CRUX: Adaptive Querying for Efficient Crowdsourced Data Extraction

Theodoros Rekatsinas
thodrek@cs.wisc.edu
University of Wisconsin-Madison
Madison, WI, USA

Amol Deshpande
amol@cs.umd.edu
University of Maryland
College Park, MA, USA

Aditya Parameswaran
adityagp@berkeley.edu
University of California-Berkeley
Berkeley, CA, USA

ABSTRACT

Crowdsourcing is essential for collecting information about real-world entities. Existing crowdsourced data extraction solutions use fixed, non-adaptive querying strategies that repeatedly ask workers to provide entities from a fixed domain until a desired level of coverage is reached. Unfortunately, such solutions are highly impractical as they yield many duplicate extractions. We design an adaptive querying framework, CRUX, that maximizes the number of extracted entities for a given budget. We show that the problem of budgeted crowdsourced entity extraction is NP-Hard. We leverage two insights to focus our extraction efforts: *exploiting the structure of the domain of interest*, and *using exclude lists to limit repeated extractions*. We develop new statistical tools to reason about the number of new distinct extracted entities of *additional queries* under the presence of little information, and embed them within adaptive algorithms that maximize the distinct extracted entities under budget constraints. We evaluate our techniques on synthetic and real-world datasets, demonstrating an improvement of up to 300% over competing approaches for the same budget.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**.

KEYWORDS

crowdsourcing; extraction; structured domains

ACM Reference Format:
Theodoros Rekatsinas, Amol Deshpande, and Aditya Parameswaran. 2019. CRUX: Adaptive Querying for Efficient Crowdsourced Data Extraction. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3357976

1 INTRODUCTION

Structured data repositories, such as knowledge bases, taxonomies, and hierarchies, enable many end-user applications including keyword search, product catalogs, event detection and recommender systems in a variety of companies, such as Google [23], Microsoft [6], Yahoo [7], Walmart [8], and Amazon [2]. The majority of these repositories are constructed using automated extraction schemes that target popular text and web corpora. As a result, they contain information primarily on *head data*, i.e., popular entities and their attributes, comprising a tiny fraction of all entities. There has therefore been increasing interest in augmenting this with information about the so called *long-tail*, not-so-popular entities, to reduce the sparsity of present-day structured data repositories. Crowdsourcing is a natural solution to extract these entities; examples of such efforts include Google's Guides[1], and Facebook's Professional Services[2]. In fact, a recent study reports that a number of companies use crowdsourcing for entity extraction [17].

Table 1: Extracting people from the news. Percentage of entities reported by at least two different workers for each ⟨occupation, news portal⟩.

News Portal	Actors/Singers	Athletes	Politicians
WashPost	47%	55%	56%
NY Times	41%	65%	53%
HuffPost	45%	63%	60%
USA Today	54%	45%	57%
WSJ	42%	77%	57%

Motivated by the aforementioned applications, we study *crowdsourced entity extraction*, i.e., the problem of collecting entities by asking crowd workers to *list entities* from a domain of interest. Recent work from Trushkowsky et al. [25, 26] has studied crowdsourced entity extraction using fixed questions like "give me another entity", extending species estimation techniques to assess the degree of completeness of their extraction. Unfortunately, due to the cost of human labor and the inherent redundancy in the answers of crowd workers, crowdsourced entity extraction using this fixed form of questions can become highly impractical. We use a small pilot experiment on Amazon's Mechanical Turk [1] to illustrate this.

Example 1.1. Our goal was to extract people from the news. We asked workers to list people of different occupations mentioned in five major US news portals during the period of a single day. In particular, we asked workers to provide us with "Actors/Singers", "Athletes", and "Politicians" listed in "The New York Times", "Huffington Post", "The Washington Post", "USA Today" and "The Wall Street Journal". For each newspaper we provided a link to the newspaper's homepage and for each (occupation, news portal) combination, e.g., "Athletes mentioned in NY Times", we asked 30 distinct workers to provide us with five entities, leading to 150 extractions per combination.

Table 1 shows the percentage of entities that were provided by at least two different workers for different <occupation, news

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3357976

[1] https://www.google.com/local/guides/
[2] https://www.facebook.com/services/

portal> combinations. As shown, on average, more than half of the extracted entities were reported by at least two workers, while there are cases where the percentage of duplicate entities was as high as 77%. This means that *at least 77%* of the cost (in compensating workers for their answers) is simply wasted. Thus, a static or fixed querying strategy that issues the same question repeatedly to crowd workers, such as in Trushkowsky et al. [25], leads to a large number of *repeated extractions of the popular entities, and low coverage of the less-popular entities*. In fact, the not-so-popular entities may only be extracted after a very long time (translating to high cost and effort) or may never be extracted at all. ***It is exactly this abundance of duplicate extractions that can make fixed-query crowdsourced data extraction impractical***.

Our Approach and Challenges. Our goal is to reduce redundancy in crowdsourced data extraction by leveraging two key insights that allow us to *adaptively issue fine-grained queries*, i.e., queries that target only a subspace of the entity-domain, to the crowd:

1) Exploit Structure to Partition the Search Space. Most domains that one may choose to extract entities from are *structured*, i.e., domains are associated with a collection of attributes, each of which typically exhibits hierarchical structure. One can leverage the existence of such attributes to choose queries from a much richer space, considering all combinations of values for these attributes. For example, if we had a hierarchy associated with athletes in our pilot experiment, we could leverage that to *partition our extraction space* and extract tennis players, marathon runners, or javelin throwers, by asking workers questions like *provide a marathon runner mentioned in the New York Times today*. It is easy to see that targeted queries which focus on disjoint partitions of the underlying entity domain can drastically limit duplicate extractions.

2) Use Exclude Lists. We can also extend typical crowd queries to include an *exclude list*, e.g., "list a person in the New York Times that is not Donald Trump". Excluding popular entities can help us identify new distinct entities much faster.

Overall, fine-grained queries of the form above enable us to design *adaptive querying strategies* that aim to limit the number of duplicate extractions. We refer to a querying strategy as adaptive if it analyzes the entities returned in previously issued queries to dynamically adapt further queries in two ways: (i) either by limiting their scope to a subdomain of the overall entity domain, thus, exploiting the structure of the entity domain, or (ii) by introducing an exclude list to them to prevent already extracted entities from bing extracted again. To guarantee that our methods will lead to efficient crowdsourced entity extraction techniques, we study a budgeted version of the problem, where our goal is to identify the adaptive querying strategy that maximizes the number of distinct retrieved entities for a given budget.

Unfortunately, designing adaptive querying policies comes with new challenges. The main question we seek to address is: *what is the most profitable query to issue next?* Given a cost model for crowd queries, we need to *estimate the gain* of a new query, i.e., estimate how many new entities would be extracted from a query in expectation, given the already extracted data. In addition, we also need to deal with the *sparsity* and the *exponential size* of the query space. Many of the attribute combinations are likely to be empty, i.e., the corresponding queries are likely to have no answers (e.g., there may not be few, if any, javelin throwers living in New York); avoiding such queries is essential to keep monetary cost low. Finally, we need to deal with the *interrelationships* across queries. Many of the queries are *coupled*. For example, the results from a few queries to "list an athlete in today's New York Times" can inform whether issuing queries to "list a Baseball player in today's New York Times" is useful or not. Thus, identifying the right order to ask queries is highly important. Overall, we want to maximize the number of distinct entities extracted with the minimal number of queries against crowd workers.

To address the above challenges we introduce CRUX, a framework for efficient CRowdsoUrced data eXtraction under budget constraints. We propose a collection of statistical techniques for estimating the *gain* of further queries, i.e., the number of new distinct entities extracted, for any attribute combination. Our techniques make use of the extracted entities in previously issued queries to derive accurate estimates for new queries. Eventually we covert the problem of budgeted crowdsourced entity extraction to an adaptive optimization problem to detect the optimal queries to be issued against the crowd.

Prior Work. As mentioned previously, Trushkowsky et al. [25, 26] describe the use of species estimation techniques to estimate the completeness of extracted entities from the crowd. Other work [3] has studied the problem of leveraging crowd workers to provide recommendations to users by listing entities relevant to user queries. These papers ask human workers to "list one more entity", without leveraging structure or an exclude list. For example, if we consider the task of enumerating all people mentioned in today's issue of the New York Times, previous techniques focus on issuing the query "list one person in today's New York Times" repeatedly against the crowd until a certain desired degree of completeness is achieved. However, all these approaches suffer from the same problem identified earlier: severe wasted cost, with only the popular entities being extracted repeatedly. As we will see in Section 6, there are scenarios where the existing state-of-the-art methods focus only on the head entities and are only able to retrieve less than 20% of the total entities for a fixed budget while our techniques can extract more than 75% of the total entities. Other recent work employs adaptive querying for filtering, as opposed to extraction, problems [16]. We discuss other related work in Section 7.

Contributions. Our main contributions are:

- *Formalization and Characterization of Hardness.* We formalize the notion of *generalized entity extraction queries* that can also include an *exclude list*. Such queries are of the type "List k entities with attributes $\bar{X}$ that belong in domain D and are not in $\{A, B, \ldots\}$". We also provide statistical techniques to estimate the gain, i.e., the number of newly extracted distinct entities, for generalized queries. We prove that this budgeted crowdsourced entity extraction problem is NP-Hard.
- *Gain Estimation for Generalized Queries.* We develop a new technique to estimate the gain of generalized queries in the presence of little information, i.e., when only a small portion of the underlying entity population has been observed. We empirically demonstrate its effectiveness when extracting entities from sparse domains.

- *Adaptive Querying Strategies.* We propose an algorithmic framework that exploits the structure of the entity domain to maximize the number of extracted entities under budget constraints by targeting tail entities. We view the problem of entity extraction as a *multi-round adaptive optimization problem.* At each round we exploit the available information about entities obtained by previous queries to adaptively select the next query that maximizes the overall *cost-gain* trade-off.

We evaluate our techniques on both real-world and synthetic data and show that CRUX extracts up to *300%* more entities compared to a collection of baselines, and for large entity domains is at *most 25%* away from an omniscient adaptive querying strategy with perfect information.

The remainder of the paper is organized as follows: In Section 2 we formalize the notion of a structured data domain, in Section 3 we define the problem of budgeted crowdsourced entity extraction and show that the problem is NP-hard. In Section 4, we describe techniques for estimating the gain of further queries. Then in Section 5, we introduce an algorithm for discovering the most profitable queries to be issued against the crowd. In Section 6 we present an empirical evaluation for CRUX on both real-world and synthetic data from two distinct application domains.

2 PRELIMINARIES

We formalize the problem of crowdsourced entity extraction over structured domains.

2.1 Structured Data Domains

Let $\mathcal{D}$ be a data domain described by a set of discrete attributes $\mathcal{A}_D = \{A_1, A_2, \ldots, A_d\}$. Let $dom(A_i)$ denote the domain of each attribute $A_i \in \mathcal{A}_D$. Each attribute A_i can also be hierarchically organized. Consider Eventbrite (www.eventbrite.com), an online event aggregator, that uses crowdsourcing to compile a directory of events, such as political rallies and concerts. Events are fully described by their location, type, date and category. Here, entities in the data domain $\mathcal{D}$ correspond to events. The attributes describing the entities in $\mathcal{D}$ are $\mathcal{A}_D$ = {"Event Type", "Location", "Date"}, with "Location" and "Date" being hierarchically organized.

The domain $\mathcal{D}$ can be viewed as a *poset*, i.e., a partially ordered set, corresponding to the cross-product of all available hierarchies[3]. Part of the poset corresponding to the previous example is shown in Figure 1. We denote the poset for a domain $\mathcal{D}$ as $\mathcal{H}_D$. As shown in Figure 1, nodes in the poset correspond to configurations where only a subset of the attributes in $\mathcal{A}_D$ are specified while others are allowed to take any value. For example the root of the poset {} has no specified attributes, corresponding to queries of the form "list an event". Nodes at lower levels, such as $\{X1\}$ and $\{C1\}$, correspond to queries where the event type and location are specified respectively.

2.2 Entities and Entity Extraction Queries

Entities. Our goal is to extract entities from domain $\mathcal{D}$. Each entity e can be uniquely associated with one of the leaf nodes in the hierarchy $\mathcal{H}_D$; that is, there is a unique combination of attribute values $A_1, \ldots, A_d$ characterizing each entity. For example, in Eventbrite, each event is of a specific type, takes place in a specific city, and on a specific day.

[3]Note that $\mathcal{D}$ is not a lattice since there is no unique infimum.

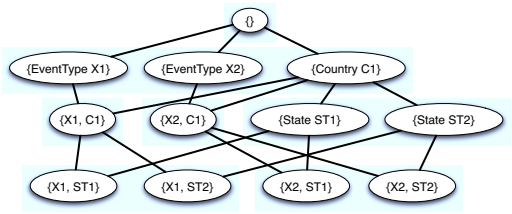

Figure 1: Part of the poset for the domain of Eventbrite.

Queries. We focus on *generalized extraction queries* that can be issued to crowd workers. A query q^v is said to be associated with a node $v \in \mathcal{H}_D$ when q^v contains predicates that correspond to the value combination for $\mathcal{A}_D$ associated with v. For example, if we consider the poset in Figure 1 a query for node $\{X1\}$ has a predicate $EventType = X1$. Hence, workers are required to provide events that satisfy this predicate.

There are three types of queries q^v: (i) *Single entity queries* where workers are required to provide only "one more" entity matching the predicates of the query, (ii) *queries of size k* where workers are asked to provide k distinct entities for a query q^v, and (iii) *exclude list queries* where workers are additionally provided with a list E of l entities that have already been extracted and they are required to provide k distinct entities that are not present in the exclude list. It is easy to see that the last variation generalizes the previous two. Therefore, in the remainder of the paper, we will only consider queries using the third configuration. We refer to these queries as *generalized* queries. To describe a generalized query, we use the notation $q^v(k, E)$ denoting a query of size k with an exclude list E of length l that is associated with node $v \in \mathcal{H}_D$. We denote the configuration for a query as (k, l, v).

2.3 Query Response

We consider a querying interface that asks human workers to not only list entities but to also provide, for each entity, the values for its attributes in $\mathcal{A}_D$ that are not specified in the predicates of q. For example, if the query is "list one concert in Manhattan, New York", with $k = 1, E = \emptyset$, the worker gives us one concert in Manhattan, New York, but also gives us the day on which the concert will take place (here, the missing, unspecified attribute) and the type of concert, i.e., rock concert. If the query is "a concert in the US", with $k = 1, E = \emptyset$, the human worker gives us one concert in the US, but also gives the day on which the concert will take place, as well as the specific city. If less than k entities are present in the underlying population, workers have the flexibility to report either an empty answer or a smaller number of entities.

While getting additional attribute values for entities is not strictly necessary, this information allows us to assign an extracted entity to all relevant nodes in $\mathcal{H}_D$. Furthermore, in most practical applications, it is useful to get the values of the missing attributes to organize and categorize the extracted entities better. Similar query interfaces that ask users to fully specify the attributes of entities have been proposed in recent work [19]. That said, our techniques still apply even if workers do not provide all attributes: in such a setting, entities will be assigned to interior nodes but not to leaves.

Often, workers unwittingly provide the same entity, resulting in duplicates. Resolving duplicates during extraction is crucial for two reasons: (i) they are used to estimate the completeness of extracted entities, and thus, reason about the gain of additional queries, and (ii) they can be used to resolve erroneous values for the attributes of the extracted entities. In practice, standard entity resolution and data fusion techniques [11] can be used to address this problem. Given that we obtain information about several attributes of entities from the crowd, we can apply simple rules that match the entity attribute values to detect duplicates and resolve erroneous values. Additional worker quality detection techniques that consider limited ground truth data [9] can be used to determine the accuracy of workers. During our experiments, we found that simple data fusion techniques were sufficient to resolve noisy labels and duplicates. So, for this paper, we focus on devising near-optimal adaptive crowd querying strategies for entity extraction.

2.4 Query Cost

In a typical crowdsourcing marketplace, tasks have different costs based on their difficulty. While our algorithms works with any cost function, we consider a cost function $c(\cdot)$ that obeys the following properties: (a) given a query with fixed size, its cost should increase (or remain the same) as the size of its exclude list should increase, (b) given a query with a fixed exclude list size, its cost should increase (or remain the same) as the number of requested answer increases, and (c) given a query with fixed size and exclude list size, its cost should increase (or remain the same) as the query contains more predicates, i.e., it corresponds to nodes v at the lower-levels of $\mathcal{H}_D$. The cost function is fixed upfront by the interface-designer based on the amount of work involved.

3 BUDGETED EXTRACTION

We now define *budgeted crowd entity extraction* over structured domains and present an overview of our framework.

3.1 Problem Definition

The problem of *crowdsourced entity extraction* seeks to extract entities that belong to $\mathcal{D}$. For large entity structured domains, one may need to issue a series of entity extraction queries at multiple nodes in $\mathcal{H}_D$—often overlapping with each other—to ensure that the coverage across the domain is maximized. We refer to a series of generalized $q^v(k, E)$ queries at different nodes $v \in \mathcal{H}_D$ as a *querying policy*.

Let π denote a querying policy. A policy π can select a query $q^v(k, E)$ multiple times. Let $C(\pi)$ denote the overall monetary cost of policy π. We define the gain of π, denoted by $\mathcal{E}(\pi)$, to be the total number of unique entities extracted when following π. There is a natural trade-off between the gain (i.e., the number of extracted entities) and the cost of policies.

We require that the user will *only* provide a monetary budget τ_c. The poset $\mathcal{H}_D$ and the possible query size and exclude list size configurations (k, l) for each node are given as input by the application designer. Our goal is to identify the querying policy that maximizes the number of retrieved entities under the given budget constraint:

Problem 1 (**Budgeted Crowd Entity Extraction**). *Let $\mathcal{D}$ be an entity domain characterized by a poset $\mathcal{H}_D$. For each node $v \in \mathcal{H}_D$, let K^v and L^v be the sets of allowed query sizes and exclude list sizes for queries at node v. Let τ_c be a budget on the total cost of issued queries. Find a querying policy π^* using queries $q^v(k, E)$ with $k \in K^v$ and $|E| = l \in L^v$ over nodes $v \in \mathcal{H}_D$ that maximizes $\mathcal{E}(\pi^*)$, the number of unique entities extracted, under the constraint $C(\pi^*) \leq \tau_c$.*

Note that the optimal policy not only specifies the nodes at which queries will be executed but also the size and exclude list of each query. The cost of a querying policy π is $C(\pi) = \sum_{q \in \pi} c(q)$, where the cost of each query q^v is defined according to a cost model specified by the user, is easy to compute. However, the number of unique entities extracted by a policy is not known upfront and needs to be estimated as we discuss in Section 4. Moreover, the problem of finding an optimal querying policy is NP-hard.

Theorem 3.1 (**NP-Hardness**). *Problem 1 is NP-hard.*

The proof is provided in our technical report [20] (with all other proofs) and is based on a reduction from the *unbounded knapsack* problem. This problem is a variation of the original 0-1 knapsack problem that places no upper bound on the number of copies of each kind of item.

Finally, computing the total cost of a policy π is easy. However, the gain $\mathcal{E}(\pi)$ of a policy π is unknown as we do not know in advance the entities corresponding to each node in $\mathcal{H}_D$, and hence, needs to be estimated (see Section 4).

3.2 Query Response Model

To reason about the occurrence of entities as response to specific queries, we assume that each entity has an *unknown popularity value* with respect to crowd workers. Since workers are likely to return different answers based on how the query is phrased, this popularity can be different for different nodes in $\mathcal{H}_D$, and thus, is query-dependent. Given a query $q^v(1, \emptyset)$, the probability that we get entity e in the result of q^v is simply the popularity value of e divided by the popularity value of all entities e' that also satisfy the same predicate constraints. For example, if there are only two entities e_1, e_2 that satisfy the constraints of a query q_1, with popularity values 3 and 2, then the probability that we get e_1 on issuing a query $q_1(1, \emptyset)$ is 3/5. If an exclude list E is specified, then the probability that we get $e \notin E$ is the popularity value of e divided by the popularity values of all entities $e' \notin E$ also satisfying the predicates of q^v.

Since workers are asked to provide a limited number of entities, each query can be viewed as taking a random sample from an unknown population of entities. We refer to the distribution characterizing the popularities of entities in a population as the *popularity distribution* of the population. We do not know the popularity distribution in advance; rather we use the samples retrieved by previous queries as a proxy to reason about this distribution. Also, it is not necessary that workers follow the same popularity distribution. Rather, the overall popularity distribution can be seen as an average of the popularity distributions across all workers.

Estimating the gain of a query $q^v(k, E)$ at a node $v \in \mathcal{H}_D$ is equivalent to estimating the number of new entities extracted by taking additional samples from the population of v given all the retrieved entities (*running sample*) associated with node v. Due to the structure of the poset we may retrieve entities for a node when

issuing queries at other nodes. We discuss this form of *indirect sampling* in Section 4.2.

3.3 CRUX Overview

CRUX casts the budgeted crowd entity extraction problem as a multi-round adaptive optimization problem where at each round we solve the following subproblems:

- **Estimating the Gain for a Query** (Section 4). For each node $v \in \mathcal{H}_D$, consider the retrieved entities associated with v and estimate the number of new unique entities that will be retrieved by a new query $q^v(k, E)$. The query gain is estimated for different query size and exclude list configurations.
- **Detecting the Optimal Querying Policy** (Section 5). Using the gain estimates from the previous step, identify the query configuration (k, l, v) that maximizes the total gain across all rounds given the budget constraint. When identifying the next query we do not explicitly optimize for the exclude list to be used. We rather optimize for the exclude list size l. Once the size is selected, the exclude list is constructed in a randomized fashion. We elaborate more on this in Section 5.1.

CRUX iteratively solves the aforementioned problems until the entire budget is used.

4 ESTIMATING THE GAIN OF QUERIES

As discussed previously, crowd entity extraction queries are equivalent to retrieving samples from a population of items following an unknown popularity distribution. Deciding which query to issue is dictated by the estimated gain of each query, i.e., the expected number of newly extracted entities. Estimating the gain of a query requires reasoning about the population percentage covered by all entities extracted by previous queries as well as the expected number of unseen entities in the underlying population. Prior work [25, 26] studied the problem of estimating the aforementioned quantities for queries of the form "list one more entity", drawing from species richness estimation [5], the problem of estimating the number of distinct species using samples from the underlying population. The proposed techniques extend an estimator from Chao et al. [5] and do not consider queries with exclude lists or queries on non-root nodes of the poset. Moreover, the original Chao estimator has been shown to exhibit negative biases [12, 21], i.e., it underestimates the expected gain. The latter is worse in the presence of little information. Negative biases can severely impact entity extraction since nodes that contain entities from the long tail of the popularity distribution may never be queried as they may be deemed to have zero population. In this section, we first review the existing methodology for estimating the gain of a query, and then discuss how these estimators can be extended to consider support a poset (Section 4.2) and exclude lists (Section 4.3). Finally, we propose a new gain estimator for queries $q^v(k, E)$ that exhibits lower biases, and thus, improved performance, in the presence of little information compared to previous techniques (Section 4.4).

4.1 Single-Node Estimators without Exclusion

Consider a specific node $v \in \mathcal{H}_D$. Let Q be the set of all existing samples retrieved by issuing queries at v without an exclude list. These samples can be combined into a single sample, referred to as a *running sample*, corresponding to a multiset of size $n = \sum_{q \in Q} \text{size}(q)$. Let f_i denote the number of entities that appear i times in this unified sample, f_0 denote the number of unseen entities from the population under consideration, and C be the population coverage of the unified sample, i.e., the fraction of the population covered by the sample. We have that $C = \frac{f_1+f_2+..}{f_0+f_1+...}$. A new query $q^v(k, \emptyset)$ is equivalent to increasing the size of the unified sample by k, thus, its gain is the number of new distinct entities included in the increased sample. Originally, Shen et al. [21] proposed an estimator for the number of new species $\hat{N}_{Shen}$ that would be found in an increased sample of size k. The approach assumes that unobserved entities have equal relative popularity. An estimate of the unique elements found in an increased sample of size k is given by:

$$\hat{N}_{Shen} = f_0 \left(1 - \left(1 - \frac{1-C}{f_0}\right)^k\right) \quad (1)$$

For crowdsourced entity extraction, the term of Shen's formula in parenthesis corresponds to the probability that at least one unseen entity will be present in the result of a query asking for k more entities. Thus, multiplying this quantity with the number of unseen entities f_0 corresponds to the expected number of unseen entities in the result of $q^v(k, \emptyset)$.

The quantities f_0 and C are unknown and need to be estimated for the population of node v using the observed entities in the running sample for v. The coverage can be estimated by the Good-Turing estimator $\hat{C} = 1 - \frac{f_1}{n}$ for the retrieved sample. To estimate f_0, the number of unseen items, the Chao et al. [5] estimator is known to do well [25, 26]. The Chao estimator relies on sample coverage C and the estimated skew of the underlying popularity distribution. The latter is estimated via the information in the available f_i counts [5]. Recently, Hwang et al. [12] proposed an alternative estimator for f_0 that is more robust in the presence of little information and utilizes a regression technique that exploits the information available in all f_i counts.

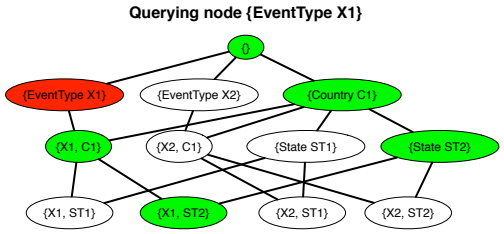

Figure 2: Querying the red node reveals entities from the green nodes.

4.2 Indirect Sampling

Given a structured domain, the extracted entities for a node v can be obtained either by querying v directly or by indirect information flowing to v by queries at other nodes connected to v. We refer to the latter case as *indirect sampling*. Eventually, we have two different kinds of samples: (i) those that are extracted by the **entire population** of v, and (ii) those that are extracted by sampling **only a part of the population** of v. We use an example (Figure 2) using the poset in Figure 1, to illustrate these two cases. Assume a query $q(k, 0)$ at node {EventType X1} whose result contains entities only from node {X1,ST2}. The green nodes in Figure 2 are nodes

for which samples are obtained indirectly without querying them. Notice, that all these nodes are ancestors of {X1,ST2}. We have:

- The samples for nodes {X1, C1} and {X1,ST2} are obtained by their *entire population* since node {EventType X1} is an ancestor of both and its population fully contains the populations of {X1,C1} and {X1,ST1}.
- The samples for nodes { }, {Country C1} and {State ST2} are obtained only by a part of their population since the population of node {EventType X1} does not fully contain the populations of these nodes.

Samples of both types are used to estimate the gain of a query. To do so we merge the extracted entities for each node into a single sample and treat the unified sample as being extracted from the entire population. As we discuss later in Section 5, we develop querying strategies that traverse the poset $\mathcal{H}_D$ in a top-down approach, hence, the number of samples belonging in the first category, i.e., samples retrieved considering the entire population of a node, dominates the number of samples retrieved by considering only part of a node's population.

4.3 Exclude Lists and Negative Answers

A query $q^v(k, E)$ with $E \neq \emptyset$ issued at node $v \in H_D$ effectively limits the sampling to a subset of the population corresponding to node v. To estimate the expected return of such a query, we need to update the estimates $\hat{f}_0$ and $\hat{C}$ before applying Equation (1), by removing entities in E from the running sample for node v and updating the frequency counts f_i and sample size n. The above requires that the exclude list is known, discussed in Section 5.1.

Finally, we consider the effect of *negative answers* on estimating the gain of future queries. It is possible to issue a query at a specific node $v \in \mathcal{H}_D$ and receive no entities, i.e., we receive a negative answer. This is an indication that either underlying entity population of v is empty. In such a scenario, we assign the expected gain of future queries at v and all its descendants to zero. Another type of negative answer corresponds to issuing a query at an ancestor node u of v and receiving no entities for v. In this case, we do not update our estimates for node u as entities from other descendants of u may be more popular than entities associated with u.

4.4 Direct Gain Estimation

The techniques reviewed in Section 4.1 result in negative bias when the number of observed entities represents only a *small fraction* of the entire population [12, 21]. This holds for the large and sparse domains we consider. Bias is introduced as all techniques rely on Equation (1). To eliminate negative bias, we propose a direct estimator for the gain of queries $q^v(k, E)$ that does not use Equation (1). We extend the approach in Hwang [12] and use a regression based technique that captures the structural properties of the expected gain function. The proofs for the results below are in [20].

Let S be the total number of entities in the population and p_i the abundance probability (i.e., popularity) of entity i. Given a sample of size n (where n corresponds to the total sum of sizes for all previously issued queries), define $K(n)$ to be $K(n) = \frac{\sum_{i=1}^{S}(1-p_i)^n}{\sum_{i=1}^{S} p_i(1-p_i)^{n-1}}$. First, we focus on queries without an exclude list. Later we relax this and discuss queries with exclude lists. We have the following theorem on query gain:

Theorem 4.1 (**Direct Gain Estimation**). *Given a node $v \in \mathcal{H}_D$ and a corresponding entity sample of size n, let f_1 and f_2 denote the number of entities that appear exactly once (i.e., singletons) and exactly twice respectively. Let G denote the number of new items retrieved by a query $q(m, \emptyset)$. We have:*

$$G = \frac{1}{(1 + \frac{K'}{n+m})}(K\frac{f_1}{n} - K'\frac{f_1(1 - \frac{1}{n}2\frac{f_2}{f_1})^m}{n+m}) \quad (2)$$

where $K = K(n)$ and $K' = K(n+m)$.

All quantities apart from K and K' in Equation (2) are known. The value of K can be estimated using the regression approach of Hwang and Shen [12]. To estimate the value of K' for an increased sample of size $n+m$, we first show that K increases monotonically as the size of the running sample increases.

Lemma 4.2. *The function $K(n) = \frac{\sum_{i=1}^{S}(1-p_i)^n}{\sum_{i=1}^{S} p_i(1-p_i)^{n-1}}$ increases monotonically, i.e., $K(n+m) \geq K(n), \forall n, m > 0$.*

Given its monotonicity, we model K as a generalized logistic function of the form $K(x) = \frac{A}{1+exp(-G(x-D))}$. As we observe samples of different sizes for different queries we estimate K as described above and therefore we observe different realizations of $f(\cdot)$. Thus, we can learn the parameters of f and use it to estimate K'. In the presence of an exclude list of size l we follow the approach described in Section 4.3 to update the quantities f_i and n used above.

5 DISCOVERING QUERYING POLICIES

We now focus on the core component of CRUX responsible for discovering querying policies that maximize the total number of extracted entities by exploiting the structure of the input domain. We introduce a multi-round adaptive optimization algorithm for identifying good querying strategies. At each round we assume access to a gain estimator for any query $q^v(k, E)$, constructed using the techniques in the previous section. The gain of each query can be viewed as a random variable. By issuing a query we get to observe the value of this random variable, and using the previous observations we decide which query to issue next. Our framework builds upon ideas from the multi-armed bandit literature [4, 10], with additional challenges:

- The number of nodes in $\mathcal{H}_D$ is exponential in the number of attributes $\mathcal{A}_D$ describing $\mathcal{D}$. Querying every node to estimate the expected return for queries $q^v(k, E)$ is prohibitively expensive.
- Balancing the trade-off between *exploitation*/*exploration* [4] is hard. The first refers to querying nodes with sufficient retrieved entities, and hence, accurate estimates for their expected gain; the latter refers to exploring nodes to avoid myopic policies.
- Optimizing all potential exclude lists (i.e., all subsets of observed entities) leads to an exponential explosion. To limit the query space we instead optimize over all potential query configurations (k, l, v).

5.1 Optimizing over Query Configurations

Instead of optimizing over all potential queries $q^v(k, E)$, we optimize over all potential query configurations (k, l, v). That is, we

do not optimize directly for the exclude list but rather for its size l. Once we decide on l, the exclude list E is constructed following a randomized approach, where l of the retrieved entities are included in the list uniformly at random. Below, we denote a query configuration (k, l, v) and a query $q^v(k, E)$ with $|E| = l$ using the same symbol q for convenience. We now need to estimate the gain for a configuration (k, l, v) instead of a query $q^v(k, E)$. When $l = 0$, i.e., $E = \emptyset$, then we use the techniques introduced in Section 4 directly. When $l \neq 0$, we generate multiple instances of $q^v(k, E)$ queries using the configuration (k, l, v) and use the techniques from Section 4 to estimate the expected gain of each. The exclude list of each query is generated following the randomized approach described above. Finally, we estimate the expected gain of configuration (k, l, v) by considering the average gain of all generated instances. The variance of the gain is also used to compute an upper bound on the gain of the configuration (k, l, v) as described next.

5.2 Balancing Exploration and Exploitation

The estimate of the expected gain of a query configuration (k, l, v) is based on a rather small sample of the underlying population—exploiting this information at every round may lead to suboptimal decisions. Hence, we need to balance the trade-off between exploiting configurations with high estimated gain and those that have not been selected many times. Formally, the latter corresponds to upper-bounding the expected gain of each configuration with a *confidence interval* that depends on the variance of the expected gain and the number of times a query was issued [4].

Next, we use q to denote a query configuration (k, l, v). Let $r(q)$ denote the expected gain of q. This is an estimate of the true gain $r^*(q)$. Let $\sigma(q)$ be an error component on the gain of configuration q chosen such that $r(q) - \sigma(q) \leq r^*(q) \leq r(q) + \sigma(q)$ with high probability. The parameter $\sigma(q)$ should take into account both the **empirical variance** of the expected gain and our **uncertainty** about the gain of query configuration q if its has only been chosen a few times. Given $r(q)$ and $\sigma(q)$, we assign a score to each configuration by using a linear function of the quantity $r(q) + \sigma(q)$. This score prioritizes exploration when the variance or our uncertainty is high, and thus, we could potentially discover a profitable new configuration, and exploitation when the estimated gain is high. Next, we discuss how we set $\sigma(q)$.

We proceed in rounds and at each round select a query configuration q. Let $n_{q,t}$ be the number of times we have chosen q by round t, and $v_{q,t}$ be the maximum value between some constant (e.g., 0.01) and the empirical variance for the gain for q at round t. The latter can be computed via bootstrapping over the retrieved sample and applying the estimators from Section 4 over the bootstrapped samples. Several techniques have been proposed in the multi-armed bandits literature to compute parameter $\sigma(q)$ [24], that we can reuse.

5.3 A Multi-Round Querying Policy Algorithm

We now introduce a multi-round algorithm for solving the budgeted entity enumeration problem (Algorithm 1). It takes as input the poset $\mathcal{H}_D$, a set K of query size assignments, a set L of exclude list size assignments and a budget τ_c. The algorithm also assumes access to an oracle providing $r(q)$ and $\sigma(q, t) - t$ denotes the round count — which characterize the upper gain of q, and an oracle providing the query cost $c(q)$. The algorithm has also access to method ActiveQueryConf (see Section 5.4) returning the allowed configurations per round.

Algorithm 1 Multi-round Extraction Algorithm

1: **Input:** $\mathcal{H}_D$: the hierarchy describing the entity domain; K: set of valid query size assignments; L: set of valid exclude-list size assignments; τ_c: budget; r, σ: value oracle access to upper-bounded gain for different query configurations; c: value oracle access to the query configuration costs;
2: **Output:** $\mathcal{E}$: a set of extracted distinct entities;
3: $\mathcal{E} \leftarrow \{\}$
4: $t \leftarrow 1$ /* Initialize round counter */
5: Rembudget $\leftarrow \tau_c$ /* Initialize remaining budget */
6: $Q \leftarrow$ ActiveQueryConf($\mathcal{H}_D$, $\emptyset$, NULL, K, L, t) /* Initialize active query configurations */
7: **while** Rembudget > 0 and $Q \neq \{\}$ **do**
8: $q^* \leftarrow \arg\max_{q \in Q} \frac{r(q)+\sigma(q,t)}{c(q)}$ s.t. Rembudget $- c(q^*) > 0$
9: **if** q^* is NULL **then**
10: break;
11: Rembudget $\leftarrow$ Rembudget $- c(q^*)$ /* Update budget */
12: Given configuration $q^* = (k^*, l^*, v^*)$ generate an exclude list E such that $|E| = l^*$;
13: Issue query $q^{v^*}(k^*, E)$
14: $E \leftarrow$ entities extracted by query $q^{v^*}(k^*, E)$
15: $\mathcal{E} \leftarrow \mathcal{E} \cup E$ /* Update unique entities */
16: $Q \leftarrow$ ActiveQueryConf($\mathcal{H}_D$, Q, v^*, K, L, t) /* Update active queries */
17: $t \leftarrow t + 1$ /* Increase round counter */
18: **return** $\mathcal{E}$

Algorithm 2 ActiveQueryConf

1: **Input:** $\mathcal{H}_D$: the hierarchy describing the entity domain; Q_{old}: the running set of active query configurations; v^*: the node in $\mathcal{H}_D$ associated with the last query; K: set of valid query size assignments; L: set of valid exclude-list size assignments; t: running round counter; r, σ: value oracle access to upper-bounded gain for different query configurations;
2: **Output:** Q_{new}: the updated set of active query configurations;
3: **if** Q_{old} is empty **then**
4: /* **Initialize Set of Active Query Configurations** */
5: /* Populate Q_{new} with all possible (k, l) configuration for the root of $\mathcal{H}_D$. */
6: $Q_{new} \leftarrow \bigcup_{k \in K \times l \in L} \{(k, l, root)\}$
7: **else**
8: /* **Extend Set of Active Query Configurations** */
9: $Q_{new} \leftarrow S_{old}$
10: **for all** $d \in$ Set of Direct Descendant Nodes of v^* **do**
11: $Q_{new} \leftarrow Q_{new} \cup \bigcup_{k \in K \times l \in L} \{(k, l, d)\}$
12: /* **Remove Bad Query Configurations***/
13: /* Find maximum lower-bounded gain over all q in Q_{new}*/
14: $\psi \leftarrow \max_{q' \in Q_{new}} (r(q') - \sigma(q', t))$
15: $\mathcal{B} \leftarrow$ All configurations q in Q_{new} with $r(q) + \sigma(q, t) < \psi$
16: $Q_{new} \leftarrow Q_{new} \setminus \mathcal{B}$
17: **return** Q_{new}

Algorithm 1 proceeds as follows: First it initializes the set $\mathcal{E}$ of extracted entities, the round count t, the remaining budget Rembudget and the set of candidate queries Q (i.e., query configurations to be considered) at the first round (Ln.3-6). Then it proceeds in rounds and iteratively selects one query configuration to be used until the total budget is utilized or the set of candidate queries is empty (Ln. 7). At each round our algorithm performs the following steps. It first detects a configuration in Q that maximizes the score quantity $\frac{r(q)+\sigma(q,t)}{c(q)}$ under the constraint that the cost $c(q)$ is less or equal to the remaining budget (Ln. 8). If no such configuration exists the algorithm terminates (Ln. 9-10). Otherwise, the algorithm proceeds by executing a query corresponding to the selected configuration q^*, updates the set of extracted entities, the remaining budget, the set of active queries, and the round counter

(Ln. 11 - 17). Finally, Algorithm 1 returns the set of distinct entities extracted (Ln. 18).

5.4 Updating the Set of Actions

We now introduce an algorithm to effectively limit the number of query configurations considered at each round of Algorithm 1. To do so we exploit the structure of poset $\mathcal{H}_D$ and the gain estimates for different configurations $q = (k, l, v)$. Let Q denote the set of candidate query configurations. We propose an algorithm that **adds** configurations to Q by traversing the input poset in a top-down manner. These configurations (k, l, v) correspond to nodes v that are *direct descendants* of already queried nodes.

To limit the number of candidate configurations, we also **remove** any *bad query configurations* from Q. A configuration q is defined to be bad at round t when $r(q)+\sigma(q,t) < \max_{q' \in Q}(r(q')-\sigma(q',t))$. Intuitively, we do not need to consider a configuration as long as there exists another such that the upper-bounded gain of the former is lower than the lower-bounded gain of the latter. This technique is also adopted in multi-armed bandits to limit the number of actions considered [10].

Our algorithm for determining the set of active queries for each round is shown in Algorithm 2. It proceeds as follows: If the running set of active queries is empty (Ln. 3), i.e., if we have issued no queries previously, the set of active query configurations is initialized to contain all potential configurations (k, l) for the root of the poset $\mathcal{H}_D$ (Ln. 6). Otherwise, the algorithm first adds all configurations for the direct descendants of node v^* to Q_{old} (Ln. 9 - 11) and then removes any bad configurations from the new set Q_{new} (Ln.13-16). The algorithm terminates by returning the set Q_{new}.

6 EXPERIMENTAL EVALUATION

We present an empirical evaluation of CRUX on both real and synthetic datasets. The evaluation is performed on an Intel Core i7 3.7 GHz 32GB machine; algorithms are implemented in Python 2.7.

6.1 Experimental Setup

Gain Estimators. We evaluate the gain estimators below:

- **Chao92Shen**, combines the methodology proposed by Chao [5] with Shen's formula, i.e., Equation (1).
- **HwangShen**, combines the regression-based approach proposed by Hwang and Shen [12] with Shen's formula.
- **NewRegr**, our new technique proposed in Section 4.4.

All estimators are coupled with bootstrapping to estimate the gain variance to upper bound the return of a query as in Section 5.2.

Entity Extraction Algorithms. We evaluate the following algorithms for crowdsourced entity extraction:

- **Rand**, executes random queries until all the available budget is used. It selects a random node from $\mathcal{H}_D$ and a random query configuration (k, l) valid with the k, l input.
- **RandL**, same as Rand but only executes queries *at the leaf nodes* of $\mathcal{H}_D$ until all the available budget is used.
- **BFS**, performs a breadth-first traversal of $\mathcal{H}_D$, executing one query at each node. The query configuration is randomly selected from all valid combinations. Negative-answers are used to prune non-populated parts of $\mathcal{H}_D$.
- **RootChao**, corresponds to the entity extraction scheme of Trushkowsky et al. [25, 26] that uses Chao92Shen to measure the gain of an additional query. RootChao is agnostic to the structure of the input domain, thus, equivalent to issuing queries only at the root node of $\mathcal{H}_D$. Since the authors only propose a pay-as-you-go scheme, we coupled this algorithm with Alg. 1 to optimize for the input budget constraint. We allow the algorithm to consider different query configurations (k, l) but restrict the possible queries to the root node.
- **GSChao, GSWang, GSNewR**, are different variations of CRUX. These algorithms correspond to our proposed graph search querying policy algorithm (Section 5.3) coupled with Chao92Shen, HwangShen and NewRegr respectively.
- **GSExact**, is a near-optimal, omniscient baseline that allows us to see how far off our algorithms are from an algorithm with perfect information. We combine the algorithm proposed in Section 5.3 with an exact computation of the return or gains from queries.

For the results reported below, we run each algorithm ten times and report the average gain achieved under the given budget.

Querying Interface. For all datasets we consider generalized queries $q^v(k, E)$. The nodes v are set based on the input poset and (k, l) takes values in $\{(5, 0), (10, 0), (20, 0), (5, 2), (10, 5), (20, 5), (20, 10)\}$. The cost of each query is computed using an additive model over three terms that depend on the characteristics of the query. The cost terms are: (i) CostK that depends on the number of responses k requested from a user, (ii) CostL that depends on the size of the exclude list l in the query, and (iii) CostSpec that depends on the *specificity* of the query q_s. We define the specificity of a query to be equal to the number of attributes assigned non-wildcard values for the node $u \in \mathcal{H}_D$ the query corresponds to.

We consider two types of cost functions. The first is a linear function where the overall cost for a query with configuration (k, l) with specificity s is:$Cost(q) = \alpha \cdot \frac{k}{\text{max. query size}} + \beta \cdot \frac{l}{\text{max. ex. list size}} + \gamma \cdot \frac{s}{\text{max. specificity}}$. The cost of a query should be significantly increased when an exclude list is used, thus, β should be set to a larger value than α and γ. Different (α, β, γ) configurations were tested. We also considered a step cost function CostK + CostL, where CostK and CostL are set as follows: $(k, \text{CostK}) = \{(5, 0.20), (10, 0.60), (20, 0.80)\}$ and $(l, \text{CostL}) = \{(0, 0), (2, 0.10), (5, 0.50), (10, 0.70)\}$. We observed that the *relative performance* of the extraction algorithms for different cost functions was the similar. Below, we mention the cost function we used for each experiment we report.

Data. We use two datasets, Eventbrite and PeopleInNews, where the entities in the first correspond to events while the entities in the second to people. Eventbrite is a large real-world dataset where responses from workers are simulated using real-world events, while PeopleInNews is a small real-world dataset where responses from workers are obtained via Amazon's Mechanical Turk (AMT). Next, we describe each dataset in detail.

The first dataset was extracted from Eventbrite (Section 1). We collected a dataset spanning events of 19 different types, such as rallies, tournaments, conferences, and conventions, over the months of October and November 2014. The dataset has three dimensions: (i) event type, (ii) location, and (iii) time, with location and time being hierarchically structured. The poset of the domain can be fully specified if we consider the cross product across the possible

values for location, event type and time. For each of the location, time, type dimensions we also consider a special *wildcard* value. The poset has 8,508,160 nodes and 57,805 distinct events overall. Only 175,068 nodes are populated leading to a rather sparsely populated domain. Due to lack of popularity proxies for the extracted events, we assigned a *popularity value* in (0, 10] to each event.

To construct the second dataset, we used Amazon's Mechanical Turk to extract people "in the news", like in Example 1.1. We asked workers to extract the names of people belonging to four different occupations from five different news portals. We issued 20 HITS for each leaf node of the domain's poset, resulting in 600 HITS in total, and 1,245 unique people in total. The popularity value of each extracted entity was assigned to be equal to the number of distinct workers that reported it.

6.2 Experimental Results

We evaluate different aspects of the extraction techniques.

How does CRUX compare against baselines? We evaluated the performance using the total entities extracted for different budgets. The results for Eventbrite and PeopleInNews are shown in Figure 3(a) and Figure 3(b) respectively. As shown, CRUX, i.e., GSChao, GSHwang, GSNewR, *outperforms all baselines by at least 30% across both datasets.* This is expected as CRUX not only exploits the structure of the domain to diversify entity extraction and target even less-popular entities but also optimizes the queries for the given budget. Against the naive baselines Rand, RandL, and BFS, we see that GSChao, GSHwang and GSNewR extracted more than 200% more entities for the sparse Eventbrite domain and around 100% more entities for small budgets and 54% for larger ones when considering the dense PeopleInNews. For Eventbrite and a budget of 50 all CRUX schemes extracted more than 600 events while Rand and RandL extracted 1.1 and 0.2 events and BFS extracted 207.7 events, an improvement of over 180%.

Against RootChao, we see that GSChao, GSHwang and GSNewR were able to retrieve up to 30% more entities for Eventbrite and 400% more entities for the PeopleInNews dataset. This difference is because the gain achieved by RootChao saturates at a faster rate than GSChao, GSHwang and GSNewR as the cost increases. This is because RootChao issues queries at the root of the poset, and hence, it is not able to extract entities belonging to the long tail. For PeopleInNews, RootChao performs poorly even compared to the naive baselines Rand, RandL and BFS. Again, this is due to the skew of the popularity distribution.

How does CRUX compare against a near-optimal policy discovery algorithm? We evaluated the different variations of CRUX, i.e., GSChao, GSHwang and GSNewR, against the near-optimal querying policy discovery algorithm GSExact. The results can be found in our technical report [20]. In short, we find that our proposed techniques are not too far away from GSExact which has perfect information about the gain of each query; for example, for Eventbrite, the difference in extractions was less than 25%.

How do the different techniques compare with respect to the total number of queries issued during extraction? We compared RootChao against the CRUX algorithms GSChao, GSHwang and GSNewR with respect to the *total number of queries issued during extraction.* This evaluation metric is a surrogate for the overall latency of the crowd-extraction process. Figure 4 shows the results for a run for Eventbrite and a budget of 80. As shown, *RootChao requires almost up to 200% more queries* to extract the same number of entities as our techniques, thus, exhibiting significantly larger latency compared to GSChao, GSHwang and GSNewR.

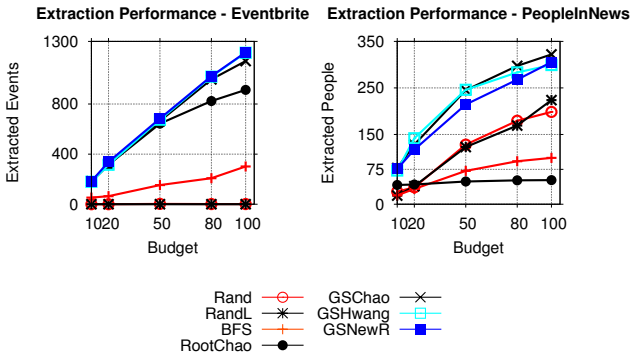

Figure 3: A comparison of the CRUX techniques against baselines for (a) Eventbrite and (b) PeopleInNews.

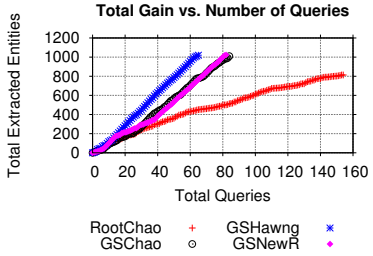

Figure 4: The number of events extracted by different algorithms for Eventbrite versus the total number of queries.

How do different CRUX versions traverse the poset and use query configurations? We first measure how many queries each algorithm issues at various levels of the poset. In Figure 5, we plot the number of queries (per level) issued by our algorithms when the budget is set to 10 and 100 respectively for 10 runs of PeopleInNews. For a small budget, all algorithms prefer queries at higher levels. The inner nodes are preferred and only a small number of queries is issued at the root (i.e., level one) of the poset. This is justified if we consider that due to their popularity, certain entities are repeatedly extracted, thus, leading to a lower gain. As the budget increases, we see that all algorithms tend to consider more specialized queries at deeper levels of the poset. It is interesting to observe that all algorithms issue the majority of their queries at level two nodes, while GSExact, which has perfect information, focuses mostly on leaf nodes. Thus, our techniques could benefit from being more aggressive at traversing the poset and reaching deeper levels; overall, our techniques may end up being more conservative to cater to a larger space of posets and popularity distributions.

In Figure 6, we plot the query configurations chosen by our algorithms when the budget is set to 10 and 100. GSExact always prefers queries with $k = 20$ and $l = 0$ for both small and large budgets. On the other hand, our algorithms issue more queries of smaller size when operating under a limited budget and prefer queries of larger size for larger budgets. GSNewR was the only one issuing queries with exclude lists of different sizes, thus, exploiting the rich diversity of query interfaces.

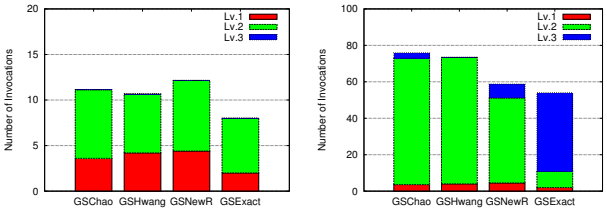

Figure 5: The number of queries issued at different levels when budget is set at 10 or 100.

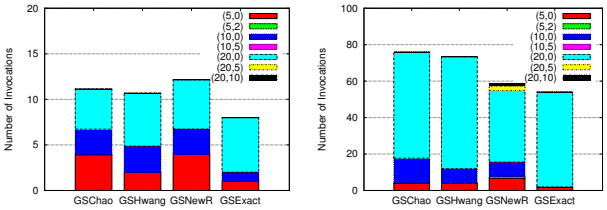

Figure 6: The query configurations used when budget is set at 10 or 100.

How effective are different estimators at predicting the gain of additional queries? GSNewR was able to outperform GSChao and GSHwang for Eventbrite but the opposite behavior was observed for PeopleInNews. To understand the relative performance of the estimators, we measure their error at predicting the number of new retrieved entities for different query configurations for both Eventbrite and PeopleInNews. In summary, for large and sparse domains, NewRegr slightly outperforms Chao92Shen and HwangShen for certain queries. For example, for $k = 10, l = 5$, Chao92Shen has a relative error of 0.58, HwangShen had a relative error of 0.7, and NewRegr had a relative error of 0.29. For smaller, dense domains, such as PeopleInNews, NewRegr offers better gain estimates for small query sizes, but as the query size increases, hence, a larger portion of the population is observed, Chao92Shen outperforms both regression-based techniques. Detailed results can be found in our technical report [20].

7 RELATED WORK

Prior work related to the techniques proposed in this paper can be placed in a few categories; we describe each of them in turn: We have already discussed prior work on crowdsourced extraction or enumeration [18, 25] in the introduction.

Knowledge Acquisition Systems. Recent work has also considered the problem of using crowdsourcing within knowledge acquisition systems [13, 15, 27]. This line of work suggests using the crowd for curating knowledge bases and for gathering additional information to be added to the knowledge base, instead of augmenting the set of entities themselves.

Deep Web Crawling. A different line of work has focused on data extraction from the deep web [14, 22] where data is obtained by querying a form-based interface over a hidden database and extracting results. Sheng et al. [22] provide near-optimal algorithms that exploit the exposed structure of the underlying domain to extract all the tuples present in the hidden database. Our goal is similar in that we also extract entities via a collection of interfaces. Unlike our setting, answers from a hidden database are deterministic, i.e., a query will always retrieve the same top-k tuples. So, it suffices to ask each query precisely once, making it much simpler.

8 CONCLUSION

We studied the problem of crowdsourced entity extraction over large and diverse data domains. We proved that the problem of budgeted crowdsourced entity extraction is NP-hard. We introduced CRUX, a novel crowdsourced entity extraction framework, that combines statistical techniques with an adaptive optimization algorithm to maximize the total number of unique entities extracted. We proposed a new regression-based technique for estimating the gain of further querying when the number of retrieved entities is small with respect to the total size of the underlying population. We also introduced a new algorithm that exploits the often known structure of the underlying data domain to devise adaptive querying strategies. CRUX extracts up to 300% more entities compared to baselines, and for large sparse entity domains is at most 25% away from an omniscient adaptive querying strategy.

Acknowledgements. We acknowledge support from grant IIS-121-8367, IIS-1652750, and IIS-1815538 awarded by the NSF and grant W911NF-18-1-0335 awarded by the Army.

REFERENCES

[1] Mechanical Turk. *http://mturk.com.*
[2] Amazon Product Categories, http://services.amazon.com/services/soa-approval-category.htm . 2015.
[3] Y. Amsterdamer et al. OASSIS: query driven crowd mining. SIGMOD, 2014.
[4] P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *JMLR*, 3, 2003.
[5] A. Chao and S. M. Lee. Estimating the Number of Classes via Sample Coverage. *JASA*, 87(417):210–217, 1992.
[6] T. Cheng, H. W. Lauw, and S. Paparizos. Fuzzy matching of web queries to structured data. In *ICDE*, 2010.
[7] N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu. A web of concepts (keynote). In *PODS*, 2009.
[8] O. Deshpande et al. Building, maintaining, and using knowledge bases: A report from the trenches. In *SIGMOD*, 2013.
[9] P. Donmez, J. G. Carbonell, and J. Schneider. Efficiently learning the accuracy of labeling sources for selective sampling. In *KDD*, 2009.
[10] E. Even-Dar et al. Action elimination and stopping conditions for the multi-armed bandit and reinforcement learning problems. *JMLR*, 7, 2006.
[11] L. Getoor and A. Machanavajjhala. Entity resolution in big data. In *KDD*, 2013.
[12] W.-H. Hwang and T.-J. Shen. Small-sample estimation of species richness applied to forest communities. *Biometrics*, 66(4), 2010.
[13] L. Jiang, Y. Wang, J. Hoffart, and G. Weikum. Crowdsourced entity markup. In *CrowdSem*, 2013.
[14] X. Jin, N. Zhang, and G. Das. Attribute domain discovery for hidden web databases. SIGMOD, 2011.
[15] S. K. Kondredi et al. Combining information extraction and human computing for crowdsourced knowledge acquisition. In *ICDE*, 2014.
[16] D. Lan, K. Reed, A. Shin, and B. Trushkowsky. Dynamic filter: Adaptive query processing with the crowd. In *HCOMP' 17*, pages 118–127, 2017.
[17] A. Marcus and A. Parameswaran. Crowdsourced data management: Industry and academic perspectives. *Foundations and Trends in DB*, 6(1-2):1–161, 2015.
[18] H. Park and J. Widom. Crowdfill: A system for collecting structured data from the crowd. In *WWW*, 2014.
[19] A. J. Quinn and B. B. Bederson. Asksheet: Efficient human computation for decision making with spreadsheets. CSCW, 2014.
[20] T. Rekatsinas, A. Deshpande, and A. Parameswaran. Technical report, https://arxiv.org/abs/1502.06823. 2019.
[21] T. Shen, A. Chao, and C. Lin. Predicting the number of new species in further taxonomic sampling. *Ecology*, 84(3), 2003.
[22] C. Sheng, N. Zhang, Y. Tao, and X. Jin. Optimal algorithms for crawling a hidden database in the web. *PVLDB*, 5(11):1112–1123, July 2012.
[23] A. Singhal. Introducing the knowledge graph: things, not strings. *Official Google Blog, May*, 2012.
[24] O. Teytaud, S. Gelly, and M. Sebag. Anytime many-armed bandits. In *CAP*, 2007.
[25] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. ICDE, pages 673–684, 2013.
[26] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Answering enumeration queries with the crowd. *Commun. ACM*, 59(1):118–127, 2016.
[27] R. West, E. Gabrilovich, K. Murphy, S. Sun, R. Gupta, and D. Lin. Knowledge base completion via search-based question answering. WWW, 2014.