime}$ 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 \(M\)
    ptree \(\leftarrow\) create a new partition tree rooted at \(r\)
    for all \(m \in M\) do
        \(p u t(m, r, 1)\)
    end for
    return all the buckets in ptree
    cedure put(mapping m,node \(n\), integer \(k\) )
        if \(n\) is a bucket then
            put \(m\) in \(n\)
        else
        \(a_{k} \leftarrow\) the \(k\)-th attribute in \(q_{T}\)
        for all out-going edge \(e\) of \(n\) do
            if \(e\) is labeled as \(a_{k}^{\prime},\left(a_{k}^{\prime}, a_{k}\right) \in m\) then
                let \(n^{\prime}\) be the child of \(n\) connected by \(e\)
                    \(\operatorname{put}\left(m, n^{\prime}, k+1\right)\)
            end if
        end for
        if the above \(e\) does not exist then
            create an edge \(e\) labeled with \(a_{k}^{\prime},\left(a_{k}^{\prime}, a_{k}\right) \in m\)
            if \(a_{k}\) is the last target attribute then
                create a new bucket \(n^{\prime}\) connected with \(e\)
            else
                create a new node \(n^{\prime}\) for \(a_{k+1}\) connected with \(e\)
            end if
            \(\operatorname{put}\left(m, n^{\prime}, k+1\right)\)
        end if
    end if
```


## Appendix B <br> The Top-K Algorithm

Algorithm 4 first partitions the mapping set and finds out the representative mappings $M^{\prime}$ (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 $M^{\prime}$ (Step 3). Afterwards, it creates a heap, and set the initial values for $L B$ and $U B$ (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., $U B>L B$, then it inserts $r$ into the heap, and set its $l b$ and $u b$ properly (Steps 67). After all the tuples are processed, it updates $L B$ and $U B$ 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 $r^{\prime}$ in the heap whose lower bound probability ranks larger than $k, r^{\prime} . u b \leq L B$ (which implies that $r^{\prime}$ cannot be a top- $k$ answer), and 2) whether $U B \leq L B$ (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
    \(P_{1}, \ldots, P_{n} \leftarrow \operatorname{partition}\left(q_{T}, M\right)\)
    \(M^{\prime} \leftarrow \operatorname{represent}\left(P_{1}, \ldots, P_{n}\right)\)
    \(u_{1} \leftarrow i n i t \_u \_t r a c e\left(q_{T}, M^{\prime}\right)\)
    heap \(\leftarrow\) new heap of tuples, sorted by tuple's \(l b\)
    \(L B \leftarrow 0, U B \leftarrow 1\)
    run_qt_topk \(\left(u_{1}, D\right)\)
    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)
    if \(u . T\) has no operator then
        return decide_result( \(u\) )
    else if \(u . T\) contains empty relation then
        \(u . T \leftarrow \emptyset\)
        return decide_result(u)
    else
        \(\left(P_{1}, \ldots, P_{g}\right) \leftarrow \operatorname{next}(u)\)
        reorder_op \((u\), u.onext \()\)
        for all \(P_{i} \in P_{1}, \ldots, P_{g}\) do
            \(m \leftarrow\) arbitrary mapping in \(P_{i}\)
            \(o^{\prime} \leftarrow\) reformulate_op \(\left(o_{\text {next }}, m\right)\),
            \(a n s_{S} \leftarrow\) run_qs \(\left(o^{\prime}, D\right)\)
            \(u_{i} \cdot T \leftarrow\) create_qtree \(\left(u . T\right.\), u.onext,\(\left.a n s_{S}\right)\)
            \(u_{i} \cdot M \leftarrow P_{i}\)
            if run_qt_topk \(\left(u_{i}, D\right)=\) TRUE then
                    delete \(\bar{u}\), return TRUE
            end if
        end for
        delete \(u\), return FALSE
    end if
    nction decide_result(E-Unit u)
    utput: TRUE/FALSE (i.e., if all top- \(k\) results decided, or not)
    remove duplicate tuples in \(u . T\)
    for all \(r \in u . T\) do
        if \(\exists r^{\prime} \in h e a p, r^{\prime}=r\) then
            \(\exists r^{\prime} \in h e a p, r=r\) then
\(r^{\prime} . l b+\sum_{m \in u . M} \operatorname{Pr}(m)\)
        else if \(U B>L B\) then
            \(r . u b \leftarrow U B, r . l b \leftarrow \sum_{m \in M^{\prime}} \operatorname{Pr}(m)\)
            heap.push(r)
        end if
    end for
    \(U B \leftarrow U B-\sum_{m \in u . M} \operatorname{Pr}(m)\)
    \(L B \leftarrow r^{\prime} . l b, r^{\prime}\) is the \(k\)-th (or the last) tuple in heap
    if \(\forall r^{\prime} \in h e a p[k+1,|h e a p|], r^{\prime} . u b \leq L B\) and \(U B \leq L B\) then
        delete \(u\), return TRUE
    else
        delete \(u\), return FALSE
    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 $U B$.
Case 3: (Steps 7-20) Similar to Algorithm 2, it finds u.o next and reorder $u . T$; for each partition $P_{i}$ of $o_{\text {next }}$, it reformulates $o_{\text {next }}$ 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.


[^0]:    ${ }^{1}$ Step 3 does not use an MQO algorithm, because it is slower than basic. More details can be found in Section VIII.

[^1]:    ${ }^{2}$ 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 utrace explains why $o$-sharing needs slightly more time than $q$-sharing.

