Progressive Deep Web Crawling Through Keyword Queries For Data Enrichment

Pei Wang◇ Ryan Shea◇ Jiannan Wang◇ Eugene Wu†
Simon Fraser University◇ Columbia University†
{peiw, rws1, jnwang}@sfu.ca ew2493@columbia.edu

ABSTRACT

Data enrichment is the problem of extending a local database with new attributes from external data sources. In this paper, we study how to enrich a local database using deep web (i.e., 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. We find that query selection is the most challenging part. We start with a simple query selection strategy which only considers the query frequency in a local database. We identify three factors that may affect its performance. For each factor, we first theoretically analyze the negative impact, and then propose effective approaches to mitigate the impact. 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 $2-10\times$ in a large variety of situations.

1. INTRODUCTION

Data scientists often spend more than 80% of their time on data preparation—the process of turning raw data into a form suitable for further analysis. This process involves many tasks such as data cleaning, data integration, and data enrichment. In this paper, we focus on data enrichment, the act of extending a local database with new attributes from external data sources. For example, suppose a data scientist collects a list of VLDB papers and wants to know the citation of each paper, or she collects a list of newly opened restaurants in Canada and wants to know the category of each restaurant. A natural use case of data enrichment is to augment a training set with new features. A recent study has shown that it is also beneficial to some other data preparation tasks such as error detection [9].

Data enrichment is not a new research topic. However, existing work is mainly focused on how to enrich data using web tables (i.e., HTML tables) [22, 23, 47, 46, 35, 13]. In this paper, we argue that there is a strong need to study how to enrich data using *deep web*, another invaluable but widely ignored data source. Deep web (a.k.a. hidden databases), such as Yelp and The ACM Digital Library, curate a large database of entities but only provide a restrictive query interface for the public to access their data. For example, Yelp maintains a large database of restaurants and provides a keyword-search interface that allows a user to enter a set of keywords (e.g., "Lotus of Siam") to search for a restaurant.

There are many benefits to leverage deep web for data enrichment. First, the data stored in a hidden database (e.g., Yelp) is typically of high quality and keeps up to date. Second, a hidden database is often very big and covers a large number of entities in a certain domain. Third, a hidden database may contain some useful data (e.g., restaurant ratings) that cannot be found from elsewhere. Fourth, many hidden databases provide easy-to-use APIs (e.g., Yelp APIs [8]) to facilitate data extraction.

However, there is no free lunch: in order to achieve these advantages, we have to solve a challenging problem – *deep web crawling*. A hidden database can only be accessed by a restrictive query interface. In order to crawl its data, we need to decide which queries should be issued. A deep website may have different kinds of query interfaces such as keyword search [12, 34, 14, 26, 10], form-like search [36, 28, 31, 29, 38, 39], and graph-browsing [33]. We consider the *keyword-search interface* in this paper because it is widely supported by a variety of deep web sites (e.g., Yelp [8], IMDb [3], SoundCloud [6], GoodReads [1],The ACM Digital Library [7]). Specifically, we assume that a hidden database can take a set of keywords as input, and returns the top-k records that match the keywords, where the top-k records are selected based on an *unknown* ranking function.

It is prohibitively expensive or even impossible to issue a lot of queries to a hidden database. For example, Yelp API [8] is restricted to 25,000 free requests per day [8] and Goolge Maps API [2] only allows 2,500 free requests per day [2]. Progressive deep web crawling is a promising idea to mitigate this issue. Given a query budget b, progressive deep web crawling aims to select a set of the best b queries in order to maximize the *benefit* of issuing the queries. In the existing deep web crawling literature, the benefit is defined as the total number of the records crawled from a hidden database. In our work, if a hidden record cannot be used to enrich a local database, there is no need to crawl it, thus we have a totally different way to quantify the benefit. We call our problem *progressive deep web crawling driven by data enrichment*, shorted as DeepEnrich. Informally, given a local database $\mathcal{D}$, a hidden database $\mathcal{H}$, and a query budget b, the goal of DeepEnrich is to issue a set of b queries to $\mathcal{H}$ such that the union of the query results can *match* as many records in $\mathcal{D}$ as possible. We say a hidden record matches a local record if and only if they refer to the same real-world entity (e.g., the same restaurant). There are two straightforward solutions to DeepEnrich:

- **FullCrawl** is to apply an existing deep-web crawling approach [36, 31, 26, 38, 10, 28, 39, 34] to crawl the entire hidden 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.

- **NaiveCrawl** is to enumerate each record in $\mathcal{D}$ and then generate a query to cover it. Each generated query tends to be very *specific* and contain many keywords. For example, a query can be the entire paper title or the full restaurant name. NaiveCrawl suffers from two limitations. First, it needs to generate a lot of queries for a large $|\mathcal{D}|$. Suppose $|\mathcal{D}| = 100,000$. Then, it requires a query budget of $b = 100,000$ in order to cover the entire $\mathcal{D}$. Second, it is sensitive to data errors. Suppose a restaurant name is dirty (e.g., "Lotus of Siam" is falsely written as "Lotus of Siam 12345"). If we issue "Lotus of Siam 12345" to a hidden database (e.g., Yelp), due to the data error, the hidden database may not return the matching restaurant to us. Despite that NaiveCrawl has these limitations, OpenRefine (a state-of-the-art data wrangling tool) is using this approach to crawl data from web services [4].

To overcome the limitations of FullCrawl and NaiveCrawl, we propose the SmartCrawl framework. Given $\mathcal{D}$, $\mathcal{H}$, and b, SmartCrawl first constructs a query pool from $\mathcal{D}$, and then it iteratively selects the query with the largest benefit from the pool, and issues it to $\mathcal{H}$ until the budget b is exhausted. The key insights of SmartCrawl are as follows:

- **Query Sharing.** Unlike NaiveCrawl, SmartCrawl generates both specific and *general* queries, where a general query (e.g., "Lotus") can cover multiple local records at a time. Typically, a hidden database sets the top-k restriction with k in the range between 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 cover 100 records, which is 100 times better than NaiveCrawl. Even better, since a general query tends to contain fewer keywords, it is less sensitive to data errors compared to NaiveCrawl.
- **Local-Database-Aware Crawling.** Unlike FullCrawl, SmartCrawl seeks to evaluate the benefit of each query based on how many records the query 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 $|\mathcal{H}| = 10^7$ and $|\mathcal{D}| = 10^5$. Then, FullCrawl needs to cover 100 times more records than SmartCrawl.

We find that the query-selection stage is the most challenging part. A bad query-selection strategy can have a huge negative impact on the SmartCrawl's performance (i.e., the coverage of $\mathcal{D}$). Ideally, SmartCrawl wants to select the query such that, if the query was issued, the query result can cover the maximum number of (uncovered) records in $\mathcal{D}$. We call it QSel-Ideal. However, QSel-Ideal cannot be achieved in reality due to the "chicken-and-egg" problem (i.e., it needs to issue the query in order to decide whether or not the query should be issued). Therefore, the key problem is how to effectively estimate the query benefit without issuing a query. This paper presents a comprehensive solution to this problem.

We start with a simple approach, called QSel-Simple, which uses the *query frequency w.r.t.* $\mathcal{D}$ as an estimation of query benefit. Query frequency w.r.t. $\mathcal{D}$ is defined as the number of records in $\mathcal{D}$ that contain the query. For example, consider a query q = "Noodle House". If there are 100 records in $\mathcal{D}$ that contain "Noodle" as well as "House", this simple approach will set the query benefit to 100. While this approach sounds simple, it is non-trivial to analyze its performance theoretically. There are many interesting questions that worth investigating, including that *is there any situation when* QSel-Simple *and* QSel-Ideal *are equivalent*, *what factors may lead* QSel-Simple *to perform less effective than* QSel-Ideal, and *how to reduce their performance gap*. Being able to answer these questions will not only help us gain a deep understanding of the QSel-Simple's performance, but also guide us to develop a set of new techniques to improve its performance.

We identify three factors that may affect the QSel-Simple's performance, and we prove that by making the following three assumptions, QSel-Simple and QSel-Ideal are equivalent. Furthermore, we discuss that if an assumption does not hold, how to optimize QSel-Simple to reduce the performance gap between QSel-Simple and QSel-Ideal.

Factor 1: Local Database Coverage. The first assumption is that $\mathcal{D}$ can be fully covered by $\mathcal{H}$. If this assumption does not hold, there will be some records in $\mathcal{D}$ that cannot be found from $\mathcal{H}$. We denote the number of such records by $|\Delta\mathcal{D}| = |\mathcal{D} - \mathcal{H}|$. Recall that in the previous example, QSel-Simple sets the benefit of q = "Noodle House" to 100. However, if all the 100 records are in $|\Delta\mathcal{D}|$, there will be no benefit to issue the query, i.e., the true benefit is 0. Therefore, we need to use the *query frequency w.r.t.* $\mathcal{D} - \Delta\mathcal{D}$ rather than $\mathcal{D}$ to estimate query benefit. For this reason, we first study how to bound the performance gap between QSel-Simple, which uses query frequency w.r.t. $\mathcal{D}$, and QSel-Ideal, which uses query frequency w.r.t. $\mathcal{D} - \Delta\mathcal{D}$. If $|\Delta\mathcal{D}|$ is big, then the performance gap can be large, thus we propose effective techniques to mitigate the negative impact of big $|\Delta\mathcal{D}|$.

Factor 2: Top-k Constraint. The second assumption is that there is no top-k constraint. If this assumption does not hold, we need to be aware that the benefit of any query cannot be larger than k. For example, suppose k = 50. The true benefit of any query cannot be larger than 50 no matter how big $|q(\mathcal{D})|$ is. We study how to leverage a hidden database sample to estimate query benefits. There is a large body of work on deep web sampling [27, 11, 49, 20, 17, 18, 19, 48, 42]. Recently, Zhang et al. [48] proposed an efficient technique that can create an unbiased sample of a hidden database as well as an unbiased estimate of the sampling ratio through the keyword-search interface. We apply this technique 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.

The key challenge is that a hidden database has an *unknown* ranking function. To address this challenge, we divide queries into two types: *solid query* and *overflowing query*. Intuitively, a solid query will not be affected by the top-k constraint, but an overflowing query will. We study how to use a hidden database sample to predict query type. We investigate both unbiased estimators as well as biased estimators to estimate the benefit of each type of queries. We say an estimator is unbiased if and only if the expected value of the estimated benefit is equal to the true benefit. We show that the biased estimators typically have a small bias and they are superior to the unbiased ones especially for a small sampling ratio.

Factor 3: Fuzzy Matching. The third assumption is that there is no fuzzy matching situation, i.e., a local record and a hidden record refer to the same entity if and only if they look exactly the same. If this assumption does not hold, our estimators may give less accurate estimates. We first show that our unbiased estimators are still unbiased but our biased estimators will have a large bias. Nevertheless, we demonstrate that the SmartCrawl framework tends to be more robust to the fuzzy-matching situation compared to NaiveCrawl. In the end, we discuss how to apply an existing entity resolution technique to mitigate the impact of the fuzzy-matching situation.

We have implemented all the above optimization techniques, and call the new query selection strategy QSel-Est. We demonstrate the superiority of QSel-Est over QSel-Simple in the cases of when $\mathcal{D}$ is not fully covered by $\mathcal{H}$, the search interface enforces the top-k constraint, or there are many fuzzily matching record pairs.

We experimentally compare the different crawling frameworks: SmartCrawl, NaiveCrawl, and FullCrawl. The results show that SmartCrawl can significantly outperform the two baselines in terms of the number of covered records in a large variety of situations. We have built an end-to-end data enrichment system[1] based on the SmartCrawl framework and made a video to demonstrate how it works[2]. The system was used in a data science class at Simon Fraser University and helped the students to reduce the time spent on data enrichment from hours to minutes. More details can be found in our demo paper [43].

To summarize, our main contributions are:

- To the best of our knowledge, we are the first to study the DeepEnrich problem. We formalize the problem and present two straightforward solutions, NaiveCrawl and FullCrawl.
- We propose the SmartCrawl framework based on the ideas of query sharing and local-database-aware crawling. We present a simple query selection strategy called QSel-Simple and prove that it is equivalent to QSel-Ideal under certain assumptions.
- We identify three factors that may affect the effectiveness of QSel-Simple. We analyze the negative impact of each factor and propose new techniques to mitigate the impact.
- We conduct extensive experiments over simulated and real hidden databases. The results show that SmartCrawl outperforms NaiveCrawl and FullCrawl by a factor of $2-10\times$ in a large variety of situations.

2. PROBLEM FORMALIZATION

In this section we formulate the DeepEnrich problem and discuss the challenges. Without loss of generality, we model a local database and a hidden database as two relational tables. 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; hidden records can be accessed only by issuing queries through a *keyword-search interface*. Let q denote a *keyword query* consisting of a set of keywords (e.g., q = "Thai Cuisine"). The keyword-search interface returns top-k hidden records $q(\mathcal{H})_k$ of a keyword query q. We say a local record d is covered by the query q if and only if there exists $h \in q(\mathcal{H})_k$ such that d and h refer to the same real-world entity. Since top-k results are returned, we can cover multiple records using a single query. To make the best use of resource access, our goal is to cover as many records as possible. Specifically, given a query budget b, a local database $\mathcal{D}$, and a hidden database $\mathcal{H}$, we try to find a set of b queries such that they can cover as many local records as possible.

In this paper, we will mainly focus on the deep web crawling part. In order to build an end-to-end data enrichment system, however, we also need to implement many other components such as schema matching (i.e., match the inconsistent schemas between a local database and a hidden database) and entity resolution (i.e., check whether a local record and a hidden record refer to the same real-world entity). We have discussed how to implement these components using existing approaches as well as how to put them together in the demo paper [43]. Therefore, we assume that schemas have been aligned and we treat entity resolution as a black box.

Problem Statement. We model $\mathcal{H}$ and $\mathcal{D}$ as two sets[3]. We define the intersection between $\mathcal{D}$ and $\mathcal{H}$ as

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

$\mathsf{match}(d,h)$ returns True if d and h refer to the same real-world entity; otherwise, $\mathsf{match}(d,h)$ returns False. This intersection contains all the local records that can be covered by $\mathcal{H}$. Note that $\mathcal{D}$ is not necessarily a subset of $\mathcal{H}$.

Let $q(\mathcal{D})_{\text{cover}}$ denote the set of local records that can be covered by q. The goal of DeepEnrich is to select a set $\mathcal{Q}_{\text{sel}}$ of queries within the budget such that $|\bigcup_{q\in\mathcal{Q}_{\text{sel}}} q(\mathcal{D})_{\text{cover}}|$ is maximized.

Problem 1 (DeepEnrich). *Given a budget b, a local database $\mathcal{D}$, and a hidden database $\mathcal{H}$, the goal of DeepEnrich is to select a set of queries, $\mathcal{Q}_{sel}$, to maximize the coverage of $\mathcal{D}$, i.e.,*

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

Unfortunately, DeepEnrich is an NP-Hard problem, which can be proved by a reduction from the maximum-coverage problem (a variant of the set-cover 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 reasons in the next section).

Keyword-search Interface. In this paper, we consider the widely used *keyword-search interface*, and defer other interfaces (e.g., form-like search, graph-browsing) to future work. We investigated a number of deep websites to understand the keyword-search interface in real-word scenarios. We found that most of them (e.g., IMDb, The ACM Digital Library, GoodReads, and SoundCloud) support the conjunctive keyword search interface. That is, they only return the records that contain all the query keywords (we do not consider stop words as query keywords). Even if they violate this assumption (e.g., Yelp), they tend to rank the records that contain all the query keywords to the top. In the experiments, we demonstrate the superiority of our framework in both situations.

Definition 1 (Conjunctive Keyword Search). Each record is modeled as a document, denoted by $\mathsf{document}(\cdot)$, which concatenates all[4] the attributes of the record. Given a query, we say a record h (resp. d) *satisfies* the query if and only if $\mathsf{document}(h)$ (resp. $\mathsf{document}(d)$) contains *all* the keywords in the query.

[1] http://deeper.sfucloud.ca/

[2] http://tiny.cc/deeper-video

[3] 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 $\mathcal{H}$ or treat them as one record.

[4] 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.

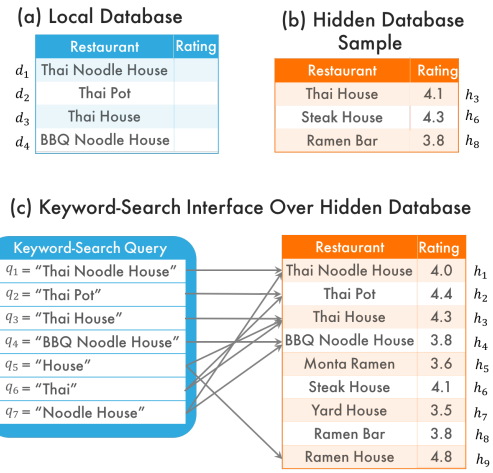


Figure 1: A running example ($k = 2$, $\theta = \frac{1}{3}$). There are four record pairs (i.e., $\langle d_1, h_1\rangle$, $\langle d_1, h_1\rangle$, $\langle d_3, h_3\rangle$, and $\langle d_4, h_4\rangle$) that refer to the same real-world entity. Each arrow points from a query to its result. (Please ignore Figure 1(b) for now and we will discuss it later in Section 5).

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

Due to the top-k constraint, a search interface 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 formally defines the keyword-search interface.

DEFINITION 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 1. *Figure 1 shows an example local database(Figure 1(a)), hidden database and the correspondence (grey line) between queries and returned results(Figure 1(c))(Ignore Figure 1(b) for now). For example, consider q_5 = "House". According to the conjunctive search interface assumption, we have $q_5(\mathcal{D}) = \{d_1, d_3, d_4\}$ and $q_5(\mathcal{H}) = \{h_1, h_3, h_6, h_7, h_9\}$. Suppose only top-2 results will be returned. After issuing the query, we get the query result $q_5(\mathcal{H})_k = \{h_3, h_9\}$ and we obtain $q_5(\mathcal{D})_{cover} = \{d_3\}$. Suppose $b = 2$. The goal of the* DeepEnrich *problem is to select two queries q_i, q_j from $\{q_1, q_2, \cdots, q_7\}$ in order to maximize $|q_i(\mathcal{D})_{cover} \cup q_j(\mathcal{D})_{cover}|$. The ultimate goal is to enrich the local table with a rating attribute from a hidden database. We can see that the optimal solution should select q_6 and q_7 since $|q_6(\mathcal{D})_{cover} \cup q_7(\mathcal{D})_{cover}| = 4$ reaches the maximum. The key challenge is how to decide which queries should be selected in order to cover the largest number of local records.*

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 a hidden database as possible. However, they fail to exploit the ideas of query sharing and local-database-aware crawling. In response, we propose the SMARTCRAWL framework.

Framework Overview. The SMARTCRAWL framework 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 $q^*(\mathcal{D})_{\text{cover}}$, and updates the query pool. This iterative process will repeat until the budget is exhausted or the local database is fully covered.

3.1 Query Pool Generation

Let $\mathcal{Q}$ denote a query pool. If a query q does not appear in any local record, i.e., $|q(\mathcal{D})| = 0$, we do not consider the query. Therefore, there is a finite number of queries that need to be considered, i.e.,

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

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, we hope the query pool to include the queries that can cover multiple local records at a time.

- To satisfy the first principle, SMARTCRAWL adopts the same method as NAIVECRAWL. That is, for each local record, SMARTCRAWL generates a very specific query. For example, a query can be a concatenation of the attributes in a candidate key. Let $\mathcal{Q}_{naive}$ denote the collection of the queries generated in this step. We have $|\mathcal{Q}_{naive}| = |\mathcal{D}|$.
- To satisfy the second principle, SMARTCRAWL finds the queries such that $|q(\mathcal{D})| \ge t$ ($t = 2$ by default). We can efficiently generate these queries using Frequent Pattern Mining algorithms (e.g., [24]). 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:

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

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.

Algorithm 1: QSel-Ideal Algorithm

```
  Input: Q, D, 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-Simple Algorithm

```
Replace Line 3 in Algorithm 1 with the following lines:
    benefit(q) = |q(D)|;
```

Example 2. *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 $\mathcal{Q}_{naive} = \{q_1, q_2, q_3, q_4\}$, 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".*

3.2 Query Selection

After a query pool is generated, SmartCrawl enters the query-selection stage.

Let us first take a look at how QSel-Ideal works. QSel-Ideal assumes that we know the true benefit of each query in advance. 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 $\mathcal{Q}$, 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 of each query until the query is issued, but it needs to know the true benefit in order to decide which query should be issued. To overcome the problem, we need to estimate the benefit of each query and then use the estimated benefit to determine which query should be issued.

A simple solution is to use the query frequency w.r.t. $q(\mathcal{D})$ as the estimated benefit. Algorithm 2 depicts the pseudo-code. We can see that SmartCrawl$_s$ differs from IdealCrawl only in the benefit calculation part. Intuitively, SmartCrawl$_s$ tends to select high-frequent keyword queries. For example, consider q_5 = "House" and q_7 = "Noodle House" in Figure 1. Since $|q_5(\mathcal{D})| = 3$ and $|q_7(\mathcal{D})| = 2$, SmartCrawl$_s$ prefers q_5 to q_7. In fact, despite that q_5 is more frequent, issuing q_5 can only cover a singe local record but issuing q_7 can cover two local records. Therefore, IdealCrawl prefers q_7 to q_5.

QSel-Ideal vs. QSel-Simple. As discussed in the Introduction, the QSel-Simple's performance may be affected by the three factors. We can prove that by making certain assumptions about the three factors, QSel-Simple and QSel-Ideal are equivalent.

Algorithm 3: QSel-Bound Algorithm

```
Input: Q, D, H, b
Result: SmartCrawl_b covers at least (1 − |ΔD|/b) · N_ideal records.
while b > 0 and D ≠ ϕ do
  for each q ∈ Q do
    benefit(q) = |q(D)|;
  end
  Issue q* to H, and then get the query result q*(H)_k;
  q*(ΔD) = q*(D) − q*(D)_cover;
  if |q*(ΔD)| = 0 then
    D = D − q*(D)_cover;    Q = Q − {q*};
  else
    D = D − q*(ΔD);        // Note that q* is not removed;
  end
  b− = 1;
end
```

Assumption 1 (Factor 1). We assume that $\mathcal{D}$ can be fully covered by $\mathcal{H}$. That is, for each $d \in \mathcal{D}$, there exist a hidden record $h \in \mathcal{H}$ such that $\mathsf{match}(d, h) = True$.

Assumption 2 (Factor 2). We assume that $\mathcal{H}$ does not enforce a top-k constraint. That is, for each query $q \in \mathcal{Q}$, we have that $q(\mathcal{H})_k = q(\mathcal{H})$.

Assumption 3 (Factor 3). We assume that there is no fuzzy-matching situation. That is, for any $d \in \mathcal{D}$ and $h \in \mathcal{H}$, d and h are matching if and only if $\mathsf{document}(d) = \mathsf{document}(h)$.

Lemma 1. *If Assumptions 1-3 hold, then* QSel-Ideal *and* QSel-Simple *are equivalent.*

Proof. All the proofs in this paper can be found in the technical report [5]. □

It is easy to see that these assumptions are strong and may not hold in practice. In the following sections, we will relax each assumption and discuss how to optimize QSel-Simple accordingly.

4. LOCAL DATABASE COVERAGE

In this section, we assume that Assumptions 2 and 3 hold, but Assumption 1 does not. Let $\Delta\mathcal{D} = \mathcal{D} - \mathcal{H}$ denote the set of the records in $\mathcal{D}$ that cannot be covered by $\mathcal{H}$. We want to explore how $|\Delta\mathcal{D}|$ will affect the performance gap between QSel-Simple and QSel-Ideal. For example, suppose $\mathcal{D} = 10000$ and $|\Delta\mathcal{D}| = 10$, which means that $\mathcal{D}$ only has a very small number of records that cannot be covered by $\mathcal{H}$. In this situation, how big the performance gap can be? Is it possible that QSel-Ideal can cover a significantly more number of records than QSel-Simple? We seek to answer these questions in this section.

4.1 Bounding the Performance Gap

We find that it is not easy to directly reason about the performance gap between QSel-Ideal and QSel-Simple. Thus, we construct a new algorithm, called QSel-Bound, as a proxy. We first bound the performance gap between QSel-Ideal and QSel-Bound, and then compare the performance between QSel-Bound and QSel-Simple.

As the same as QSel-Simple, 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-Simple. (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 $q(\Delta\mathcal{D})$ 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-Ideal and QSel-Bound, let $\mathcal{Q}_{\text{sel}} = \{q_1, q_2, \cdots, q_b\}$ and $\mathcal{Q}'_{\text{sel}} = \{q'_1, q'_2, \cdots, q'_b\}$ 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.,

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

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

Lemma 2. *Given a query pool $\mathcal{Q}$, the worst-case performance of QSel-Bound is bounded w.r.t. QSel-Ideal, i.e., $N_{\text{bound}} \geq (1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}$.*

Proof 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., $\{q_i \mid 1 \leq i \leq b - |\Delta\mathcal{D}|\} \subseteq \mathcal{Q}'_{sel}$. 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 lemma indicates that when $|\Delta\mathcal{D}|$ is relatively small w.r.t. b, QSel-Bound performs almost as well as QSel-Ideal. For example, consider a local database having $|\Delta\mathcal{D}| = 10$ records not in a hidden database. Given a budget $b = 1000$, 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-Bound vs. QSel-Simple. Note that QSel-Simple and QSel-Bound are both applicable in practice, but we empirically find that QSel-Simple tends to perform better. 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-Simple.

4.2 Mitigate the Negative Impact of Big $|\Delta\mathcal{D}|$

When $|\Delta\mathcal{D}|$ is small, QSel-Simple has a similar performance with QSel-Ideal; however, when $|\Delta\mathcal{D}|$ is very large, QSel-Simple may perform much worse than QSel-Ideal. Thus, we need to study how to mitigate the negative impact of big $|\Delta\mathcal{D}|$.

Our basic idea is to predict the local records in $\Delta\mathcal{D}$ and then remove them from $\mathcal{D}$. In other words, we want to predict which local record cannot be covered by $\mathcal{H}$. Our prediction works as follows. (1) Issue a selected query to a hidden database and get the query result $q(\mathcal{H})$, (2) use $q(\mathcal{H})$ to cover $\mathcal{D}$ and obtain $q(\mathcal{D})_{cover}$, and (3) return the prediction result $q(\mathcal{D}) - q(\mathcal{D})_{cover}$ w.r.t. q.

The prediction says that for any record in $q(\mathcal{D}) - q(\mathcal{D})_{cover}$, it must not be covered by $\mathcal{H}$. We can prove the correctness of this prediction result by contradiction. Assume that there exists a record $d \in q(\mathcal{D}) - q(\mathcal{D})_{cover}$ which can be covered by $h \in \mathcal{H}$. This is impossible because since d satisfies q, then h must satisfy q (Assumption 2), thus h must be retrieved by q (Assumption 3). Therefore, we can deduce that $d \in q(\mathcal{D})_{cover}$ must hold, which contradicts that $d \in q(\mathcal{D}) - q(\mathcal{D})_{cover}$.

Table 1: A summary of our estimators for query benefits.

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

Remarks. When a hidden database enforces the top-k constraint, we cannot apply the above prediction method directly. Recall that there are two types of queries: solid query and overflowing query. For a solid query, since the query result will *not* be affected by the top-k constraint, the prediction method can be applied directly. For an overflowing query, since the query result is not trustful, the prediction method cannot be applied. We will discuss some other challenges of the top-k constraint in the next section.

5. TOP-K CONSTRAINT

In this section, we study how to handle the top-k constraint. Recall that QSel-Simple estimates the benefit of each query as $|q(\mathcal{D})|$. This is problematic because when the top-k constraint is enforced, the query benefit is guaranteed to be no larger than k, but QSel-Simple does not take this important information into account.

To overcome this limitation, our key idea is to leverage a hidden database sample to estimate query benefits. We first present how to use a hidden database sample to predict query type (solid or overflowing) in Section 5.1, and then propose new estimators to estimate the benefits of solid queries in Section 5.2 and the benefits of overflowing queries in Section 5.3, respectively. Table 1 summarizes the proposed estimators.

5.1 Query Type Prediction

Sampling from a hidden database is a well-studied topic in the Deep Web literature [27, 11, 49, 20, 17, 18, 19, 48, 42]. We create a hidden database sample offline, and reuse it for any user who wants to match their local database with the hidden database. 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 [11, 49, 48]. 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 [48] in the experiments and evaluate the performance of SmartCrawl using the sample created by the technique (Section 7.3).

To estimate the benefit of a query, we first use the sample $\mathcal{H}_s$ to predict whether the query is solid or overflowing, and then applies a corresponding estimator. Specifically, we compute the query frequency w.r.t. $\mathcal{H}_s$ and use 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.

Example 3. *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 it is predicated 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 it is predicated as an overflowing query, which is also a correct prediction. In summary, q_1, q_2, q_4, q_7 are predicted as solid queries and q_3, q_5, q_6 are predicted as overflowing queries. The only wrong prediction is to predict q_3 as a solid query.*

5.2 Estimators For Solid Queries

We propose an unbiased estimator and a biased estimator for solid queries, respectively.

Unbiased Estimator. The true benefit of a query is defined as:

$$\mathsf{benefit}(q) = |q(\mathcal{D})_{cover}| = |q(\mathcal{D}) \cap q(\mathcal{H})_k|. \quad (1)$$

According to the definition of solid queries in Definition 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

$$\mathsf{benefit}(q) = |q(\mathcal{D}) \cap q(\mathcal{H})|. \quad (2)$$

The benefit estimation problem can be modeled as a selectivity estimation problem, which aims to estimate the selectivity of the following query:

SELECT d, h FROM $\mathcal{D}$, $\mathcal{H}$
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:

$$\mathsf{benefit}(q) \approx \frac{|q(\mathcal{D}) \cap q(\mathcal{H}_s)|}{\theta}, \quad (3)$$

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

LEMMA 3. Given a solid query q, then $\frac{|q(\mathcal{D}) \cap q(\mathcal{H}_s)|}{\theta}$ is an unbiased estimator of $|q(\mathcal{D}) \cap q(\mathcal{H})|$.

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 $q \in \mathcal{Q}$. 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.

Biased Estimator. We propose another estimator to overcome the limitations. The benefit of a solid query (Equation 2) can be denoted by

$$\mathsf{benefit}(q) = |q(\mathcal{D}) - q(\Delta\mathcal{D})| = |q(\mathcal{D})| - |q(\Delta\mathcal{D})|. \quad (4)$$

A hidden database (e.g., Yelp, IMDb) often has a very good coverage of the entities in some domain (e.g., Restaurant, Movie, etc.). As a result, $\Delta\mathcal{D}$ is often small, and thus $|q(\Delta\mathcal{D})|$, as a subset of $\Delta\mathcal{D}$, is even much smaller. For this reason, we derive the following estimator:

$$\mathsf{benefit}(q) \approx |q(\mathcal{D})|, \quad (5)$$

where the bias of the estimator is $|q(\Delta\mathcal{D})|$. In the experiments, we compare the biased estimator with the unbiased one, and find that the biased one tends to perform better, especially for a small sampling ratio.

5.3 Estimators For Overflowing Queries

We propose a (conditionally) unbiased estimator and a biased estimator for overflowing queries, respectively.

Intuitively, the benefit of an overflowing query can be affected by three variables: $|q(\mathcal{D})|$, $|q(\mathcal{H})|$, and k. How should we systematically combine them together in order to derive an estimator? We call this problem *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.

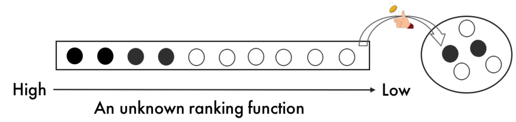


Figure 2: An illustration of breaking the top-k constraint.

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

$$E[X] = \sum_{i=0}^{n} i \cdot P(X = i) = n\frac{k}{N}. \quad (6)$$

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 2, 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 6, we have

$$E[\mathsf{benefit}(q)] = n \cdot \frac{k}{N} = |q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|} \quad (7)$$

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.

Estimators. Note that Equation 7 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.

$$|q(\mathcal{H})| \approx \frac{|q(\mathcal{H}_s)|}{\theta}. \tag{8}$$

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 5.2. The unbiased estimator (see Equation 3) is

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

The biased estimator (see Equation 5) is

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

By plugging Equations 8 and 9 into $|q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|}$, we obtain the first estimator for an overflowing query:

$$\mathsf{benefit}(q) \approx |q(\mathcal{D}) \cap q(\mathcal{H}_s)| \cdot \frac{k}{|q(\mathcal{H}_s)|} \tag{11}$$

This estimator is derived from the ratio of two unbiased estimators. Since $E[\frac{X}{Y}] \neq \frac{E[X]}{E(Y)}$, Equation 11 is not an unbiased estimator, but it is conditionally unbiased (Lemma 4). For simplicity, we omit "conditionally" if the context is clear.

Lemma 4. 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 4. *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 $\mathsf{benefit}(q_3) = 1$.*

By plugging Equations 8 and 10 into $|q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|}$, we obtain another estimator:

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

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

$$bias = |q(\Delta\mathcal{D})| \cdot \frac{k}{|q(\mathcal{H})|} \tag{13}$$

Lemma 5. 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 5.2, $q(\Delta\mathcal{D})$ 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. *Consider q_3 = "Thai House" again. Since $|q_3(\mathcal{D})| = 1$ and $|q_3(\mathcal{H}_s)| = 1$, then $\mathsf{benefit}(q_3)$ 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 4.*

Table 2: True benefits along with estimated benefits using the biased estimators. q_1, q_2, q_4, q_7 are predicted as solid queries; q_3, q_5, q_6 are predicted as overflowing queries.

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

Example 6. *We now illustrate how to use our biased estimators to select queries. Table 2 (the last column) shows the estimated benefits for all queries using the biased estimator. Suppose $b = 2$. In the first iteration, we select q_6 which has the largest estimated benefit (if there is a tie, we break the tie randomly). The returned result can cover two local records $q_6(\mathcal{D})_{cover} = \{d_1, d_4\}$. We remove 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, we select q_7 which has the largest estimated benefit among the remaining queries. The returned result can cover $q_7(\mathcal{D})_{cover} = \{d_2, d_3\}$. Since the budget is exhausted, we stop the iterative process and return $\mathcal{H}_{crawl} = \{d_1, d_2, d_3, d_4\}$. We can see that in this running example, using the biased estimator leads to the optimal solution.*

6. PRACTICAL ISSUES

In this section, we present a number of issues that one may encounter when applying the estimators in practice. We first present the extension to fuzzy matching in Section 6.1, and then discuss how to deal with the problem of inadequate sample size in Section 6.2, and finally present some implementations details in Section 6.3.

6.1 Fuzzy Matching

So far, we have assumed that there is no fuzzy matching situation (Assumption 3). That is, d and h are matching if and only if $\mathsf{document}(d) = \mathsf{document}(h)$. Next, we eliminate the assumption and explore its impact on our estimators.

Let $|A \mathbin{\tilde{\cap}} B|$ 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 6 that by replacing $|q(\mathcal{D}) \cap q(\mathcal{H}_s)|$ with $|q(\mathcal{D}) \mathbin{\tilde{\cap}} q(\mathcal{H}_s)|$, they are still unbiased estimators.

Lemma 6. Lemmas 3 and 4 hold without Assumption 3.

For the two biased estimators, their biases could get larger when Assumption 3 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 3 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 QSel-Est. In general, however, SmartCrawl (with QSel-Est) is a more robust framework than NaiveCrawl when facing 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

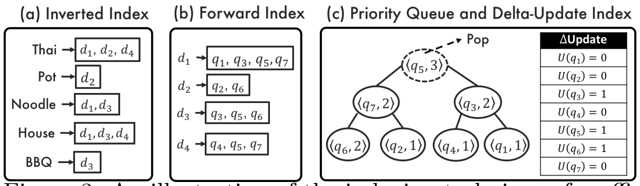


Figure 3: An illustration of the indexing techniques for efficient implementations of our estimators.

likely it is to contain a fuzzy-matching word like "Rest vs. Restaurant". We further investigate this finding in the experiments (Section 7.2.5).

Another change that may need to be made to QSel-Est is about the computation of $q^*(\mathcal{D}) \widetilde{\cap} q^*(\mathcal{H})_k$ at each iteration. One idea is to only search for the exactly matching record pairs between $q^*(\mathcal{D})$ and $q^*(\mathcal{H})_k$, and remove the matched records 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 $\mathsf{Jaccard}(d,h) \geq 0.9$.

6.2 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 1, 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.

6.3 Implementation Details

In this section, we present some implementation details of QSel-Est. More details such as time complexity analysis and pseudo code can be found in [5].

How to compute $|q(\mathcal{D})|$ efficiently? We build an *inverted index* on $\mathcal{D}$ to compute $|q(\mathcal{D})|$ efficiently. 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 3(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 $I(\text{Noodle}) = \{d_1, d_4\}$ and $I(\text{House}) = \{d_1, d_3, d_4\}$, and then compute $q_3(\mathcal{D}) = I(\text{Noodle}) \cap I(\text{House}) = \{d_1, d_4\}$.

How to update $|q(\mathcal{D})|$ efficiently? 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 $F(d)$. For example, Figure 3(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.

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).

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 3(c) illustrates the delta-update index. For example, $U(q) = 1$ 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., $P(q) = |q(\mathcal{D})|$. Figure 3(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 3(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.

7. 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?

7.1 Experimental Settings

7.1.1 Simulated Hidden Database

We designed a simulated experiment based on DBLP dataset[5].

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

[5] `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%

('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 $\mathcal{H} - \mathcal{D}$ 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.

7.1.2 Real Hidden Database

We evaluated SmartCrawl over the Yelp's hidden database.

Local Database. We constructed a local database based on the Yelp dataset[6]. 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 [48] 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.

7.2 Simulated Hidden Databases

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

7.2.1 Sampling Ratio

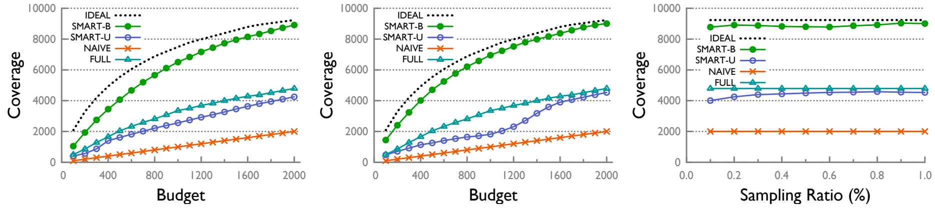
We first examine the impact of sampling ratios on the performance of SmartCrawl. Figure 4 shows the result. In Figure 4(a), we set the sampling ratio to 0.2%, leading to a sample size = $100,000 \times 0.2\% = 200$. We can see that with such a small sample, SmartCrawl-B still had similar performance 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 4(b), which increased the sampling ratio to 1%. In Figure 4, 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.

7.2.2 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.

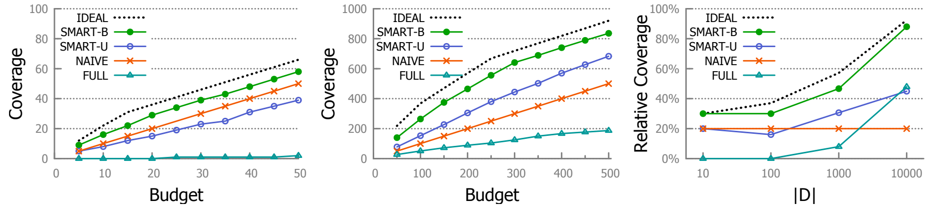
Figure 9(a) shows the result when $|\mathcal{D}|$ has only 100 records. We can see that FullCrawl only covered 2 records after issuing 50 queries while the other approaches all covered 39 more records. Another interesting observation is that even for such a small local database, SmartCrawl-B can still outperform NaiveCrawl due to the accurate benefit estimation as well as the power of query sharing. Figure 9(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 9(c). We can see that as $|\mathcal{D}|$ increased, all the approaches except NaiveCrawl

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



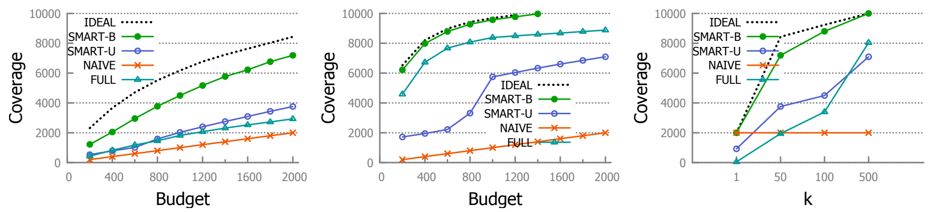
(a) Sampling Ratio = 0.2% (b) Sampling Ratio = 1% (c) Budget = 2000

Figure 4: Comparisons of different approaches with various sampling ratios.



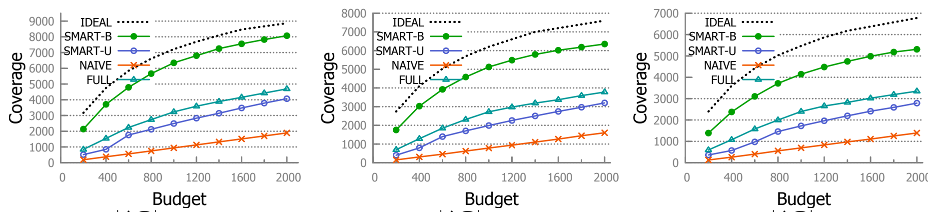
(a) Budget = 100 (b) Budget = 1000 (c) Budget = 20% of $|\mathcal{D}|$

Figure 5: Comparisons of different approaches with various local database size $|\mathcal{D}|$.



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

Figure 6: Comparisons of different approaches with various result # limit 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 7: Comparisons of different approaches with various $|\Delta\mathcal{D}|$ size.

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.

7.2.3 *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 6(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 6(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 6(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.

7.2.4 *Increase of Bias*

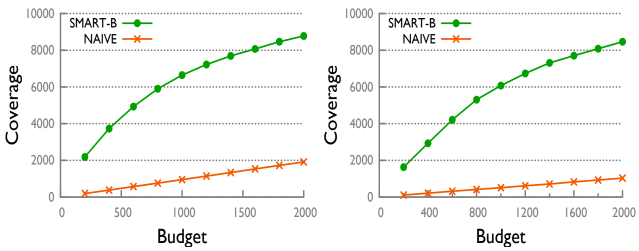
SmartCrawl-B is a biased estimator, where the bias depends on the size of $|\Delta\mathcal{D}|$. A larger $|\Delta\mathcal{D}|$ will increase the bias. In this section, we explore the impact of $|\Delta\mathcal{D}|$ on SmartCrawl-B. Figure 7(a), (b), (c) show the results when $|\Delta\mathcal{D}|$ is 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.

7.2.5 *Fuzzy Matching*

We compared SmartCrawl-B with NaiveCrawl in the fuzzy matching situation. Figure 9(a),(b) show the results for the cases when adding 5% and 50% data errors to local databases. As discussed in Section **??**, 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 (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.

7.3 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-



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

Figure 8: Comparisons of SmartCrawl-B and NaiveCrawl in the fuzzy matching situation.

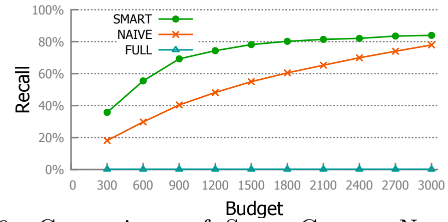


Figure 9: Comparisons of SmartCrawl, NaiveCrawl, and FullCrawl over the Yelp's hidden database.

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 9 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.

8. RELATED WORK

Deep Web. There are three lines of work about deep web related to our problem: deep web crawling [36, 31, 26, 38, 10, 28, 39, 34], deep web integration [15, 25, 45, 32], and deep web sampling [27, 11, 49, 20, 17, 18, 19, 48, 42].

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 [26, 10, 34] and form-like search interface [36, 31, 38, 28, 39]. Unlike these work, our goal is to have a good coverage of a local database rather than the underlying hidden database.

Deep web integration [15, 25, 45, 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.

Deep web sampling studies how to create a random sample of a hidden database using keyword-search interfaces [11, 49, 48] or form-like interfaces [42, 17, 17]. 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.

Data Enrichment. There are some works on data enrichment with web table [22, 23, 47, 46, 35, 13, 30], 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. Knowledge fusion studies how to enrich a knowledge base using Web content (e.g., Web Tables, Web pages) [21]. They assume that all the Web content have been crawled rather than study how to crawl a deep website progressively. There are also some works on entity set completion [40, 44, 41, 37, 50]. Unlike our work, they aim to enrich data with new rows (rather than new columns). We plan to extend SmartCrawl to support this scenario in the future.

Blocking Techniques in ER. 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 [16]. DeepEnrich 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.

9. CONCLUSION

This paper studied a novel problem called DeepEnrich. We proposed the SmartCrawl framework based on the ideas of query sharing and local-database-aware crawling. A key challenge is how to select the best query at each iteration. We started with a simple query selection algorithm called QSel-Simple, and found it ineffective because it ignored three key factors: local database coverage, top-k constraint, and fuzzy matching. We theoretically analyzed the negative impact of each factor, and proposed a new query selection algorithm called QSel-Est. We also discussed how to deal with inadequate sample size and how to implement QSel-Est efficiently. Our detailed experimental evaluation has shown that (1) QSel-Est is much more effective than QSel-Est in many situations; (2) the biased estimators are superior to the unbiased estimators; (3) SmartCrawl can significantly outperform NaiveCrawl and FullCrawl over both simulated and real hidden databases; (4) SmartCrawl is more robust than NaiveCrawl when facing the fuzzy-matching situation.

This paper has shown that it is a promising idea to leverage deep web for data enrichment. 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 a sample in runtime 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, we want to study how to crawl a hidden database for other purpose such as data cleaning and row population.

10. REFERENCES

[1] GoodReads. `https://www.goodreads.com/`.

[2] Google Maps API. `https://developers.google.com/maps/documentation/geocoding/usage-limits`.

[3] IMDb. `http://www.imdb.com/`.

[4] OpenRefine Reconciliation Service. `https://github.com/OpenRefine/OpenRefine/wiki/Reconciliation-Service-API`. Accessed: 2018-01-15.

[5] P. Wang, R. Shea, J. Wang and E. Wu. Technical Report: Progressive Deep Web Crawling Through Keyword Queries For Data Enrichment. `http://deeper.sfucloud.ca/static/deeperweb/pdf/deeper%200.0.1.pdf`.

[6] SoundCloud. https://soundcloud.com/.
[7] The ACM Digital Library. https://dl.acm.org/.
[8] Yelp API. https://www.yelp.com/developers/faq.
[9] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.
[10] 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.
[11] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's index. *J. ACM*, 55(5):24:1–24:74, 2008.
[12] L. Barbosa and J. Freire. Siphoning hidden-web data through keyword-based interfaces. In *SBBD*, pages 309–321, 2004.
[13] M. J. Cafarella, A. Y. Halevy, and N. Khoussainova. Data integration for the relational web. *PVLDB*, 2(1):1090–1101, 2009.
[14] M. J. Cafarella, J. Madhavan, and A. Y. Halevy. Web-scale extraction of structured data. *SIGMOD Record*, 37(4):55–61, 2008.
[15] 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.
[16] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. Knowl. Data Eng.*, 24(9):1537–1555, 2012.
[17] 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.
[18] 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.
[19] 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.
[20] 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.
[21] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *ACM SIGKDD*, pages 601–610, 2014.
[22] 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.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] X. Jin, N. Zhang, and G. Das. Attribute domain discovery for hidden web databases. In *ACM SIGMOD*, pages 553–564, 2011.
[29] G. Z. Kantorski, V. P. Moreira, and C. A. Heuser. Automatic filling of hidden web forms: A survey. *SIGMOD Record*, 44(1):24–35, 2015.
[30] O. Lehmberg, D. Ritze, P. Ristoski, R. Meusel, H. Paulheim, and C. Bizer. The mannheim search join engine. *J. Web Sem.*, 35:159–166, 2015.
[31] 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.
[32] W. Meng, C. T. Yu, and K. Liu. Building efficient and effective metasearch engines. *ACM Comput. Surv.*, 34(1):48–89, 2002.
[33] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, 2007.
[34] A. Ntoulas, P. Zerfos, and J. Cho. Downloading textual hidden web content through keyword queries. In *JCDL*, pages 100–109, 2005.
[35] R. Pimplikar and S. Sarawagi. Answering table queries on the web using column keywords. *PVLDB*, 5(10):908–919, 2012.
[36] 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.
[37] A. D. Sarma, L. Fang, N. Gupta, A. Y. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu. Finding related tables. In *ACM SIGMOD*, pages 817–828, 2012.
[38] 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.
[39] 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.
[40] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *IEEE ICDE*, pages 673–684, 2013.
[41] C. Wang, K. Chakrabarti, Y. He, K. Ganjam, Z. Chen, and P. A. Bernstein. Concept expansion using web tables. In *WWW*, pages 1198–1208, 2015.
[42] 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.
[43] P. Wang, Y. He, R. Shea, J. Wang, and E. Wu. Deeper: A data enrichment system powered by deep web. In *2018 ACM SIGMOD Demo, June 10-15, 2018, Houston, Texas, USA.*, 2018.
[44] R. C. Wang and W. W. Cohen. Iterative set expansion of named entities using the web. In *IEEE ICDM*, pages 1091–1096, 2008.
[45] 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.
[46] 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.
[47] 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.
[48] 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.
[49] 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.
[50] S. Zhang and K. Balog. Entitables: Smart assistance for entity-focused tables. In *ACM SIGIR*, pages 255–264, 2017.

APPENDIX

APPENDIX

A. PROOFS

Proof of Lemma 3

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 1

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 $\mathcal{Q}$ 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 2

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.,

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

<u>*Basis:*</u> Obviously, the statement holds for $b \le |\Delta\mathcal{D}|$.

<u>*Inductive Step:*</u> 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 $\mathcal{Q}'_{\text{sel}}$. 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

$$q'_1 = \operatorname*{argmax}_{q\in\mathcal{Q}} |q(\mathcal{D})|.$$

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

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

Since $q_1 = \operatorname{argmax}_{q\in\mathcal{Q}} \mathsf{benefit}(q)$, 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 $\mathsf{benefit}(q'_1) \neq |q'_1(\mathcal{D})|$, since $b \ge |\Delta\mathcal{D}|$ and each $q' \in \mathcal{Q}'_{\text{sel}}$ can cover at most one uncovered local record in $\Delta\mathcal{D}$, there must exist $q' \in \mathcal{Q}'_{\text{sel}}$ 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

$$\mathcal{D}_i - \Delta\mathcal{D} = \mathcal{D} - \Delta\mathcal{D}. \tag{14}$$

Based on Equation 14, we can deduce that ,

$$|q(\mathcal{D}_i)_{\text{cover}}| = |q(\mathcal{D})_{\text{cover}}| \quad \text{for any } q \in \mathcal{Q}. \tag{15}$$

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

$$q'_i = \operatorname*{argmax}_{q\in\mathcal{Q}} |q(\mathcal{D}_i)|. \tag{16}$$

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

$$|q'_i(\mathcal{D}_i)| = |q'_i(\mathcal{D}_i)_{\text{cover}}|. \tag{17}$$

For any query $q \in \mathcal{Q}$, we have

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

By plugging Equations 17 and 18 into Equation 16, we obtain

$$q'_i = \operatorname*{argmax}_{q\in\mathcal{Q}} |q(\mathcal{D}_i)_{\text{cover}}|. \tag{19}$$

By plugging Equation 15 into Equation 19, we obtain

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

Since $q_1 = \operatorname{argmax}_{q\in\mathcal{Q}} \mathsf{benefit}(q)$, 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

$$N_1 < (1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}. \tag{20}$$

$$N_1 + N_2 = N_{\text{ideal}} \tag{21}$$

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

$$\frac{N_1}{b - \Delta\mathcal{D}} \geq \frac{N_2}{\Delta\mathcal{D}}. \tag{22}$$

By plugging Equation 21 into Equation 22, we obtain

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

Algebraically:

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

which contradicts Equation 20. 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 4

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

$$E\big[|q(\mathcal{D}) \cap q(\mathcal{H}_s)| \cdot \frac{k}{|q(\mathcal{H}_s)|}\big] = \frac{k}{|q(\mathcal{H}_s)|} \cdot E\big[|q(\mathcal{D}) \cap q(\mathcal{H}_s)|\big] \tag{23}$$

Based on Lemma 3, we obtain

$$E\big[|q(\mathcal{D}) \cap q(\mathcal{H}_s)|\big] = \theta |q(\mathcal{D}) \cap q(\mathcal{H})| \tag{24}$$

By plugging Equation 25 into Equation 23, we have that the expected value of our estimator is:

$$\frac{k\theta}{|q(\mathcal{H}_s)|} |q(\mathcal{D}) \cap q(\mathcal{H})| = \frac{k}{|q(\mathcal{H})|} |q(\mathcal{D}) \cap q(\mathcal{H})|, \tag{25}$$

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 7).

Proof of Lemma 5

The expected value of the estimator is

$$E\big[|q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|}\big] = k\theta \cdot |q(\mathcal{D})| \cdot \frac{1}{E[|q(\mathcal{H}_s)|]} \tag{26}$$

$$= k\theta \cdot |q(\mathcal{D})| \cdot \frac{1}{|q(\mathcal{H})|\theta} \tag{27}$$

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

Therefore, the bias of the estimator is:

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

Proof of Lemma 6

We first prove that Lemma 3 holds without Assumption **??**. 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 3, we have

$$\mathbf{E}\Big[\frac{q(\mathcal{D}) \cap q(\mathcal{H}'_s)}{\theta}\Big] = |q(\mathcal{D}) \cap q(\mathcal{H}')| \tag{30}$$

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

$$|q(\mathcal{D}) \cap q(\mathcal{H}')| = |q(\mathcal{D}) \widetilde{\cap} q(\mathcal{H})| \quad |q(\mathcal{D}) \cap q(\mathcal{H}'_s)| = |q(\mathcal{D}) \widetilde{\cap} q(\mathcal{H}_s)| \tag{31}$$

By plugging Equation 31 into Equation 30, 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 3 holds without Assumption **??**.

We can use a similar idea to prove that Lemma 4 holds without Assumption **??**

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 $\mathcal{O}\big(|\mathcal{D}||d| + |\mathcal{H}_s||h|\big)$; (2) create a forward index with the time complexity of $\mathcal{O}(|\mathcal{Q}||q(\mathcal{D})|)$; (3) create a priority queue with the time complexity of $\mathcal{O}(|\mathcal{Q}|\log(|\mathcal{Q}|))$; (4) compute the query frequency w.r.t. $\mathcal{D}$ and $\mathcal{H}_s$ with the time complexity of $\mathcal{O}\big(cost_q \cdot |\mathcal{Q}|\big)$, 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 $\mathcal{O}\big(b \cdot t \cdot \log|\mathcal{Q}|\big)$, 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 $\mathcal{O}\big(|\mathcal{D}||F(d)|\big)$, where $|F(d)|$ denotes the average number of queries that can cover d, which is much smaller than $|\mathcal{Q}|$.

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.

Algorithm 4: QSEL-EST Algorithm (Biased Estimators)

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

Build inverted indices I_1 and I_2 based on $\mathcal{D}$ and $\mathcal{H}_s$, respectively;
for each $q \in \mathcal{Q}$ **do**
  $|q(\mathcal{D})| = |\cap_{w \in q} I_1(w)|$; $|q(\mathcal{H}_s)| = |\cap_{w \in q} I_2(w)|$;
end
Initialize a forward index F, where $F(d) = \phi$ for each $d \in \mathcal{D}$;
for each $q \in \mathcal{Q}$ **do**
  for each $d \in q(\mathcal{D})$ **do**
    Add q into $F(d)$;
  end
end
Let P denote an empty priority queue;
for each $q \in \mathcal{Q}$ **do**
  if $\frac{|q(\mathcal{H}_s)|}{\theta} \leq k$ **then** P.push($\langle q, |q(\mathcal{D})| \rangle$) ;
  else P.push($\langle q, |q(\mathcal{D})| \cdot \frac{k\theta}{|q(\mathcal{H}_s)|} \rangle$) ;
end
Initialize a hash map U, where $U(q) = 0$ for each $q \in \mathcal{Q}$;
while $b > 0$ and $\mathcal{D} \neq \phi$ **do**
  $\langle q^*, \text{old\_priority} \rangle$ = P.pop();
  if $|U(q^*)|\ != \ 0$ **then**
    if $\frac{|q^*(\mathcal{H}_s)|}{\theta} \leq k$ **then**
      new_priority $= |q^*(\mathcal{D})| - |U(q^*)|$
    else
      new_priority $= \big(|q^*(\mathcal{D})| - |U(q^*)|\big) \cdot \frac{k\theta}{|q^*(\mathcal{H}_s)|}$
    end
    P.push($\langle q, \text{new\_priority} \rangle$); $|U(q^*)| = 0$;
    continue;
  end
  Issue q^* to the hidden database, and then get the result $q^*(\mathcal{H})_k$;
  if q^* is a solid query **then** $\mathcal{D}_{\text{removed}} = q^*(\mathcal{D})$;
  else $\mathcal{D}_{\text{removed}} = q^*(\mathcal{D})_{cover}$;
  for each $d \in \mathcal{D}_{removed}$ **do**
    for each $q \in F(d)$ **do**
      $U(q) += 1$;
    end
  end
  $\mathcal{D} = \mathcal{D} - \mathcal{D}_{\text{removed}}$; $\mathcal{Q} = \mathcal{Q} - \{q\}$; $b = b - 1$;
end

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.