Entity Search Strategies for Mashup Applications

Stefan Endrullis, Andreas Thor, Erhard Rahm

University of Leipzig
Leipzig, Germany
{endrullis,thor,rahm}@informatik.uni-leipzig.de

***Abstract*—Programmatic data integration approaches such as mashups have become a viable approach to dynamically integrate web data at runtime. Key data sources for mashups include entity search engines and hidden databases that need to be queried via source-specific search interfaces or web forms. Current mashups are typically restricted to simple query approaches such as using keyword search. Such approaches may need a high number of queries if many objects have to be found. Furthermore, the effectiveness of the queries may be limited, i.e., they may miss relevant results. We therefore propose more advanced search strategies that aim at finding a set of entities with high efficiency and high effectiveness. Our strategies use different kinds of queries that are determined by source-specific query generators. Furthermore, the queries are selected based on the characteristics of input entities. We introduce a flexible model for entity search strategies that includes a ranking of candidate queries determined by different query generators. We describe different query generators and outline their use within four entity search strategies. These strategies apply different query ranking and selection approaches to optimize efficiency and effectiveness. We evaluate our search strategies in detail for two domains: product search and publication search. The comparison with a standard keyword search shows that the proposed search strategies provide significant improvements in both domains.**

I. Introduction

Many web applications require the dynamic querying of domain-specific web sources such as entity search engines or hidden databases of the deep web. For example, queries to Google Product Search or Amazon can be used to determine the price or other features for products of interest. Such queries are especially relevant for many types of mashups that combine query results and other kinds of data from different sources. Mashups typically implement interactively executed data integration workflows consisting of steps for data acquisition (querying, result extraction), data transformation and matching, analysis and visualization. The easy development of mashups is supported by many frameworks and prototypes, including Yahoo pipes[1], Deri Pipes [1], Mashmaker [2], and Mash-o-matic [3].

Current mashups are still limited to rather simple integration tasks involving relatively small amounts of data. Query access to data sources or search engines is typically based on simple keyword searches that may provide limited result quality or need many queries to find all results of interest. Such simple queries do not exploit the advanced (domain-specific) search facilities provided by current entity search engines.

[1]http://pipes.yahoo.com/

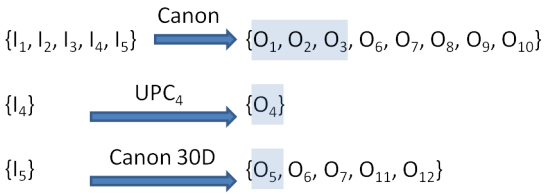


Fig. 1. Motivating example of a query strategy that employs three queries for finding all relevant entities (O_1-O_5) for a given set of input entities (I_1-I_5).

Motivating example: Assume a user is interested in buying a new digital photo camera. She has already identified 5 candidates (I_1-I_5, see Figure 1) and wants to find corresponding entities using an entity search engine such as Amazon or Google Product Search. A simple approach would be to use a keyword query on the name of every camera resulting into five queries. This results in a relatively high number of queries that may not necessarily return relevant results for all cameras. Alternatively, we may use a series of different queries to iteratively improve the query results. Since all cameras in the example are from Canon we might first submit a query manufacturer:Canon. Such a query can, in principle, find all products of interest but is likely to also retrieve many irrelevant products manufactured by Canon. In our example, the query retrieves three relevant entities (O_1-O_3) and five irrelevant (O_6-O_{10}). We may then send additional queries using the UPC (universal product code) information if available. In the example, we assume that I_4's UPC is known so that the corresponding query can precisely return the corresponding entity O_4. For the remaining entity I_5 another query can be issued, e.g. using the product name and the manufacturer name (Canon 30D). The query result contains the product of interest (O_5) but also other products, e.g., accessory products for the camera in question. In total, we need only three queries instead of five keyword queries. Furthermore, the diverse queries may improve the quality of the search result by finding relevant information on more cameras.

The example illustrates a common task in web data integration applications that we address in this paper: finding a set of given entities at an entity search engine with high efficiency (few queries) and high effectiveness (e.g. good recall). The specific entities to be searched for may be interactively determined by a mashup user based on previous searches. There may also be a predetermined set of entities that should be evaluated at certain points of time, e.g. to determine

the current prices for a set of products or to determine the current citation counts for a set of papers. Determining the most effective and efficient set of queries for a set of entities is a challenging problem since entity search engines typically provide many advanced search predicates and other query options. Furthermore, the complexity increases with the number and heterogeneity of input entities to be found.

To solve this problem we advocate for the use of so-called search strategies to automatically determine the most promising queries. We use a set of search engine-specific query generators to determine relevant queries and rank these queries according to different criteria. Queries are either executed in parallel or iteratively until the entities have been found or a certain number of queries has been executed. Choosing different kinds of query generators and ranking approaches permits a high flexibility to deal with diverse sets of input entities and to deal with different entity search strategies.

Our specific contributions are as follows:

- We introduce a framework for entity search strategies that utilize multiple query generators for improved efficiency and effectiveness. The framework supports different approaches for ranking and selecting candidate queries determined by different query generators.
- We propose four specific search strategies that use different kinds of information for query ranking and selection. The simplest approach is to execute all determined queries in parallel. A sequential strategy uses a predetermined order among query generators. Another strategy ranks queries according to their estimated number of covered input objects. The most sophisticated approach utilizes knowledge from previous query executions.
- We provide a detailed evaluation of our search strategies for two domains: product search and publication search. We propose new measures to determine the effectiveness, efficiency, and cost-effectiveness of search strategies. We use these measures to analyze the introduced search strategies and compare them with a basic keyword search approach.

The rest of the paper is organized as follows. The next section provides an overview about the new framework for entity search. We then explain the concept of query generators and introduce different kinds of such query generators (Section III). In Section IV we describe four search strategies, in particular their approaches for query ranking and selection. A detailed evaluation of query generators and search strategies is presented in Section V. We review related work (Section VI) before we conclude.

II. Entity Search Framework

We propose an entity search framework for efficient and effective retrieval of web entities that match a given set of input entities. The framework is able to execute the most promising queries adaptively out of a large set of queries (constructed by so-called query generators) by analyzing query results in an iterative way.

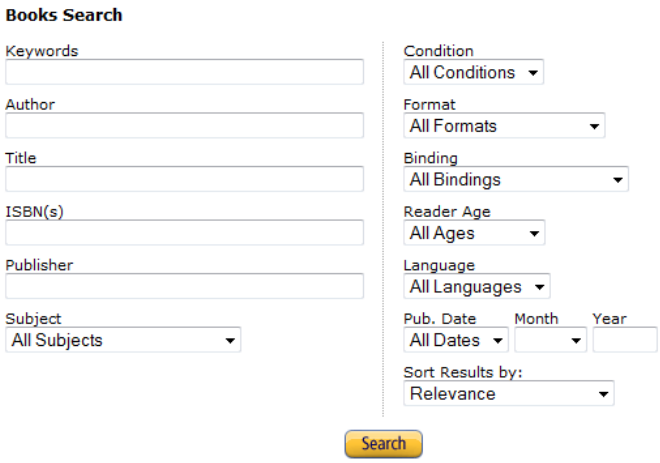


Fig. 2. Amazon's advanced search interface for Books

Our framework assumes that web entities can be acquired by corresponding queries to an **entity search engine (ESE)**. An ESE is restricted to a specific entity type (e.g., products) and supports a set of search predicates which allow for a structured search over the underlying ("hidden") entity database. Most commonly, the interface of an ESE is an HTML form consisting of one or more input fields where each input field is assigned to a predicate. For example, Figure 2 shows a screenshot of Amazon's advanced search form for books. The interface provides a free search predicate `Keywords` as well as specific search predicates for authors, title, publisher, and product-specific identifiers such as `ISBN`.

Figure 3 illustrates the schematic workflow of our entity search framework. The input is a set of entities I of the same type (e.g., product, publication, or person) for which matching web entities should be found. The output is therefore a match result, i.e., pairs (i, w) representing correspondences between input entities $i \in I$ and retrieved web entities $w \in W$.

The workflow has three consecutive phases: query generation, query ranking, and query selection followed by query execution and entity matching.

During the query generation phase the input entities I are input to a set of **query generators**. Each query generator generates one or more search queries for the input entities. The goal is that the corresponding query results match the input entities as good as possible, i.e., it aims to find the maximal number of input entities (high recall, i.e., all relevant entities appear in the result) at a good precision (few irrelevant results). To reduce the number of queries and thus improve performance, query generators may try to find multiple entities simultaneously with one or few queries. For example, it is more efficient to pose one query returning all relevant results for 10 input entities than to use 10 queries each returning only one relevant result entity. The output of each query generator is a set of pairs (q, I') where q denotes a generated query and $I' \subseteq I$ represents the set of **covered input entities**, i.e., entities that are supposed to have matching counterparts in q's query result.

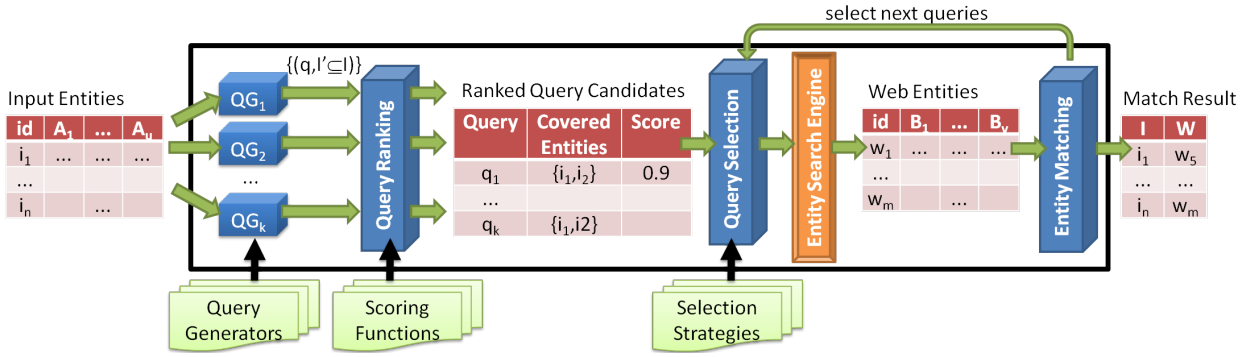


Fig. 3. Schematic workflow of an adaptive search strategy to search for given input entities

For our framework we assume the existence of query generators that are constructed beforehand, e.g., by a domain expert that is familiar with the ESE. Query generators usually utilize the capabilities of ESEs to generate a certain kind of queries for a given set of entities. They can implement similar strategies to the ones used by humans to quickly find certain entities. For example, since the ISBN unambiguously identifies a book, searching for a set of ISBN's (if available) is an efficient and precise approach to retrieve a set of books using Amazon's search interface (Figure 2). Alternatively, a specific book can be searched using the authors' names and some significant keywords from the title. Further examples will be discussed in the next section as well as in the evaluation section.

Query generators are orchestrated by a **search strategy** that determines not only what queries will eventually be executed but also their execution order. We will outline search strategies in Section IV. However, the quality of the underlying query generators has a significant impact on the effectiveness and efficiency of the search strategy and a search strategy may only employ some of the available query generators (e.g., due to manual specification or automatic selection; see Section IV). A search strategy basically controls the second and the third phase of the workflow.

The resulting queries of the query generators are ranked during the second phase. To this end a **scoring function** is applied to estimate the efficiency and effectiveness for every query individually. Scoring functions may take diverse information into account as will be discussed in Section IV. For example, queries may be ranked according to the number of entities they cover to give preference to queries that are likely to return many relevant results. Advanced approaches can employ background knowledge from previous query execution results to derive an average cost-effectiveness per query generator. For example, a query generator that utilizes entity-specific codes (e.g., UPC for products) is probably more effective than a simple keyword-based generator because the latter is likely to return many irrelevant results.

The last phase is the query selection (and execution), i.e., an **selection strategy** targets an efficient approach to find all relevant web entities with a minimal number of queries. To this end a selection strategy specifies the order in which available queries are selected and executed, respectively. The selection is not only based on the ranked list of queries but also adapts to the actual query results. For example, if an input entity is not found by an initial query or if we want to find additional entity representations (near-duplicates, e.g. product offers from different merchants) further queries (from different query generators) can be issued to improve the result. To support reasonable execution times, we can also limit the number of queries by enforcing a maximal number (***maxTrials***) of queries per input entity.

For their execution, queries are transformed into HTTP requests of the corresponding ESEs. Since ESEs may limit the number of results that can be retrieved with one request, we also consider to retrieve further results by following `next` links if provided. Thus, depending on the number of `next` links to be followed, a query can lead to multiple search engine requests.

Finally our framework assumes the existence of a reliable, domain-specific **entity matching** approach to decide whether two entities actually refer to the same real-world object. The high importance and difficulty of the entity matching problem has triggered a huge amount of research on different variations of the problem (see [4], [5] for recent surveys). However, our search framework considers entity matching as a "black box".

III. Query Generators

The first step towards finding input entities at an ESE comprises an appropriate search query generation. This task can be performed by query generators which have been introduced in [6].

A query generator takes as input a set I of entities of the same type and generates queries for a specific search engine so that the queries are likely to retrieve matching web entities for I. The output of a query generator is a set of pairs (q, I') where q denotes a generated query and $I' \subseteq I$ represents the set of **covered input entities**, i.e., entities that have been used to derive q and are supposed to have matching counterparts in q's query result. Note that the generated queries of a query generator do not necessarily need to cover all input entities I. For example, entities with missing attributes may not be considered if the query generator relies on these attribute

values. As we will see, the covered input entities I' can be used for query scoring and selection, so it is important that they are determined by the query generators for further use in the search framework.

All generated queries correspond to the following general entity search engine model. An entity search engine (ESE) supports a set of m predicates $p_1, \ldots, p_m$. Every predicate typically corresponds to a condition in a search form. For example, Amazon's advanced book search (see Figure 2) supports a general free text predicate (Keywords) as well as specific search predicates, e.g., for authors and title.

A **basic query** q is a conjunction $p_1(v_1) \wedge \ldots \wedge p_m(v_m)$ specifying a matching condition between each value v_i and predicate p_i. Typically only a subset of the available predicates is used in a query (i.e., a search value $v_i = \perp$ is possible). The conjunction of predicates is not necessarily interpreted as a strict logical AND but the search result may actually contain entities matching only some of the specified predicates. Depending on the search engine capabilities, the search values v_i may represent a single value, a set of keywords, an exact phrase, or a pattern utilizing wildcard symbols.

A query generator may apply an arbitrary algorithm to generate basic queries. However, most query generators follow a general pattern for constructing queries. First, the query generator splits the input entities I into possibly overlapping subsets $I'_1, \ldots, I'_k$. One query is then generated for each subset so that k queries are generated for the entire set I.

We distinguish between naïve and frequent-value generators. A **naïve** approach generates one query per entity by using the entity's attribute values as search values. This approach is quite expensive but simple and applicable to all search engines. In contrast, a **frequent-value** strategy aims at reducing the number of queries by identifying search values covering several entities. We use a variation of the well-known Apriori algorithm [7] to determine common values in attributes, e.g., manufacturer, that occur in at least $minSupport$ entities. The entities covered by a frequent value set form a subset I'_j for which one query is generated. Depending on the sets of frequent values the input entities may thus be divided into several subsets of variable size. More details on the realization of frequent-value query generators can be found in [6].

For query construction the relevant attribute values of the input entities are mapped to their corresponding search engine predicates. The attribute-predicate mapping is usually determined beforehand, e.g., based on a manually or automatically determined schema matching [8]. Different attributes may map to the same predicate (e.g., the free search predicate) and, in principle, an attribute may map to different predicates. Different functions can be applied on the actual attribute values to determine the predicate values. Common functions include phrase generation (putting a string in quotation marks) or keywords determination, e.g., by removing stop words from a string. Further transformation functions may be specific to a search engine.

Figure 4 shows the output of different query generators for the same set of five input entities I_1 through I_5. The generators QG:Keyword and QG:UPC are naïve generators whereas QG:Manufacturer and QG:FrequentValue are frequent-value generators. The number of generated queries varies from 1 (QG:Manufacturer) to 5 (QG:Keyword). Furthermore not all query generators are capable to cover all input entities, e.g., QG:UPC can only be applied for 3 of the 5 entities due to the missing UPC values for I_2 and I_5. The example queries also illustrate the wide variety in the precision of query results. For example, a QG:UPC query is likely to find a matching entity (if it is in the web source) whereas a "Manufacturer" query can return many irrelevant entities.

Input Entities

Id	Product Title	UPC
I_1	Canon Powershot A800 10 MP Digital Camera	0013803133059
I_2	Canon EOS Rebel XS Digital SLR Camera	
I_3	Canon PowerShot SX 20 IS 12.1 MP Digital Camera	0013803113662
I_4	Canon SX30IS 14.1MP Digital	0013803127348
I_5	Canon EOS 7D 18 MP CMOS Digital SLR Camera	

QG: Keyword

Query	Cov. Ent.
Canon Powershot A800 10 MP Digital Camera	{I_1}
Canon EOS Rebel XS Digital SLR Camera	{I_2}
Canon PowerShot SX 20 IS 12.1 MP Digital Camera	{I_3}
Canon SX30IS 14.1MP Digital	{I_4}
Canon EOS 7D 18 MP CMOS Digital SLR Camera	{I_5}

QG: Manufacturer

Query	Covered Entities
Canon	{I_1, I_2, I_3, I_4, I_5}

QG: Frequent Value

Query	Cov. Ent.
Canon Powershot	{I_1, I_3}
Canon EOS	{I_2, I_5}
Canon SX30IS	{I_4}

QG: UPC-Combined

Query	Cov. Ent.
UPC: 0013803133059 **OR** UPC: 0013803113662 **OR** UPC: 0013803127348	{I_1, I_3, I_4}

QG: UPC

Query	Cov. Ent.
UPC: 0013803133059	{I_1}
UPC: 0013803113662	{I_3}
UPC: 0013803127348	{I_4}

Fig. 4. Example queries generated by five different query generators for the same set of input entities I_1-I_5

Furthermore the search engine may allow the **disjunction** (`OR` combination) **of basic queries**. Combining several basic queries is an important feature to reduce the overall number of posed queries and thus to improve the efficiency of search engine access. For example, consider QG:UPC-Combined in Figure 4. This generator combines the basic queries as generated by QG:UPC with `OR`. The (ESE-specific) query generator takes into account that the overall number of basic queries in a combined query is limited by the ESE, e.g., due to size constraints for each predicate value. Assuming that at least three UPC queries can be combined, QG:UPC-Combined returns one query only.

IV. Search Strategies

After all query candidates have been generated, a search strategy selects a subset of queries and determines their order of execution. For example, a search strategy might avoid executing queries for entities that have already been found by a previous query if duplicates are not of interest. Search strategies usually aim to find all relevant web entities with a minimal number of queries, i.e., they usually analyze the

achieved query results for the selection of further queries. Search strategies thereby strive for a good balance between high efficiency (i.e., few queries) and high effectiveness (i.e., finding all entities).

To this end we propose four specific search strategies:

- **Parallel:** This naïve search strategy executes all queries of all available query generators.
- **Sequential:** This strategy executes queries according a fixed order of query generators (as realized in our motivating example shown in Figure 1).
- **Optimistic:** This search strategy executes queries according to the number of covered entities and thereby prefers queries with a large coverage over other queries.
- **Pre-Evaluated:** This sophisticated strategy executes the most-promising queries based on the performance (i.e., effectiveness and efficiency) of previously executed queries of the same query generator. The approach is based on a preceding evaluation of the search results for all query generators on a common training set of input entities.

All search strategies employ a query scoring function and a query selection method. The scoring function assigns a query score to every query candidate. All queries are ranked according their scores (in ascending order) and a selection method then processes the ranking and filters the queries of interest.

A. Query Scoring and Ranking

For query ranking a **scoring function** is applied to estimate the effectiveness and efficiency for every query individually. The queries' scores will be used to provide a ranked list of query candidates for the subsequent query selection. Recall that query generators return pairs (q, I') of a query q and a set of covered input entities I'. The score value for a query q can therefore be based on q's query generator (QG) and/or the covered entities (E). We therefore distinguish between four classes of scoring functions:

- **QG:** The query score solely depends on the query's generator, i.e., all queries of the same query generator receive the same score value. This approach basically is employed by the sequential strategy and defines the order in which the query generators should be applied. Our motivating example (see Figure 1) is of this type where the user first employs the manufacturer information, then the UPC, and finally title keywords. The scoring function would assign all queries of QG:Manufacturer a score of 1. All queries of QG:UPC are assigned a score of 2 and all queries of QG:Keyword receive a score of 3. In general, the query generator scoring can be provided manually by a domain expert or automatically, e.g. based on the query generator performance in previous evaluations.
- **E:** This strategy does not take into account the query generator but only looks at the set of covered entities. For example, the optimistic search strategy ranks queries based on their number of covered entities. The underlying assumption is that the query will eventually retrieve corresponding web entities for all covered input entities.
- **QG+E:** This method combines both information, i.e., the score of a query depends on the query generator and the covered entities. Approaches of this type – like the pre-evaluated strategy – may take into account how well queries of the same generator have performed previously for a similar number of entities.
- **Uniform:** This approach takes into account neither the query generator nor the covered entities but assigns the same score to all queries. This function is used by the parallel strategy.

The ranking of the query candidates has a major impact on the workflow of the search strategy. The ranking defines which of the queries may be executed in parallel (queries with the same score) or sequentially (queries with different scores).

B. Query Selection

After the query candidates have been ranked they are iteratively processed by the **query selection** function according to their ranking. The selection groups together all query candidates sharing the same score and processes these groups iteratively (in ascending order of their score). In each iteration all query candidates of the respective group are handed over to the selection function which determines a subset of these queries to be eventually sent to the ESE.

The iterative query execution has several advantages. First, queries that have already been executed are eliminated (in the case that different query generators produced the same queries). Second, it allows for an iterative result improvement since the ranking score reflects the effectiveness and efficiency of queries. Mashup applications that employ search strategies may therefore already present intermediate (or approximated) results to the user while performing additional iterations in the background. Third, the iterative model enables a search strategy to be executed efficiently by considering previous query results. In general, the query selection function may take into account statistics about previously executed queries and their results.

For a query candidate q and any entity $i \in I'$ covered by q the selection functions has to decide if q should be executed. This selection process is controlled by three parameters:

- $maxResults$ is the number of corresponding web entities for i that is considered to be sufficient. For example, if it is sufficient to find one web entity per input entity ($maxResults = 1$, i.e., we are not interested in duplicates) then the query selection avoids executing queries that search for already found entities.
- $maxTrials$ is the maximal number of queries that should be executed to find i. A selection strategy can thereby avoid to execute too many queries if the entity of interest seems not to be in the web data source.
- The option $favorDistinct$ forces the query selection to give preference to diverse queries when searching for the same entity i (if the $maxTrials$ threshold has not yet been exceeded). This can be achieved by rejecting queries

Algorithm 1: Search Strategy Execution

input : search strategy consisting of
- set of query generators G
- scoring function $score$
- selection properties: $maxTrials$, $maxResults$, $favorDistinct$

input : set of input entities I
output: aggregated match result M

$allQueries \leftarrow \bigcup_{g \in G} g(I)$;
$rankedQueries \leftarrow \texttt{rank}(allQueries, score)$;
$M \leftarrow \emptyset$;
while $rankedQueries \neq \emptyset$ **do**
  $queries \leftarrow \texttt{pullTopWithSameScore}(rankedQueries)$;
  $selected \leftarrow \emptyset$;
  for $(q, I') \in queries$ **do**
    $g \leftarrow \texttt{generator}(q)$;
    if $\forall i \in I' : i \notin \texttt{processed}(g)$, $\texttt{trials}(i) < maxTrials$, $\texttt{results}(i) < maxResults$ **then**
      if $favorDistinct$ **then**
        if $\forall g' \in G, g' \sim g : i \notin \texttt{processed}(g')$ **then**
          $selected \leftarrow selected \cup \{(q, I')\}$;
      else
        $selected \leftarrow selected \cup \{(q, I')\}$;
  $results \leftarrow \texttt{sendToESE}(selected)$;
  $M' \leftarrow \texttt{match}(I', results)$;
  $M \leftarrow M \cup M'$;

of the same or very similar query generators. As we will see in the evaluation (Section V-E), this option can avoid ineffective queries and thus improve performance. Furthermore, it reduces the need to preselect suitable query generators from a larger pool for use in search strategies.

Algorithm 1 shows the pseudo-code of the query selection. Based on a given ranking, all queries are processed iteratively (while loop). In each iteration the top queries with the same (lowest) score are taken from the stack of ranked queries and further processed by the query select. A query passes the selection process only if all of its processed input entities satisfy the conditions for $maxTrials$, $maxResults$ and $favorDistinct$, i.e., they did not yet exceed the maximal number of trials and results and have not yet been processed ($i \notin \texttt{processed}(g')$) by another similar ($g \sim g'$) query generator in case of the $favorDistinct$ option. The remaining queries are sent to the ESE (e.g. in parallel), the query result is matched to the input entities, and the overall match result is updated accordingly.

Figure 5 illustrates an example mode of action for all proposed search strategies using the query generators shown in Figure 4. The parallel strategy assigns the same score to all queries and executes all 13 queries.

The sequential strategy implements the procedure of the motivating example (see Figure 1). It employs only 3 out of the 5 query generators and orders (scores) them as follows: QG:Manufacturer (score=1), QG:UPC (2), and QG:Keyword (3). In this example we are not interested in duplicates ($maxResult$=1) which is why after the first query we eliminate queries "upc1" and "upc3" (denoted by a score value in parentheses) because the corresponding entities have already been found (see Figure 1). Hence during the second iteration

Query Generator	Query	Covered Entities	Parallel	Sequential (maxRes=1)	Optimistic (maxRes=2)	Pre-Evaluated (maxTrials=1)
Manufacturer	Canon	{I_1, I_2, I_3, I_4, I_5}	1	1	1	(5)
UPC-Combined	upc1 OR upc3 OR upc4	{I_1, I_3, I_4}	1	--	2	1
UPC	upc1	{I_1}	1	(2)	(4)	(2)
UPC	upc3	{I_3}	1	(2)	(4)	(2)
UPC	upc4	{I_4}	1	2	4	(2)
FrequentValue	Canon Powershot	{I_1, I_3}	1	--	(3)	(4)
FrequentValue	Canon EOS	{I_2, I_5}	1	--	3	4
FrequentValue	Canon SX30IS	{I_4}	1	--	4	(3)
Keyword	Canon Powershot A800 10 MP Digital Camera	{I_1}	1	(3)	(4)	(6)
Keyword	Canon EOS Rebel XS Digital SLR Camera	{I_2}	1	(3)	(4)	(6)
Keyword	Canon PowerShot SX 20 IS 12.1 MP Digital Camera	{I_3}	1	(3)	(4)	(6)
Keyword	Canon SX30IS 14.1MP Digital	{I_4}	1	(3)	4	(6)
Keyword	Canon EOS 7D 18 MP CMOS Digital SLR Camera	{I_5}	1	3	4	(6)
Overall number of executed queries			13	3	7	2

Fig. 5. Illustration of the four different search strategies for input entities I_1-I_5 using the query generators of Figure 4. The numbers in columns 4-7 denote the scores of corresponding queries. Scores in parentheses indicate that the query does not pass query selection.

we only execute "upc4". Similarly, the third iteration ignores all but the last query for I_5.

For the optimistic strategy we allow duplicates, i.e., we set $maxResult$=2 to allow for up to two matching web entities per input entity. The optimistic strategy basically orders the query based on the number of covered entities. After the execution of the first two queries, I_1 and I_3 have already been found twice (I_4 could not be retrieved by the manufacturer query, see Figure 1). Hence only the query "Canon EOS" is executed during the third iteration. Finally, all queries for I_4 and I_5 are executed during the last iteration (covered entity size equals 1).

The pre-evaluated strategy ranks the queries as indicated in Figure 5 based on previous query execution results. For this strategy we set $maxTrials$=1, i.e., we send at most one query per input entity. Therefore, after the first query (QG:UPC-Combined) we are only searching for I_2 and I_5 which is why query "Canon EOS" is the only other query to be executed though it was initially ranked fourth.

V. Evaluation

We evaluate the introduced query generators and search strategies for two domains: product search and publication search. We first describe how we measure the effectiveness (quality) and efficiency of query generators and search strategies. We then provide details about the experiment settings, in particular the considered search engines (Amazon, Google Scholar), the query generators as well as the chosen sets of input entities. The evaluation starts with the comparative analysis of selected query generators. We then evaluate the new search strategies for different parameter settings and compare them with a baseline approach using keyword queries only.

A. Evaluation Measures

To evaluate query generators and search strategies we use three measures: a quality (effectiveness) measure, an effi-

TABLE I
EVALUATION MEASURES FOR QUERY GENERATORS AND QUERY STRATEGIES

Measure	Global	Local
Quality	$\frac{\lvert domain(M)\rvert}{\lvert I\rvert}$	$\frac{\lvert domain(M)\rvert}{\lvert I'\rvert}$
Efficiency	$\frac{\lvert domain(M)\rvert}{requests}$	$\frac{\lvert domain(M)\rvert \cdot \lvert I\rvert}{requests \cdot \lvert I'\rvert}$

ciency measure, and a combination of both to measure cost-effectiveness. The measures can be used for search strategies combining several query generators as well as for individual query generators. However, certain query generators may only be applicable for a subset of input entities, e.g. due to missing values such as UPC or manufacturer. We therefore distinguish between global and local versions of the measures. The global measures relate to the set of all input entities, I, and are of primary interest for entire search strategies but can also be used for query generators. The local measures, on the other hand, are only useful for evaluating query generators and relate only to the subset of input entities I' that can be processed by a query generator.

Table I gives the definitions of the global and local quality and efficiency measures. In the formulae, M denotes the aggregated match result containing all found entities that match with input entities. Thus, $domain(M)$ refers to the set of input entities for which a matching entity could be found. Moreover, $requests$ represents the number of query requests sent to the ESE. These requests include the original queries as determined by the query generators as well as possible requests to follow `next` links to obtain additional entities of larger result sets.

The quality measures are recall-oriented and determine the fraction of input entities that could be found at the ESE. Note that the optimal value 1 cannot be achieved if the ESE does not keep some of the input entities. We use rather comprehensive ESEs for our evaluation to limit this effect. Also, the focus here is more on a comparative evaluation of different search strategies rather than maximizing the absolute quality values. The efficiency measures determine the number of found input entities per request. Hence, finding several input entities per query allows efficiency values larger than 1.

Since we want to achieve both high quality and high efficiency, we also determine a joint evaluation measure to determine the cost-effectiveness of a query generator or search strategy. We calculate this measure using a weighted harmonic mean of the (global or local) quality and efficiency measures:

$$cost-effectiveness_{q,e} = \frac{q+e}{\frac{q}{quality} + \frac{e}{efficiency}}$$

In this study we set the weights q and e to 1, i.e. we use the standard harmonic mean between quality and efficiency. We can give preference to quality or efficiency by choosing larger values for q than for e or vice versa.

TABLE II
SEARCH CAPABILITIES OF SELECTED ESES

Capability	Amazon	Google Scholar
Search predicates	*free*, title, manufacturer, min_price, max_price, ...	*free*, author, title, published_in, min_year, max_year
Search values	keywords, phrases	keywords, phrases, patterns
OR aggregation	yes	yes
Max. number of results per request	10	100

B. Experiment settings

We evaluate our approaches for two entity search engines from the e-commerce and bibliographic domains: Amazon's Product Advertising API for searching products and Google Scholar[2] for searching publications. Selected features of these search engines are summarized in Table II. Both ESEs allow searching using a free search predicate or by choosing among several domain-specific search predicates such as (product) manufacturer or (publication) author. Both search engines also allow the OR aggregation (disjunction) of several simpler queries (search predicates) within a combined query that can help to improve efficiency. The maximal number of result entities per query is restricted to only 10 for Amazon, and to 100 for Google Scholar. For our evaluation, we found these result sizes sufficient to pose only one request per query, i.e., we do not consider requests to follow `next` links. We have evaluated the consideration of `next` links in [6].

Based on the available search capabilities we defined ten query generators per search engine for use within search strategies. The considered query generators are listed in Table III and Table IV. For each query generator we show its short name, type of query generator and a short descriptions how search values are derived from the input entities. For both ESEs we include all three types of query generators introduced in Section III: naïve, frequent-value, and use of OR aggregation. For Amazon, all search values are mapped to the free search predicate. For instance, the baseline approach, *kw*, performs a free text search with keywords from the product title, whereas *pc* tries to extract product codes (such as `PowerShot SX220`) or at least parts of them from the title. Note that some of specific query generators can likely cover only a subset of the input entities, e.g. if they rely on data such as UPCs or product codes. For Google Scholar, we also utilize specific search predicates such as author or title.

For evaluating the query generators and search strategies we use many different sets of input entities. As usual for data integration applications, we obtain the input entities from data sources different than the search engines. For the e-commerce domain, we choose the entities from a collection of more than 114 thousand electronic product offers provided by a price comparison portal. For the bibliographic domain, we obtain the entities to be searched from the DBLP Computer Science

[2]http://scholar.google.com

TABLE III
QUERY GENERATORS FOR AMAZON

Name	Type	Search Values
kw	naïve	title keywords
pw	naïve	pure words from title (terms without digits and punctuations)
pc	naïve	product code from title
pcm	naïve	product code from title + manufacturer name
f1	f. value	4 common title keywords
f1m	f. value	f1 + manufacturer name
f2	f. value	5 common title keywords
fv	f. value	f2 + manufacturer name
f3	f. value	1 common title keywords
upc	OR aggr.	up to 8 UPCs combined with OR

TABLE IV
QUERY GENERATORS FOR GOOGLE SCHOLAR

Name	Type	Search Values → Predicate
kw	naïve	title keywords → free
at	naïve	first author → author; title keywords → title
aty	naïve	all authors → author; title keywords → title; year → min_year; year → max_year
pa1	naïve	title pattern → title
ph1	naïve	title phrase → title
fa	f. value	1 common author → author
ft	f. value	1 common title keyword → author
faty	f. value	2 common terms in authors, title, or year, mapped to corresponding predicates
pa	OR aggr.	up to 10 title patterns combined with OR → title
ph	OR aggr.	up to 10 title phrases combined with OR → title

Bibliography[3].

For the e-commerce domain we automatically generate ten data sets of 30 products for each of the following three categories:

- type: only products of the same product type
- manufacturer: products of the same manufacturer
- random: random collection of products

The products are selected so that 50% of them contain a UPC; which is about the same share than in the full set of product offers.

For the bibliographic domain, we also choose 30 data sets evenly distributed among three categories:

- author: only publications of the same author
- venue: publications of the same journal or conference
- random: random collection of publications.

The specific categories are chosen to reflect common search scenarios where input entities share some common property. Some query generators, e.g. frequent value query generators, might also be able to utilize the existence of dominating values for attributes like manufacturer or author.

For each of the considered categories we use 5 data sets for a pre-evaluation of query generators that is used by some search strategies; the 5 remaining datasets are used for the final evaluation of query generators and search strategies.

The matching between the input entities and search results is performed by utilizing fine-tuned and manually verified match approaches. For product matching we utilize the UPC, product code and the title values. Publication matching is based on a comparison of the authors, publication titles, and publication years.

C. Evaluation of Query Generators

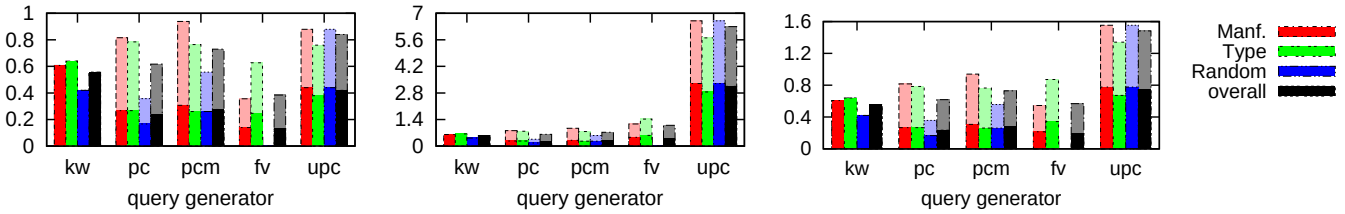
We first evaluate different query generators for the two domains since they are the building blocks for search strategies. In favor of readability, we focus on five of the ten query generators per domain; their names are bold-faced in Table III and Table IV.

Figure 6 and Figure 7 show the quality, efficiency, and cost-effectiveness results of the Amazon query generators for product search and the Google Scholar query generators for publication search, respectively. For each query generator and measure four bars are shown: the average results for the three considered categories and the average over all categories and input datasets. The half transparent bars in the figures illustrate the local measures and the nontransparent bars the global ones. For example, the local quality of the upc query generator in Figure 6a is more than 0.8 meaning that more than 80% of the input entities with an UPC could be found. However, the global quality of this query generator is only half as high since only half of the input products have an UPC. Other query generators with significant differences between local and global measures include pc, pcm and some frequent value query generators. For query generators applicable to all input entities, the global and local measures are the same.

We first discuss the result for the product query generators (Figure 6). In terms of global quality, we observe that the baseline query generator *kw* achieves the best result. Hence, this makes it an excellent choice for maximizing the number of relevant search results and a strong competitor for our search strategies. On the other hand, we see that some specialized query generators like upc and pcm have higher local quality indicating that they may be good choices for a search strategy combining several query generators. There are relatively small differences between the different categories of input entities. The pcm query generator that is based on product code and manufacturer values can achieve the best local quality for category manufacturer (where all input entities have a defined manufacturer value). For efficiency, the best results by far (values of about 6) are achieved by the *upc* query generator due to its OR aggregation of UPC values for several products. By contrast, the efficiency of the naïve query generators is generally below 1. As a consequence, the *upc* query generator achieves the best global efficiency and global cost-effectiveness and can outperform kw in this respect. As we will see, its performance can still be topped by some of the search strategies exploiting multiple query generators.

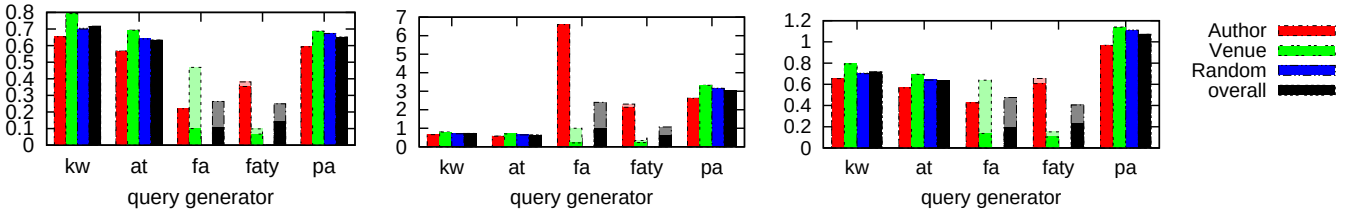
The results for the bibliographic domain in Figure 7 show that the baseline query generator, *kw*, achieves again the best quality. It is a naïve query generator which maps keywords from the publication title to the free search predicate. The

[3]http://www.informatik.uni-trier.de/~ley/db/



(a) Quality (b) Efficiency (c) Cost-effectiveness (q=e=1)

Fig. 6. Dataset-specific evaluation results for Amazon query generators



(a) Quality (b) Efficiency (c) Cost-effectiveness (q=e=1)

Fig. 7. Dataset-specific evaluation results for Google Scholar query generators

frequent value query generators show unstable quality and efficiency results for different input categories. The best efficiency of more than 6 is achieved for the author category and the fa query generator that can find many relevant publications by a single author query. The best cost-effectiveness is achieved by the *pa* query generator that utilizes OR aggregation of title patterns. Such patterns are supported by Google Scholar and allow using only substrings of titles with wildcard characters (*) for insignificant parts. For instance, "`Evaluation * * Generators * * Engines`" would be a valid title pattern to query for [6]. Up to 10 such title patterns are aggregated in a disjunctive query; the query size limit of 256 characters is also observed. The efficiency results indicate that on average about 3 publications are found per pa query.

In summary, the results show that there are big quality and efficiency differences between query generators. While naïve approaches can achieve good quality they suffer from poor efficiency. OR aggregation is highly efficient but typically limited to rather short queries. Frequent value query generators may achieve good efficiency but are restricted to specific categories of input data. We therefore see the need for search strategies combining several query generators and automatically choosing the most suitable queries.

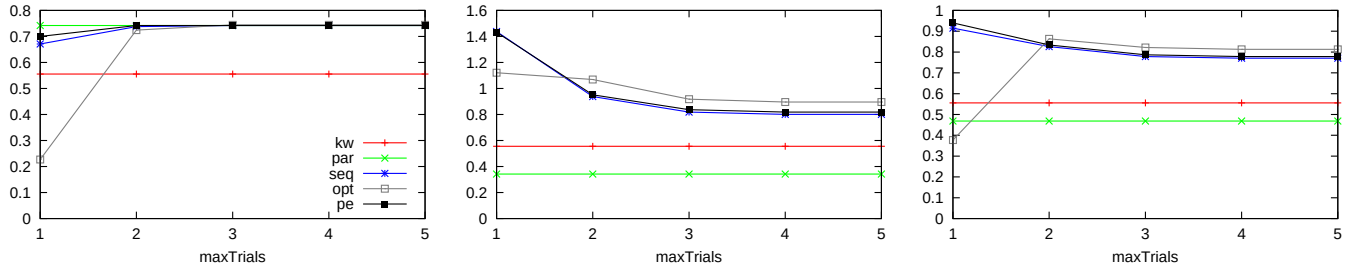
D. Evaluation of search strategies and $maxTrials$

We now comparatively evaluate the search strategies introduced in Section IV for both domains. For both search engines we use all ten query generators for query generation and use the input datasets of all categories. For the sequential search strategy, we apply the query generators according to their previously determined local efficiency, i.e., queries of more efficient query generators are executed first. We only look for one matching result per input entity ($maxResults = 1$) and apply the $favorDistinct$ option.

Figure 8 and Figure 9 show the quality, efficiency and cost-effectiveness of the search strategies for Amazon and Google Scholar, respectively. As baseline approaches we include the results for the previously introduced keyword query generators *kw*. We compare the strategies for different values of the $maxTrials$ parameter that determines the maximal number of search trials per input entity. Increasing the number of queries by using higher $maxTrials$ values is likely to decrease efficiency but can help finding more entities thereby improving quality.

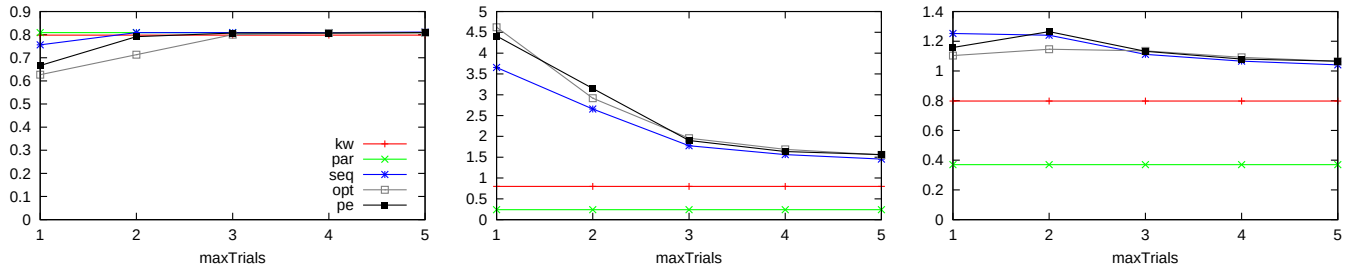
We first discuss the results for the Amazon product search engine (Figure 8). In terms of quality, we observe that the baseline query generator *kw* is clearly outperformed by all new search strategies. The best possible quality is achieved by the parallel search strategy *par* which is, like *kw*, independent of $maxTrials$ since it submits all generated queries (subject to the elimination of redundant queries due to the $favorDistinct$ option). The sequential (*seq*), optimistic (*opt*) and pre-evaluated (*pe*) strategies can achieve about the same good quality for $maxTrials = 2$ indicating an effective ranking and query selection. Even for $maxTrials = 1$, *seq* and *pe* achieve already surprisingly good results. By contrast, the optimistic approach depends on more than one query per input entity ($maxTrials > 1$), apparently since not all queries return the entities they are meant to cover.

With respect to efficiency, the new search strategies (except parallel) outperform the baseline approach even to a larger degree. As expected their efficiency drops with increasing values of $maxTrials$ but remains ahead of kw. The parallel search strategy is not competitive and serves only as a reference point for the best quality in our evaluation. The best cost-effectiveness is achieved for the two strategies *seq* and *pe* for $maxTrials = 1$. They both use knowledge about the query generator performance from previous executions. The simpler optimistic strategy achieves a comparable cost-



(a) Quality (b) Efficiency (c) Cost-effectiveness

Fig. 8. Evaluation results for search strategies for Amazon subject to $maxTrials$



(a) Quality (b) Efficiency (c) Cost-effectiveness

Fig. 9. Evaluation results for search strategies for Google Scholar subject to $maxTrials$

effectiveness for $maxTrials > 1$. Hence, all three advanced search strategies can successfully combine different queries to achieve significantly improved quality and efficiency over the baseline approach.

The results for Google Scholar (Figure 9) largely confirm the observations for Amazon. Here, the baseline approach is very effective and achieves already near-perfect results, i.e., almost every query returns the respective publication. The advanced search strategies can achieve the same high quality for $maxTrials$ values of 2 and 3. On the other hand, the efficiency advantage of the search strategies over the baseline approach is much more pronounced due to the combined use of efficient query generators. The best cost-effectiveness is again achieved by the *seq* and *pe* strategies, here for $maxTrials = 2$.

E. Effect of query generator selection

The results presented so far used all query generators and the seq and pe strategies utilized knowledge about the relative performance of different query generators. We performed two experiments to analyze the dependency on such previous knowledge and to see whether our approaches suffer from having to consider all query generators instead of only a subset with the most cost-effective ones.

In the first experiment we compare the cost-effectiveness of two sequential search strategies for the Amazon search engine called *seq1* and *seq2* (see Figure 10). *seq1* is the previously studied strategy that utilizes a pre-optimized order of query generators. It turned out that a successful ordering can be found by ranking query generators (and their queries) according to their local efficiency determined for training data sets beforehand (the first query generator is upc, followed by

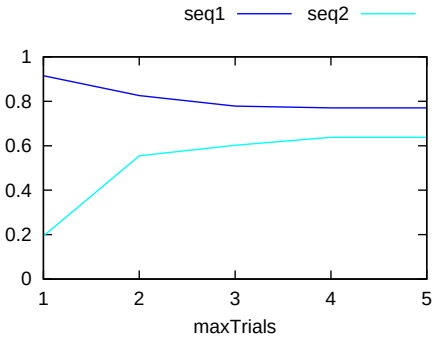


Fig. 10. Cost-effectiveness of two sequential search strategies, one optimally ranked query generators and one with the reversed ranking

some frequent value query generators and later naïve query generators). For strategy *seq2* we simply reversed the order, i.e., we start with the least efficient query generator. The results in Figure 10 show that, as expected, *seq2* performs much worse than *seq1* since it requires many more queries to find the input entities. *seq2* needs higher values for $maxTrials$ to eventually find the entities but the increased query overhead prevents that a sufficient cost-effectiveness can be reached. The experiment underlines the value but also the dependency on pre-evaluations of query generators.

Given the quality and efficiency differences between query generators it seems promising to restrict search strategies to a preselected subset of the most promising query generators. In our framework we did not want to introduce this additional tuning complexity and therefore aim at an automatic removal of ineffective queries. This is controlled by the $favorDistinct$ option that leads to ignoring queries that are highly similar

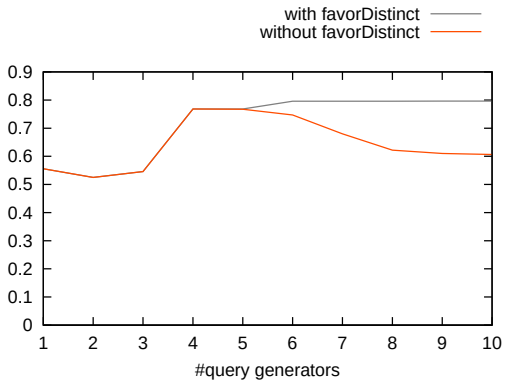


Fig. 11. Cost-effectiveness of optimistic search strategies for an increasing number of query generators (Amazon product search)

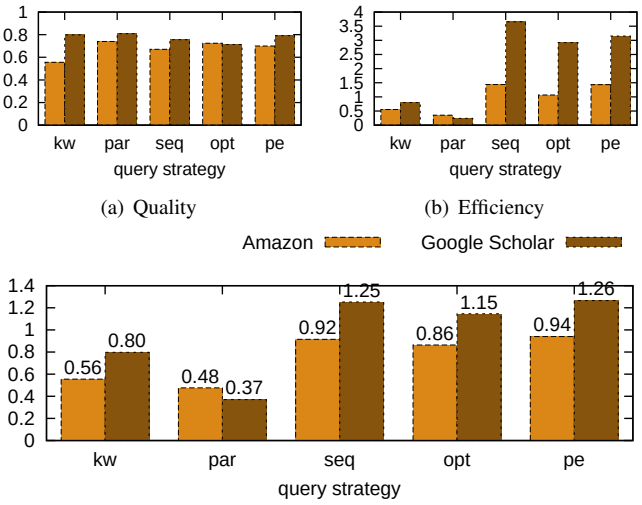


Fig. 12. Evaluation results of selected search strategies

to already executed queries and therefore unlikely to identify additional results. For example, the pc query generator searches for the product code of a product while the pcm query generator searches for both the product code and manufacturer. If a product has already been searched by one of these query generators, we avoid a query by the other query generator that also uses product code information. In our implementation we group the query generators into disjoint sets of similar query generators based on their used attributes. We then drop queries for entities that have already been queried by a similar query generator.

We evaluate the usefulness of the $favorDistinct$ option for optimistic search strategies and the Amazon search engine. Figure 11 shows how the cost-effectiveness for both strategies (with and without this option set) evolves when we successively increase the number of query generators from 1 to 10. The first four query generators we add are not similar to each other so that the same results are achieved. Hence, $favorDistinct$ affects only the last part of the curves (from 5 to 10 query generators). Here we can see that the cost-effectiveness of the search strategy not skipping similar queries declines with more query generators, whereas the other strategy can even improve or maintain the best cost-effectiveness. Hence, we see that the $favorDistinct$ option works as desired and allows us to keep all query generators (or add more). This highly desirable behavior was also observed for the other search strategies and therefore applied per default.

F. Final comparison

We finally summarize the best results for the considered search strategies. All results are based on the use of all query generators, the $favorDistinct$ option and $maxResults = 1$. $maxTrials$ was set to 2 for Google Scholar and when using the optimistic search strategy; and to 1 otherwise.

The evaluation results of the introduced search strategies are shown in Figure 12. The result show similar trends for both domains despite that the absolute values are higher for the bibliographic domain (Google Scholar). The parallel search strategy was not meant to be a true competitor as its many queries lead to poor efficiency and cost-effectiveness. However, it illustrates the best possible quality and we can see that for both domains the top quality can (almost) be achieved by the alternate, more efficient search strategies seq, opt and pe. These advanced strategies outperform the efficiency and cost-effectiveness of the baseline search strategy kw in both domains by about 50-70%. For the e-commerce domain (Amazon) they could also clearly improve the result quality of kw.

The cost-effectiveness is best for the sequential and pre-evaluated search strategies. These approaches are close together and benefit from knowledge about the query generator performance. The simpler optimistic approach does not depend on a pre-evaluation of query generators but uses the simple coverage information for ranking. Its cost-effectiveness is less than 10% lower than for the other strategy making the optimistic search strategy a good default approach.

VI. Related Work

This work extends our preliminary work on query generators [6] that deals with query generators for Google Scholar. The present paper introduces the new concept of search strategies that combine multiple query generators in order to find relevant entities efficiently. New aspects include the proposed iterative workflow model, several scoring functions for ranking query candidates, and query selection techniques. The evaluation is by far more extensive and considers two domains.

In [9] we presented a framework for the development of data integration mashups. The framework consists of components for query generation, online matching, and other data transformations. A developer can then define data integration data flows using a script language and can thereby realize simple search strategies. As an example for such mashups we introduced our Online Citation Service (OCS) that employs three different query generators within a sequential search strategy.

The capabilities to search for sets of entities at entity search engines are still very limited in current mashup systems such as Yahoo pipes or IBM Damia [10]. Those systems typically perform entity search via simple keyword searches which may result in a high number of queries and many irrelevant results. Here our search strategies could be incorporated to increase both the quality and the performance of search results.

In general, the automatic query generation for deep web sources can be considered from two perspectives: crawling the hidden web and virtual data integration. Different systems for crawling the hidden web have been described in [11], [12], and [13]. These systems automatically generate search queries for deep web sources with the intent to download large portions of hidden databases. The crawling process is usually an iterative process, where new search queries are generated based on previously retrieved search results. This has quite some similarities to our work, since the task of generating queries based on sets of values (or structured entities) is focus of our work. However, while Deep Web crawling aims at finding new/unknown information, our intention with search strategies is to find a set of known entities more efficiently.

Virtual data integration approaches translate queries posed against a global schema into sub-queries of the underlying web sources at runtime. Meta search engines, such as MetaQuerier [14], generate sub-queries based on the user input independently from one another. In contrast to that, our approach is instance-based, i.e., search engine queries are generated from a set of entities instead of a user query. However, in both cases the query generation or transformation has to take into account the query capabilities of web sources.

Querying data sources with limited access capabilities has been widely investigated in the literature. The automatic extraction of ESE interfaces from web pages is part of the discussions in [15], [16], and [17]. Together with schema matching techniques [8] those methods could be used to (partially) automate the process of building query generators for ESEs which will be part of our future work.

Query capabilities of deep web sources are typically described using *binding patterns* [18], [19]. Binding patterns can be used to define what combinations of form elements may be used to generate valid search queries and what limitations regarding the domains of form elements have to be taken into account. At the moment, defining query generators for search strategies is a manual process and users have to take care of the search capabilities themselves. In future, binding patterns could support the process of building query generators by ensuring the validity of their queries. Anyway, our objective in this paper is to increase the efficiency of entity search.

VII. Conclusions

We presented and evaluated a new framework for adaptive entity search that aims at finding entities that are given as input with high quality and efficiency. The framework utilizes multiple query generators per search engine and supports several approaches to rank queries. Queries are iteratively selected and executed to incrementally improve result quality while limiting the total number of queries. We proposed four specific search strategies and showed for two domains that they can substantially outperform simple search approaches based on keyword queries. The best cost-effectiveness can be obtained if previous knowledge on the cost-effectiveness of query generators is exploited for query ranking. But even the relatively simple optimistic search strategy achieves already very good results.

In future work we plan to further automate the search framework by an automatic generation of query generators rather than their manual creation by domain experts. We also want to integrate the search framework in existing mashup environments and mashup applications.

References

[1] D. L. Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni, "Rapid prototyping of semantic mash-ups through semantic web pipes," in *WWW*, 2009.

[2] R. J. Ennals and M. N. Garofalakis, "Mashmaker: mashups for the masses," in *SIGMOD*, 2007.

[3] S. Murthy, D. Maier, and L. Delcambre, "Mash-o-matic," in *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, 2006.

[4] H. Köpcke and E. Rahm, "Frameworks for entity matching: A comparison," *Data Knowl. Eng.*, vol. 69, no. 2, 2010.

[5] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, 2007.

[6] S. Endrullis, A. Thor, and E. Rahm, "Evaluation of query generators for entity search engines," in *Workshop on Using Search Engine Technology for Information Management (USETIM)*, 2009.

[7] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB*, 1994.

[8] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB Journal*, vol. 10, no. 4, 2001.

[9] A. Thor, D. Aumueller, and E. Rahm, "Data integration support for mashups," in *Int. Workshop on Information Integration on the Web*, 2007.

[10] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh, "Damia: data mashups for intranet applications," in *SIGMOD Conference*, 2008.

[11] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *VLDB*, 2001.

[12] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Y. Halevy, "Google's deep web crawl," *PVLDB*, vol. 1, no. 2, 2008.

[13] L. Barbosa and J. Freire, "Siphoning hidden-web data through keyword-based interfaces," *JIDM*, vol. 1, no. 1, 2010.

[14] B. He, Z. Zhang, and K. C.-C. Chang, "Metaquerier: querying structured web sources on-the-fly," in *SIGMOD*, 2005.

[15] Z. Zhang, B. He, and K. C.-C. Chang, "Understanding web query interfaces: best-effort parsing with hidden syntax," in *SIGMOD*, 2004.

[16] T. Kabisch, E. C. Dragut, C. T. Yu, and U. Leser, "A hierarchical approach to model web query interfaces for web source integration," *PVLDB*, vol. 2, no. 1, 2009.

[17] H. He, W. Meng, C. T. Yu, and Z. Wu, "Wise-integrator: A system for extracting and integrating complex web search interfaces of the deep web," in *VLDB*, 2005.

[18] A. Rajaraman, Y. Sagiv, and J. D. Ullman, "Answering queries using templates with binding patterns," in *PODS*, 1995.

[19] R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman, "Computing capabilities of mediators," in *SIGMOD*, 1999.