Progressive Deep Web Crawling Through Keyword Queries For Data Enrichment

Pei Wang
Simon Fraser University
peiw@sfu.ca

Ryan Shea
Simon Fraser University
rws1@sfu.ca

Jiannan Wang
Simon Fraser University
jnwang@sfu.ca

Eugene Wu
Columbia University
ew2493@columbia.edu

ABSTRACT

Data enrichment is the act of extending a local database with new attributes from external data sources. In this paper, we study a novel problem—how to progressively crawl the deep web (i.e., a hidden database) through a keyword-search API to enrich a local database in an effective way. This is challenging because these interfaces often limit the data access by enforcing the top-k constraint or limiting the number of queries that can be issued within a time window. In response, we propose SmartCrawl, a new framework to collect results effectively. Given a query budget b, SmartCrawl first constructs a query pool based on the local database, and then iteratively issues a set of most beneficial queries to the hidden database such that the union of the query results can cover the maximum number of local records. The key technical challenge is how to estimate query benefit, i.e., the number of local records that can be covered by a given query. A simple approach is to estimate it as the query frequency in the local database. We find that this is ineffective due to i) the impact of $|\Delta\mathcal{D}|$, where $|\Delta\mathcal{D}|$ represents the number of local records that cannot be found in the hidden database, and ii) the top-k constraint enforced by the hidden database. We study how to mitigate the negative impacts of the two factors and propose effective optimization techniques to improve performance. The experimental results show that on both simulated and real-world hidden databases, SmartCrawl significantly increases coverage over the local database as compared to the baselines.

1 INTRODUCTION

Data scientists spend more than 80% of their time on data preparation [2]—the process of turning raw data into a form suitable for subsequent analysis. This process involves many tasks such as data collection, data cleaning, and data enrichment. In this paper, we focus on data enrichment, the act of extending a local database with new attributes extracted from external data sources. As a simple example, a data scientist collected a list of newly opened restaurants and she wants to know the category and the rating of each restaurant. Data enrichment can uncover new insights about the data. A natural use case of data enrichment is to augment a training set with new features. Enriched data can also be used for error detection [11].

Data enrichment is not a new research topic. Existing work mainly focuses on the use of *web tables* (i.e., HTML tables) to enrich data via table augmentation [14, 22, 23, 34, 45, 46]. However, data scientists often do not have the Web Tables corpus downloaded. They just have a website in mind for data enrichment. The challenge is that the data in the website is hidden behind a restrictive query interface. This is often called the *deep web*.

An important class of deep websites is those hidden behind keyword search interfaces. For instance, IMDb [5], SoundCloud [7], ACM Digital Library [9], GoodReads [3], DBLP [1], Spotify [8], along with a multitude of modern websites, expose paginated data records through a keyword search interface. The results of the search API, including the above examples, are commonly based on conjunctive keyword search, where each result contains all the keywords. Abstractly, search APIs to these hidden databases take a set of keywords as input, identify results that contain all the keywords, and return the top-k records using an *unknown* ranking function.

The challenge is that the data scientist wants to enrich her local data quickly. However search APIs can be rate limited, or simply have a cap on the number of search calls that can be issued. For example, the Yelp API is restricted to 25,000 free requests per day [10] and the Google Maps API only allows 2,500 free requests per day [4]. Thus, it is important to judiciously choose a specific set of queries so that once issued, the returned hidden records can *cover* the most records in the user's local database.

We call this problem CrawlEnrich. Let a local record be *covered* by a query if and only if the query result contains a hidden record that refers to the same real-world entity as the local record[1]. Given a local database $\mathcal{D}$, a hidden database $\mathcal{H}$ that provides a search API, a fixed query budget b of the number of API calls, the goal of CrawlEnrich is to issue a set of b queries to $\mathcal{H}$ such that the union of the query results can *cover* as many records in $\mathcal{D}$ as possible.

[1]To focus on the algorithmic problem, we assume that, if a hidden record in the query result refers to the same real-world entity as the local record, it is possible to identify the match by using existing entity resolution techniques.

We find that even simple variants of this problem, as described in this paper, is challenging. The obvious approach, which we call NaiveCrawl, is to generate a search query for each record in the local database $\mathcal{D}$. This maximizes the likelihood that every matching record in the hidden database will *eventually* returned, and is used by tools such as OpenRefine [6] to crawl data. However, the number of queries increases with the size of the local database. Further, these queries may not be robust to data errors. For instance, if the query for the restaurant "Lotus of Siam" is incorrectly issued as "Lotus of Siam 12345", then the search query will likely not return the matching record.

The challenge of the naive approach is two-fold. First, queries should not be overly precise, meaning that a search query should not be generated for only a single local record. This wastes API calls that could have covered more local records. On the other hand, queries should not be overly general by *trying* to cover too many local records, because the covering hidden records may not be in the top-k results of an overly general query.

The fundamental issue is *query-benefit estimation*: how can we estimate the expected number of local records that will be covered by a given search query? This expected number is called *query benefit*. Once query benefits are estimated, the problem is reduced to a well-known Maximum Coverage problem, where a greedy algorithm can give both good theoretical and empirical performance [32].

To this end, we develop SmartCrawl, a framework that iteratively selects queries to maximally cover the local database. It first constructs a query pool from $\mathcal{D}$, and then iteratively selects the query with the largest estimated benefit from the pool, issues it to $\mathcal{H}$ until the budget b is exhausted. The key technical challenge is the benefit estimation.

We start with a simple approach, called QSel-Simple. This approach uses *query frequency w.r.t. the local database* $\mathcal{D}$ as an estimation of query benefit, where the 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", QSel-Simple will estimate the query benefit as 100. We analytically compare QSel-Simple with the ideal approach (called QSel-Ideal), which selects queries based on *true* query benefits. Obviously, QSel-Ideal is hypothetical and cannot be achieved in reality. The purpose of the comparison aims to investigate that *Under which conditions are QSel-Simple and QSel-Ideal equivalent? What factors may lead QSel-Simple to perform worse than QSel-Ideal? For those factors, are there ways to improve QSel-Simple's performance?* Answering these questions not only provide insights on QSel-Simple's performance, but also guides us to develop a set of optimization techniques.

We identify two factors that may significantly affect QSel-Simple's performance, and we prove that QSel-Simple and QSel-Ideal are equivalent under certain assumptions. Furthermore, we discuss effective optimizations for QSel-Simple when breaking each assumption.

Factor 1: Impact of $|\Delta\mathcal{D}|$. The first factor is whether $\mathcal{D}$ can be *fully* covered by $\mathcal{H}$ or not. If not, we denote $\Delta\mathcal{D} = \mathcal{D} - \mathcal{H}$ as the set of the records in $\mathcal{D}$ that cannot be found in $\mathcal{H}$. Recall that in the previous example, QSel-Simple sets the benefit of q = "Noodle House" to 100. However, if all the 100 records cannot be found in $\mathcal{H}$ (i.e., they are in $\Delta\mathcal{D}$), there will be no benefit to issue the query. Therefore, we need to use the *query frequency w.r.t.* $\mathcal{D} - \Delta\mathcal{D}$ rather than w.r.t. $\mathcal{D}$ to estimate query benefit. For this reason, we first study how to bound the performance gap between QSel-Simple and QSel-Ideal, where the former uses query frequency w.r.t. $\mathcal{D}$ and the latter 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 $|\Delta\mathcal{D}|$.

Factor 2: Top-k Constraint. The second factor is whether the selected queries will be affected by the top-k constraint or not. We say a query will be affected by the top-k constraint if it can match more than k records in the hidden database. In this case, the hidden database will sort these matched records according to an unknown ranking function and only return the top-k records as a query result. Intuitively, QSel-Simple tends to select very frequent keywords (e.g., "Restaurant") as a query, but this kind of query is also more likely to be affected by the top-k constraint. For this example, it may return many unrelated restaurants to the local database. To solve this problem, our main idea is to leverage deep web sampling [13, 17–20, 27, 41, 47, 48]. We first create a sample of the hidden database offline [47], then apply the sample to predict which queries will be affected by the top-k constraint, and finally develop new estimators for the affected queries. Note that the sample only needs to be created once and can be reused by any user who wants to enrich their local database with the hidden database.

We call the new query selection approach QSel-Est, which equipped with the above optimization techniques. In the experiments, we find that, when compared to QSel-Simple, QSel-Est improves SmartCrawl across a wide range of experimental settings and local databases. SmartCrawl also outperforms NaiveCrawl by up to 7×, and is more robust to data errors in the local database.

To summarize, our main contributions are:

- To the best of our knowledge, we are the first to study the CrawlEnrich problem. We formalize the problem and propose SmartCrawl to solve the problem.
- We present a simple query selection strategy called QSel-Simple and identify two factors (i.e., $\Delta\mathcal{D}$ and the top-k constraint) that may affect its performance.
- We analyze how $|\Delta\mathcal{D}|$ has a negative impact on the performance of QSel-Simple, and propose effective techniques to mitigate the negative impact.

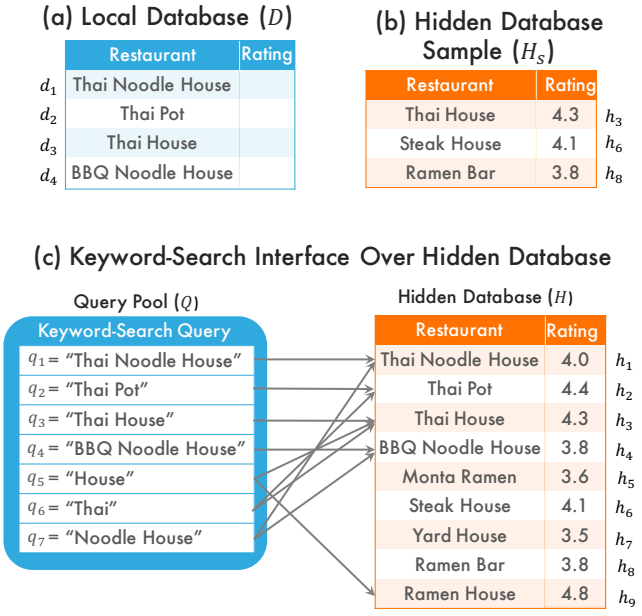


Figure 1: A running example ($k = 2$, $\theta = \frac{1}{3}$). There are four record pairs (i.e., $\langle d_1, h_1\rangle$, $\langle d_1, h_1\rangle$, $\langle d_3, h_3\rangle$, and $\langle d_4, h_4\rangle$) that refer to the same real-world entity. Each arrow points from a query to its result. (Please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

- We study how to break the top-k constraint using a deep web sample and develop novel estimators to further improve QSel-Simple's performance.
- We conduct extensive experiments over simulated and real hidden databases. The results show that SmartCrawl outperforms baselines by up to 7× and is more robust to data errors.

The remaining of the paper is organized as follows. We formalize the problem in Section 2, and present the SmartCrawl framework in Section 3. Section 4 and Section 5 discuss the impact of $|\Delta\mathcal{D}|$ and the top-k constraint, respectively. The experimental results are shown in Section 6. We review the related work in Section 7 and conclude the paper in Section 8.

2 PROBLEM FORMALIZATION

In this section, we formulate the CrawlEnrich 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.

Table 1: Illustration of notations for the running example. (Please ignore $q(\mathcal{H}_s)$ for now.)

	Known before issuing q		**Unknown** before issuing q		
Q	$q(\mathcal{D})$	$q(\mathcal{H}_s)$	$q(\mathcal{H})$	$q(\mathcal{H})_k$	$q(\mathcal{D})_{\textbf{cover}}$
q_1	$\{d_1\}$	ϕ	$\{h_1\}$	$\{h_1\}$	$\{d_1\}$
q_2	$\{d_2\}$	ϕ	$\{h_2\}$	$\{h_2\}$	$\{d_2\}$
q_3	$\{d_3\}$	$\{h_3\}$	$\{h_3\}$	$\{h_3\}$	$\{d_3\}$
q_4	$\{d_4\}$	ϕ	$\{h_4\}$	$\{h_4\}$	$\{d_4\}$
q_5	$\{d_1, d_3, d_4\}$	$\{h_3, h_6\}$	$\{h_1, h_3, h_4$ $h_6, h_7, h_9\}$	$\{h_3, h_9\}$	$\{d_3\}$
q_6	$\{d_1, d_2, d_3\}$	$\{h_3\}$	$\{h_1, h_2, h_3\}$	$\{h_2, h_3\}$	$\{d_2, d_3\}$
q_7	$\{d_1, d_4\}$	ϕ	$\{h_1, h_4\}$	$\{h_1, h_4\}$	$\{d_1, d_4\}$

This paper focuses on the crawling part. A full end-to-end data enrichment system would need additional functionalities such as schema matching (i.e., match the 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). However, they can be treated as an orthogonal issue. We have discussed how to apply existing schema-matching and entity-resolution techniques to build an end-to-end data enrichment system in the demo paper [42]. 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[2]. We define the intersection between $\mathcal{D}$ and $\mathcal{H}$ as

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

match(d, h) returns True if d and h refer to the same real-world entity; otherwise, match(d, h) returns False. This intersection contains all the local records that can be covered by $\mathcal{H}$. Note that $\mathcal{D}$ may not be 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 is to select a set Q_{sel} of queries within the budget such that $|\bigcup_{q \in Q_{\text{sel}}} q(\mathcal{D})_{\text{cover}}|$ is maximized.

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

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

Unfortunately, CrawlEnrich is an NP-Hard problem, which can be proved by a reduction from the maximum-coverage problem (a variant of the set-cover problem) [32]. In fact, what makes this problem exceptionally challenging is that

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

the greedy algorithm that can be used to solve the maximum-coverage problem is not applicable (see the "chicken-and-egg" dilemma in Section 3.2).

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.

Keyword-search Interface. A keyword search interface accepts a query and returns the top-k of the matched hidden records to the user. We say a query *overflows* when the actual matched record number is larger than k and only the top-k records are exposed to the user; on the other hand, a query is a *solid* one if the matched record number is smaller than or equal to k, i.e., all the matched records are returned.

For the keyword matching rule, we investigated a number of deep websites to understand the keyword-search interface in real-world scenarios. We find that most of them (e.g., IMDb, DBLP, ACM Digital Library, GoodReads, Spotify, and SoundCloud) adopt 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). Thus we assume the keyword-search interface is a conjunctive one. In the experiments, we found that our approach also performed well without the assumption. This is because that even if a keyword-search interface violates this assumption, it tends to rank the records that contain all the query keywords to the top.

Definition 2.1 (Conjunctive Keyword Search). Each record is modeled as a document, denoted by document$(\cdot)$, which concatenates all[3] the attributes of the record. Given a query, we say a record h (resp. d) satisfies the query if and only if document(h) (resp. document(d)) contains all *the keywords in the query.*

Let $q(\mathcal{H})$ ($q(\mathcal{D})$) 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 whenever it is executed. Definition 2.2 formally defines the keyword-search interface.

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

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

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

[3]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.

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

Example 2.3. Figure 1 shows an example. Figure 1(a) represents a local database. Figure 1(c) represents a hidden database and the correspondence (grey line) between queries and returned top-k records ($k = 2$).

Consider q_5 = "House" in Table 1. Since d_1, d_3, d_4 contain "House", we have $q_5(\mathcal{D}) = \{d_1, d_3, d_4\}$. Since $h_1, h_3, h_4, h_6, h_7, h_9$ contain "House", we have $q_5(\mathcal{H}) = \{h_1, h_3, h_4, h_6, h_7, h_9\}$. Note that $k = 2$. As shown in Figure 1(c), only h_3, h_9 are returned for q_5, thus $q_5(\mathcal{H})_k = \{h_3, h_9\}$. We can see that $q_5(\mathcal{H})_k$ covers one local record d_3. Therefore, $q_5(\mathcal{D})_{\text{cover}} = \{d_3\}$.

Suppose $b = 2$. We aim to select two queries q_i, q_j from $\{q_1, q_2, \cdots, q_7\}$ in order to maximize $|q_i(\mathcal{D})_{\text{cover}} \cup q_j(\mathcal{D})_{\text{cover}}|$. We can see that the optimal solution should select q_6 and q_7 since $|q_6(\mathcal{D})_{\text{cover}} \cup q_7(\mathcal{D})_{\text{cover}}| = 4$ reaches the maximum. 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

We propose the SmartCrawl framework to solve the CrawlEnrich problem. The framework has two stages: i) *Query Pool Generation* initializes a *query pool* by extracting keyword queries from $\mathcal{D}$; ii) *Query Selection* iteratively selects the *most beneficial* query to maximize local database coverage until the budget is exhausted.

3.1 Query Pool Generation

Let Q denote a query pool. If a query q does not appear in any local record, i.e., $|q(\mathcal{D})| = 0$, we do not consider the query. Given $\mathcal{D}$, 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 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 to cover the record. Let Q_{naive} denote the collection of the queries generated in this step. We have $|Q_{naive}| = |\mathcal{D}|$.
- To satisfy the second principle, SmartCrawl finds the queries such that $|q(\mathcal{D})| \ge t$. 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:

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

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

Example 3.1. The seven queries, $\{q_1, q_2, \cdots, q_7\}$, in Figure 1(c) are generated using the method above. Suppose $t = 2$. Based on the first principle, we generate $\mathcal{Q}_{naive} = \{q_1, q_2, q_3, q_4\}$, 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".

Threshold Selection. Here, t is a threshold, which balances the trade-off between the number of generated queries and the time spend in generating the queries. A user needs to generate a set of at least b queries, but the upper bound depends on the user's time constraint. In the experiment, we set t to a value so that our frequent pattern mining algorithm will generate around $5b$ queries. We empirically find when using this threshold, the performance is close to the performance when generating the queries in an exhaustive way, and save the query generation efforts at the mean time.

3.2 Query Selection

After the query pool is generated, SmartCrawl enters the query-selection stage. Section 1 briefly introduces three approaches: QSel-Ideal, QSel-Simple, and QSel-Est. They start with the same query pool but use different query selection strategies.

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 removes the matched records from $\mathcal{D}$ and updates $|q(\mathcal{D})_{cover}|$ for each q accordingly, and goes to the next iteration.

Example 3.2. Suppose QSel-Ideal needs to select $b = 2$ queries. Consider $q(\mathcal{D})_{\text{cover}}$ in Table 1. We can see $|q_1(\mathcal{D})_{\text{cover}}| = 1, |q_2(\mathcal{D})_{\text{cover}}| = 1, |q_3(\mathcal{D})_{\text{cover}}| = 1, |q_4(\mathcal{D})_{\text{cover}}| = 1, |q_5(\mathcal{D})_{\text{cover}}| = 1, |q_6(\mathcal{D})_{\text{cover}}| = 2, |q_7(\mathcal{D})_{\text{cover}}| = 2$. At the first iteration, both q_6 and q_7 has the maximum $|q(\mathcal{D})_{\text{cover}}|$. Suppose we break the tie by choosing the smallest query id. Thus, q_6 will be selected. After issuing the query q_6, we can cover d_2 and d_3. QSel-Ideal removes d_2 and d_3 from $\mathcal{D}$, and then obtains $|q_1(\mathcal{D})_{\text{cover}}| = 1, |q_2(\mathcal{D})_{\text{cover}}| = 0, |q_3(\mathcal{D})_{\text{cover}}| = 0, |q_4(\mathcal{D})_{\text{cover}}| = 1, |q_5(\mathcal{D})_{\text{cover}}| = 0, |q_7(\mathcal{D})_{\text{cover}}| = 2$. At the second iteration, q_7 has the maximum $|q(\mathcal{D})_{\text{cover}}|$, thus it will be selected. After issuing the query q_7, we can cover d_1 and d_4. Now, the budget is exhausted and QSel-Ideal terminates. In the end, QSel-Ideal selects q_6 and q_7, which covers d_1, d_2, d_3, d_4.

Algorithm 1: QSel-Ideal Algorithm

Input: $\mathcal{Q}, \mathcal{D}, \mathcal{H}, b$
Result: Iteratively select the query with the largest benefit.
1 **while** $b > 0$ and $\mathcal{D} \neq \phi$ **do**
2 **for** each $q \in \mathcal{Q}$ **do**
3 benefit(q) = $|q(\mathcal{D})_{cover}|$;
4 **end**
5 Select q^* with the largest benefit from $\mathcal{Q}$;
6 Issue q^* to the hidden DB, and then get the result $q^*(\mathcal{H})_k$;
7 $\mathcal{D} = \mathcal{D} - q^*(\mathcal{D})_{\text{cover}}$; $\mathcal{Q} = \mathcal{Q} - \{q^*\}$; $b = b - 1$;
8 **end**

Algorithm 2: QSel-Simple Algorithm

1 Replace Line 3 in Algorithm 1 with the following lines:
2 benefit(q) = $|q(\mathcal{D})|$;

Chicken-and-Egg Dilemma. In reality, however, QSel-Ideal suffers from a "chicken and egg" dilemma. 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 to issue. To overcome the dilemma, we use the estimated benefits 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 of QSel-Simple. We can see that QSel-Simple differs from QSel-Ideal only in the benefit calculation part. Intuitively, QSel-Simple tends to select high-frequent keyword queries.

Example 3.3. Suppose QSel-Simple needs to select $b = 2$ queries. Consider $q(\mathcal{D})$ in Table 1. We can see $|q_1(\mathcal{D})| = 1, |q_2(\mathcal{D})| = 1, |q_3(\mathcal{D})| = 1, |q_4(\mathcal{D})| = 1, |q_5(\mathcal{D})| = 3, |q_6(\mathcal{D})| = 3, |q_7(\mathcal{D})| = 2$. At the first iteration, both q_5 and q_6 has the maximum $|q(\mathcal{D})|$. We select q_5 since it has a smaller query id. After issuing the query q_5, we can cover $q_5(\mathcal{D})_{\text{cover}} = \{d_3\}$. QSel-Simple removes d_3 from $\mathcal{D}$, and then obtains $|q_1(\mathcal{D})| = 1, |q_2(\mathcal{D})| = 1, |q_3(\mathcal{D})| = 0, |q_4(\mathcal{D})| = 1, |q_6(\mathcal{D})| = 2, |q_7(\mathcal{D})| = 2$. At the second iteration, both q_6 and q_7 has the maximum $|q(\mathcal{D})|$. We select q_6 since it has a smaller query id. After issuing the query q_6, we can cover d_2 and d_3. Now, the budget is exhausted and QSel-Simple terminates. In the end, QSel-Simple selects q_5 and q_6, which covers d_2, d_3.

QSel-Ideal vs. QSel-Simple. As discussed in the introduction, the QSel-Simple's performance may be affected by

two factors, the impact of $|\Delta\mathcal{D}|$ and the top-k constraint. We prove that if $\mathcal{D}$ can be fully covered by $\mathcal{H}$ (Assumption 1) and $\mathcal{H}$ does not enforce the top-k constraint (Assumption 2), then QSel-Simple and QSel-Ideal are equivalent (i.e., they select the same set of queries). The formal proof can be found in Lemma 3.4.

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 $d = h$.

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 $q(\mathcal{H})_k = q(\mathcal{H})$.

Lemma 3.4. *If Assumptions 1 and 2 hold, then* QSel-Ideal *and* QSel-Simple *are equivalent.*

Proof. All the proofs can be found in the appendix. □

In our running example (Figure 1), Assumption 1 holds since all the four restaurants in the local database can be found in the hidden database, but Assumption 2 does not hold since only top-2 records are returned for each query. Due to the violation of Assumption 2, QSel-Ideal and QSel-Simple are *not* equivalent. Therefore, as shown in Examples 3.2 and 3.3, they select a different set of queries, respectively.

So far, we have not seen how the violation of Assumption 1 could have a negative impact on the QSel-Simple's performance. We will answer this question in Section 4 and also propose an effective technique to mitigate its negative impact. To further reduce the performance gap between QSel-Ideal and QSel-Simple, we relax both assumptions in Section 5, and propose novel benefit estimation approaches to handle the general situation.

4 IMPACT OF $|\Delta\mathcal{D}|$

In this section, we assume that Assumptions 2 holds, i.e., no top-k constraint, but Assumption 1 does not, i.e., $\Delta\mathcal{D} \neq 0$. We want to explore how $|\Delta\mathcal{D}|$ will affect the performance gap between QSel-Simple and QSel-Ideal. For example, suppose $|\mathcal{D}| = 10,000$ and $|\Delta\mathcal{D}| = 10$. How big the performance gap (between QSel-Simple and QSel-Ideal) can be? Is it likely that QSel-Ideal covers a much larger number of records than QSel-Simple? We first answer these questions in Section 4.1, and then propose an effective technique to mitigate the negative impact of $|\Delta\mathcal{D}|$ in Section 4.2.

4.1 Understand the Impact of $|\Delta\mathcal{D}|$

Rather than *directly* reason about the performance gap between QSel-Ideal and QSel-Simple, 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. Intuitively, QSel-Bound does not want an incorrectly selected query to affect the benefits of the following queries. Thus, if it finds a query incorrectly selected (i.e., $q^*(\mathcal{D}) \neq q^*(\mathcal{D})_{\text{cover}}$), it will put it back to the query pool.

Algorithm 3: QSel-Bound Algorithm

Input: $\mathcal{Q}$, $\mathcal{D}$, $\mathcal{H}$, b
Result: SmartCrawl$_b$ covers at least $(1 - \frac{|\Delta\mathcal{D}|}{b}) \cdot N_{\text{ideal}}$ 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
```

Table 2: We make two changes to the running example in Figure 1 tailored for Section 4. i) We remove h_4 from the hidden database. With this change, the local database cannot be fully covered by $\mathcal{H}$ (since d_4 cannot be found in $\mathcal{H}$). ii) We remove the top-k constraint. With this change, $q(\mathcal{H})$ and $q(\mathcal{H})_k$ have no difference.

	Known before issuing q	**Unknown** before issuing q	
$\mathcal{Q}$	$q(\mathcal{D})$	$q(\mathcal{H}) = q(\mathcal{H})_k$	$q(\mathcal{D})_{\textbf{cover}}$
q_1	$\{d_1\}$	$\{h_1\}$	$\{d_1\}$
q_2	$\{d_2\}$	$\{h_2\}$	$\{d_2\}$
q_3	$\{d_3\}$	$\{h_3\}$	$\{d_3\}$
q_4	$\{d_4\}$	ϕ	ϕ
q_5	$\{d_1, d_3, d_4\}$	$\{h_1, h_3, h_6, h_7, h_9\}$	$\{d_1, d_3\}$
q_6	$\{d_1, d_2, d_3\}$	$\{h_2, h_3\}$	$\{d_2, d_3\}$
q_7	$\{d_1, d_4\}$	$\{h_1\}$	$\{d_1\}$

Example 4.1. Consider the running example in Figure 1. Since Section 4 focuses on when $\mathcal{D}$ cannot be fully covered by $\mathcal{H}$, we make two changes to the example (see Table 2).

Consider $q(\mathcal{D})$ in Table 2. We can see that $|q_1(\mathcal{D})| = 1$, $|q_2(\mathcal{D})| = 1$, $|q_3(\mathcal{D})| = 1$, $|q_4(\mathcal{D})| = 1$, $|q_5(\mathcal{D})| = 3$, $|q_6(\mathcal{D})| = 3$, $|q_7(\mathcal{D})| = 2$. We will illustrate how QSel-Bound works at the first iteration. The remaining iterations are similar. At the first iteration, since $|q_5(\mathcal{D})| = 3$ has the maximum benefit, QSel-Bound selects q_5 and issues it to the hidden database. The returned result is $q_5(\mathcal{H})_k =$

$\{h_1, h_3, h_6, h_7, h_9\}$, which covers $|q_5(\mathcal{D})_{\text{cover}}| = \{d_2, d_3\}$. However, since $q_5(\mathcal{D}) \neq q_5(\mathcal{D})_{\text{cover}}$, then the query belongs to the second situation (see Line 9-10 in Algorithm 3). We compute $q_5(\Delta\mathcal{D}) = q_5(\mathcal{D}) - q_5(\mathcal{D})_{\text{cover}} = \{d_4\}$, and then set $\mathcal{D}$ to $\mathcal{D} - q_5(\Delta\mathcal{D}) = \{d_1, d_2, d_3\}$. Since d_4 has been removed from $\mathcal{D}$, we obtain $|q_1(\mathcal{D})| = 1$, $|q_2(\mathcal{D})| = 1$, $|q_3(\mathcal{D})| = 1$, $|q_4(\mathcal{D})| = 0$, $|q_5(\mathcal{D})| = 2$, $|q_6(\mathcal{D})| = 3$, $|q_7(\mathcal{D})| = 1$. After that, QSel-Bound starts the second iteration. Note that at the first iteration, q_5 is *not* removed from the query pool. Thus, q_5 may be selected again in the remaining iterations.

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.,

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

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

Lemma 4.2. 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 Sketch. The proof consists of two parts. In the first part, we prove that the first $\left(b - |\Delta\mathcal{D}|\right)$ 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 $\left(b - |\Delta\mathcal{D}|\right)$ 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 good as QSel-Ideal. For example, consider a local database having $|\Delta\mathcal{D}| = 10$ records not in the 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 both QSel-Simple and QSel-Bound are applicable in practice since they select queries based on $|q(\mathcal{D})|$, 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 10 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 $|\Delta\mathcal{D}|$

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 study how to mitigate the negative impact of $|\Delta\mathcal{D}|$.

We aim to find the local records in $\Delta\mathcal{D}$ and then remove them from $\mathcal{D}$. In other words, we want to identify which local record cannot be covered by $\mathcal{H}$. We first use a toy example to illustrate our key insight, and then present our technique in detail.

Consider a local database with a list of restaurant names (e.g., "Thai Pot", "Thai House") and a list of conference names (e.g., "SIGMOD 2019", "SIGMOD 2018"). Suppose the user wants to enrich the local database with a hidden restaurant database (e.g., Yelp). Here, all the conference names are in $\Delta\mathcal{D}$ since they do not match any hidden record. Suppose "SIGMOD" is selected and issued to the hidden database, and the query result is empty. It indicates that Yelp has no hidden record that contains "SIGMOD". Therefore, all the local records that contain "SIGMOD" cannot be covered by Yelp. We can safely remove them from $\mathcal{D}$. This optimization technique will help us avoid selecting many worthless queries (such as "SIGMOD 2019" and "SIGMOD 2018") in future iterations.

The above example shows a special case of our technique when the query result is empty. In the general situation, our technique 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) predict that the local records in $q(\mathcal{D}) - q(\mathcal{D})_{cover}$ cannot be covered by $\mathcal{H}$. The correctness can be proved 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 also satisfies q, thus h will be retrieved by q. Therefore, we can deduce that $d \in q(\mathcal{D})_{cover}$ should hold, which contradicts that $d \in q(\mathcal{D}) - q(\mathcal{D})_{cover}$.

Example 4.3. Consider the example in Table 2. Suppose q_5 is selected. (1) Issue q_5 to the hidden database and get the query result $q_5(\mathcal{H}) = \{h_1, h_3, h_6, h_7, h_9\}$, (2) use $q_5(\mathcal{H})$ to cover $\mathcal{D}$ and obtain $q_5(\mathcal{D})_{cover} = \{d_1, d_3\}$, and (3) compute $q_5(\mathcal{D}) - q_5(\mathcal{D})_{cover} = \{d_4\}$ and predict that d_4 cannot be covered by $\mathcal{H}$. Thus, we remove d_4 from $\mathcal{D}$. Now, $\mathcal{D}$ has only d_1 left. Note that without this optimization technique, $\mathcal{D}$ has both d_1 and d_4 left, where the existence of d_4 will have a negative impact on the estimated benefits of future queries.

5 TOP-K CONSTRAINT

In this section, we study how to further improve QSel-Simple by breaking the top-k constraint. Our key idea is to leverage a hidden database sample to estimate query benefits. Recall that there are two types of queries: solid query and overflowing query. 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 query benefits for solid queries in Section 5.2 and for overflowing

Table 3: Summary of query-benefit estimators.

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

queries in Section 5.3, respectively. Table 3 summarizes the proposed estimators and Table 4 illustrates how they work for the running example.

5.1 Query Type Prediction

Sampling from a hidden database is a well-studied topic in the Deep Web literature [13, 17–20, 27, 41, 47, 48]. 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 [13, 47, 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 [47] in the experiments and evaluate the performance of SmartCrawl using the sample created by the technique (Section 6.3).

Given a query q, we aim to predict whether q is solid or overflowing before issuing it. In other words, the goal is to predict whether $|q(\mathcal{H})|$ is greater than k or not. Since $\mathcal{H}_s$ is a random sample of $\mathcal{H}$, then we have $|q(\mathcal{H})| \approx \frac{|q(\mathcal{H}_s)|}{\theta}$. Thus, if $\frac{|q(\mathcal{H}_s)|}{\theta} \leq k$, q will be predicated as a solid query; otherwise, it will be predicated as an overflowing query.

Example 5.1. Consider the running example in Figure 1. Figure (b) shows a hidden database sample $\mathcal{H}_s$ with the sampling ratio of $\theta = \frac{1}{3}$. Table 1 shows $q(\mathcal{H}_s)$ for each query. Suppose $k = 2$.

For q_1 = "Thai Noodle House", since $\frac{|q(\mathcal{H}_s)|}{\theta} = \frac{0}{1/3} \leq k$, it is predicated as a solid query. This is a correct prediction. For q_5 = "House", since $\frac{|q(\mathcal{H}_s)|}{\theta} = \frac{2}{1/3} = 6 > k$, it is predicated as an overflowing query. This 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 study how to estimate query benefits for solid queries. We first propose an unbiased estimator, i.e., in expectation the estimator's estimated query benefit is equal to the true query benefit.

Unbiased Estimator. The query's true benefit is defined as:

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

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

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

Table 4: An illustration of query-benefit estimators for the running example ($q(\mathcal{D})$ and $q(\mathcal{H}_s)$ are copied from Table 1). According to Example 5.1, q_1, q_2, q_4, q_7 are predicted as solid queries; q_3, q_5, q_6 are predicted as overflowing queries. *Unbiased, Biased, and True* represent true benefit, unbiased estimator's estimated benefit, and biased estimator's estimated benefit, respectively.

Q	$q(\mathcal{D})$	$q(\mathcal{H}_s)$	Unbiased	Biased	True
q_1	$\{d_1\}$	ϕ	0	1	1
q_2	$\{d_2\}$	ϕ	0	1	1
q_4	$\{d_4\}$	ϕ	0	1	1
q_7	$\{d_1, d_4\}$	ϕ	0	2	2
q_3	$\{d_3\}$	$\{h_3\}$	2	$\frac{2}{3}$	1
q_5	$\{d_1, d_3, d_4\}$	$\{h_3, h_6\}$	1	1	1
q_6	$\{d_1, d_2, d_3\}$	$\{h_3\}$	2	2	2

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

```
SELECT d, h FROM D, H
WHERE d = h AND d satisfies q.
```

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

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

We prove the estimator is unbiased in Lemma 5.2.

Lemma 5.2. Given a solid query q, then $\frac{|q(\mathcal{D})\cap q(\mathcal{H}_s)|}{\theta}$ is an unbiased estimator of $|q(\mathcal{D}) \cap q(\mathcal{H})|$.

Example 5.3. Recall that q_1, q_2, q_4, q_7 are predicted as solid queries (Example 5.1). Table 4 illustrates how to use the proposed *unbiased* estimator to estimate their benefits. For q_1, we have $q_1(\mathcal{D}) = \{d_1\}$ and $q(\mathcal{H}_s) = \phi$, thus the estimated benefit is $\frac{|q_1(\mathcal{D})\cap q(\mathcal{H}_s)|}{\theta} = \frac{0}{\frac{1}{3}} = 0$. Similarly, for q_2, q_4, q_7, their estimated benefits are 0 as well.

From the above example, we can see that the unbiased estimator does not perform well. This is because that for solid queries, $|q(\mathcal{D})|$ is typically small, thus $|q(\mathcal{D}) \cap q(\mathcal{H}_s)|$ tends to be zero. Therefore, the query benefit will be estimated as 0, which is not useful for query selection.

Biased Estimator. We propose another estimator to overcome this limitation. The benefit of a solid query is shown in Equation 2. We can rewrite it as

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

Many hidden databases (e.g., Yelp, IMDb) often have a very good coverage of the entities in some domain (e.g., Restaurant, Movie, etc.). As a result, $\Delta\mathcal{D}$ could be small, and thus $|q(\Delta\mathcal{D})|$, as a subset of $\Delta\mathcal{D}$, is even much smaller. Even if $\Delta\mathcal{D}$ is big, we can use the technique proposed in Section 4.2 to reduce its size. For these reasons, we omit $|q(\Delta\mathcal{D})|$ and derive the following estimator:

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

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.

Example 5.4. Table 4 illustrates how to use the proposed *biased* estimator to estimate the benefits of q_1, q_2, q_4, q_7. For q_1, as $q_1(\mathcal{D}) = \{d_1\}$, we have benefit$(q_1) \approx |q_1(\mathcal{D})| = 1$. Similarly, for q_2, q_4, q_7, we have benefit$(q_2) \approx |q_2(\mathcal{D})| = 1$, benefit$(q_4) \approx |q_4(\mathcal{D})| = 1$, and benefit$(q_7) \approx |q_7(\mathcal{D})| = 2$. We can see that this estimator's estimated benefit is the same as the true benefit for all the queries.

5.3 Estimators For Overflowing Queries

We study how to estimate query benefits for overflowing queries. 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.

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

$$P(X = i) = \frac{\binom{k}{i}\binom{N-k}{n-i}}{\binom{N}{n}}. \tag{6}$$

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

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

Intuitively, every draw chooses $\frac{k}{N}$ black ball in expectation. After n draws, $n\frac{k}{N}$ black ball(s) will be chosen. For example, in Figure 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 benefit$(q) = |q(\mathcal{D}) \cap q(\mathcal{H})_k|$, where $q(\mathcal{H})_k$ denotes the top-k records in $q(\mathcal{H})$. We model $q(\mathcal{H})$ as a list of balls, $q(\mathcal{H})_k$ as black balls, $q(\mathcal{D}) - q(\mathcal{H})_k$ as white balls, and $q(\mathcal{D}) \cap q(\mathcal{H})$ as a set of balls randomly drawn from $q(\mathcal{H})$. Then, estimating the benefit of a query is reduced as estimating the number of black balls in draws. Based on Equation 7, we have

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

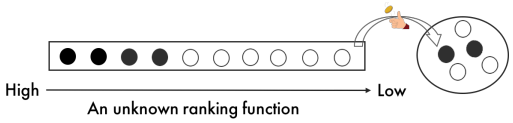


Figure 2: Analogy: breaking the top-k constraint.

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

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

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

For $|q(\mathcal{D}) \cap q(\mathcal{H})|$, which is the number of hidden records that satisfy q and are also in $\mathcal{D}$, we have studied how to estimate it in Section 5.2. The unbiased estimator (see Equation 3) is

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

The biased estimator (see Equation 5) is

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

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

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

This estimator is derived from the ratio of two unbiased estimators. Since $E[\frac{X}{Y}] \neq \frac{E[X]}{E(Y)}$, Equation 12 is not an unbiased estimator, but it is conditionally unbiased (Lemma 5.5). For simplicity, we omit "conditionally" if the context is clear.

Lemma 5.5. Given an overflowing query q, if $q(\mathcal{D}) \cap q(\mathcal{H})$ is a random sample of $q(\mathcal{H})$, then $|q(\mathcal{D}) \cap q(\mathcal{H}_s)| \cdot \frac{k}{|q(\mathcal{H}_s)|}$ is a conditionally unbiased estimator of the true benefit given $|q(\mathcal{H}_s)|$ regardless of the underlying ranking function.

Example 5.6. Recall that q_3, q_5, q_6 are predicted as overflowing queries (Example 5.1). Table 4 illustrates how to use the above unbiased estimator (Equation 12) to estimate

their benefits. For q_3, since $q_3(\mathcal{D}) = \{d_3\}$ and $q_3(\mathcal{H}_s) = \{h_3\}$, then we have $|q_3(\mathcal{D}) \cap q_3(\mathcal{H}_s)| = 1$ and $|q_3(\mathcal{H}_s)| = 1$. By plugging them into Equation 12, we have benefit$(q_3) \approx |q_3(\mathcal{D}) \cap q_3(\mathcal{H}_s)| \cdot \frac{k}{|q_3(\mathcal{H}_s)|} = 1 \cdot \frac{2}{1} = 2$. Similarly, for q_5, q_6, we have benefit$(q_5) \approx |q_5(\mathcal{D}) \cap q_5(\mathcal{H}_s)| \cdot \frac{k}{|q_5(\mathcal{H}_s)|} = 1 \cdot \frac{2}{2} = 1$ and benefit$(q_6) \approx |q_6(\mathcal{D}) \cap q_6(\mathcal{H}_s)| \cdot \frac{k}{|q_6(\mathcal{H}_s)|} = 1 \cdot \frac{2}{1} = 2$.

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

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

This estimator is biased, where the bias is

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

Lemma 5.7. 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 5.8. Consider q_3, q_5, q_6 again. Table 4 illustrates how to use the above unbiased estimator (Equation 14) to estimate their benefits. For q_3, since $q_3(\mathcal{D}) = \{d_3\}$ and $|q_3(\mathcal{H}_s)| = \{h_3\}$, then we have $|q_3(\mathcal{D})| = 1$ and $|q(\mathcal{H}_s)| = 1$. By plugging them into Equation 14, we have benefit$(q_3) \approx |q_3(\mathcal{D})| \cdot \frac{k\theta}{|q_3(\mathcal{H}_s)|} = 1 \cdot \frac{2 \cdot 1/3}{1} = \frac{2}{3}$. Similarly, for q_5, q_6, we have benefit$(q_5) \approx |q_5(\mathcal{D})| \cdot \frac{k\theta}{|q_5(\mathcal{H}_s)|} = 3 \cdot \frac{2 \cdot 1/3}{2} = 1$ and benefit$(q_6) \approx |q_6(\mathcal{D})| \cdot \frac{k\theta}{|q_6(\mathcal{H}_s)|} = 3 \cdot \frac{2 \cdot 1/3}{1} = 2$.

Putting Everything Together (QSel-Est). We can now put everything together. We call this new query selection approach QSel-Est, which improves QSel-Simple using the optimization techniques proposed in Sections 4 and 5. Example 5.9 illustrates how QSel-Est (biased estimator) works. For QSel-Est (unbiased estimator), the query-selection process is similar. In Appendix B, we discuss some implementation details, such as how to handles small sample size and how to implement QSel-Est efficiently.

Example 5.9. Table 4 (the 'Biased' column) shows the estimated benefits. Suppose $b = 2$. In the first iteration, QSel-Est selects q_6 which has the largest estimated benefit. The returned result can cover two local records $q_6(\mathcal{D})_{\text{cover}} = \{d_1, d_4\}$. QSel-Est removes the covered records from $\mathcal{D}$ and re-estimates the benefit of each query w.r.t. the new $\mathcal{D}$. In the second iteration, QSel-Est selects q_7 which has the largest estimated benefit among the remaining queries. The returned result can cover $q_7(\mathcal{D})_{\text{cover}} = \{d_2, d_3\}$. Since the budget is exhausted, QSel-Est stops the iterative process and returns the crawled hidden records $\{d_1, d_2, d_3, d_4\}$. We can see that in this running example, QSel-Est gets the optimal solution.

Table 5: 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%

6 EXPERIMENTS

We conduct extensive experiments to evaluate the performance of SmartCrawl over simulated and real hidden databases. We aim to examine five major questions: (1) How well does SmartCrawl perform compared to NaiveCrawl? (2) How well does QSel-Est perform compared to QSel-Simple? (3) Which estimator (biased or unbiased) is preferable? (4) Is SmartCrawl more robust to data errors compared to NaiveCrawl? (5) Does SmartCrawl still perform well over a hidden database with a conjunctive (non-conjunctive) keyword-search interface?

6.1 Experimental Settings

We first provide the experimental settings of simulated and real-world hidden databases, and then present the implementation of different crawling approaches.

6.1.1 Simulated Hidden Database. We designed a simulated experiment based on DBLP[4]. The simulated scenario is as follows. Suppose a data scientist collects a list of papers in some domains (e.g., database, data mining, AI), and she wants to know the BibTex URL of each paper from DBLP.

Local and Hidden Databases. The DBLP dataset has 5 million records. We construct a local database which contains papers whose authors have published on top conference. We first got the list of the authors who have published papers in SIGMOD, VLDB, ICDE, CIKM, CIDR, KDD, WWW, AAAI, NIPS, and IJCAI, and then collected their papers. We used sampled data from this dataset as the local database. We assumed that a local database $\mathcal{D}$ was randomly drawn from the union of the publications of the authors. Since $\mathcal{D}$ may not be fully covered by $\mathcal{H}$, we assumed that a hidden database consists of two parts: $\mathcal{H} - \mathcal{D}$ and $\mathcal{H} \cap \mathcal{D}$, where $\mathcal{H} \cap \mathcal{D}$ was randomly drawn from $\mathcal{D}$, and $\mathcal{H} - \mathcal{D}$ was randomly drawn from the entire DBLP dataset.

Keyword Search Interface. We implemented a search engine over the 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.

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

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 5 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 robustness of different approaches to data errors. Suppose *error%* = 10%. We will randomly select 10% records from $\mathcal{D}$. For each record, we removed a word, added a new word, or replaced an existing word with a new word with the probability of 1/3, respectively.

6.1.2 Real Hidden Database. We evaluated SmartCrawl over the Yelp's hidden database and the Spotify's hidden database, respectively. Note that Spotify Search API [8] uses conjunctive keyword search but Yelp does not force queries to be conjunctive [10]. This experiment aims to examine how well SmartCrawl performs compared to baselines over real-world hidden databases, with/without conjunctive-keyword-search assumptions.

Local Database. We constructed a local database based on the Yelp dataset[5]. The dataset contains 36,500 records in Arizona, where each record describes a local business. We randomly chose 3000 records as a local database.

We downloaded an Amazon Music dataset[6] as a local database for Spotify. The original dataset contains 55,959 song tracks. We selected a subset of tracks from 12 popular artists (Ed Sheeran, Taylor Swift, Beyonce, Kelly Clarkson, Elvis Presley, Lady Gaga, Justin Bieber, Hilary Duff, Nelly Furtado, Whitney Houston, Lana Del Rey, and U2). The size of the local database is 2808.

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, and thus we only needed to generate keyword queries. For each API call, Yelp returns the top-50 related results.

We treated all the music tracks in the Spotify database as the hidden database. Unlike Yelp, Spotify provides a conjunctive keyword search interface. It allows public users to search over albums, tracks, artists information with keyword queries. We used keywords from track titles for the query-pool generation. We used their track search service. For each API call, Spotify returns the top-50 results

Hidden Database Sample. We adopted an existing technique [47] 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.

We applied the same sampling approach to Spotify. We used the keywords from the whole Amazon Music dataset to initialize a query pool. A 0.25% sample with size 500 was constructed by issuing 7017 queries.

Evaluation Metric. We manually labeled the data by searching each local record over Yelp (Spotify) and identifying its matching hidden record. Since entity resolution is an independent component of our framework, we assumed that once a record is crawled, the entity resolution component can perfectly find its matching local record (if any). Let $\mathcal{H}_{\text{crawl}}$ denote a set of crawled hidden records. We compared the *recall* of different approaches, where the 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 $\mathcal{H}$.

6.1.3 Implementation of Different Approaches. **NaiveCrawl.** For DBLP dataset, NaiveCrawl concatenated the title, venue, and author attributes of each local record as a query and issued the queries to a hidden database in a random order. For Yelp dataset, NaiveCrawl concatenated the business name and city attributes.

HiddenCrawl. Deep web crawling has been extensively studied in the literature [12, 26, 28, 30, 33, 35, 37, 38]. They aim to crawl the hidden database data as more as possible rather than cover the content relating to the local database. We compared with the state-of-the-art approach (keyword-search interface) HiddenCrawl [33]. To make the comparison fairer, we assume that the hidden database sample is available for HiddenCrawl so that it can leverage it to generate a query pool.

SmartCrawl-S, SmartCrawl-U, SmartCrawl-B. We consider three variants of SmartCrawl. They adopted the same query pool generation method (Section 3.1), but different query selection strategies. SmartCrawl-S used QSel-Simple, which issues queries only based on the query frequency w.r.t. the local database (Algorithm 2). SmartCrawl-U used QSel-Est (unbiased), which estimates query benefits based on the unbiased estimator. SmartCrawl-B used QSel-Est, which estimates query benefits based on the biased estimator. Note that both SmartCrawl-U and SmartCrawl-B implemented the optimization technique in Section 4.2 to mitigate the negative impact of $\Delta\mathcal{D}$.

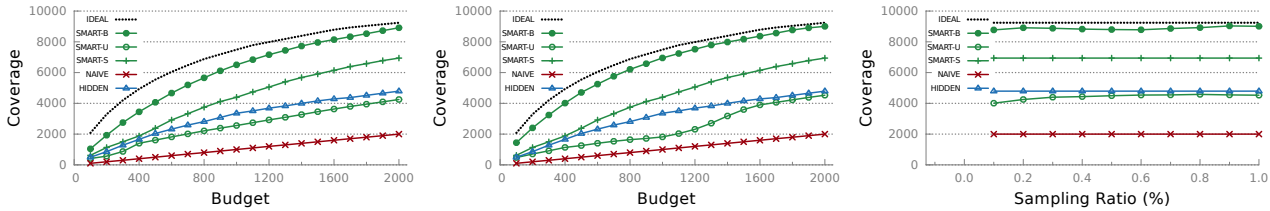
IdealCrawl. IdealCrawl used the same query pool as SmartCrawl but select queries using the ideal greedy algorithm QSel-Ideal (Algorithm 1) based on true benefits.

6.2 Simulation Experiments

We evaluated the performance of SmartCrawl and compared it with the approaches mentioned above in a large variety of situations.

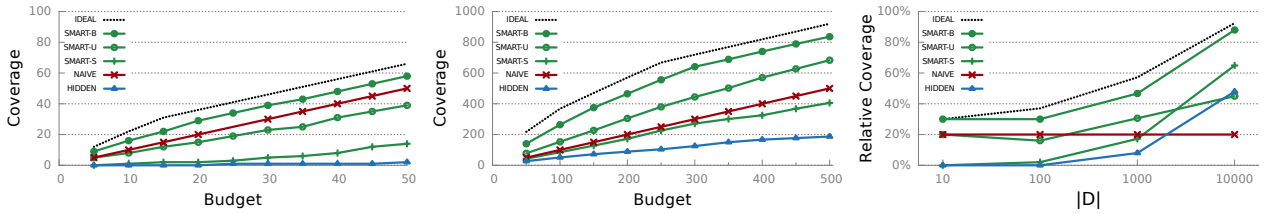
Sampling Ratio. We first examine the impact of sampling ratios on the performance of SmartCrawl. Figure 3 shows

[5]https://www.yelp.com/dataset_challenge
[6]https://sites.google.com/site/anhaidgroup/useful-stuff/data



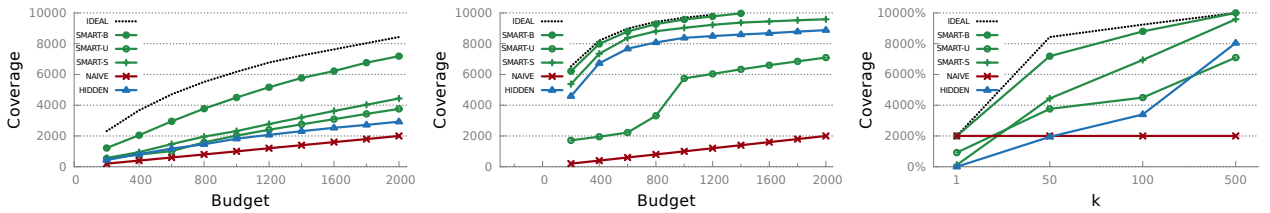
(a) Sample Ratio = 0.2% (b) Sample Ratio = 1% (c) Budget = 2000

Figure 3: Comparisons of different approaches with various sampling ratios θ (DBLP).



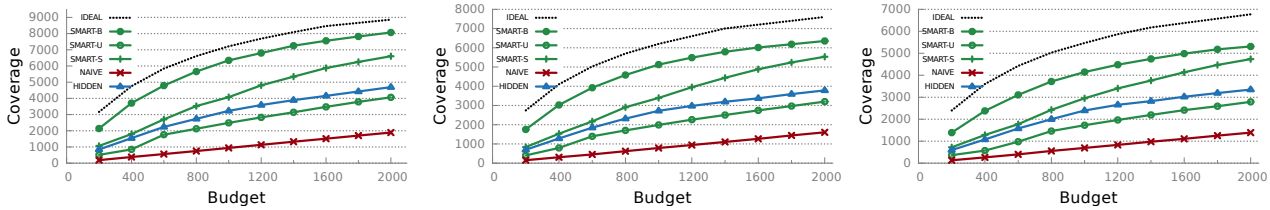
(a) $|\mathcal{D}| = 100$ (b) $|\mathcal{D}| = 1000$ (c) Budget = 20% of $|\mathcal{D}|$

Figure 4: Comparisons of different approaches with various local database size $|\mathcal{D}|$ (DBLP).



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

Figure 5: Comparisons of different approaches with various result# limits k (DBLP).



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

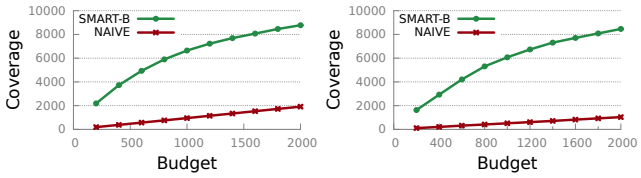
Figure 6: Comparisons of different approaches with various $|\Delta\mathcal{D}|$ size (DBLP).

the result. In Figure 3(a), we set the sampling ratio to 0.2%, leading to the sample size of $100,000 \times 0.2\% = 200$. We can see that with such a small sample, SmartCrawl-B still had a similar performance with IdealCrawl and covered about 2× more records than HiddenCrawl and about 4× more records than NaiveCrawl.

We found that SmartCrawl-B outperformed SmartCrawl-S. This shows that the biased estimator is very effective; but for SmartCrawl-S, since it only considered the query frequency w.r.t. the local database, it tended to issue many overflowing queries which have low benefits.

Furthermore, we can see that SmartCrawl-U did not have a good performance on such a small sample, even worse than HiddenCrawl. In fact, we found that SmartCrawl-U tended to select queries randomly because many queries had the same benefit values. This phenomenon was further manifested in Figure 3(b), which increased the sampling ratio to 1%. In Figure 3(c), 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 ratio was increased to 1%, SmartCrawl-B is very close to IdealCrawl, where IdealCrawl covered 92% of the local database while SmartCrawl covered 89%.

In summary, this experimental result shows that (1) biased estimators are much more effective than unbiased estimators; (2) biased estimators work well even with a very small sampling ratio 0.1%; (3) SmartCrawl-B outperformed HiddenCrawl and NaiveCrawl by a factor of 2 and 4, respectively;



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

Figure 7: Comparing the robustness of SmartCrawl-B and NaiveCrawl (DBLP).

(4) SmartCrawl-B outperformed SmartCrawl-S due to the proposed optimization techniques.

Local Database Size. The main reason that HiddenCrawl performed so well in the previous 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 4(a) shows the result when $|\mathcal{D}|$ has only 100 records. We can see that HiddenCrawl only covered 2 records after issuing 50 queries, while the other approaches (except SmartCrawl-S) all covered 39 more records. SmartCrawl-S also did not perform well either since it did not consider the top-k constraint and selected many overflowing queries. 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 selection of more general queries (e.g., "SIGMOD").

Figure 4(b) shows the result for $|D| = 1000$. We can see that HiddenCrawl 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 4(c). We can see that as $|\mathcal{D}|$ increased, all the approaches except NaiveCrawl showed improved performance. This is because the larger $|\mathcal{D}|$ is, the more local records an issued query can cover. But for NaiveCrawl, its performance remained the same. The reason is that NaiveCrawl issued overly specific queries. With the same query budget b, it covered b local records regardless of whether $|\mathcal{D}|$ increased or not.

Result Number Limit. Next, we investigate the impact of k on different approaches.

Figure 5(a) shows the result when $k = 50$. In this case, SmartCrawl-B can cover about 3.5 times more records than NaiveCrawl after issuing 2000 queries. In other words, for SmartCrawl-B, each query covered 3.5 local records on average while NaiveCrawl only covered one record per query. When we increased k to 500 (Figure 5(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 5(c) compared different approaches by varying k. We can see that IdealCrawl, SmartCrawl-B, and NaiveCrawl achieved the same performance when $k = 1$, while

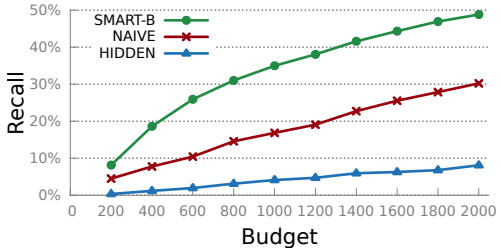


Figure 8: Comparisons of SmartCrawl-B, NaiveCrawl, and HiddenCrawl (Spotify).

HiddenCrawl and SmartCrawl-S can hardly cover any records. As k increased, NaiveCrawl kept unchanged because it covered one local record at a time regardless of k, while the other approaches all got the performance improved. The performance gap between SmartCrawl-B and SmartCrawl-S got smaller as k grew. This is because SmartCrawl-S ignored the top-k constraint. As k grows, the impact of the ignorance of the top-k constraint will be reduced.

Impact of $|\Delta\mathcal{D}|$. We explore the impact of $|\Delta\mathcal{D}|$ on different approaches. Figure 6(a), (b), (c) show the results when $|\Delta\mathcal{D}|$ is 5%, 20%, and 30% of $|\Delta\mathcal{D}|$. We can see that increasing $|\Delta\mathcal{D}|$ will make all approaches perform less effectively. However, SmartCrawl-B still outperformed all the other approaches since it mitigated the negative impact of $|\Delta\mathcal{D}|$ and considered the top-k constraint.

Robustness to Data Errors. We compared SmartCrawl-B with NaiveCrawl when the local database contains data errors. Figure 7(a),(b) show the results for the cases when adding 5% and 50% data errors to the local databases, respectively. We find that 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 is because that the queries selected by NaiveCrawl typically contain more keywords. The more keywords a query contains, the more likely it will be affected by data errors. We have also observed this interesting phenomenon in the next section.

6.3 Real-world Hidden Databases

We evaluate the performance of SmartCrawl using real-world hidden databases, Spotify and Yelp, where Spotify uses a conjunctive keyword search interface, but Yelp's keyword search interface may return the hidden records that partially match the query.

We first compare the performance of SmartCrawl-B, NaiveCrawl, and HiddenCrawl over the Spotify hidden database. Figure 9 shows the recall of each approach by varying the budget from 200 to 2000. We can see that SmartCrawl-b achieved the best recall among the three approaches. This further validate the effectiveness of SmartCrawl-b in a real-world hidden database.

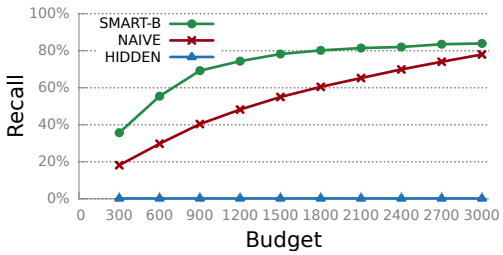


Figure 9: Comparisons of SmartCrawl, NaiveCrawl, and HiddenCrawl (Yelp).

So far, we have evaluated the performance of SmartCrawl under the conjunctive keyword-search assumption. One natural question is that how well SmartCrawl would perform without the assumption. To answer this question, we compare the performance of SmartCrawl-B, NaiveCrawl, and HiddenCrawl on the Yelp dataset. We have three observations from the figure. Firstly, SmartCrawl-B can achieve the recall above 80% by issuing 1800 queries while NaiveCrawl only achieved a recall of 60%. This shows that for a real-life hidden database, it is still the case that only issuing very specific queries is less effective than issuing both specific and general queries. Secondly, HiddenCrawl performed poorly on this dataset because the local database $|\mathcal{D}|$ is small. This further validated that the ineffective of existing deep web crawling techniques. Thirdly, NaiveCrawl got a recall smaller than SmartCrawl-B even after issuing all the queries (one for each local record). This is because NaiveCrawl is not as robust as SmartCrawl-B to tolerate data errors.

7 RELATED WORK

Deep Web. There are three lines of work about deep web related to our problem: deep web crawling [12, 26, 28, 30, 33, 35, 37, 38], deep web integration [15, 25, 31, 44], and deep web sampling [13, 17–20, 27, 41, 47, 48].

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 good coverage of the underlying database. Along this line of research, various types of query interfaces were investigated, such as keyword search interface [12, 26, 33] and form-like search interface [28, 30, 35, 37, 38]. 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, 31, 44] 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 [13, 47, 48] or form-like interfaces [17, 17, 41]. 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 [14, 22, 23, 29, 34, 45, 46], 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 [36, 39, 40, 43, 49]. 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]. CrawlEnrich 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.

8 CONCLUSION

This paper studied a novel problem called CrawlEnrich. We proposed the SmartCrawl framework, which progressively selects a set of queries to maximize the local database coverage. The 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 two key factors: the impact of $\Delta\mathcal{D}$ and the top-k constraint. We theoretically analyzed the negative impacts of these factors, and proposed a new query selection algorithm called QSel-Est. Our detailed experimental evaluation has shown that (2) the biased estimators are superior to the unbiased estimators; (1) QSel-Est is more effective than QSel-Simple in various situations; (3) SmartCrawl can significantly outperform NaiveCrawl and HiddenCrawl over both simulated and real hidden databases; (4) SmartCrawl is more robust than NaiveCrawl to data errors.

This paper has shown that it is a promising idea to leverage the 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. For example, how to create a sample at 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 purposes such as data cleaning and row population.

APPENDIX

A PROOFS

Proof of Lemma 3.4

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

For Algorithm 1 (Line 3), it sets benefit$(q) = |q(\mathcal{D})_{\text{cover}}|$.

For Algorithm 2 (Line 3), it sets benefit$(q) = |q(\mathcal{D})| = |q(\mathcal{D})_{\text{cover}}| + |q(\Delta\mathcal{D})|$. Since $\mathcal{D} \subseteq \mathcal{H}$, then we have $|q(\Delta\mathcal{D})| = 0$. Thus, the above two equations are equal. Hence, the lemma is proved.

Proof of Lemma 4.2

Part I. We prove by induction that the first $\left(b-|\Delta\mathcal{D}|\right)$ queries selected by QSel-Ideal must be selected by QSel-Bound, i.e.,

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

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

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

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

(1) If 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

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

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

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

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

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

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

Based on Equation 15, we can deduce that ,

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

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

$$q'_i = \operatorname*{argmax}_{q \in Q} |q(\mathcal{D}_i)|. \tag{17}$$

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

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

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

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

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

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

By plugging Equation 16 into Equation 20, we obtain

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

Since $q_1 = \text{argmax}_{q \in Q}$ 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 $\left(b - |\Delta\mathcal{D}|\right)$ 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 $\left(b - |\Delta\mathcal{D}|\right)$ queries, and N_2 denote the number of local records covered by the remaining $|\Delta\mathcal{D}|$ queries. Then, we have

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

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

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

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

By plugging Equation 22 into Equation 23, we obtain

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

Algebraically:

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

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

Proof of Lemma 5.2

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:

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

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

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

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

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

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

Proof of Lemma 5.5

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

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

Based on Lemma 5.2, we obtain

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

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

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

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

Proof of Lemma 5.7

The expected value of the estimator is

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

Therefore, the bias of the estimator is:

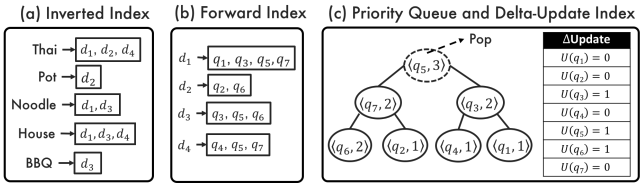


Figure 10: An illustration of the indexing techniques for efficient implementations of our estimators.

$$bias = \frac{k \cdot |q(\mathcal{D})|}{|q(\mathcal{H})|} - |q(\mathcal{D}) \cap q(\mathcal{H})| \cdot \frac{k}{|q(\mathcal{H})|} = |q(\Delta\mathcal{D})| \cdot \frac{k}{|q(\mathcal{H})|} \tag{27}$$

B IMPLEMENTATION DETAILS

This section presents implementation details of QSel-Est. We first discuss how to handle small sample size in Section B.1, and then propose efficient techniques to implement QSel-Est in Section B.2.

B.1 Inadequate Sample Size

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

- *Query Type.* For the queries with $|q(\mathcal{H}_s)| = 0$, since $\frac{|q(\mathcal{H}_s)|}{\theta} = 0 \leq k$, the current QSel-Est will always predict them as solid queries. With the idea of treating $\mathcal{D}$ as a random sample, QSel-Est will continue to check whether $\frac{|q(\mathcal{D})|}{\alpha} > k$ holds. If yes, QSel-Est will predict q as an overflowing query instead.
- *Query Benefit.* For the queries with $|q(\mathcal{H}_s)| = 0$, as shown in Table 3, 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.

B.2 Efficient Implementation

This section discusses how to implement QSel-Est efficiently.

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 10(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 10(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 10(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 10(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 10(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.

Algorithm 4 depicts the pseudo-code of our efficient implementation of QSel-Est.

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.

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

At the initialization stage (Lines 1-15), QSel-Est needs to (1) create two inverted indices based on $\mathcal{D}$ and $\mathcal{H}_s$ with the time complexity of $O\left(|\mathcal{D}||d| + |\mathcal{H}_s||h|\right)$; (2) create a forward index with the time complexity of $O(|\mathcal{Q}||q(\mathcal{D})|)$; (3) create a priority queue with the time complexity of $O(|\mathcal{Q}| \log(|\mathcal{Q}|))$; (4) compute the query frequency w.r.t. $\mathcal{D}$ and $\mathcal{H}_s$ with the time complexity of $O\left(cost_q \cdot |\mathcal{Q}|\right)$, where $cost_q$ denotes the average cost of using the inverted index to compute $|q(\mathcal{D})|$

and $|q(\mathcal{H}_s)|$, which is much smaller than the brute-force approach (i.e., $cost_q \ll |\mathcal{D}||q| + |\mathcal{H}_s||q|$).

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

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

REFERENCES

[1] DBLP. http://dblp.org/.
[2] Forbes Survey. https://tinyurl.com/y7evy7en.
[3] GoodReads. https://www.goodreads.com/.
[4] Google Maps API. https://developers.google.com/maps/documentation/geocoding/usage-limits.
[5] IMDb. http://www.imdb.com/.
[6] OpenRefine Reconciliation Service. https://github.com/OpenRefine/OpenRefine/wiki/Reconciliation-Service-API. Accessed: 2018-10-15.
[7] SoundCloud. https://soundcloud.com/.
[8] Spotify API. https://developer.spotify.com/documentation/web-api/reference/search/search.
[9] The ACM Digital Library. https://dl.acm.org/.
[10] Yelp API. https://www.yelp.com/developers/faq.
[11] 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.
[12] E. Agichtein, P. G. Ipeirotis, and L. Gravano. Modeling query-based access to text databases. In *WebDB*, pages 87–92, 2003.
[13] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's index. *J. ACM*, 55(5):24:1–24:74, 2008.
[14] M. J. Cafarella, A. Y. Halevy, and N. Khoussainova. Data integration for the relational web. *PVLDB*, 2(1):1090–1101, 2009.
[15] K. C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In *CIDR*, 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 *ACM SIGMOD*, 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 *ACM SIGMOD*, pages 855–866, 2010.
[19] A. Dasgupta, N. Zhang, and G. Das. Leveraging COUNT information in sampling hidden databases. In *ICDE*, 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*, 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 *SSDBM*, 2015.
[23] J. Fan, M. Lu, B. C. Ooi, W. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. In *ICDE*, pages 976–987, 2014.
[24] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD*, 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 *WSDM*, pages 355–364, 2013.
[27] P. G. Ipeirotis and L. Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *VLDB*, 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] 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.
[30] 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.
[31] W. Meng, C. T. Yu, and K. Liu. Building efficient and effective metasearch engines. *ACM Comput. Surv.*, 34(1):48–89, 2002.
[32] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.*, 14(1):265–294, 1978.
[33] A. Ntoulas, P. Zerfos, and J. Cho. Downloading textual hidden web content through keyword queries. In *JCDL*, pages 100–109, 2005.
[34] R. Pimplikar and S. Sarawagi. Answering table queries on the web using column keywords. *PVLDB*, 5(10):908–919, 2012.
[35] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB*, pages 129–138, 2001.
[36] 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.
[37] 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.
[38] S. Thirumuruganathan, N. Zhang, and G. Das. Breaking the top-k barrier of hidden web databases? In *ICDE*, pages 1045–1056, 2013.
[39] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, pages 673–684, 2013.
[40] 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.
[41] F. Wang and G. Agrawal. Effective and efficient sampling methods for deep web aggregation queries. In *EDBT*, pages 425–436, 2011.
[42] P. Wang, Y. He, R. Shea, J. Wang, and E. Wu. Deeper: A data enrichment system powered by deep web. In *ACM SIGMOD Demo*, 2018.
[43] R. C. Wang and W. W. Cohen. Iterative set expansion of named entities using the web. In *ICDM*, pages 1091–1096, 2008.
[44] 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 *ACM SIGMOD*, pages 95–106, 2004.
[45] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *ACM SIGMOD*, pages 97–108, 2012.
[46] M. Zhang and K. Chakrabarti. Infogather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *ACM SIGMOD*, pages 145–156, 2013.
[47] M. Zhang, N. Zhang, and G. Das. Mining a search engine's corpus: efficient yet unbiased sampling and aggregate estimation. In *ACM SIGMOD*, pages 793–804, 2011.
[48] M. Zhang, N. Zhang, and G. Das. Mining a search engine's corpus without a query pool. In *CIKM*, pages 29–38, 2013.
[49] S. Zhang and K. Balog. Entitables: Smart assistance for entity-focused tables. In *ACM SIGIR*, pages 255–264, 2017.