SmartCrawl: Deep Web Crawling Driven by Data Enrichment

ABSTRACT

Entity resolution is defined as finding different records that refer to the same real-world entity. In this paper, we study deep entity resolution (DeepER) which aims to find pairs of records that describe the same entity between a local database and a hidden database. The local database can be accessed freely but the hidden database can only be accessed by a keyword-search query interface. To the best of our knowledge, we are the first to study this problem. We first show that straightforward solutions are inefficient because they fail to exploit the ideas of query sharing and local-database-aware crawling. In response, we propose SMARTCRAWL, a novel framework to overcome the limitations. Given a budget of b queries, SMARTCRAWL first constructs a query pool based on the local database and then iteratively issues b queries to the hidden database such that the union of the query results can cover the maximum number of records in the local database. Finally, it performs entity resolution between the local database and the crawled records. We find that query selection is the most challenging aspect, and we investigate how to select the query with the largest benefit at each iteration. SMARTCRAWL seeks to use a hidden database sample to estimate the query benefit. We propose unbiased estimators as well as biased estimators (with small biases) to achieve this goal, and devise efficient algorithms to implement them. We found that (1) biased estimators are much more effective than unbiased estimators, especially when the sample is small (e.g., 0.1%); (2) SMARTCRAWL is more robust to data errors than straightforward solutions. Experimental results over simulated and real hidden databases show that SMARTCRAWL can cover a large portion of the local database with a small budget, outperforming straightforward solutions by a factor of $2-7\times$ in a large variety of situations.

1 INTRODUCTION

Many websites, such as Yelp and The ACM Digital Library, curate a large database of entities but only provide a keyword-search interface for the public to access their data. For example, Yelp maintains a large database of restaurants and allows a user to enter a set of keywords (e.g., "Lotus of Siam") to search for a restaurant. These databases are called *deep* web or *hidden* databases. The hidden databases typically contain *rich* as well as *high-quality* information about a substantial number of entities, which is an invaluable external resource for *data enrichment* (enriching data with new attributes) or *data cleaning* (detecting and removing errors from data).

Data enrichment and cleaning are both essential but time-consuming steps in data analysis [21]. Consider a data scientist at a bank who wants to analyze the credit card transaction data of their customers, aiming to infer customer preference (e.g., having a higher preference for Thai food) and provide them better services (e.g., sending them promotional messages about Thai restaurants). She first collects the list of restaurants that all customers have visited and treats it as a local restaurant database.

- *Data Enrichment.* She may find that the local database misses a number of important attributes such as restaurant category or rating, without which it is hard to infer customer preference accurately. For example, suppose a customer often visits a restaurant called "Lotus of Siam". If it is unknown that "Lotus of Siam" is a Thai restaurant, the customer's preference for Thai food may not be inferred. To address this problem, she can find the missing attributes in Yelp and use them to enrich the data.
- *Data Cleaning.* She may also find that some restaurants' information is out of date or incorrect. For instance, if "Lotus of Siam" has an incorrect zip code in the local database, she cannot infer which commercial area that "Lotus of Siam" belongs to, thus the customer's preference of staying in this area may not be inferred. To address this problem, she can use Yelp as a reference database to detect and repair the error in the local database.

To achieve these goals, we need to determine how to retrieve the records in a hidden database $\mathcal{H}$ that match the ones in a local database $\mathcal{D}$, where two records are matching if and only if they refer to the same-world entity (e.g., the same restaurant). We call this problem DeepER (short for *Deep Entity Resolution*). DeepER differs from a typical ER problem only in that one of the databases is hidden behind a keyword-search query interface. This difference raises a couple of new challenges.

First, it is prohibitively expensive and even impossible to issue a lot of queries to a hidden database. For example, Yelp API is restricted to 25,000 free requests per day [1] and Goolge Maps API only allows 2,500 free requests per day [2]. This is a new constraint that has not been considered by existing ER techniques. Second, a hidden database often ranks query results based on an *unknown* ranking function and returns the top-k results only. Since query results are unknown, it is hard to determine which queries are more beneficial (i.e., match more records in the local database).

There are two straightforward solutions to DeepER. (1) NAIVECRAWL. A naive approach is to enumerate each record in $\mathcal{D}$ and then generate a query to match it. For example, a query can be a concatenation of restaurant name and address attributes. However, this approach turns out to be very expensive when $|\mathcal{D}|$ gets larger. For example, suppose $|\mathcal{D}| = 100,000$. Then, this approach has to issue $100,000$ queries to a hidden database. (2) FULLCRAWL. Another approach is to apply a deep-web crawling approach [3, 17, 20, 22, 24, 26–28] to crawl the entire hidden database and then perform ER between the crawled records and the local database. This approach ignores the fact that the goal is to cover the content relating to the local database rather than crawl the entire hidden database.

To overcome the limitations of NAIVECRAWL and FULLCRAWL, we propose the SMARTCRAWL framework. The key insights of SMARTCRAWL are as follows:

- *Query Sharing.* Unlike NAIVECRAWL, SMARTCRAWL generates queries that can match multiple records at a time. Typically, a hidden database sets the top-k restriction with k in a range of 10 to 1000 (e.g., $k = 50$ for Yelp API, $k = 100$ for Google Search API). Suppose $k = 100$. At best, SMARTCRAWL can use a single query to match 100 records, which is 100 times better than NAIVECRAWL.
- *Local-Database-Aware Crawling.* Unlike FULLCRAWL, SMARTCRAWL evaluates the *benefit* of each query based on how many records the query's result can cover in $\mathcal{D}$ (rather than in $\mathcal{H}$). Typically, a hidden database is orders of magnitude larger than a local database. For example, suppose $|\mathcal{D}| = 10^5$ and $|\mathcal{H}| = 10^7$.

Then, FULLCRAWL needs to cover 100 times more records than SMARTCRAWL.

Given a budget for the number of issued queries, SMARTCRAWL first constructs a query pool from $\mathcal{D}$, and then it iteratively selects the query with the largest benefit from the query pool and issues it to the hidden database until the budget is exhausted. Finally, it performs ER to find matching record pairs between $\mathcal{D}$ and $\mathcal{H}_{crawled}$, where $\mathcal{H}_{crawled}$ represents the union of the returned records of all the issued queries.

The last step can be solved by any existing ER technique. Thus, the main focus of our paper is on the query-selection step. We find that this step suffers from a "chicken-and-egg" problem (i.e., it requires knowing the benefit of issuing a query before issuing the query). We solve the problem by estimating the query benefit based on a *hidden database sample*. There is a large body of work on deep web sampling [4, 8–11, 19, 29, 35, 36], aiming to randomly choose a set of records from a hidden database. Recently, Zhang et al. [35] propose efficient techniques that can create an unbiased sample of a hidden database as well as an unbiased estimate of the sampling ratio by issuing keyword-search queries. SMARTCRAWL applies these techniques to create a hidden database sample *offline*. Note that the sample only needs to be created once and can be reused by any user who wants to match their local database with the hidden database.

To have a good estimate of the query benefit, SMARTCRAWL divides the queries into two classes: *solid query* and *overflowing query*, where a solid query will not be affected by the top-k restriction but an overflowing query will. For solid queries, we propose an unbiased estimator as well as a biased estimator to estimate their benefits; for overflowing queries, we also propose two estimators: a conditionally unbiased estimator and a biased estimator. We theoretically analyze the performance of the estimators, and show that the biased estimators are superior to the unbiased ones especially for a small sampling ratio. We finally develop effective indexing techniques to implement these estimators in the SMARTCRAWL framework efficiently.

Extensive experiments over simulated hidden databases show that SMARTCRAWL can significantly outperform NAIVECRAWL and FULLCRAWL in terms of the number of covered records in a large variety of situations.

To summarize, our main contributions are:

- To the best of our knowledge, we are the first to study the DeepER problem. We formalize the problem and present two straightforward solutions.
- To address the DeepER problem, we propose the SMARTCRAWL framework based on the ideas of query sharing and local-database-aware crawling.
- We propose an unbiased estimator and a biased estimator to estimate the benefits of solid queries, and propose a conditionally unbiased estimator and a biased estimator for overflowing queries.
- We develop effective indexing techniques to implement the proposed estimators efficiently.
- We conduct extensive experiments over simulated and real hidden databases. Experiments show that SMARTCRAWL outperforms NAIVECRAWL and FULLCRAWL by a factor of $2-7\times$ in a large variety of situations.

2 PROBLEM FORMALIZATION

Consider a local database $\mathcal{D}$ with $|\mathcal{D}|$ records and a hidden database $\mathcal{H}$ with $|\mathcal{H}|$ (unknown) records. Each record describes a real-world entity. We call each $d \in \mathcal{D}$ a *local record* and each $h \in \mathcal{H}$ a *hidden record*. Local records can be accessed freely but hidden records can only be accessed by issuing queries through a *keyword-search interface*. Without loss of generality, we model a local database and a hidden database as two relational tables over the same set of attributes.

Let $\mathcal{H}_s$ denote a *hidden database sample* and θ denote the corresponding *sampling ratio*. There are a number of sampling techniques that can be used to obtain $\mathcal{H}_s$ and θ [4, 35, 36]. In this paper, we treat deep web sampling as an orthogonal issue and assume that $\mathcal{H}_s$ and θ are given. We implement an existing deep web sampling technique in the experiments and evaluate the performance of SMARTCRAWL using the sample created by the technique (Section 8.3).

Keyword-search Interface. Let q denote a *keyword query* consisting of a set of keywords (e.g., q = "Thai Cuisine"). Each record is modeled as a document, denoted by document$(\cdot)$, which concatenates all[1] the attributes of the record. We consider *conjunctive keyword search*. There are many websites such as IMDb and The ACM Digital Library that supports this type of search interface.

Definition 2.1 (Conjunctive Keyword Search). Given a query, we say a record h (resp. d) *satisfies* the query if and only if document(h) (resp. document(d)) contains *all* the keywords in the query.

Let $q(\mathcal{H})$ (resp. $q(\mathcal{D})$) denote the set of the records in $\mathcal{H}$ (resp. $\mathcal{D}$) that satisfy q. The larger $|q(\mathcal{H})|$ (resp. $|q(\mathcal{D})|$) is, the more frequently the query q appears in $\mathcal{H}$ (resp. $\mathcal{D}$). We call $|q(\mathcal{H})|$ (resp. $|q(\mathcal{D})|$) the *query frequency* w.r.t $\mathcal{H}$ (resp. $\mathcal{D}$).

Due to the top-k restriction, a search interface often enforces a limit on the number of returned records, thus if $|q(\mathcal{H})|$ is larger than k, it will rank the records in $q(\mathcal{H})$ based on an *unknown* ranking function and return the top-k records. We consider deterministic query processing, i.e., the result of a query keeps the same if it is executed again. Definition 2.2 formally defines the keyword-search interface.

Definition 2.2 (Keyword-search Interface). Given a keyword query q, the keyword-search interface of a hidden database $\mathcal{H}$ with the top-k constraint will return $q(\mathcal{H})_k$ as the query result:

$$q(\mathcal{H})_k = \begin{cases} q(\mathcal{H}) & \text{if } |q(\mathcal{H})| \le k \\ \text{The top-}k \text{ records in } q(\mathcal{H}) & \text{if } |q(\mathcal{H})| > k \end{cases}$$

where q is called a *solid query* if $|q(\mathcal{H})| \le k$; otherwise, it is called an *overflowing query*.

Intuitively, for a solid query, we can trust its query result because it has no false negative; however, for an overflowing query, it means that the query result is not completely returned.

Example 2.3. Consider the running example in Figure 1. Figure (a) shows a local database $\mathcal{D}$ with four records, where each record refers to a real-world restaurant. For simplicity, we only keep the name attribute. Figure (b) shows a hidden database sample $\mathcal{H}_s$ with three records randomly drawn from the hidden database (sampling ratio $\theta = \frac{1}{3}$). Figure (c) illustrates the keyword-search interface of the hidden database $\mathcal{H}$ with the top-2 constraint. Each hidden record has an additional rating attribute. The ultimate goal is to enrich the local database with the rating attribute.

[1] If a keyword-search interface does not index all the attributes (e.g., rating and zip code attributes are not indexed by Yelp), we concatenate the indexed attributes only.

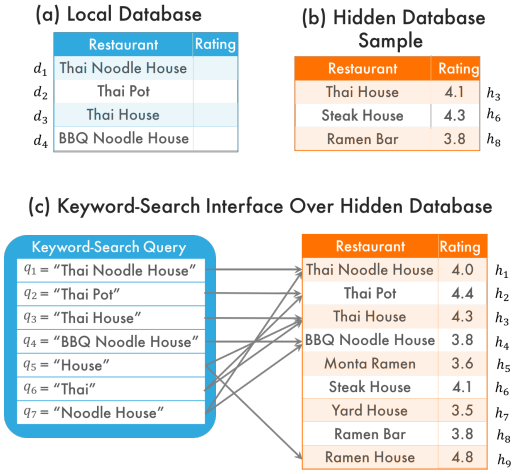


Figure 1: A running example ($k = 2$, $\theta = \frac{1}{3}$). There are four record pairs (i.e., $\langle d_1, h_1\rangle$, $\langle d_1, h_1\rangle$, $\langle d_3, h_3\rangle$, and $\langle d_4, h_4\rangle$) that refer to the same real-world entity. Each arrow points from a query to its result.

For query q_1 = "Thai Noodle House", there is one record (i.e., d_1) that satisfies it in the local database, thus the q_1's frequency w.r.t. $\mathcal{D}$ is $|q_1(\mathcal{D})| = 1$; there is one record (i.e., h_1) that satisfies it in the hidden database, thus the q_1's frequency w.r.t. $\mathcal{H}$ is $|q_1(\mathcal{H})| = 1$. Since $|q_1(\mathcal{H})| \leq k = 2$, q_1 is a solid query. The arrow (starting from q_1) points to the returned record, i.e., $q_1(\mathcal{H})_k = \{h_1\}$. Similarly, for query q_5 = "House", we obtain $|q_5(\mathcal{H})| = 6$. Since $|q_5(\mathcal{H})| > k = 2$, q_5 is an overflowing query. The query result is $q_5(\mathcal{H})_k = \{h_3, h_9\}$.

Deep Entity Resolution. Let b denote a budget for the number of issued queries and Q_{sel} denote the set of the queries selected to be issued. Given a local record d and a hidden record h, $\text{match}(d, h)$ returns True if d and h refer to the same real-world entity; otherwise, $\text{match}(d, h)$ returns False. Each query q returns a subset $q(\mathcal{H})_k$ of $\mathcal{H}$. We say a query q *covers* a local record d if and only if there exists $h \in q(\mathcal{H})_k$ such that $\text{match}(d, h) = \text{True}$. Definition 2.2 formally defines the DeepER problem.

Definition 2.4 (Deep Entity Resolution). Given a local database $\mathcal{D}$, the keyword search interface (with the top-k restriction) over a hidden database $\mathcal{H}$, a hidden database sample $\mathcal{H}_s$ with the sampling ratio of θ, and a budget of b queries, the deep entity resolution problem aims to find all matching record pairs between $\mathcal{D}$ and $\mathcal{H}_{\text{crawl}}$, i.e.,

$$\Big\{\langle d, h\rangle \mid d \in \mathcal{D}, h \in \mathcal{H}_{\text{crawl}}, \text{ and } \text{match}(d, h) = \text{True}\Big\},$$

where $|Q_{sel}| \leq b$ and $\mathcal{H}_{\text{crawl}} = \bigcup_{q \in Q_{\text{sel}}} q(\mathcal{H})_k$.

Our solution to the DeepER problem consists of two steps.

(1) *ER-driven Deep Web Crawling*. The first step is to select no more than b queries, issue them to the hidden database, and get the union of their query results, $\mathcal{H}_{\text{crawl}}$. $\mathcal{H}_{\text{crawl}}$ will be used to match with the records in $\mathcal{D}$ in a downstream ER job. The more records $\mathcal{H}_{\text{crawl}}$ can cover, the more effective the crawling process is. Thus, a key challenge is how to decide which queries should be selected such that $\mathcal{H}_{\text{crawl}}$ can cover the largest number of local records.

(2) *Entity Resolution*. The second step is to perform ER between $\mathcal{D}$ and $\mathcal{H}_{\text{crawl}}$. This turns to be a typical ER problem and can be solved by any existing ER technique. In this paper, we treat this step as a black box and mainly focus on the first step.

Example 2.5. Continuing Example 2.3, suppose the budget is $b = 2$. If the selected query set is $Q_{\text{sel}} = \{q_1, q_5\}$, then after issuing the queries to the hidden database, we obtain their query results, $q_1(\mathcal{H})_k = \{h_1\}$ and $q_5(\mathcal{H})_k = \{h_3, h_9\}$, respectively, thus the crawled record set is $\mathcal{H}_{\text{crawl}} = q_1(\mathcal{H})_k \cup q_5(\mathcal{H})_k = \{h_1, h_3, h_9\}$. We can see that $\text{match}(h_1, d_1) = \text{True}$, $\text{match}(h_3, d_3) = \text{True}$, but h_9 does not match with any local record, thus DeepER (w.r.t. Q_{sel}) returns $\{\langle d_1, h_1\rangle, \langle d_3, h_3\rangle\}$.

However, if the selected query set is $Q'_{\text{sel}} = \{q_6, q_7\}$, their query results are $q_6(\mathcal{H})_k = \{h_2, h_3\}$ and $q_7(\mathcal{H})_k = \{h_1, h_4\}$, thus the crawled record set is $\mathcal{H}_{\text{crawl}} = q_6(\mathcal{H})_k \cup q_7(\mathcal{H})_k = \{h1, h2, h3, h4\}$, which can match all local records. DeepER (w.r.t. Q'_{sel}) returns $\{\langle d_1, h_1\rangle, \langle d_2, h_2\rangle, \langle d_3, h_3\rangle, \langle d_4, h_4\rangle\}$. This example illustrates the importance of query selection.

Problem Statement. We now formally define the problem of ER-driven Deep Web Crawling. To facilitate analysis, we consider exact matching in later text (Assumption 1) and will discuss how to extend our framework to fuzzy matching in Section 7.2.

ASSUMPTION 1. *For any $d \in \mathcal{D}$ and $h \in \mathcal{H}$, we assume that match (d, h) = True if and only if document(d) = document(h).*

We model both $\mathcal{H}$ and $\mathcal{D}$ as a set (i.e., a collection of *distinct* records)[2]. Given $\mathcal{D}' \subseteq \mathcal{D}$ and $\mathcal{H}' \subseteq \mathcal{H}$, we define the intersection between $\mathcal{D}'$ and $\mathcal{H}'$ as

$$\mathcal{D}' \cap \mathcal{H}' = \{d \in \mathcal{D}' \mid h \in \mathcal{H}', \text{match}(d, h) = True\}$$

Let $q(\mathcal{D})_{\text{cover}} \subseteq q(\mathcal{D})$ denote the set of local records that can be covered by q, i.e.,

$$q(\mathcal{D})_{\text{cover}} = q(\mathcal{D}) \cap q(\mathcal{H})_k \tag{1}$$

The goal of ER-driven deep web crawling is to select a set of queries within the budget such that $|\bigcup_{q \in Q_{\text{sel}}} q(\mathcal{D})_{\text{cover}}|$ is maximized.

PROBLEM (ER-DRIVEN DEEP WEB CRAWLING). *Given a local database $\mathcal{D}$, the keyword search interface (with the top-k restriction) over a hidden table $\mathcal{H}$, a hidden database sample $\mathcal{H}_s$ with the sampling ratio of θ, and a budget of b queries, the goal of ER-driven deep web crawling is to select a set of queries, Q_{sel}, to maximize the coverage of $\mathcal{D}$, i.e.,*

$$\max \quad \Big|\bigcup_{q \in Q_{sel}} q(\mathcal{D})_{cover}\Big|$$
$$s.t. \quad |Q_{sel}| \leq b$$

Example 2.6. Consider q_5 = "House" in Figure 1. After issuing the query, we get the query result $q_5(\mathcal{H})_k = \{h_3, h_9\}$. Since $q_5(\mathcal{D}) = \{d_1, d_3, d_4\}$, based on Equation 1 we obtain $q_5(\mathcal{D})_{\text{cover}} = \{d_3\}$. Table 1 lists the values of $q(\mathcal{D})_{\text{cover}}$ for all the queries (Please only look at the first two columns; the other columns will be explained in later sections.). Suppose $b = 2$. The goal of the problem is to select two queries q_i, q_j from $\{q_1, q_2, \cdots, q_7\}$ in order to maximize $|q_i(\mathcal{D})_{\text{cover}} \cup q_j(\mathcal{D})_{\text{cover}}|$. It is easy to see that the optimal solution should select q_6 and q_7 since $|q_6(\mathcal{D})_{\text{cover}} \cup q_7(\mathcal{D})_{\text{cover}}| = 4$ reaches the maximum.

[2]Since the data in a hidden database $\mathcal{H}$ is of high-quality, it is reasonable to assume that $\mathcal{H}$ has no duplicate record. For a local database $\mathcal{D}$, if it has duplicate records, we will remove them before matching it with $\mathcal{H}$.

Table 1: True benefit along with two estimated benefits. SMARTCRAWL predicts q_1, q_2, q_4, q_7 as solid queries and adopts the estimators proposed in Section 4 for these queries; it predicts the other queries as overflowing and applies the estimators presented in Section 5 to them.

Queries	$q(\mathcal{D})_{\text{cover}}$	True Benefit	Unbiased Estimator	Biased Estimator
q_1	$\{d_1\}$	1	0	1
q_2	$\{d_2\}$	1	0	1
q_4	$\{d_4\}$	1	0	1
q_7	$\{d_2, d_3\}$	2	0	2
q_3	$\{d_3\}$	1	2	$\frac{2}{3}$
q_5	$\{d_3\}$	1	1	1
q_6	$\{d_1, d_4\}$	2	2	2

Unfortunately, ER-driven Deep Web Crawling is NP-Hard, which can be proved by a reduction from the maximum-coverage problem. In fact, what makes this problem exceptionally challenging is that the greedy algorithm that can be used to solve the maximum-coverage problem is not applicable (see the reason in the next section).

3 SMARTCRAWL FRAMEWORK

In the Introduction section, we present two straightforward crawling approaches: (1) NAIVECRAWL selects a set of queries, where each query is to cover one local record at a time; (2) FULLCRAWL selects a set of queries in order to crawl as many records from the hidden database as possible. However, they fail to exploit the ideas of query sharing and local-database-aware crawling. In response, we propose SMARTCRAWL, a novel framework to overcome the limitations.

Framework Overview. Figure 2 depicts the SMARTCRAWL framework. It has two stages: query pool generation and query selection.

In order to leverage the power of query sharing, SMARTCRAWL initializes a *query pool* by extracting queries from $\mathcal{D}$, where the query pool does not only contain the queries that can cover a single local record (like NAIVECRAWL) but also the ones that can cover multiple local records.

In order to leverage the power of local-database-aware crawling, SMARTCRAWL selects the *best* query at each iteration, where the best query is determined by the query frequency w.r.t. not only the hidden database (like FULLCRAWL) but also the local database. Once a query q^* is selected, SMARTCRAWL issues q^* to the hidden database, gets the covered records $q^*(\mathcal{D})_{\text{cover}}$, updates the query pool, and merge $q^*(\mathcal{H})_k$ into $\mathcal{H}_{\text{crawl}}$. This iterative process will repeat until the budget is exhausted or the local database is fully covered.

We now present these two stages in detail.

(1) Query Pool Generation. Let Q denote a query pool. If a query q does not appear in any local record, i.e., $|q(\mathcal{D})| = 0$, there is no need to issue the query because it cannot cover any local record. Therefore, there is a finite number of queries that need to be considered, i.e.,

$$\mathbb{Q} = \{q \mid |q(\mathcal{D})| \geq 1\}.$$

Let $|d|$ denote the number of distinct keywords in d. Since each local record can produce $2^{|d|} - 1$ queries, the total number of all possible queries is still very large, i.e., $|\mathbb{Q}| = \sum_{d \in \mathcal{D}} 2^{|d|} - 1$. Thus, we adopt a heuristic approach to generate a subset of $\mathbb{Q}$ as the query pool.

There are two basic principles underlying the design of the approach. First, we hope the query pool to be able to take care of every local record. Second, the query pool should include queries that can cover multiple local records at a time.

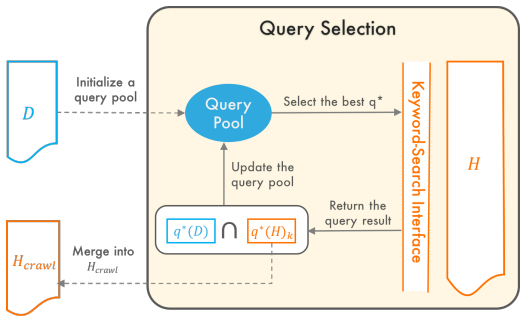


Figure 2: The SMARTCRAWL Framework.

- To satisfy the first principle, SMARTCRAWL adopts the method used by NAIVECRAWL. That is, for each local record, SMARTCRAWL concatenates a set of attributes that can uniquely identify the record within the hidden database and adds it into the query pool. Let Q_{naive} denote the collection of the queries generated in this step. We have $|Q_{naive}| = |\mathcal{D}|$.
- To satisfy the second principle, SMARTCRAWL finds the queries whose frequency w.r.t. $\mathcal{D}$ are no less than t, where t is a user-specified threshold. We can efficiently generate these queries using Frequent Pattern Mining algorithms (e.g., [15]). Specifically, we treat each keyword as an item, then use a frequent pattern mining algorithm to find the itemsets that appear in $\mathcal{D}$ with frequency no less than t, and finally converts the frequent itemsets into queries.

From the above two steps, SMARTCRAWL will generate a query pool as follows:

$$Q = Q_{naive} \cup \{q \mid |q(\mathcal{D})| \geq t\}.$$

Furthermore, we remove the queries *dominated* by the others in the query pool. We say a query q_1 dominates a query q_2 if $|q_1(\mathcal{D})| = |q_2(\mathcal{D})|$ and q_1 contains all the keywords in q_2.

Example 3.1. The seven queries, $\{q_1, q_2, \cdots, q_7\}$, in Figure 1(c) are generated using the method above. Suppose $t = 2$. Based on the first principle, we generate $Q_{naive} = \{q_1, q_2, q_3, q_4\}$, where each query uses the full restaurant name; based on the second principle, we first find the itemsets {"House", "Thai", "Noodle House", "Noodle"} with frequency no less than 2, and then remove "Noodle" since this query is dominated by "Noodle House", and finally obtain q_5 = "House", q_6 = "Thai", and q_7 = "Noodle House".

(2) Query Selection. After a query pool is generated, SMARTCRAWL enters the query-selection stage. Let us first take a look at how the *ideal* greedy algorithm works.

As shown in Algorithm 1, QSEL-IDEAL iteratively selects the query with the largest *benefit* from the query pool, where the benefit is defined as $|q(\mathcal{D})_{\text{cover}}|$. That is, in each iteration, the query that covers the largest number of uncovered local records will be selected. After a query q^* is selected, the algorithm issues q^* to the hidden database, and gets the query result. Then, it updates $\mathcal{D}$ and Q, and goes to the next iteration.

Chicken and Egg Problem. The greedy algorithm suffers from a "chicken and egg" problem. That is, it cannot get the true benefit, $|q(\mathcal{D})_{\text{cover}}|$, of each query until the query is issued, but it needs to know $|q(\mathcal{D})_{\text{cover}}|$ in order to decide which query should be issued. To overcome the problem, we seek to use a hidden database sample to estimate the benefit of each query and then use the estimated benefit to determine which query should be issued.

Algorithm 1: QSEL-IDEAL Algorithm

Input: Q, $\mathcal{D}$, $\mathcal{H}$, b
Result: Iteratively select the query with the largest benefit.

```
1 while b > 0 and D ≠ φ do
2     for each q ∈ Q do
3         benefit(q) = |q(D)_cover|;
4     end
5     Select q* with the largest benefit from Q;
6     Issue q* to the hidden database, and then get the result q*(H)_k;
7     D = D − q*(D)_cover;    Q = Q − {q*};    b = b − 1;
8 end
```

Algorithm 2: QSEL-EST Algorithm

```
1 Replace Line 3 in Algorithm 1 with the following lines:
2     if |q(H_s)|/θ ≤ k then
3         Set benefit(q) to the estimated benefit (Section 4) // Solid Query;
4     else
5         Set benefit(q) to the estimated benefit (Section 5) //Overflowing;
6     end
```

Algorithm 2 depicts the pseudo-code of the algorithm. We can see that QSEL-EST differs from QSEL-IDEAL only in the benefit calculation part. To estimate the benefit of a query, QSEL-EST first uses the sample $\mathcal{H}_s$ to predict whether the query is solid or overflowing, and then applies a corresponding estimator. Specifically, QSEL-EST computes the query frequency w.r.t. $\mathcal{H}_s$ and uses it to estimate the query frequency w.r.t. $\mathcal{H}$. If the estimated query frequency, $\frac{|q(\mathcal{H}_s)|}{\theta}$, is no larger than k, it will be predicated as a solid query; otherwise, it will be considered as an overflowing query. We will detail how to estimate the benefits for the two different types of queries in Section 4 and Section 5, respectively.

Example 3.2. Consider q_1 = "Thai Noodle House" in Figure 1. Since the q_1's frequency w.r.t. $\mathcal{H}_s$ is zero, the q_1's estimated frequency w.r.t. $\mathcal{H}$ is $\frac{|q(\mathcal{H}_s)|}{\theta} = \frac{0}{1/3} \leq k$, thus QSEL-EST predicates it as a solid query, which is a correct prediction. Consider q_5 = "House". Since the q_5's frequency w.r.t. $\mathcal{H}_s$ is 2, the q_5's estimated frequency w.r.t. $\mathcal{H}$ is $\frac{|q(\mathcal{H}_s)|}{\theta} = \frac{2}{1/3} = 6 > k$, thus QSEL-EST predicates it as an overflowing query, which is also a correct prediction. In summary, QSEL-EST predicts q_1, q_2, q_4, q_7 as solid queries and q_3, q_5, q_6 as overflowing queries. The only wrong prediction is to predict q_3 as a solid query.

4 ESTIMATORS FOR SOLID QUERIES

In this section, we present benefit estimation for solid queries. Section 4.1 provides an unbiased estimator as well as the reasons for its ineffectiveness. Section 4.2 proposes a biased estimator and proves its good performance theoretically.

4.1 Unbiased Estimator

The true benefit of a query is defined as:

$$\text{benefit}(q) = |q(\mathcal{D})_{cover}| = |q(\mathcal{D}) \cap q(\mathcal{H})_k|. \tag{2}$$

According to the definition of solid queries in Definition 2.2, if q is a solid query, all the hidden records that satisfy the query can be returned, i.e., $q(\mathcal{H})_k = q(\mathcal{H})$. Thus, the benefit of a solid query is

$$\text{benefit}(q) = |q(\mathcal{D}) \cap q(\mathcal{H})|. \tag{3}$$

Once the top-k constraint is eliminated, the problem of benefit estimation can be modeled as a selectivity estimation problem, which aims to estimate the selectivity of the following query:

```
SELECT d, h FROM D, H
WHERE d satisfies q AND h satisfies q AND match(d, h) = True.
```

An *unbiased* estimator of the selectivity based on the hidden database sample $\mathcal{H}_s$ is:

$$\text{benefit}(q) \approx \frac{|q(\mathcal{D}) \cap q(\mathcal{H}_s)|}{\theta}, \tag{4}$$

which means that the estimator's expected value is equal to the true benefit.

LEMMA 4.1. *Given a solid query q, then $\frac{|q(\mathcal{D}) \cap q(\mathcal{H}_s)|}{\theta}$ is an unbiased estimator of $|q(\mathcal{D}) \cap q(\mathcal{H})|$.*

PROOF. All the proofs in this paper can be found in Appendix A. □

However, this estimator tends to produce highly inaccurate results. In practice, the hidden database sample $\mathcal{H}_s$ cannot be very large. As a result, $|q(\mathcal{D}) \cap q(\mathcal{H}_s)|$ will be 0 for most $q \in Q$. For example, consider a sampling ratio of $\theta = 1\%$. For any query q with $|q(\mathcal{D})| < 100$, $|q(\mathcal{D}) \cap q(\mathcal{H}_s)| = 0$ (in expectation). Furthermore, the possible values of estimated benefits are very coarse-grained, which can only be 0, 100, 200, 300, etc. As a result, many queries will have the same benefit which is not helpful for query selection.

Example 4.2. Consider q_1 = "Thai Noodle House" in Figure 1. Since $q_1(\mathcal{D}) = \{d_1\}$ and $q_1(\mathcal{H}_s) = \phi$, then benefit($q_1$) is estimated as $\frac{|q_1(\mathcal{D}) \cap q_1(\mathcal{H}_s)|}{\theta} = \frac{0}{1/3} = 0$, which suggests that selecting this query has no benefit. As shown in Table 1 (see the top half part of the 4th column), there are four queries (i.e., q_1, q_2, q_4, q_7) that are predicted as solid queries and their benefits are all being incorrectly estimated.

4.2 Biased Estimator

We propose another estimator to overcome the limitations. While this estimator is biased, the bias is often very small in practice. Furthermore, we prove that the impact of the bias on the query selection algorithm can be bounded.

Let $\Delta\mathcal{D} = \mathcal{D} \setminus \mathcal{H}$ denote the set of the records in $\mathcal{D}$ but not in $\mathcal{H}$, and $q(\Delta\mathcal{D})$ denote the set of the records in $\Delta\mathcal{D}$ that satisfy q. Then, the benefit of a solid query (Equation 3) can be denoted by

$$\text{benefit}(q) = |q(\mathcal{D}) - q(\Delta\mathcal{D})| = |q(\mathcal{D})| - |q(\Delta\mathcal{D})|. \tag{5}$$

Key Observation. We observe that a hidden database (e.g., Yelp, IMDb) often has a very good coverage of the entities in some domain (e.g., Restaurant, Movie, etc.). If one wants to match a local database with a hidden database, it is reasonable to assume that most of the records in the local database can be found in the hidden database.

This observation implies that $\Delta\mathcal{D}$ is often small, thus $|q(\Delta\mathcal{D})|$, as a subset of $\Delta\mathcal{D}$, is even much smaller. For this reason, we derive the following estimator:

$$\text{benefit}(q) \approx |q(\mathcal{D})|, \tag{6}$$

where the bias of the estimator is $|q(\Delta\mathcal{D})|$.

Algorithm 3: QSEL-BOUND Algorithm

Input: Q consists of solid queries only; $\mathcal{D}$, $\mathcal{H}$, b

Result: QSEL-BOUND covers at least $(1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot n_{\text{ideal}}$ local records.

```
1  while b > 0 and D ≠ φ do
2      for each q ∈ Q do
3          benefit(q) = |q(D)|;
4      end
5      Issue q* to the hidden database, and then get the result q*(H)_k;
6      q*(ΔD) = q*(D) − q*(D)_cover;
7      if |q*(ΔD)| = 0 then
8          D = D − q*(D)_cover;   Q = Q − {q*};
9      else
10         D = D − q*(ΔD);   // Note that q* is not removed from the pool.;
11     end
12     b− = 1;
13 end
```

Example 4.3. Consider q_1 = "Thai Noodle House" again. Since $q_1(\mathcal{D}) = \{d_1\}$, then benefit($q_1$) will be estimated as $|q_1(\mathcal{D})| = 1$, which is equal to the true benefit. As shown in Table 1 (see the top half part of the 5th column), the benefits of q_1, q_2, q_4, q_7 are all being correctly estimated.

We now provide theoretical analysis to justify the effectiveness of the estimator.

CASE 1 ($\mathcal{D} \subseteq \mathcal{H}$). Intuitively, the estimator suggests that we only need to use the query frequency w.r.t. $\mathcal{D}$ to quantify how beneficial a query is. As it ignores the query frequency w.r.t. $\mathcal{H}$, one may be tempted to think that this will only work if $\mathcal{D}$ and $\mathcal{H}$ follow the same distribution, i.e., $\mathcal{D}$ has to be a random sample of $\mathcal{H}$. But, in fact, as long as $\mathcal{D}$ is a subset of $\mathcal{H}$, this estimator will always return the true benefit. Since we have $\mathcal{D} \subseteq \mathcal{H}$, then we have $|q(\Delta\mathcal{D})| = 0$, thus benefit($q$) = $|q(\mathcal{D})|$ (based on Equation 5), i.e., the true benefit is equal to the estimated benefit. More interestingly, for a query pool with solid queries only, QSEL-EST is equivalent to QSEL-IDEAL, where the former selects queries based on estimated benefits (Equation 6) while the latter selects queries based on true benefits (Equation 3).

LEMMA 4.4. *Given a query pool Q that consists of solid queries only , if $\mathcal{D} \subseteq \mathcal{H}$, then* QSEL-IDEAL *and* QSEL-EST *are equivalent.*

CASE 2 ($\mathcal{D} \not\subseteq \mathcal{H}$). In this case, we find that it is not easy to directly reason about the performance difference between QSEL-IDEAL and QSEL-EST. Thus, we construct a new algorithm, called QSEL-BOUND, as a proxy. We first prove that the performance of QSEL-BOUND w.r.t. QSEL-IDEAL can be bounded, and then demonstrate that QSEL-EST is preferable to QSEL-BOUND.

As the same as QSEL-EST, QSEL-BOUND selects the query with the largest $|q(\mathcal{D})|$ at each iteration. The difference between them is how to react to the selected query. Suppose the selected query is q^*. There are two situations about q^*. (1) $|q(\mathcal{D})|$ is equal to the true benefit. In this situation, QSEL-BOUND will behave the same as QSEL-EST. (2) $|q(\mathcal{D})|$ is *not* equal to the true benefit. In this situation, QSEL-BOUND will keep q^* in the query pool and remove $q(\Delta\mathcal{D})$ (rather than $q^*(\mathcal{D})_{\text{cover}}$ like QSEL-EST) from $\mathcal{D}$. To know which situation q^* belongs to, QSEL-BOUND first issues q^* to the hidden database and then checks whether $q^*(\mathcal{D}) = q^*(\mathcal{D})_{\text{cover}}$ holds. If yes, it means that $|q^*(\Delta\mathcal{D})| = 0$, thus q^* belongs to the first situation; otherwise, it belongs to the second one. Algorithm 3 depicts the pseudo-code of QSEL-BOUND.

To compare the performance of QSEL-BOUND and QSEL-IDEAL, let $Q_{\text{sel}} = \{q_1, q_2, \cdots, q_b\}$ and $Q'_{\text{sel}} = \{q'_1, q'_2, \cdots, q'_b\}$ denote the set of the queries selected by QSEL-IDEAL and QSEL-BOUND, respectively. Let N_{ideal} and N_{bound} denote the number of local records that can be covered by QSEL-IDEAL and QSEL-BOUND, respectively i.e.,

$$N_{\text{ideal}} = |\cup_{q \in Q_{\text{sel}}} q(\mathcal{D})_{\text{cover}}|, \quad N_{\text{bound}} = |\cup_{q' \in Q'_{\text{sel}}} q'(\mathcal{D})_{\text{cover}}|.$$

We find that $N_{\text{bound}} \geq (1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}$. The following lemma proves the correctness.

LEMMA 4.5. *Given a query pool Q that consists of solid queries only, the worst-case performance of* QSEL-BOUND *is bounded w.r.t.* QSEL-IDEAL*, i.e., $N_{bound} \geq (1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{ideal}$.*

PROOF SKETCH. The proof consists of two parts. In the first part, we prove that the first $(b - |\Delta\mathcal{D}|)$ queries selected by QSEL-IDEAL must be selected by QSEL-BOUND, i.e.,

$$\{q_i \mid 1 \leq i \leq b - |\Delta\mathcal{D}|\} \subseteq Q'_{sel}.$$

This can be proved by induction. In the second part, we prove that the first $(b - |\Delta\mathcal{D}|)$ queries selected by QSEL-IDEAL can cover at least $(1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}$ local records. This can be proved by contradiction. The full proof of the lemma is provided in the Appendix. □

The lemma indicates that when $|\Delta\mathcal{D}|$ is relatively small w.r.t. b, QSEL-BOUND performs almost as good as QSEL-IDEAL. For example, consider a local database having $|\Delta\mathcal{D}| = 10$ records not in a hidden database. Given a budget of 1000 queries, if QSEL-IDEAL covers $N_{\text{ideal}} = 10,000$ local records, then QSEL-BOUND can cover at least $(1 - \frac{10}{1000}) \cdot 10,000 = 9,900$ local records, which is only 1% smaller than N_{ideal}.

QSEL-EST vs. QSEL-BOUND. Note that both QSEL-EST and QSEL-BOUND are applicable in practice, but we empirically find that QSEL-EST often performs better than QSEL-BOUND. The reason is that, to ensure the theoretical guarantee, QSEL-BOUND is forced to keep some queries, which have already been selected, into the query pool (see Line 11 in Algorithm 3). These queries may be selected again in later iterations and thus waste the budget. Because of this, although the worse-case performance of QSEL-BOUND can be bounded, we still stick to QSEL-EST.

5 ESTIMATORS FOR OVERFLOW QUERIES

In this section, we present benefit estimation for overflowing queries. We first explain why the estimator $|q(\mathcal{D})|$ does not work for overflowing queries in Section 5.1, and then propose two new estimators in Section 5.2.

5.1 Why Does $|q(\mathcal{D})|$ Not Work Anymore?

The estimator $|q(\mathcal{D})|$ only considers the query frequency w.r.t. $\mathcal{D}$ without considering the query frequency w.r.t. $\mathcal{H}$, i.e., $|q(\mathcal{H})|$. It works well for solid queries because for a solid query, all of the hidden records that satisfy the query can be returned. But, for an overflowing query, only the top-k hidden records can be returned. The larger $|q(\mathcal{H})|$ is, the harder the *targeted hidden records* (i.e., the ones that covers local records) can be retrieved. For example, consider the overflowing query q_5="House" in Figure 1, where $|q_5(\mathcal{D})| = 3$

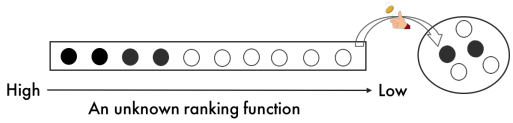


Figure 3: An illustration of breaking the top-k constraint.

and $|q_5(\mathcal{H})| = 6$. If the benefit of q_5 is estimated as $|q_5(\mathcal{D})| = 3$, then the query will have the largest estimated benefit and should be selected right away. However, $|q_5(\mathcal{H})| = 6$ is also the largest among all queries. As shown in Table 1, the true benefit of the query is only 1, which is smaller than the true benefits of q_6 and q_7. Thus, the selection of q_5 should be postponed.

5.2 Benefit Estimation

How should we combine $|q(\mathcal{D})|$, $|q(\mathcal{H})|$, and k systematically in order to derive an estimator for an overflowing query? We call it the problem of *breaking the top-k constraint*. Note that the ranking function of a hidden database is unknown, thus the returned top-k records *cannot* be modeled as a random sample of $q(\mathcal{H})$. Next, we present the basic idea of our solution through an analogy.

Basic Idea. Suppose there are a list of N balls that are sorted based on an *unknown* ranking function. Suppose the first k balls in the list are black and the remaining ones are white. If we randomly draw a set of n balls without replacement from the list, how many black balls will be chosen in draws? This is a well studied problem in probability theory and statistics. The number of black balls in the set is a random variable X that follows a *hypergeometric distribution*, where the probability of $X = i$ (i.e., having i black balls in the set) is

$$P(X = i) = \frac{\binom{k}{i}\binom{N-k}{n-i}}{\binom{N}{n}}.$$

It can be proved that the expected number of black balls is

$$E[X] = \sum_{i=0}^{n} i \cdot P(X = i) = n\frac{k}{N}. \tag{7}$$

Intuitively, every draw chooses $\frac{k}{N}$ black ball in expectation. After n draws, $n\frac{k}{N}$ black ball(s) will be chosen. For example, in Figure 3, there are 10 balls in the list and the top-4 are black. If randomly choosing 5 balls from the list, the expected number of the black balls that are chosen is $5 \times \frac{4}{10} = 2$.

Breaking the Top-k Constraint. We apply the idea to our problem. Recall that the benefit of an overflowing query is defined as

$$\mathsf{benefit}(q) = |q(\mathcal{D}) \cap q(\mathcal{H})_k|,$$

where $q(\mathcal{H})_k$ denotes the top-k records in $q(\mathcal{H})$. We model $q(\mathcal{H})$ as a list of balls, $q(\mathcal{H})_k$ as black balls, $q(\mathcal{D}) - q(\mathcal{H})_k$ as white balls, and $q(\mathcal{D}) \cap q(\mathcal{H})$ as a set of balls randomly drawn from $q(\mathcal{H})$. Then, estimating the benefit of a query is reduced as estimating the number of black balls in draws. Based on Equation 7, we have

$$E[\mathsf{benefit}(q)] = n \cdot \frac{k}{N} = |q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|} \tag{8}$$

The equation holds under the assumption that $q(\mathcal{D}) \cap q(\mathcal{H})$ is a random sample of $q(\mathcal{H})$. If $q(\mathcal{D}) \cap q(\mathcal{H})$ is a biased sample (i.e., each black ball and white ball have different weights to be sampled), the number of black balls in draws follow *Fisher's noncentral hypergeometric distribution*. Suppose the probability of choosing each ball is proportional to its weight. Let ω_1 and ω_2 denote the weights of each black and white ball, respectively. Let $\omega = \frac{\omega_1}{\omega_2}$ denote the odds ratio. Then, the expected number of black balls in draws can be represented as a function of ω. As an analogy, a higher weight for black balls means that top-k records are more likely to cover a local table than non-top-k records . Since a local table is provided by a user, it is hard for a user to specify the parameter ω for the table, thus we assume $\omega = 1$ (i.e., $q(\mathcal{D}) \cap q(\mathcal{H})$ is a random sample of $q(\mathcal{H})$) in the paper.

Table 2: A summary of our estimators for query benefits.

	Unbiased	**Biased (w/ small biases)**
Solid	$\frac{\lvert q(\mathcal{D}) \cap q(\mathcal{H}_s)\rvert}{\theta}$	$\lvert q(\mathcal{D})\rvert$
Overflowing	$\lvert q(\mathcal{D}) \cap q(\mathcal{H}_s)\rvert \cdot \frac{k}{\lvert q(\mathcal{H}_s)\rvert}$	$\lvert q(\mathcal{D})\rvert \cdot \frac{k\theta}{\lvert q(\mathcal{H}_s)\rvert}$

Estimators. Note that Equation 8 is not applicable in practice because $q(\mathcal{H})$ and $|q(\mathcal{D}) \cap q(\mathcal{H})|$ are unknown. We estimate them based on the hidden database sample $\mathcal{H}_s$.

For $|q(\mathcal{H})|$, which is the number of hidden records that satisfy q, the unbiased estimator is.

$$|q(\mathcal{H})| \approx \frac{|q(\mathcal{H}_s)|}{\theta}. \tag{9}$$

For $|q(\mathcal{D}) \cap q(\mathcal{H})|$, which is the number of hidden records that satisfy q and are also in $\mathcal{D}$, we have studied how to estimate it in Section 4. The unbiased estimator (see Equation 4) is

$$|q(\mathcal{D}) \cap q(\mathcal{H})| \approx \frac{|q(\mathcal{D}) \cap q(\mathcal{H}_s)|}{\theta} \tag{10}$$

The biased estimator (see Equation 6) is

$$|q(\mathcal{D}) \cap q(\mathcal{H})| \approx |q(\mathcal{D})| \tag{11}$$

By plugging Equations 9 and 10 into $|q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|}$, we obtain the first estimator for an overflowing query:

$$\mathsf{benefit}(q) \approx |q(\mathcal{D}) \cap q(\mathcal{H}_s)| \cdot \frac{k}{|q(\mathcal{H}_s)|} \tag{12}$$

This estimator is derived from the ratio of two unbiased estimators. Since $E[\frac{X}{Y}] \neq \frac{E[X]}{E(Y)}$, Equation 30 is not an unbiased estimator, but it is conditionally unbiased (Lemma 5.1). For simplicity, we omit "conditionally" if the context is clear.

LEMMA 5.1. *Given an overflowing query q, if $q(\mathcal{D}) \cap q(\mathcal{H})$ is a random sample of $q(\mathcal{H})$, then $|q(\mathcal{D}) \cap q(\mathcal{H}_s)| \cdot \frac{k}{|q(\mathcal{H}_s)|}$ is a conditionally unbiased estimator of the true benefit given $|q(\mathcal{H}_s)|$ regardless of the underlying ranking function.*

This estimator suffers from the same issue as the unbiased estimator proposed for a solid query. That is, the possible values of $|q(\mathcal{D}) \cap q(\mathcal{H}_s)|$ are coarse-grained and have a high chance to be 0.

Example 5.2. Consider q_3 = "Thai House" in Figure 1, where $q_3(\mathcal{D}) = \{d_3\}$ and $q_3(\mathcal{H}_s) = \{h_3\}$. Since $\mathsf{match}(d_3, h_3)$ = True, then $|q(\mathcal{D}) \cap q(\mathcal{H}_s)| = 1$, thus $\mathsf{benefit}(q_3)$ is estimated as $|q_3(\mathcal{D}) \cap q_3(\mathcal{H}_s)| \cdot \frac{k}{|q_3(\mathcal{H}_s)|} = 1 \cdot \frac{2}{1} = 2$. In comparison, the true benefit of the query is $\mathsf{benefit}(q_3) = 1$ (see Table 1).

By plugging Equations 9 and 11 into $|q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|}$, we obtain another estimator:

$$\mathsf{benefit}(q) \approx |q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|} \tag{13}$$

We can deduce that this estimator is biased, where the bias is

$$bias = |q(\Delta\mathcal{D})| \cdot \frac{k}{|q(\mathcal{H})|} \tag{14}$$

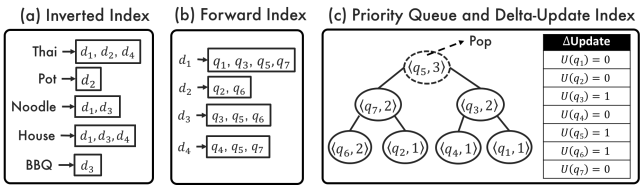


Figure 4: An illustration of the indexing techniques for efficient implementations of our estimators.

LEMMA 5.3. *Given an overflowing query q, if $q(\mathcal{D}) \cap q(\mathcal{H})$ is a random sample of $q(\mathcal{H})$, then $|q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|}$ is a biased estimator where the bias is $|q(\Delta\mathcal{D})| \cdot \frac{k}{|q(\mathcal{H})|}$ regardless of the underlying ranking function.*

As discussed in Section 4.2, $q(\Delta\mathcal{D})$ is often very small in practice. Since q is an overflowing query, then $\frac{k}{|q(\mathcal{H})|} < 1$. Hence, the bias of the estimator is small as well.

Example 5.4. Consider q_3 = "Thai House" again. Since $|q_3(\mathcal{D})| = 1$ and $|q_3(\mathcal{H}_s)| = 1$, then benefit(q_3) is estimated as $|q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|} = 1 \cdot \frac{2 \cdot 1/3}{1} = \frac{2}{3}$, which is more close to the true benefit compared to Example 5.2.

Example 5.5. We now illustrate how QSEL-EST uses our biased estimators to select queries. Table 1 (the fifth column) shows the estimated benefits for all queries using the biased estimator. Suppose $b = 2$. In the first iteration, QSEL-EST selects q_6 which has the largest estimated benefit (if there is a tie, we break the tie randomly), and issues it to the hidden database. The returned result can cover two local records $q_6(\mathcal{D})_{\text{cover}} = \{d_1, d_4\}$. QSEL-EST removes the covered records from $\mathcal{D}$ and re-estimate the benefit of each query w.r.t. the new $\mathcal{D}$. In the second iteration, QSEL-EST selects q_7 which has the largest estimated among the remaining queries, and issue it to the hidden database. The returned result can cover $q_7(\mathcal{D})_{\text{cover}} = \{d_2, d_3\}$. Since the budget is exhausted, QSEL-EST stops and returns $\mathcal{H}_{\text{crawl}} = \{d_1, d_2, d_3, d_4\}$. We can see that QSEL-EST achieves the optimal solution for our running example.

6 EFFICIENT IMPLEMENTATIONS

Table 2 summarizes the four estimators presented in Sections 4 and 5. In this section, we present their efficient implementations, which involve three problems: (1) how to compute them efficiently; (2) how to update them efficiently (when $\mathcal{D}$ is updated); (3) how to select the best query efficiently? We will first study these problems for the estimator $|q(\mathcal{D})|$ and then extend to other estimators.

How to compute $|q(\mathcal{D})|$ efficiently? We build an *inverted index* on $\mathcal{D}$ to compute $|q(\mathcal{D})|$ efficiently. An inverted index maps each keyword to a list of local records that contain the keyword. Such a list is called an inverted list. To build the inverted index, we initialize a hash map I and let $I(w)$ denote the inverted list of keyword w. For each local record $d \in \mathcal{D}$, we enumerate each keyword in document(d) and add d into $I(w)$. Please note that the inverted index only needs to be built once and will benefit to all future queries. Given a query q, to compute $|q(\mathcal{D})|$, we first find the inverted list of each keyword in the query, and then get the intersection of the lists, i.e., $|q(\mathcal{D})| = |\bigcap_{w \in q} I(w)|$. Figure 4(a) shows the inverted index built on the local database of the running example. Given the query q_7 = "Noodle House", we get the inverted lists I("Noodle") $= \{d_1, d_4\}$ and I("House") $= \{d_1, d_3, d_4\}$, and then compute $q_3(\mathcal{D}) = I$("Noodle") $\cap\, I$("House") $= \{d_1, d_4\}$.

How to update $|q(\mathcal{D})|$ efficiently? After each iteration, some records (covered by the selected query) need to be removed from $\mathcal{D}$, thus $|q(\mathcal{D})|$ needs to be updated accordingly. A naive way to update $|q(\mathcal{D})|$ is to check whether each removed record satisfies q or not. If yes, decrement $|q(\mathcal{D})|$ by one; otherwise, keep $|q(\mathcal{D})|$ unchanged. That is, whenever a record is removed from $\mathcal{D}$, the naive approach needs to check this for all $q \in \mathcal{Q}$, thus the time complexity of removing one record is $O(|\mathcal{Q}| \cdot |q|)$. Since there are $|\mathcal{D}|$ records, the total time complexity can be as high as $O(|\mathcal{Q}| \cdot |\mathcal{D}| \cdot |q|)$.

To reduce the cost, we build a *forward index* on $\mathcal{D}$ to update $|q(\mathcal{D})|$ efficiently. A forward index maps a local record to all the queries that the record satisfies. Such a list is called a forward list. To build the index, we initialize a hash map F and let $F(d)$ denote the forward list for d. For each query $q \in \mathcal{Q}$, we enumerate each record $d \in q(\mathcal{D})$ and add q into $F(d)$. Like the inverted index, the forward index only needs to be built once and can be repeatedly used to update $|q(\mathcal{D})|$ for every iteration. For example, Figure 4(b) illustrates the forward index built on the local database in our running example. Suppose d_3 is removed. Since $F(d_3) = \{q_3, q_5, q_6\}$ contains all the queries that d_3 satisfies, only $\{q_3, q_5, q_6\}$ need to be updated. The time complexity of finding these queries is $O(|F(d)|)$ while the naive approach needs $O(|\mathcal{Q}| \cdot |q|)$ time.

How to select the largest $|q(\mathcal{D})|$ efficiently? QSEL-EST iteratively selects the query with the largest $|q(\mathcal{D})|$ from a query pool, i.e., $q^* = \text{argmax}_{q \in \mathcal{Q}} |q(\mathcal{D})|$. Note that $|q(\mathcal{D})|$ is computed based on the *up-to-date* $\mathcal{D}$ (that needs to remove the covered records after each iteration). A naive approach to solve the problem is to first update $|q(\mathcal{D})|$ in-place and then scan the query pool to find the one with the largest $|q(\mathcal{D})|$. Since there are b iterations, the total time complexity is $O(b \cdot |\mathcal{Q}|)$, which is expensive when the budget b and the pool size $|\mathcal{Q}|$ are large.

We propose an *on-demand updating mechanism* to reduce the cost. The basic idea is to update $|q(\mathcal{D})|$ in-place only when the query has a chance to be selected. We use a hash map U, called *delta-update index*, to maintain the update information of each query. Figure 4(c) illustrates the delta-update index, where $U(q)$ (e.g., = 1) means that $|q(\mathcal{D})|$ should be decremented by one.

Initially, QSEL-EST creates a priority queue P for the query pool, where the priority of each query is the estimated benefit, i.e., $P(q) = |q(\mathcal{D})|$. A priority queue is able to return the query with the largest priority or insert a new query in both $O(\log |\mathcal{Q}|)$ time. Figure 4(c) illustrates the priority queue.

In the 1st iteration, QSEL-EST pops the top query q_1^* from the priority queue and treats it as the first query that needs to be selected. Then, it stores the update information into U rather than update the priority of each query in-place in the priority queue. For example, in Figure 4(c), suppose q_5 is popped. Since q_5 can cover d_3, then d_3 will be removed from $\mathcal{D}$. We get the forward list $F(d_3) = \{q_3, q_5, q_6\}$, and then set $U(q_3) = 1, U(q_5) = 1$, and $U(q_6) = 1$.

In the 2nd iteration, it pops the top query q_2^* from the priority queue. But this time, the query may not be the one with the largest estimated benefit. We consider two cases about the query:

(1) If $U(q_2^*) = 0$, then q^* does not need to be updated, thus q^* mush have the largest estimated benefit. QSEL-EST returns q_2^* as the second query that needs to be selected;

(2) If $U(q_2^*) \neq 0$, we update the priority of q_2^* by inserting q_2^* with the priority of $P(q_2^*) - U(q_2^*)$ into the priority queue, and set $U(q_2^*) = 0$.

If it is Case (2), QSEL-EST will continue to pop the top queries from the priority queue until Case (1) holds.

In the remaining iterations, QSEL-EST will follow the same procedure as the 2nd iteration until the budget is exhausted.

Extend to other estimators. For the other estimators, there are two additional variables involved, $|q(\mathcal{H}_s)|$ and $|q(\mathcal{D}) \cap q(\mathcal{H}_s)|$. For $|q(\mathcal{H}_s)|$, we use the inverted index built on $\mathcal{H}_s$ to compute it; for $|q(\mathcal{D}) \cap q(\mathcal{H}_s)|$, we can rewrite it as $|q(\mathcal{D}_{\mathcal{H}_s})|$, where $\mathcal{D}_{\mathcal{H}_s} = \mathcal{D} \cap \mathcal{H}_s$, and then use the inverted index built on $\mathcal{D}_{\mathcal{H}_s}$ to compute it. Note that there is no need to update $|q(\mathcal{H}_s)|$ because we cannot remove the covered records from a hidden database. In order to select the query with the largest estimated benefit, we extend the on-demand updating mechanism by changing the definition of the priority from $|q(\mathcal{D})|$ to a corresponding estimator (e.g., $|q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|}$).

Time complexity analysis. A naive implementation of QSEL-EST requires $O\big(|Q||\mathcal{D}||q| + |Q||\mathcal{H}_s||q| + b|Q|\big)$ time. If $|Q| = 10^5$, $|\mathcal{D}| = 10^5$, $|\mathcal{H}_s| = 10^2$, $b = 10^3$, and $|q| = 2$, then the total time will reach up to $O(2 \cdot 10^{10})$. In Appendix B, Algorithm 4 depicts the pseudo-code of our efficient implementation of QSEL-EST. We also analyze the time complexity and show that the efficient implementation can be orders of magnitude faster than the naive one.

7 PRACTICAL ISSUES

In this section, we present two issues that one may encounter when applying QSEL-EST in practice. We first show how to deal with the problem of inadequate sample size in Section 7.1 and then discuss the extension of QSEL-EST to fuzzy matching in Section 7.2.

7.1 Inadequate Sample Size

The performance of QSEL-EST depends on the size of a hidden database sample. If the sample size is not large enough, some queries in the pool may not appear in the sample, i.e., $|q(\mathcal{H}_s)| = 0$, thus the sample is not useful for these queries. To address this issue, we model the local database as another random sample of the hidden database, where the sampling ratio is denoted by $\alpha = \frac{\theta|\mathcal{D}|}{|\mathcal{H}_s|}$, and use this idea to predict the query type (solid or overflowing) and estimate the benefit of these queries.

- *Query Type.* For the queries with $|q(\mathcal{H}_s)| = 0$, since $\frac{|q(\mathcal{H}_s)|}{\theta} = 0 \leq k$, the current QSEL-EST will always predict them as solid queries. With the idea of treating $\mathcal{D}$ as a random sample, QSEL-EST will continue to check whether $\frac{|q(\mathcal{D})|}{\alpha} > k$ holds. If yes, QSEL-EST will predict q as an overflowing query instead.
- *Query Benefit.* For the queries with $|q(\mathcal{H}_s)| = 0$, as shown in Table 2, the estimator $|q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|}$ will not work since $|q(\mathcal{H}_s)|$ appears in the denominator. By using the same idea as above, QSEL-EST replaces $\mathcal{H}_s$ and θ with $\mathcal{D}$ and α, respectively, and obtains the estimator, $k\alpha$, to deal with the special case.

7.2 Fuzzy Matching

This paper develops the four estimators based on the exact matching assumption. That is, $\mathsf{match}(d, h) = \text{True}$ if and only if $\mathsf{document}(d) = \mathsf{document}(h)$ (see Assumption 1). Next, we eliminate the assumption and explore its impact on our estimators.

Let $|A \widetilde{\cap} B|$ denote the number of record pairs (including both exact and fuzzy matching pairs) that refer to the same real-world entity between A and B. For the two unbiased estimators, we prove in Lemma 7.1 that by replacing $|q(\mathcal{D}) \cap q(\mathcal{H}_s)|$ with $|q(\mathcal{D}) \widetilde{\cap} q(\mathcal{H}_s)|$, they are still unbiased estimators.

LEMMA 7.1. *Lemmas 4.1 and 5.1 hold without Assumption 1.*

For the two biased estimators, their biases could get larger when Assumption 1 does not hold. For example, consider a solid query q = "Thai Rest" and $|q(\mathcal{D})| = 5$. Suppose the five restaurants in $q(\mathcal{D})$ can also be found in $\mathcal{H}$. If Assumption 1 holds, the biased estimator can obtain the estimated benefit (i.e., 5) with the bias of 0. However, imagine that the hidden database does not abbreviate "Restaurant" as "Rest". Issuing q = "Thai Rest" to the hidden database will return none of the five records. Thus, the bias of the estimator becomes 5 in this fuzzy-matching situation.

The increase of the bias will decrease the effectiveness of SMARTCRAWL. However, SMARTCRAWL tends to perform even more effectively than NAIVECRAWL in the fuzzy-matching situation. This is because that the queries selected by NAIVECRAWL typically contain more keywords. The more keywords a query contains, the more likely it is to contain a fuzzy-matching word like "Rest vs. Restaurant". We further investigate this finding in the experiments (Section 8.2.5).

Another change that may need to be made to SMARTCRAWL is about the computation of $q^*(\mathcal{D}) \cap q^*(\mathcal{H})_k$ at each iteration. We can still run SMARTCRAWL by only searching for exactly matching record pairs between $q^*(\mathcal{D})$ and $q^*(\mathcal{H})_k$, and remove them from $\mathcal{D}$, but the downside is that some already-covered records will stay in $\mathcal{D}$, affecting the accuracy of benefit estimation. Instead, we perform a similarity join between $q^*(\mathcal{D})$ and $q^*(\mathcal{H})_k$, where the similarity between two records is quantified by a similarity function. For example, consider $\mathsf{Jaccard}(d, h) = \frac{|d \cap h|}{|d \cup h|}$ and a threshold of 0.9. Then, at each iteration, SMARTCRAWL removes d from $\mathcal{D}$ if there exists $h \in q^*(\mathcal{H})_k$ such that $\mathsf{Jaccard}(d, h) \geq 0.9$.

8 EXPERIMENTS

We conduct extensive experiments to evaluate the performance of SMARTCRAWL over simulated and real hidden databases. The experiments aim to answer four questions. (1) Which estimator is more effective, biased or unbiased? (2) How does SMARTCRAWL (with estimated benefits) compare with IDEALCRAWL (with true benefits)? (3) Can SMARTCRAWL achieve better performance than NAIVECRAWL and FULLCRAWL in a large variety of situations? (4) Does SMARTCRAWL outperform NAIVECRAWL and FULLCRAWL over a real hidden database?

8.1 Experimental Settings

8.1.1 Simulated Hidden Database. We designed a simulated experiment based on DBLP dataset[3].

Local and Hidden Databases. The dataset has 5 million records. We found all the authors who have published papers in major database and data mining conferences ('SIGMOD', 'VLDB', 'ICDE', 'CIKM', 'CIDR', 'KDD', 'WWW', 'AAAI', 'NIPS', 'IJCAI') on the dataset, and assumed that a local database $\mathcal{D}$ was randomly drawn from the union of the publications of the authors. A hidden database consists of two parts: $\mathcal{H} - \mathcal{D}$ and $\mathcal{H} \cap \mathcal{D}$, where $\mathcal{H} - \mathcal{D}$

[3] http://dblp.dagstuhl.de/xml/release/

Table 3: A summary of parameters

Parameters	Domain	Default
Hidden Database ($\mathcal{H}$)	100,000	100,000
Local Database ($\mathcal{D}$)	1, 10, 10^2, 10^3, 10^4	10,000
Result# Limit (k)	1, 50, 100, 500	100
$\Delta D = \mathcal{D} - \mathcal{H}$	[1000, 3000]	0
Budget (b)	1% - 20% of $\lvert\mathcal{D}\rvert$	20% of $\lvert\mathcal{D}\rvert$
Sample Ratio (θ)	0.1% - 1%	0.5%
error%	0% - 50%	0%

was randomly drawn from the entire DBLP dataset and $\mathcal{H} \cap \mathcal{D}$ was randomly drawn from $\mathcal{D}$. To simulate the situation that $\Delta D = \mathcal{D} - \mathcal{H}$ is not empty, we randomly drew $|\Delta\mathcal{D}|$ records from the entire dataset and added them to $\mathcal{D}$ but not $\mathcal{H}$.

Keyword Search Interface. We implemented a search engine over a hidden database. The search engine built an inverted index on title, venue, and authors attributes (stop words were removed). Given a query over the three attributes, it ranked the publications that contain all the keywords of the query by year, and returned the top-k records.

Evaluation Metrics. We used *coverage* to measure the performance of each approach, which is defined as the total number of local records that are covered by the hidden records crawled. The *relative coverage* is the percentage of the local records in $\mathcal{D} - \Delta\mathcal{D}$ that are covered by the hidden records crawled.

Parameters. Table 3 summarized all the parameters as well as their default values used in our paper. In addition to the parameters that have already been explained above, we added a new parameter *error%* to evaluate the performance of different approaches in the fuzzy matching situation. Suppose *error%* = 10%. We will randomly select 10% records from $\mathcal{D}$. For each record, we removed a word, added a new word, and replaced an existing word with a new word with the probability of 1/3.

8.1.2 Real Hidden Database. We evaluated SMARTCRAWL over the Yelp's hidden database.

Local Database. We constructed a local database based on the Yelp dataset[4]. The dataset contains 36,500 records about Arizona, where each record describes a local business. We randomly chose 3000 records as a local database. As the dataset was released several years ago, some local businesses' information are updated by Yelp since then. This experiment evaluated the performance of our approach in the fuzzy-matching situation.

Hidden Database. We treated all the Arizona's local businesses in Yelp as our hidden database. Yelp provided a keyword-search style interface to allow the public user to query its hidden database. A query contains keyword and location information. We used 'AZ' as location information, thus only needed to generate keyword queries. For each API call, Yelp returns the top-50 related results. It is worth noting that the Yelp's search API does not force queries to be conjunctive. Thus, this experiment demonstrated the performance of our approach using a keyword-search interface without the conjunctive-keyword-search assumption.

Hidden Database Sample. We adopted an existing technique [35] to construct a hidden database sample along with the sampling ratio. The technique needs an initialized query pool. We extracted all the single keywords from the 36500 records as the query pool. A 0.2% sample with size 500 was constructed by issuing 6483 queries.

Evaluation Metric. We manually labeled the data by searching each local record over Yelp and identifying its matching hidden record. Since entity resolution is an independent component of our framework, we assumed that once a hidden record is crawled, the entity resolution component can perfectly find its matching local record (if any). We compared the *recall* of SMARTCRAWL, FULLCRAWL and NAIVECRAWL, where recall is defined as the percentage of the matching record pairs between $\mathcal{D}$ and $\mathcal{H}_{\text{crawl}}$ out of all matching record pairs between $\mathcal{D}$ and $\mathcal{H}$.

Implementation of Different Approaches. We discussed the implementation details of different approaches in Appendix C.

8.2 Simulated Hidden Databases

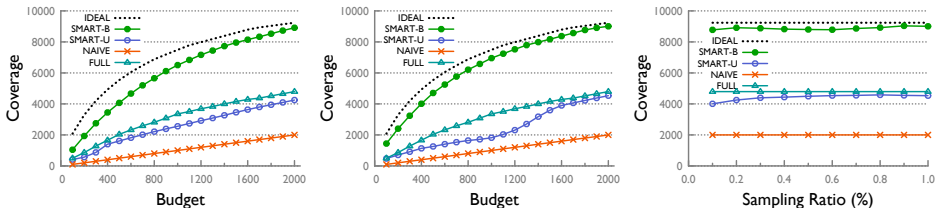
We evaluated the performance of SMARTCRAWL and compared it with baselines and the ideal solution in a large variety of situations.

8.2.1 Sampling Ratio. We first examine the impact of sampling ratios on the performance of SMARTCRAWL. Figure 5 shows the result. In Figure 5(a), we set the sampling ratio to 0.2%, leading to a sample size = $100,000 \times 0.2\% = 200$. We can see that with such a small sample, SMARTCRAWL-B still had similar performance with IDEALCRAWL and covered about 2× more records than FULLCRAWL and about 4× more records than NAIVECRAWL. Furthermore, we can see that SMARTCRAWL-U did not have a good performance on such a small sample, even worse than FULLCRAWL. In fact, we found that SMARTCRAWL-U tended to select queries randomly because many queries had the same values. This phenomenon was further manifested in Figure 5(b), which increased the sampling ratio to 1%. In Figure 5, we set the budget to 2000, varied the sampling ratio from 0.1% (sample size=100) to 1% (sample size=1000), and compared the number of covered records of each approach. We can see that as the sampling was increased to 1%, SMARTCRAWL-B is very close to IDEALCRAWL, where SMARTCRAWL-B covered 92% of the local database while IDEALCRAWL covered 89%. In summary, this experimental result shows that (1) biased estimators are much more effective than unbiased estimators; (2) biased estimators even work with a very small sampling ratio 0.1%; (3) SMARTCRAWL-B outperformed FULLCRAWL and SMARTCRAWL by a factor of 2 and 4, respectively.

8.2.2 Local Database Size. The reason that FULLCRAWL performed so well in the last experiment is that the local database $\mathcal{D}$ is relatively large compared to the hidden database ($\frac{|\mathcal{D}|}{|\mathcal{H}|} = 10\%$). In this experiment, we varied the local database size and examined how this affected the performance of each approach.

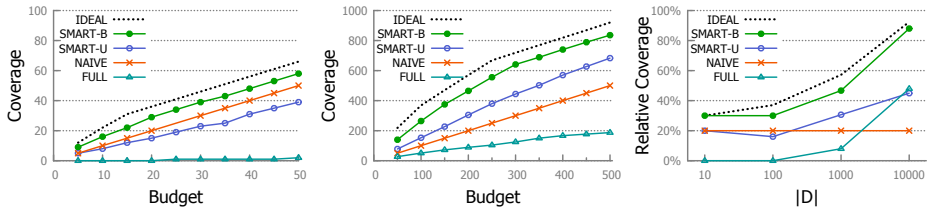
Figure 10(a) shows the result when $|\mathcal{D}|$ has only 100 records. We can see that FULLCRAWL only covered 2 records after issuing 50 queries while the other approaches all covered 39 more records. Another interesting observation is that even for such a small local database, SMARTCRAWL-B can still outperform NAIVECRAWL due to the accurate benefit estimation as well as the power of query sharing. Figure 10(b) shows the result for $|D| = 1000$. We can see that FULLCRAWL performed marginally better than before but still worse than the alternative approaches. We varied the local database size $|\mathcal{D}|$ from 10 to 10000, and set the budget to 20% of $|\mathcal{D}|$. The comparison of the relative coverage of different approaches is shown in Figure 10(c). We can see that as $|\mathcal{D}|$ increased, all the approaches except NAIVECRAWL showed improved performance. This is because that the large $|\mathcal{D}|$ is, the more local records an issued query can cover. Since NAIVECRAWL failed to exploit the query sharing idea, its performance remained the same.

[4] https://www.yelp.com/dataset_challenge



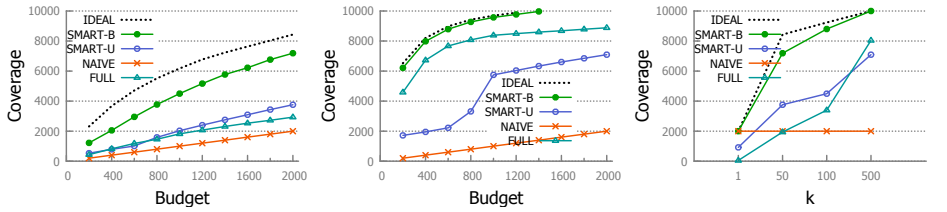
(a) Sample Ratio = 0.2% (b) Sample Ratio = 1% (c) Budget = 2000

Figure 5: Comparisons of different approaches with various sampling ratios θ.



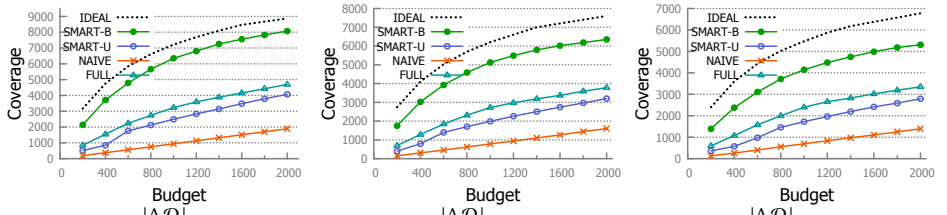
(a) |D| = 100 (b) |D| = 1000 (c) Budget = 20% of $|\mathcal{D}|$

Figure 6: Comparisons of different approaches with various local database size $|\mathcal{D}|$.



(a) k = 50 (b) k = 500 (c) Budget = 2000

Figure 7: Comparisons of different approaches with various result# limits k.



(a) $\frac{|\Delta\mathcal{D}|}{|\mathcal{D}|} = 5\%$ (b) $\frac{|\Delta\mathcal{D}|}{|\mathcal{D}|} = 20\%$ (c) $\frac{|\Delta\mathcal{D}|}{|\mathcal{D}|} = 30\%$

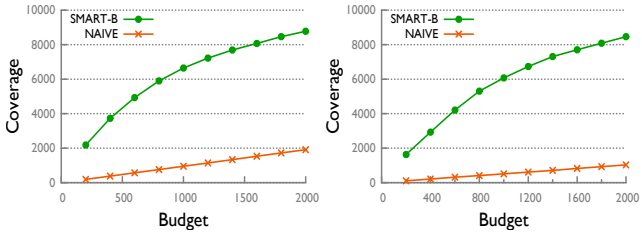
Figure 8: Comparisons of different approaches with various $|\Delta\mathcal{D}|$ size.

8.2.3 Result Number Limit. Obviously, the larger k is, the more effective the query sharing. Next, we investigate the impact of k on different approaches.

Figure 7(a) shows the result when $k = 50$. In this case, SMARTCRAWL-B can cover about 3.5 times more records than naive after issuing 2000 queries. In other words, for SMARTCRAWL-B, each query covered 3.5 local records while NAIVECRAWL only covered one query per query. When we increased k to 500 (Figure 7(b)), we found that SMARTCRAWL-B covered 99% of the local database ($|\mathcal{D}| = 10000$) with only 1400 queries while NAIVECRAWL can only cover 14% of the local database. Figure 7(c) compared different approaches by varying k. We can see that IDEALCRAWL, SMARTCRAWL-B, and NAIVECRAWL achieved the same performance when $k = 1$. As k increased, NAIVECRAWL kept unchanged because it covered one local record at a time regardless of k while all the other approaches all got the performance improved.

8.2.4 Increase of Bias. SMARTCRAWL-B is a biased estimator, where the bias depends on the size of $|\Delta\mathcal{D}|$. A larger $|\Delta\mathcal{D}|$ will increase the bias. In this section, we explore the impact of $|\Delta\mathcal{D}|$ on SMARTCRAWL-B. Figure 8(a), (b), (c) show the results when $|\Delta\mathcal{D}|$ is 5%, 20%, and 30% of $|\Delta\mathcal{D}|$. By comparing the relative performance of SMARTCRAWL-B w.r.t. IDEALCRAWL in these three figures, we can see that as $|\Delta\mathcal{D}|$ increased, SMARTCRAWL-B got more and more far away from IDEALCRAWL due to the increase of biases. Nevertheless, even with $\frac{|\Delta\mathcal{D}|}{|\mathcal{D}|}$, which means that 30% of local records cannot be found in the hidden database, SMARTCRAWL-B still outperformed all the other approaches.

8.2.5 Fuzzy Matching. We compared SMARTCRAWL-B with NAIVECRAWL in the fuzzy matching situation. Figure 10(a),(b) show the results for the cases when adding 5% and 50% data errors to local databases. As discussed in Section 7.2, SMARTCRAWL-B is more robust to data errors. For example, in the case of *error%* = 5%, SMARTCRAWL-B and NAIVECRAWL can use 2000 queries to cover 8775 and 1914 local records, respectively. When *error%* was increased to 50%, SMARTCRAWL-B can still cover 8463 local records



(a) *error%* = 5% (b) *error%* = 50%

Figure 9: Comparisons of SMARTCRAWL-B and NAIVECRAWL in the fuzzy matching situation.

(only missing 3.5% compared to the previous case) while NAIVECRAWL can only cover 1031 local records (46% less than the previous case). This result validated the robustness of SMARTCRAWL-B when dealing with the fuzzy matching situation. We have also observed this interesting phenomenon in the next section.

8.3 Yelp's Hidden Database

We theoretically prove the effectiveness of SMARTCRAWL based on a number of assumptions (e.g., conjunctive keyword search, exact matching) in the paper. However, a real-life hidden database may violate these assumptions. In this section, we evaluated the performance of SMARTCRAWL over the Yelp's hidden database, where SMARTCRAWL used the biased estimator for benefit estimation. Figure 10 shows the recall of SMARTCRAWL, NAIVECRAWL, and FULLCRAWL by varying the budget from 300 to 3000.

We have three observations from the figures. Firstly, SMARTCRAWL can achieve the recall above 80% by issuing 1800 queries while NAIVECRAWL only achieved a recall of 60%. This shows that the idea of query sharing is still very powerful for a real-life hidden database. Secondly, FULLCRAWL performed poorly on this dataset because the local database $|\mathcal{D}|$ is small. This further validated the importance of local-database-aware crawling. Thirdly, NAIVECRAWL got a recall smaller than SMARTCRAWL even after issuing all the queries (one for each local record). This is because that NAIVECRAWL is not as robust as SMARTCRAWL to tolerate data inconsistency issues. Imagine a local business has an inconsistent name with its matching one. Since NAIVECRAWL issues the full business name to Yelp, it is more likely to be affected by data errors.

9 RELATED WORK

Deep Web. There are three lines of work about deep web related to our problem: deep web crawling [3, 17, 20, 22, 24, 26–28], deep web integration [6, 16, 23, 32], and deep web sampling [4, 8–11, 19, 29, 35, 36].

Deep web crawling studies how to crawl a hidden database through the database's restrictive query interface. The main challenge is how to automatically generate a (minimum) set of queries for a query interface such that the retrieved data can have a good coverage of the underlying database. Along this line of research, various types of query interfaces were investigated, such as keyword search interface [3, 17, 24] and form-like search interface [20, 22, 26–28]. Unlike these work, our goal is to have a good coverage of a local database rather than the underlying hidden database.

Deep web integration [6, 16, 23, 32] studies how to integrate a number of deep web sources and provide a unified query interface to search the information over them. Differently, our work aims to match a hidden database with a collection of records rather than a single one. As shown in our experiments, the NAIVECRAWL solution that issues queries to cover one record at a time is highly ineffective.

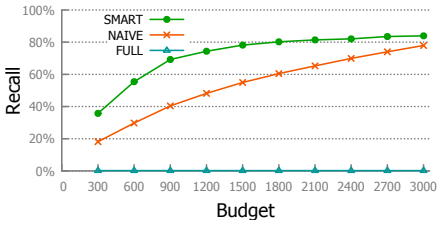


Figure 10: Comparisons of SMARTCRAWL, NAIVECRAWL, and FULLCRAWL over the Yelp's hidden database.

Deep web sampling studies how to create a random sample of a hidden database using keyword-search interfaces [4, 35, 36] or form-like interfaces [8, 8, 29]. In this paper, we treat deep web sampling as an orthogonal problem and assume that a random sample is given. It would be a very interesting line of future work to investigate how sampling and SMARTCRAWL can enhance each other.

Entity Resolution. Machine-based ER techniques have been studied for decades [18]. Recently, there is an increasing interest of leveraging crowdsourcing for ER [14, 30, 31]. However, none of existing work has studied how to match a local database with a hidden database under a restrictive query interface. There are many blocking techniques in ER, which study how to partition data into small blocks such that matching records can fall into the same block [7]. ER-driven deep web crawling is similar in spirit to this problem by thinking of a top-k query result as a block. However, existing blocking techniques are not applicable because they do not consider the situation when a database can only be accessed via a restrictive query interface.

Entity Augmentation. There are some works on entity augmentation with web table [5, 12, 13, 25, 33, 34], which study how to match with a large number (millions) of small web tables. In contrast, our work aims to match with one hidden database with a large number (millions) of records.

10 CONCLUSION

This paper studied a novel problem, called DeepER, which has many applications in data enrichment and data cleaning. We proposed the SMARTCRAWL framework based on the ideas of query sharing and local-database-aware crawling. A key challenge is how to estimate the query benefit in order to select the best query at each iteration. We classified queries into solid queries and overflowing queries, and developed unbiased as well as biased estimators for them. We devised efficient algorithms and indexing techniques to implement the estimators. Our detailed experimental evaluation has shown that (1) the biased estimators are superior to the unbiased estimators; (2) SMARTCRAWL can significantly outperform the baselines over both simulated and real hidden databases; (3) SMARTCRAWL is more robust than NAIVECRAWL when facing the fuzzy-matching situation.

We believe that DeepER is a promising new research direction. There are many interesting problems that can be studied in the future. First, the proposed estimators require a hidden database sample to be created upfront. It is interesting to study how to create the sample progressively such that the upfront cost can be amortized over time. Second, we would like to extend SMARTCRAWL by supporting not only keyword-search interfaces but also other popular query interfaces such as form-based search and graph-browsing. Third, the decoupling of deep web crawling and entity resolution makes our framework more extensible. On the other hand, this may not give us the best performance. This is an interesting trade-off worth further exploring.

REFERENCES

[1] Yelp API. https://www.yelp.com/developers/faq. Accessed: 2017-07-12.

[2] Yelp API. https://developers.google.com/maps/documentation/geocoding/usage-limits. Accessed: 2017-07-12.

[3] E. Agichtein, P. G. Ipeirotis, and L. Gravano. Modeling query-based access to text databases. In *International Workshop on Web and Databases, San Diego, California, June 12-13, 2003*, pages 87–92, 2003.

[4] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's index. *J. ACM*, 55(5):24:1–24:74, 2008.

[5] M. J. Cafarella, A. Y. Halevy, and N. Khoussainova. Data integration for the relational web. *PVLDB*, 2(1):1090–1101, 2009.

[6] K. C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 44–55, 2005.

[7] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. Knowl. Data Eng.*, 24(9):1537–1555, 2012.

[8] A. Dasgupta, G. Das, and H. Mannila. A random walk approach to sampling hidden databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 629–640, 2007.

[9] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das. Unbiased estimation of size and other aggregates over hidden web databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 855–866, 2010.

[10] A. Dasgupta, N. Zhang, and G. Das. Leveraging COUNT information in sampling hidden databases. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 329–340, 2009.

[11] A. Dasgupta, N. Zhang, and G. Das. Turbo-charging hidden database samplers with overflowing queries and skew reduction. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 51–62, 2010.

[12] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. Top-k entity augmentation using consistent set covering. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management, SSDBM '15, La Jolla, CA, USA, June 29 - July 1, 2015*, pages 8:1–8:12, 2015.

[13] J. Fan, M. Lu, B. C. Ooi, W. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 976–987, 2014.

[14] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 601–612, 2014.

[15] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 1–12, 2000.

[16] H. He, W. Meng, C. T. Yu, and Z. Wu. Automatic integration of web search interfaces with wise-integrator. *VLDB J.*, 13(3):256–273, 2004.

[17] Y. He, D. Xin, V. Ganti, S. Rajaraman, and N. Shah. Crawling deep web entity pages. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, pages 355–364, 2013.

[18] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 127–138, 1995.

[19] P. G. Ipeirotis and L. Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 394–405, 2002.

[20] X. Jin, N. Zhang, and G. Das. Attribute domain discovery for hidden web databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 553–564, 2011.

[21] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12):2917–2926, 2012.

[22] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Y. Halevy. Google's deep web crawl. *PVLDB*, 1(2):1241–1252, 2008.

[23] W. Meng, C. T. Yu, and K. Liu. Building efficient and effective metasearch engines. *ACM Comput. Surv.*, 34(1):48–89, 2002.

[24] A. Ntoulas, P. Zerfos, and J. Cho. Downloading textual hidden web content through keyword queries. In *ACM/IEEE Joint Conference on Digital Libraries, JCDL 2005, Denver, CO, USA, June 7-11, 2005, Proceedings*, pages 100–109, 2005.

[25] R. Pimplikar and S. Sarawagi. Answering table queries on the web using column keywords. *PVLDB*, 5(10):908–919, 2012.

[26] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 129–138, 2001.

[27] C. Sheng, N. Zhang, Y. Tao, and X. Jin. Optimal algorithms for crawling a hidden database in the web. *PVLDB*, 5(11):1112–1123, 2012.

[28] S. Thirumuruganathan, N. Zhang, and G. Das. Breaking the top-k barrier of hidden web databases? In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1045–1056, 2013.

[29] F. Wang and G. Agrawal. Effective and efficient sampling methods for deep web aggregation queries. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 425–436, 2011.

[30] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[31] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.

[32] W. Wu, C. T. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 95–106, 2004.

[33] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 97–108, 2012.

[34] M. Zhang and K. Chakrabarti. Infogather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 145–156, 2013.

[35] M. Zhang, N. Zhang, and G. Das. Mining a search engine's corpus: efficient yet unbiased sampling and aggregate estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 793–804, 2011.

[36] M. Zhang, N. Zhang, and G. Das. Mining a search engine's corpus without a query pool. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 29–38, 2013.

APPENDIX

A PROOFS

Proof of Lemma 4.1

Let $A = q(\mathcal{D}) \cap q(\mathcal{H}) \subseteq \mathcal{H}$. The indicator function of a subset A of $\mathcal{H}$ is defined as

$$\mathbb{1}_A(h) = \begin{cases} 1, & \text{if } h \in A \\ 0, & \text{otherwise} \end{cases}$$

The expected value of the estimated benefit is:

$$\begin{aligned} \mathbf{E}\Big[\frac{q(\mathcal{D}) \cap q(\mathcal{H}_s)}{\theta}\Big] &= \mathbf{E}\Big[\frac{\sum_{h \in \mathcal{H}_s} \mathbb{1}_A(h)}{\theta}\Big] \\ &= |\mathcal{H}| \cdot \mathbf{E}\Big[\frac{1}{|\mathcal{H}_s|} \sum_{h \in \mathcal{H}_s} \mathbb{1}_A(h)\Big] \end{aligned}$$

Since sample mean is an unbiased estimator of population mean, then we have

$$\mathbf{E}\Big[\frac{1}{|\mathcal{H}_s|} \sum_{h \in \mathcal{H}_s} \mathbb{1}_A(h)\Big] = \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \mathbb{1}_A(h)$$

By combing the two equations, we finally get

$$\begin{aligned} \mathbf{E}\Big[\frac{q(\mathcal{D}) \cap q(\mathcal{H}_s)}{\theta}\Big] = |\mathcal{H}| \cdot \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \mathbb{1}_A(h) \quad &= \sum_{h \in \mathcal{H}} \mathbb{1}_A(h) \\ = |A| = |q(\mathcal{D}) \cap q(\mathcal{H})| \end{aligned}$$

Since q is a solid query, we have the true benefit of the query is:

$$\mathsf{benefit}(q) = |q(\mathcal{D}) \cap q(\mathcal{H})|.$$

We can see that the estimator's expected value is equal to the true benefit, thus the estimator is unbiased.

Proof of Lemma 4.4

In order to prove that QSEL-IDEAL and QSEL-EST are equivalent, we only need to prove that Algorithm 1 (Line 3) and Algorithm 2 (Lines 2-6). Since Q only contains solid queries, there is no need to predict whether a query is solid or overflowing, thus we only need to prove that Algorithm 1 (Line 3) and Algorithm 2 (Line 3) set the same value to $\mathsf{benefit}(q)$ when q is solid.

For Algorithm 1 (Line 3), it sets

$$\mathsf{benefit}(q) = |q(\mathcal{D})_{\text{cover}}|$$

For Algorithm 2 (Line 3), it sets

$$\mathsf{benefit}(q) = |q(\mathcal{D})| = |q(\mathcal{D})_{\text{cover}}| + |q(\Delta\mathcal{D})|.$$

Since $\mathcal{D} \subseteq \mathcal{H}$, then we have $|q(\Delta\mathcal{D})| = 0$. Thus, the above two equations are equal. Hence, the lemma is proved.

Proof of Lemma 4.5

Part I. We prove by induction that the first $\big(b - |\Delta\mathcal{D}|\big)$ queries selected by QSEL-IDEAL must be selected by QSEL-BOUND, i.e.,

$$\{q_i \mid 1 \le i \le b - |\Delta\mathcal{D}|\} \quad \subseteq \quad Q'_{sel}.$$

Basis: Obviously, the statement holds for $b \le |\Delta\mathcal{D}|$.

Inductive Step: Assuming that the statement holds for $b = k$, we next prove that it holds for $b = k + 1$.

Consider the first selected query q'_1 in Q'_{sel}. There are two situations about q'_1.

(1) If $\mathsf{benefit}(q'_1) = |q'_1(\mathcal{D})|$, then we have $|q'_1(\mathcal{D})| = |q'_1(\mathcal{D})_{\text{cover}}|$. Since q'_1 is the first query selected from the query pool by QSEL-BOUND, then we have

$$q'_1 = \underset{q \in Q}{\text{argmax}}\, |q(\mathcal{D})|.$$

Since $|q'_1(\mathcal{D})| = |q'_1(\mathcal{D})_{\text{cover}}|$, and $|q(\mathcal{D})| \ge |q(\mathcal{D})_{\text{cover}}|$ for all $q \in Q$, we deduce that

$$q'_1 = \underset{q \in Q}{\text{argmax}}\, |q(\mathcal{D})_{\text{cover}}| = \underset{q \in Q}{\text{argmax}}\, \mathsf{benefit}(q).$$

Since $q_1 = \text{argmax}_{q \in Q}\, \mathsf{benefit}(q)$, then we have $q'_1 = q_1$ in this situation. Since the budget is now decreased to k, based on the induction hypothesis, we can prove that the lemma holds.

(2) If $\mathsf{benefit}(q'_1) \ne |q'_1(\mathcal{D})|$, since $b \ge |\Delta\mathcal{D}|$ and each $q' \in Q'_{\text{sel}}$ can cover at most one uncovered local record in $\Delta\mathcal{D}$, there must exist $q' \in Q'_{\text{sel}}$ that does not cover any uncovered local record in $\Delta\mathcal{D}$. Let q'_i denote the first of such queries. We next prove that $q'_i = q_1$.

Let $\mathcal{D}_i$ denote the local database at the i-th iteration of QSEL-BOUND. For any query selected before q'_i, they only remove the records in $\Delta\mathcal{D}$ and keep $\mathcal{D} - \Delta\mathcal{D}$ unchanged, thus we have that

$$\mathcal{D}_i - \Delta\mathcal{D} = \mathcal{D} - \Delta\mathcal{D}. \tag{15}$$

Based on Equation 15, we can deduce that ,

$$|q(\mathcal{D}_i)_{\text{cover}}| = |q(\mathcal{D})_{\text{cover}}| \quad \text{for any } q \in Q. \tag{16}$$

Since q'_i has the largest estimated benefit, we have

$$q'_i = \underset{q \in Q}{\text{argmax}}\, |q(\mathcal{D}_i)|. \tag{17}$$

Because q'_i does not cover any uncovered record in $\Delta\mathcal{D}$, we can deduce that

$$|q'_i(\mathcal{D}_i)| = |q'_i(\mathcal{D}_i)_{\text{cover}}|. \tag{18}$$

For any query $q \in Q$, we have

$$|q(\mathcal{D}_i)| \ge |q(\mathcal{D}_i)_{\text{cover}}|. \tag{19}$$

By plugging Equations 18 and 19 into Equation 17, we obtain

$$q'_i = \underset{q \in Q}{\text{argmax}}\, |q(\mathcal{D}_i)_{\text{cover}}|. \tag{20}$$

By plugging Equation 16 into Equation 20, we obtain

$$q'_i = \underset{q \in Q}{\text{argmax}}\, |q(\mathcal{D})_{\text{cover}}| = \underset{q \in Q}{\text{argmax}}\, \mathsf{benefit}(q).$$

Since $q_1 = \text{argmax}_{q \in Q}\, \mathsf{benefit}(q)$, then we have $q'_i = q_1$ in this situation. As the budget is now decreased to k, based on the induction hypothesis, we can prove that the lemma holds.

Since both the basis and the inductive step have been performed, by mathematical induction, the statement holds for b.

Part II. We prove by contradiction that the first $\big(b - |\Delta\mathcal{D}|\big)$ queries selected by QSEL-IDEAL can cover at least $(1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}$ local records. Assume this it not correct. Let N_1 denote the number of local records covered by the first $\big(b - |\Delta\mathcal{D}|\big)$ queries, and N_2 denote the number of local records covered by the remaining $|\Delta\mathcal{D}|$ queries. Then, we have

$$N_1 < (1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}. \tag{21}$$

$$N_1 + N_2 = N_{\text{ideal}} \tag{22}$$

We next prove that these two equations cannot hold at the same time. For QSEL-IDEAL, the queries are selected in the decreasing order of true benefits, thus we have

$$\frac{N_1}{b - \Delta\mathcal{D}} \ge \frac{N_2}{\Delta\mathcal{D}}. \tag{23}$$

By plugging Equation 22 into Equation 23, we obtain

$$\frac{N_1}{b - \Delta\mathcal{D}} \ge \frac{N_{\text{ideal}} - N_1}{\Delta\mathcal{D}}$$

Algebraically:

$$N_1 \ge (1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}},$$

which contradicts Equation 21. Thus, the assumption is false, and the first $\big(b - |\Delta\mathcal{D}|\big)$ queries selected by QSEL-IDEAL can cover at least $(1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}$ local records. Based on the proof in Part I, since QSEL-BOUND will also select these queries, the lemma is proved.

Proof of Lemma 5.1

Since $|q(\mathcal{H}_s)|$ is given, it can be treated as a constant value. Thus, we have

$$E\Big[|q(\mathcal{D}) \cap q(\mathcal{H}_s)| \cdot \frac{k}{|q(\mathcal{H}_s)|}\Big] = \frac{k}{|q(\mathcal{H}_s)|} \cdot E\Big[|q(\mathcal{D}) \cap q(\mathcal{H}_s)|\Big] \tag{24}$$

Based on Lemma 4.1, we obtain

$$E\Big[|q(\mathcal{D}) \cap q(\mathcal{H}_s)|\Big] = \theta|q(\mathcal{D}) \cap q(\mathcal{H})| \tag{25}$$

By plugging Equation 26 into Equation 24, we have that the expected value of our estimator is:

$$\frac{k\theta}{|q(\mathcal{H}_s)|}|q(\mathcal{D}) \cap q(\mathcal{H})| = \frac{k}{|q(\mathcal{H})|}|q(\mathcal{D}) \cap q(\mathcal{H})|, \tag{26}$$

which is equal to the true benefit when $q(\mathcal{D}) \cap q(\mathcal{H})$ is a random sample of $q(\mathcal{H})$ (See Equation 8).

Proof of Lemma 5.3

The expected value of the estimator is

$$E\Big[|q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|}\Big] = k\theta \cdot |q(\mathcal{D})| \cdot \frac{1}{E[|q(\mathcal{H}_s)|]} \tag{27}$$

$$= k\theta \cdot |q(\mathcal{D})| \cdot \frac{1}{|q(\mathcal{H})|\theta} \tag{28}$$

$$= \frac{k \cdot |q(\mathcal{D})|}{|q(\mathcal{H})|} \tag{29}$$

Therefore, the bias of the estimator is:

$$bias = \frac{k \cdot |q(\mathcal{D})|}{|q(\mathcal{H})|} - |q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|}$$

$$= |q(\Delta\mathcal{D})| \cdot \frac{k}{|q(\mathcal{H})|} \tag{30}$$

Proof of Lemma 7.1

We first prove that Lemma 4.1 holds without Assumption 1. Construct a new hidden database:

$$\mathcal{H}' = \{f(h) \mid h \in \mathcal{H}\},$$

where $f(h)$ returns h if there is no local record $d \in \mathcal{D}$ such that $\mathsf{match}(d, h) = \text{True}$; otherwise $f(h)$ returns d, where d is the local record that matches h. Similarly, we construct a new hidden database sample:

$$\mathcal{H}'_s = \{f(h) \mid h \in \mathcal{H}_s\}.$$

Based on Lemma 4.1, we have

$$\mathbf{E}\Big[\frac{q(\mathcal{D}) \cap q(\mathcal{H}'_s)}{\theta}\Big] = |q(\mathcal{D}) \cap q(\mathcal{H}')| \tag{31}$$

Since $\mathcal{D}$ and $\mathcal{H}$ have a one-to-one matching relationship, we have

$$|q(\mathcal{D}) \cap q(\mathcal{H}')| = |q(\mathcal{D}) \widetilde{\cap} q(\mathcal{H})| \quad |q(\mathcal{D}) \cap q(\mathcal{H}'_s)| = |q(\mathcal{D}) \widetilde{\cap} q(\mathcal{H}_s)| \tag{32}$$

By plugging Equation 32 into Equation 31, we have

$$\mathbf{E}\Big[\frac{q(\mathcal{D}) \widetilde{\cap} q(\mathcal{H}'_s)}{\theta}\Big] = |q(\mathcal{D}) \widetilde{\cap} q(\mathcal{H}')|$$

Therefore, Lemma 4.1 holds without Assumption 1.

We can use a similar idea to prove that Lemma 5.1 holds without Assumption 1

Algorithm 4: QSEL-EST Algorithm (Biased Estimators)

Input: Q, $\mathcal{D}$, $\mathcal{H}$, $\mathcal{H}_s$, θ, b, k
Result: Iteratively select the query with the largest *estimated* benefit.

1 Build inverted indices I_1 and I_2 based on $\mathcal{D}$ and $\mathcal{H}_s$, respectively;
2 **for** each $q \in Q$ **do**
3 $\quad |q(\mathcal{D})| = |\cap_{w \in q} I_1(w)|$; $|q(\mathcal{H}_s)| = |\cap_{w \in q} I_2(w)|$;
4 **end**
5 Initialize a forward index F, where $F(d) = \phi$ for each $d \in \mathcal{D}$;
6 **for** each $q \in Q$ **do**
7 $\quad$ **for** each $d \in q(\mathcal{D})$ **do**
8 $\quad\quad$ Add q into $F(d)$;
9 $\quad$ **end**
10 **end**
11 Let P denote an empty priority queue;
12 **for** each $q \in Q$ **do**
13 $\quad$ **if** $\frac{|q(\mathcal{H}_s)|}{\theta} \leq k$ **then** P.push($\langle q, |q(\mathcal{D})|\rangle$) ;
14 $\quad$ **else** P.push($\langle q, |q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|}\rangle$) ;
15 **end**
16 Initialize a hash map U, where $U(q) = 0$ for each $q \in Q$;
17 **while** $b > 0$ and $\mathcal{D} \neq \phi$ **do**
18 $\quad \langle q^*, \text{old\_priority}\rangle$ = P.pop();
19 $\quad$ **if** $|U(q^*)| != 0$ **then**
20 $\quad\quad$ **if** $\frac{|q^*(\mathcal{H}_s)|}{\theta} \leq k$ **then**
21 $\quad\quad\quad$ new_priority = $|q^*(\mathcal{D})| - |U(q^*)|$
22 $\quad\quad$ **else**
23 $\quad\quad\quad$ new_priority = $\big(|q^*(\mathcal{D})| - |U(q^*)|\big) \cdot \frac{k\theta}{|q^*(\mathcal{H}_s)|}$
24 $\quad\quad$ **end**
25 $\quad\quad$ P.push($\langle q$, new_priority$\rangle$); $|U(q^*)| = 0$;
26 $\quad\quad$ **continue**;
27 $\quad$ **end**
28 $\quad$ Issue q^* to the hidden database, and then get the result $q^*(\mathcal{H})_k$;
29 $\quad$ **if** q^* is a solid query **then** $\mathcal{D}_{\text{removed}} = q^*(\mathcal{D})$;
30 $\quad$ **else** $\mathcal{D}_{\text{removed}} = q^*(\mathcal{D})_{cover}$;
31 $\quad$ **for** each $d \in \mathcal{D}_{removed}$ **do**
32 $\quad\quad$ **for** each $q \in F(d)$ **do**
33 $\quad\quad\quad$ $U(q) += 1$;
34 $\quad\quad$ **end**
35 $\quad$ **end**
36 $\quad \mathcal{D} = \mathcal{D} - \mathcal{D}_{\text{removed}}$; $Q = Q - \{q\}$; $b = b - 1$;
37 **end**

B PSEUDO-CODE AND TIME COMPLEXITY ANALYSIS

Algorithm 4 depicts the pseudo-code of our efficient implementation of QSEL-EST.

At the initialization stage (Lines 1-15), QSEL-EST needs to (1) create two inverted indices based on $\mathcal{D}$ and $\mathcal{H}_s$ with the time complexity of $O\big(|\mathcal{D}||d| + |\mathcal{H}_s||h|\big)$; (2) create a forward index with the time complexity of $O(|Q||q(\mathcal{D})|)$; (3) create a priority queue with the time complexity of $O(|Q|\log(|Q|))$; (4) compute the query frequency w.r.t. $\mathcal{D}$ and $\mathcal{H}_s$ with the time complexity of $O\big(cost_q \cdot |Q|\big)$, where $cost_q$ denotes the average cost of using the inverted index to compute $|q(\mathcal{D})|$ and $|q(\mathcal{H}_s)|$, which is much smaller than the brute-force approach (i.e., $cost_q \ll |\mathcal{D}||q| + |\mathcal{H}_s||q|$).

At the iteration stage (Lines 16-37), QSEL-EST needs to (1) select b queries from the query pool with the time complexity of $O\big(b \cdot t \cdot \log|Q|\big)$, where t denotes the average number of times that Case Two (Line 19) happens over all iterations; (2) apply on-demand updating mechanism to each removed record with the total time complexity of $O\big(|\mathcal{D}||F(d)|\big)$, where $|F(d)|$ denotes the average number of queries that can cover d, which is much smaller than $|Q|$.

By adding up the time complexity of each step, we can see that our efficient implementation of QSEL-EST can be orders of magnitude faster than the naive implementation.

C EXPERIMENTAL SETTINGS

C.1 Simulated Hidden Database

SMARTCRAWL. We adopted the query-pool generation method (Section 3) to generate a query pool for our experiments ($t = 2$). For query selection, we implemented both biased and unbiased estimators. SMARTCRAWL-B denoted our framework with biased estimators; SMARTCRAWL-U represented our framework with unbiased estimators.

IDEALCRAWL. We implemented the ideal framework, called IDEALCRAWL, which used the same query pool as SMARTCRAWL but select queries using the ideal greedy algorithm (Algorithm 2) based on true benefits.

FULLCRAWL. FULLCRAWL aims to issue a query such that the query can cover a hidden database as more as possible. We assumed that there was 1% hidden database sample available for FULLCRAWL. It first generated a query pool based on the sample and then issued queries in the decreasing order of their frequency in the sample.

NAIVECRAWL. NAIVECRAWL concatenated title, venue, and author attributes of each local record as a query and issued the queries to a hidden database in a random order.

C.2 Real Hidden Database

SMARTCRAWL. SMARTCRAWL generated a query pool based on business name and city attributes ($t = 2$), and issued queries based on estimated benefits derived from biased estimators.

NAIVECRAWL. For each local record, NAIVECRAWL concatenated the business name and city attributes of the record as a query, issued it to the hidden database, and used the returned hiddent records to cover the local record.

FULLCRAWL. FULLCRAWL used the hidden database sample to generate a query pool and then issued queries in the decreasing order of their frequency in the sample.