Interactive Set Discovery

Arif Hasnat
University of Alberta
Edmonton, Canada
hasnat@ualberta.ca

Davood Rafiei
University of Alberta
Edmonton, Canada
drafiei@ualberta.ca

ABSTRACT

We study the problem of set discovery where given a few example tuples of a desired set, we want to find the set in a closed collection of sets. A challenge is that the example tuples may not uniquely identify a set, and a large number of candidate sets may be returned. Our focus is on interactive exploration to set discovery where additional example tuples from the candidate sets are shown and the user either accepts or rejects them as members of the target set. The goal is to find the target set with the least number of user interactions. The problem is cast as an optimization problem where we want to find a decision tree that can guide the search to the target set with the least number of questions to be answered by the user. We propose a general algorithm, capable of reaching an optimal solution, and two variations of it that strike a balance between the quality of a solution and the running time. We also propose a novel pruning strategy that safely reduces the search space without introducing false negatives. Our extensive evaluation on both real and synthetic data show that our approach is effective, comparable to or improving upon SOTA, while our pruning strategy reduces the running time of the search algorithms by 2-5 orders of magnitude.

1 INTRODUCTION

Consider a patient walking to a clinic and being greeted by a machine who does the triage. The patient types headache, nausea and fatigue as symptoms, and the machine checks its database of disease cases and finds over thousands matching each symptom and over hundreds matching all three. What are the best ways of narrowing down the cases? What are the next few questions the machine asks?

Many database interfaces behave in a similar fashion in that there is a large collection of sets and the user is searching for a particular set in the collection. In SQL interfaces, in particular, queries express propositions over *sets of tuples* where the grouping of tuples into sets (e.g. customers who live in Toronto vs those who do not) is *inherent* in queries, and the user is forced to precisely express all propositions in a query to describe the target set. However, writing SQL queries is a challenging task for many do-it-yourself scientists and professionals. For example, both the Sloan Digital Sky Survey Project [1] and the SQL-Share Project [2, 16, 18] allow scientists to query their data using SQL, but not many scientists using these projects are expected to know SQL.

In example-based query interfaces [24, 27, 34], the user provides a set of tuples that are expected to be in the target set (as positive examples) and a set of tuples that are not (as negative examples), and a query that satisfies the given constraints is suggested. The user may adjust the query or provide additional tuples, as positive or negative examples, in an interactive fashion until a precise query expression describing the target set is found.

In browsing-based interfaces, a repository of queries are maintained and the users can browse the repository for similar queries that can be reused with small or no changes [20, 37]. For example, both the Sloan Digital Sky Survey and the SQL-Share projects keep popular user queries and support searches over those queries. Similarly, in query recommendation, past queries that are recorded in a query log may be recommended as a whole or in part, based on the query fragments that are typed [10, 25]. If each query in the repository is treated as a set of tuples (e.g. the result of the query applied to a database instance), then the recommendation engine is searching for a set in the collection.

In all aforementioned cases, there is a collection of sets or queries and the user is searching for a particular set in the collection. We study an interactive exploration approach to set discovery where example tuples from the candidate sets are shown, and the user either accepts or rejects those tuples as members of the target set.

The number of interactions does not depend on the number of tuples in the database and is only a function of the number of sets (e.g. see Figures 6 and 7). In many cases, the number of sets that are similar to a target set is expected to be small. For example, Zhong et al. [38] show that generating 94 neighbour queries on average for a target SQL query is sufficient to distinguish queries that are different from the target query. If k denotes the number of sets that are similar to a target set, the number of interactions is $k-1$ in the worst cases and closer to $logk$ in most cases. As each interaction with the user has a cost, we want to retrieve the target set with the least number of interactions. Relevant research questions are: (1) what exploration strategies may be used, and how efficient are those strategies? (2) how long does an exploration take and what factors (e.g., set sizes, overlaps, etc.) do affect the exploration time? (3) how may the sets be organized to support an efficient exploration?

Problem statement Informally, set discovery is an interactive process that starts with an initial membership question posed to the user and continues with follow-up questions based on the user's answers. The problem is how to select the next question such that the number of interactions is minimized. More formally, given a collection C of unique sets and an initial set I, which includes a subset of the user's desired set, the goal is to find a target set G in C such that $I \subseteq G$. We want to narrow down the search through interactions, i.e. asking the user membership questions about example tuples from C, and we want to find the target set with the minimal number of interactions. With no user interaction, the problem is under-specified and more than one such set G can contain the elements of I unless $G = I$, in which case a search is meaningless since the user has listed the full target set. Also, when I is an empty set, then G is fully identified through interactions with the user.

Our approach We cast the problem as an optimization with the aim of minimizing the number of questions that the user needs to answer. The search for a target set is modeled as a tree traversal from the root to a leaf, with each node representing a step of the exploration where the user is given a question that can reduce the number of candidate sets. The problem of optimal decision tree construction is NP-hard, and there are strong results on the non-approximability of the problem (see Sec. 2). Our work improves upon a SOTA approximate algorithm, in terms of the quality of the tree, while significantly reducing the size of the search space using strongly effective pruning strategies. The tree construction is done *online* in an incremental fashion with questions answered; we also discuss a strategy which aims to reduce the online search cost with an *offline* tree construction. Assuming all candidate sets in C being equally likely to be the target set G, we consider two exploration scenarios: (1) *average-case* where the average number of questions over all possible target sets is minimized, and (2) *worst-case* where the maximum number of questions over all possible target sets is minimized. Our algorithms are general and work under both exploration scenarios.

Contributions Our contributions can be summarized as follows:

- We formalize interactive set discovery as an optimization problem, minimizing the number of questions posed to users.
- We propose cost functions to characterize the quality of a decision tree for interactive set discovery, in terms of its worst-case and average-case performance, and some lower bounds that are easy to compute but effective in pruning the search space.
- We propose a pruning strategy, based on our lower bounds, that allows certain choices of entities for decision tree nodes to be safely rejected if there is evidence that it cannot lead to a better tree than one already found.
- Based on our pruning strategy, we develop an efficient lookahead algorithm that can find near-optimal trees in many cases. We also develop two variations of our lookahead algorithm to further speed up the search process by bounding the number of entities in each step of the search.
- Through an extensive experimental evaluation, we show that our pruning strategy is effective, reducing the running time by a few orders of magnitude, and that our algorithms outperform competitive approaches from the literature.

The rest of the paper is organized as follows. We review the related work in Section 2 and present the problem and our formulation in Section 3. Our algorithms and strategies are discussed in Section 4, and in Section 5, we experimentally evaluate their performance. Finally, we conclude with a discussion in Section 6 and provide remarks for future work in Section 7.

2 RELATED WORK

Our work is related to the lines of work on (a) example-based query discovery, (b) active learning and interactive query discovery, and (c) cost-efficient decision tree construction.

2.1 Example-based query discovery

Our work is related to this line of work in that it can be applied to discover target queries based on example tuples if the candidate queries are known or can be enumerated. The problem of discovering queries based on examples has its root in QBE [39] and has been lately studied for reverse engineering queries in various domains (e.g., relational [32] and graph data [6, 26]). On discovering SQL queries, in particular, Tran et al. [33] study the problem for select-project queries, and others study project-join queries [19, 36]. Weiss et al. [34] show that the problem of discovering SPJ from examples is NP-hard when there is a bound on the size of queries. An underlying assumption in many of these works is that the queries can be discovered on small instances (where the answer tuples can be easily listed) before being applied to larger instances. Unlike the aforementioned works that focus on a specific query type to keep the complexity of query generation under control, there is no such restriction in our work. Our query discovery is done based on the query output on a sample database, hence any pair of different queries must return different results on the sample instance to be distinguishable. Also we assume the set of queries are given or can be enumerated. There is no other constraint on queries and their complexity. The approaches on reverse engineering queries are limited in terms of the type of queries they can produce, and these approaches are not applicable when the target is a set and not a query.

2.2 Active learning and interactive query discovery

Related work includes active learning [30], where a model is learned by interacting with a user, and interactive exploration to learn a desired query. Angluin [5] shows the polynomial learnability of conjunctions of horn clauses and Abouzied et al. [3] show that efficient solutions for a subset of quantified Boolean queries, referred to as role-preserving qhorn queries, are reachable. In both cases, users specify propositions that hold by answering membership questions (e.g. a row is or is not in the answer) and this helps to narrow down the search for candidate queries. Bonifati et al. [8, 9] infer join queries and Dimitriadou et al. [11] predict conjunctive queries, both based on interactions in the form of simple yes/no answers about the presence of tuples in the final output. Li et al. [24] take a sample database and a desired result and generate candidate SPJ queries that produce the result on the sample, using the approach of Tran et al. [33]. In each follow-up interaction, the user is provided with a modified database and a collection of query results to choose from, based on which candidate queries are removed until a query emerges.

In all aforementioned works, the search takes place over the space of possible queries that can be generated, and the size of this space is bounded by placing constraints on the the shape of queries. For example, this space in Abouzied et al. [3] is the cartesian product of all domains, which means with m Boolean variables, there are 3^m possible assignments of constants and don't-care values to those variables and that many propositional logic queries. This space in Bonifati et al. [8, 9] is all subsets of the Cartesian product of the two tables being joined while limiting the predicates to equijoin, in Dimitriadou et al. [11] is the set of rectangular regions defined by the conjunction of range predicates on numerical data and in Li et al. [24] is a set of conjunctive queries. In our case, the search takes place over a closed collection of sets, and there is no constraint on the shape of queries that may generate those sets. For example, the sets in one of our datasets are generated using SQL queries with CNF formulas in the where clause and in another dataset by performing union over arbitrary sets. Also unlike active learning where the classes are not explicitly known or not enumerated and achieving 100% accuracy is out of reach, in our case the sets to be discovered and their boundaries and relationships in terms of overlaps are fully known.

2.3 Cost-efficient decision tree construction

There are some strong results on the nonapproximability of the problem. Sieling [31] shows that the problem cannot be approximated up to any constant factor, based on the nonapproximability of Vertex Cover for Cubic graphs and that the problem can be mapped to an optimal decision tree construction. In a much stronger result, Dinue and Steurer [12] show that optimal set cover cannot be approximated to $(1 - o(1))lnn$ unless $P = NP$, and the same result holds for optimal decision tree construction based on a reduction from set cover [23]. Adler et al. [4] propose a greedy algorithm which achieves $(\ln n + 1)$-approximation, by simply choosing an entity at each decision node that most evenly partitions the collection of items. This greedy algorithm sets a strong baseline in terms of the approximability of the problem, and many commonly-used approaches (e.g. Information Gain [28], ID3 [29] and C4.5 [28]) are all variations of this 1-step lookahead greedy algorithm (see Sec. 4.2). Esmeir et al. [14] propose lookahead based algorithms for anytime induction of decision trees by developing k-steps entropy and information gain. Our proposed algorithm for set discovery improves upon the 1-step lookahead approaches, which are pretty strong baselines, but is 2 to 5 orders of magnitude faster than the k-steps of Esmeir et al., thanks to our powerful pruning strategies.

3 PROBLEM FORMULATION

Consider a collection of n candidate sets and a target set in the collection that needs to be identified. Without loss of generality, we assume the sets are all unique; if not, duplicates can be removed without affecting the search task. We want to find the target set through a set of membership questions that the user answers (e.g., Is A in the target set?). At a high level, we want to minimize the number of interactions.

A general approach to the search problem is to construct a decision tree with the candidate sets placed at the leaves and each internal node representing a question. With interactions limited to yes/no membership questions, the decision tree will be a full binary tree with n leaves and $n - 1$ internal nodes. The number of such decision trees that can be constructed is huge[1], and some of those trees are more efficient for finding the target set than others.

Let m denote the size of the universe from which the sets are drawn. For a collection C of finite sets, $m = |\bigcup_{s \in C} s|$. In our presentation, we may refer to the members of the universe as entities, though our approach is applicable to any sets of tuples (e.g., sets of relationships). For a fixed tree shape with $n - 1$ internal nodes, the number of possible placements of m entities or tuples on internal nodes will be $m(m-1)\dots(m-n+2) = \frac{m!}{(m-n+1)!}$, assuming that each entity appears at most once in the tree. Otherwise, this number is even larger. Searching for an efficient decision tree among all these tree shapes and possible placements of entities on internal nodes is a major computational challenge, and that is the problem studied in this paper.

To alleviate the problem, one may group entities in C into *informative* and *uninformative*. An entity that is either present in all sets in C or none is not informative, since a membership question about that entity does not reduce the search space. The rest of the entities can be considered as informative. Clearly we want to limit our questions to informative entities, and only place those entities on the internal nodes.

[1]The actual number is the $(n-1)$th Catalan number, i.e., $\frac{1}{n}\binom{2(n-1)}{n-1} = \frac{(2(n-1))!}{n!(n-1)!}$.

Example 3.1. Consider the collection of seven sets, as shown in Fig. 1. Entity a is uninformative since it is present in all sets. All the other entities $b, c, \dots, k$ are informative. Fig. 2 shows three possible decision trees that represent the sets in the collection. All the trees are full binary decision trees with 6 internal nodes and 7 leaves. The root node corresponds to all sets of the collection. In Fig. 2a, the left branch corresponds to the sub-collection $\{S1, S2, S3\}$ where entity d is present and the right branch corresponds to the sub-collection $\{S4, S5, S6, S7\}$ where d is not present. Each branch is further broken down based on the presence or absence of entities.

$$
\begin{array}{lll}
S1 = \{a, b, c, d\} & S2 = \{a, d, e\} & S3 = \{a, b, c, d, f\} \\
S4 = \{a, b, c, g, h\} & S5 = \{a, b, h, i\} & S6 = \{a, b, j, k\} \\
S7 = \{a, b, g\} & &
\end{array}
$$

Figure 1: A collection of example sets

Given a decision tree, the number of questions that are required to find a set is determined by the depth at which the set is placed. For example, in Fig. 2a, $S2$ can be detected using two questions whereas one will need three questions to find any other set. Since we do not know the target set in advance, and assuming that all sets are equally likely, the cost of a tree can be defined as the average depth of the leaves which equivalently represents the expected number of questions required to find the target set.

Definition 3.2. Let T be a full binary decision tree over a collection C of unique sets, i.e., T has exactly $|C|$ leaves and each leaf is labelled with a set in C. If *depth(s,T)* denote the depth of a set s in T, then the cost of T is defined as

$$cost(T) = \frac{\sum_{s \in C} depth(s, T)}{|C|}.$$

Alternatively, one can also define the cost of a tree as the height[2] of the tree. For a collection with n unique sets, the height of a full binary decision tree cannot be less than $\lceil \log_2 n \rceil$ for $n > 0$. This sets a lower bound on the height (H) of an optimal tree, which we refer to as $LB\_H(n)$. The next lemma gives the lower bound on the average depth of the leaves.

Lemma 3.3. *Given a collection of n unique sets such that n > 0, a lower bound on the average depth of the leaves (AD) of a full binary decision tree representing the collection, denoted as LB_AD(n), is* $\lceil n \log_2 n \rceil / n$.

Proof. The average depth of the leaf nodes of a full binary decision tree representing n sets cannot be less than $\log_2 n$. Hence, the sum of depth of the n leaf nodes cannot be less than $\lceil n \log_2 n \rceil$ since it must be an integer number. Therefore, a lower bound on AD for n unique sets is $\lceil n \log_2 n \rceil / n$. □

Now let's examine the trees in Fig. 2 again. A lower bound on AD of any full binary decision tree representing a collection of 7 sets, according to Lemma 3.3, is 2.857. The AD of the tree in Fig. 2a is 2.857, Fig. 2b is 3.0 and that of the tree in Fig. 2c is 3.857, hence the first tree is optimal but the others are not.

Given a collection of n unique sets, our goal can be stated as finding a full binary decision tree representation of the collection with the least cost where the cost metric is either AD or H.

[2]Here height refers to the depth of the leaf with the longest distance from the root, i.e., the number of questions to be answered to reach the deepest leaf.

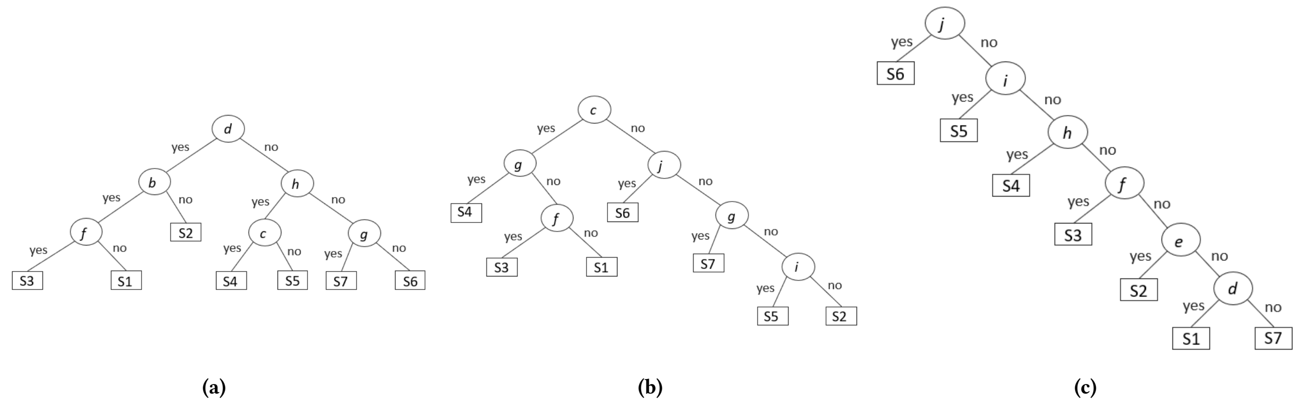


Figure 2: Example of decision tree representations of the sets in Figure 1

4 METHODOLOGY

Given a collection C of unique sets, our set discovery process constructs a decision tree with the sets in C placed at the leaves and the membership questions about entities placed at the internal nodes. Since there are many possible trees that can be constructed, and some are more efficient than others, our goal is to find a tree that leads to the least number of interactions with the user. Before presenting our algorithms, we develop a few lower bounds on cost, which will be used in pruning the search space of our algorithms.

4.1 Cost Lower Bounds

Given a collection C of unique sets, two lower bounds on cost (as discussed in Section 3) are

$$LB\_AD_0(C) = \frac{\lceil |C| * \log_2 |C| \rceil}{|C|}, and \tag{1}$$

$$LB\_H_0(C) = \lceil \log_2 |C| \rceil \tag{2}$$

for cost metrics AD and H respectively. Now consider an entity e that partitions C into two sub-collections $C1$ and $C2$. Our cost lower bounds, after placing e at the current node of the decision tree, can be written as

$$LB\_AD_1(C,e) = \frac{|C1| * LB\_AD_0(C1) + |C2| * LB\_AD_0(C2)}{|C|} + 1, \tag{3}$$

and

$$LB\_H_1(C,e) = max(LB\_H_0(C1), LB\_H_0(C2)) + 1 \tag{4}$$

for cost metrics AD and H respectively, where the index '1' in the bounds indicates that the cost is calculated using information available after looking one step ahead, i.e. one level below the current node. We use the general term LB to refer to any lower bound (including LB_AD and LB_H) when a distinction in the cost metric is not important.

Let E denotes the set of entities in collection C. A lower bound on cost over all entities with 1-step look ahead is

$$LB_1(C) = min_{e \epsilon E} LB_1(C,e). \tag{5}$$

These definitions can be extended for k-steps lookahead as

$$LB\_AD_k(C,e) = \frac{|C1| * LB\_AD_{k-1}(C1) + |C2| * LB\_AD_{k-1}(C2)}{|C|} + 1, \; and \tag{6}$$

$$LB\_H_k(C,e) = max(LB\_H_{k-1}(C1), LB\_H_{k-1}(C2)) + 1 \tag{7}$$

for cost metrics AD and H respectively. A lower bound over all entities is

$$LB_k(C) = min_{e \epsilon E} LB_k(C,e). \tag{8}$$

A desirable property of these lower bounds is their monotonicity and that the cost lower bounds never decrease. This means the lower bounds can get tighter but not looser, as we look more and more steps ahead. The next two lemmas state this more formally.

Lemma 4.1. *For any collection C, $LB_k(C)$ is a monotone non-decreasing function of k, i.e., for non-negative integers $k1$ and $k2$, if $k2 > k1$, then $LB_{k2}(C) \geq LB_{k1}(C)$.*

Proof. The proof is by induction on k. For the basis, the lowest possible cost of a binary decision tree on C, defined as $LB_0(C)$, is calculated assuming that the entity in each node of the tree partitions the sub-collection as evenly as possible. But, $LB_1(C)$ is calculated after an actual entity from the collection is assigned to the root node and assuming that all other nodes partitions the sub-collections as evenly as possible. If the entity at the root partitions the collection as evenly as possible then $LB_1(C)$ is equal to $LB_0(C)$. Otherwise, $LB_1(C)$ is greater than $LB_0(C)$. For the induction step, suppose the claim holds at step $k1$. In each additional step $k2 = k1 + 1$ of the lower bound calculation, an additional level of nodes are assigned with the best entities recursively. If any of those entities does not partition the corresponding sub-collections as evenly as possible then $LB_{k2}(C) > LB_{k1}(C)$, otherwise, $LB_{k2}(C) = LB_{k1}(C)$. Therefore, the statement holds. □

Lemma 4.2. *For any collection C and entity e in the collection, $LB_k(C,e)$ is a monotone non-decreasing function of k, i.e., for positive integers $k1$ and $k2$, if $k2 > k1$, then $LB_{k2}(C,e) \geq LB_{k1}(C,e)$.*

The proof follows the line of reasoning in Lemma 4.1.

4.2 Entity Selection

The problem of constructing an optimal binary decision tree, minimizing the cost to discover an unknown target set, is NP-complete [17], hence various greedy strategies have been studied in the literature. In this section, we briefly review these strategies and compare them with ours.

4.2.1 Most even partitioning. A greedy approximation algorithm which achieves $(\ln n + 1)$-approximation for the decision tree problem on a collection C with n sets is simply to choose an entity at each internal node that most evenly partitions the collection of sets in that node [4].

4.2.2 Information gain. Decision tree construction is a very well-understood process in machine learning and data mining. A popular heuristic used by the decision tree algorithms (such as ID3 [29] and C4.5 [28]) for selecting the next feature or entity is the information gain. The entity with the largest information gain is selected to split the collection. If we treat each set in C as a class and each distinct entity e as a feature, then the information gain of e that partitions C into sub-collections $C1$ and $C2$ can be written as

$$InfoGain(C,e) = \log_2 |C| - \frac{|C1| * \log_2 |C1| + |C2| * \log_2 |C2|}{|C|}. \tag{9}$$

4.2.3 Indistinguishable pairs. Another entity selection strategy (used by Roy et al. [7]) selects an attribute or entity that minimizes the number of indistinguishable pairs of sets. For an entity e that partitions a collection C into sub-collections $C1$ and $C2$, the number of indistinguishable pairs is given as

$$Indg(C,e) = \frac{|C1| * (|C1| - 1) + |C2| * (|C2| - 1)}{2}. \tag{10}$$

4.2.4 Cost lower bound. Entity selection can be done using our cost lower bound LB_k with $k > 0$, as discussed in Section 4.1, by selecting the entity that minimizes the cost lower bound. This is the strategy we use in this paper because of some of the desirable properties of those lower bounds. However, in some cases, two entities that do not partition a collection in the same way may have the same value of a lower bound. For example, suppose entity a partitions a collection of 16 sets into 9 and 7 sets, and entity b partitions the same collection into 10 and 6 sets. With $\lceil \log_2 9 \rceil = \lceil \log_2 10 \rceil = 4$, both entities will have the same value of the lower bound on height. When there are such ties, we select an entity that most evenly partitions the collection to differentiate between entities with the same value of cost lower bound.

Though these strategies seem different from each other, it can be shown that the existing strategies discussed above and our 1-step cost lower bound LB_1 select the same entity for the binary decision tree problem. Hence, they all achieve the same $(\ln n + 1)$-approximation factor.

Lemma 4.3. *Given a collection C, the strategies (a) information gain, (b) indistinguishable pairs, and (c) 1-step cost lower bound, LB_1, select the same entity that partitions C most evenly into two sub-collections.*

Proof. (a) In (9), since $|C|$ is constant and $|C1| + |C2| = |C|$, the quantity $|C1| * \log_2 |C1| + |C2| * \log_2 |C2|$ is minimum when C is most evenly partitioned into $C1$ and $C2$. Hence, the entity that partitions C most evenly has the largest information gain and is selected by *information gain* strategy.
(b) Similarly, in (10), $|C1| * (|C1| - 1) + |C2| * (|C2| - 1)$ is minimum when C is most evenly partitioned. Therefore, the entity that partitions C most evenly has the minimum value of Indg() and is selected by *indistinguishable pairs* strategy.
(c) It can also easily be seen from (3) and (4) (after replacing LB_AD_0 and LB_H_0 with their respective values from (1) and (2)) that, an entity that most evenly partitions the collection C into $C1$ and $C2$, gives the minimum value of LB_1 in (5), thus is selected by our *1-step cost lower bound* strategy. □

This paper builds on top of our cost lower bounds, and this offers a few benefits compared to other entity selection strategies. First, the cost functions are simple and intuitive, offering an easy choice between the average case and worse case costs. Second, we can develop efficient and effective k-steps lookahead strategies using LB_k with $k > 1$. In particular, we develop an effective pruning strategy that significantly reduces the search space and the runtime of our lookahead strategies without affecting the cost. Although k-steps lookahead strategies have been studied for entropy [8] and information gain [14], we are not aware of similar pruning strategies developed for these other measures.

4.3 Pruning

We want to find a decision tree that requires the least number of interactions with the user for a set discovery hence has the least cost. However, exhaustive searching the space of possible trees for the one with the least cost is computationally intensive and not always feasible. We propose a *pruning* strategy that allows certain choices of entities for decision tree nodes to be safely rejected to reduce the size of the search space without affecting the correctness.

Lemma 4.4. *Let $LB_k(C,e)$ denote our lower bound of cost for entity e in collection C by looking k-steps ahead, and suppose entity selection is done based on LB_k, k-steps cost lower bound for some k. Consider entities e_1 and e_2, both in C. If $LB_l(C,e_2) \geq LB_k(C,e_1)$ for $l \leq k$, then e_2 can be pruned without affecting the correctness of the search.*

Proof. Based on Lemmas 4.1 and 4.2 $LB_k(C,e_2)$ cannot be smaller than $LB_k(C,e_1)$ when $LB_l(C,e_2) \geq LB_k(C,e_1)$ for $l \leq k$. Hence e_2 can be pruned without affecting the correctness of the search. □

As an example, consider the collection of sets shown in Fig. 1, denoted as $C1$, and let H be our cost metric. The entities c and d are present in 3 sets and absent in 4 sets. Hence, the 1-step lower bound, $LB\_H_1()$, for entities c and d is $max(log_2(3), log_2(4)) + 1 = 3$. Similarly, 1-step lower bound for all other informative entities is 4. Suppose, we are using 3-steps cost lower bound for entity selection. The 3-steps lower bound for d, $LB\_H_3(C1,d)$, is 3. Since $LB\_H_1()$ for all other entities is not less than 3, any further calculation for them can be pruned safely.

Now, consider another collection where the sets are the same as in collection $C1$ except $S1 = \{a,b,c\}$ and $S4 = \{a,b,c,d,g,h\}$, and let us denote this collection with $C2$. The set counts for all entities are as before, hence the 1-step lower bound, $LB\_H_1()$, for the informative entities remain the same as in collection $C1$. But, the 3-steps lower bound for d, $LB\_H_3(C2,d)$, is 4 now. Therefore, we cannot prune the 3-steps lower bound calculation for entity c using the 1-step lower bound, $LB\_H_1(C2,c)$, which is 3. Thus, we calculate the 2-steps lower bound for c, $LB\_H_2(C2,c)$, which is 4. Now, any further lower bound calculation for c can be pruned using the 2-steps lower bound since it is not less than the already calculated least 3-steps lower bound for entity d.

4.3.1 Implementation. There are several places where our pruning is applied. First, entities are sorted based on their 1-step lower bounds in non-decreasing order, the k-steps lower bounds for entities are calculated in that order, and the least value found so far is updated accordingly. If the 1-step lower bound of an entity e is not less than the already found least k-steps lower bound, then the k-steps lower bound calculations of entity e and all the subsequent entities in the sorted order are pruned.

Second, when calculating the k-steps lower bound for an entity, the already found least value is used to set an upper limit for each of the recursive steps of the calculation. Whenever the upper limit is reached, the rest of the k-steps lower bound calculation for the current entity is pruned. Since, for an entity e to be selected, $LB_k(C, e)$ needs to be less than the already found least value of the lower bound (AFLV), if e partitions a collection C into $C1$ and $C2$, the upper limit (UL) for the accepted value of $LB_{k-1}(C1)$ can be calculated using (6), for the cost metric AD, by replacing $LB\_AD_k(C, e)$ with AFLV and $LB\_AD_{k-1}(C2)$ with the least possible value $LB\_AD_0(C2)$ as

$$UL(C1) = \frac{(AFLV - 1) * |C| - |C2| * LB\_AD_0(C2)}{|C1|}, \quad (11)$$

and similarly using (7), for the cost metric H, as

$$UL(C1) = AFLV - 1. \quad (12)$$

Once the actual value of $LB_{k-1}(C1)$ is calculated, the upper limit (UL) for the accepted value of $LB_{k-1}(C2)$ can be calculated, for the cost metric AD, as

$$UL(C2) = \frac{(AFLV - 1) * |C| - |C1| * LB\_AD_{k-1}(C1)}{|C2|}, \quad (13)$$

and for the cost metric H, as

$$UL(C2) = AFLV - 1. \quad (14)$$

4.4 Lookahead Strategies

Now that we have covered our cost functions and pruning strategies, we present our k-steps lookahead strategy and two variations of it, for selecting the next question to ask by looking k-steps ahead. These strategies choose an entity based on the k-steps lower bound for the cost of a collection, as discussed in Section 4.2. When there are ties between two or more entities in terms of cost, the entity that partitions the collection most evenly is chosen. If there are still ties for the choice of entities, then an entity is selected randomly from the set of candidates. The algorithm applies our pruning strategy in every step where the search space can be cut without compromising the required number of questions, as discussed in Section 4.3.

4.4.1 k-Lookahead with Pruning (k-LP). Algorithm 1 presents our *k-lookahead with pruning* strategy. It takes a collection C of unique sets, the number of steps k to look ahead, and an upper limit ul of the k-steps lower bound for an entity to be selected, as input. Initially, the upper limit is set to a large number. Then, it sorts the entities in the collection based on their partitioning capability from the most even to the least even (Line 11). Since, the entity that partitions a collection most evenly has the minimum value of the 1-step cost lower bound, the entities will also be sorted based on their 1-step lower bound of cost in non-decreasing order. This way, an entity with both the minimum lower bound and also the most even partitioning capability is considered first, breaking possible ties on the cost lower bound. For each entity in the sorted order that partitions the collection C into two sub-collections C^+ and C^-, the $(k-1)$-steps lower bounds for C^+ and C^- are calculated by recursively calling the algorithm (Lines 16-32). Those quantities are plugged into (6) or (7) (depending on the cost metric used) to obtain the k-steps lower bound l for each entity (Line 33). The algorithm keeps track of the entity with the least k-steps lower bound l and sets it as the upper limit ul for the next entity to be considered (Lines 33-35). Since the entities are sorted, if it finds an entity with an equal or larger 1-step lower bound than the upper limit ul, the algorithm stops early and prunes all the remaining entities (Lines 14-15). To further reduce the search space, it calculates the upper limits for C^+ (using (11) or (12)) and C^- (using (13) or (14)) and passes them to the recursive call (Lines 22-23 and 29-30). If no entity can be selected with a lower value of $(k-1)$-steps lower bound than the calculated upper limit, then it stops processing the current entity and moves to the next entity (Lines 24-25 and 31-32). Finally, the algorithm returns an entity e with the minimum k-steps lower bound of cost (Lines 7-10 or 38). To speed up the calculations, the algorithm uses memoization by storing and reusing the results for different inputs of collection C and steps, k (Lines 1-6, 9, and 37).

Algorithm 1 K-Lookahead with Pruning (K-LP)

Input: collection C of unique sets, steps k, and upper limit ul of the k-steps cost lower bound for an entity to be selected
Output: selected entity and it's k-steps cost lower bound

```
1: if $(C, k) \in Cache$ then
2:     $e, l \leftarrow Cache[(C, k)]$
3:     if $ul \leq l$ then
4:         return $null, l$
5:     else if $e \neq null$ then
6:         return $e, l$
7: if $k = 1$ then
8:     let entity $e$ to most evenly partition $C$
9:     $Cache[(C, k)] \leftarrow (e, LB_1(C, e))$
10:    return $Cache[(C, k)]$
11: $SE \leftarrow$ sort entities according to most even partitioning of $C$
12: $e \leftarrow null$
13: for each entity $e_i \in SE$ do
14:     if $LB_1(C, e_i) \geq ul$ then
15:         break
16:     let $C^+$ be the collection $\{S_j \in C \mid e_i \in S_j\}$
17:     $C^- \leftarrow C - C^+$
18:     if $|C^+| = 1$ then
19:         $l^+ \leftarrow 0$
20:     else
21:         $lb^- \leftarrow LB_0(C^-)$
22:         $ul^+ \leftarrow$ Upper-Limit $(ul, |C^+|, lb^-, |C^-|, |C|)$
23:         $e^+, l^+ \leftarrow$ K-LP $(C^+, k-1, ul^+)$
24:         if $e^+ = null$ then
25:             continue
26:     if $|C^-| = 1$ then
27:         $l^- \leftarrow 0$
28:     else
29:         $ul^- \leftarrow$ Upper-Limit $(ul, |C^-|, l^+, |C^+|, |C|)$
30:         $e^-, l^- \leftarrow$ K-LP $(C^-, k-1, ul^-)$
31:         if $e^- = null$ then
32:             continue
33:     $l \leftarrow$ K-Steps-Lower-Bound $(|C^+|, l^+, |C^-|, l^-, |C|)$
34:     if $l < ul$ then
35:         $ul \leftarrow l$
36:         $e \leftarrow e_i$
37: $Cache[(C, k)] \leftarrow (e, ul)$
38: return $e, ul$
```

Two important observations can be made about our k-LP algorithm. First, it can be shown that the algorithm finds *an optimal solution* if k is set to the height of an optimal tree or a greater value. Second, the early stopping opportunities, which are based on our pruning strategy presented earlier, sets apart our lookahead strategies from the existing lookaheads in literature [8, 14].

For a collection of n unique sets and m distinct entities, the runtime of Algorithm 1 is $O(m^k n)$ since finding 1-step lower bounds for m entities is $O(mn)$ and in each recursive step, there will be $O(m)$ calls to the next step. Our next two strategies further reduce the time by setting bounds on the number of candidate entities.

4.4.2 k-LP with Limited Entities (k-LPLE). Despite all the pruning done using our lower bounds, the runtime of our k-LP algorithm increases as a polynomial function of m (e.g., quadratic for $k = 2$), and the algorithm becomes very inefficient for large values of k. On the other hand, the chance of constructing a better tree increases as we increase k. One good trade-off is to limit the number of candidate entities in each step of the lower bound calculation to $q < m$, ranked in terms of the 1-step lower bound of cost. For $q << m$, the reduction in runtime can be significant, analogous to setting a *beam size* in deep learning algorithms [35]. This can be implemented by adding an extra input q to Algorithm 1, modifying Line 11 so that SE contains only the first q sorted entities, and passing q on the recursive calls to the algorithm in Lines 23 and 30.

4.4.3 k-LP with Limited but Variable number of Entities(k-LPLVE). The runtime of k-LPLE may further be reduced by greedily considering only a single entity in each recursive step of the k-steps lower bound calculation for an entity. The intuition here is that an entity with the smallest 1-step lower bound is more probable to be the best choice. Hence, our k-LPLVE strategy limits the number of candidate entities to only one (with the least 1-step lower bound) during each step of the lower bound calculation. With this strategy, the search time is expected to reduce further, but the quality of the results is not expected to change much (see Section 5 for evaluation results). This strategy can be implemented by performing the same modifications as k-LPLE in Algorithm 1 except that when the function is called from outside with q, SE in Line 11 takes the first q sorted entities during that call and only the first entity during the subsequent recursive calls to the function.

4.5 Set Discovery

The set discovery scheme studied in this paper is an interactive process that starts with an initial question posed to the user and continues with follow-up questions based on the user's answers. The lookahead strategies choose an entity to be the next question, which is expected to minimize the cost of discovering the user's desired set. With each user feedback, the same selection process continues until the user's desired set is discovered or the user is satisfied with the refined sub-collection of sets and does not want to answer more questions.

The general approach for *set discovery* is presented in Algorithm 2. It takes the entire collection C of unique sets and a user-provided initial set I as inputs and finds the sub-collection CS containing all the supersets of I in C (Lines 2-4). It then iteratively, selects the best entity e according to the entity selection strategy denoted by $\mathbf{Y}$, asks the user a question about the presence of that entity in the desired set, and re-calculates the sub-collection

Algorithm 2 Set Discovery

Inputs: collection C of unique sets and initial set I
Output: sets that are consistent with the user's answers
Parameter: entity selection strategy $\mathbf{Y}$ and halt condition Γ

```
1:  $CS \leftarrow \emptyset$
2:  for each set $S_i \in C$ do
3:      if $I \subseteq S_i$ then
4:          $CS \leftarrow CS \cup \{S_i\}$
5:  while $|CS| > 1$ and $\Gamma$ is $false$ do
6:      $e \leftarrow \mathbf{Y}(CS)$
7:      $\alpha \leftarrow$ query user about the presence of $e$ in target set
8:      let $P$ be the collection $\{S_i \in CS \mid e \in S_i\}$
9:      if $\alpha$ is $true$ then
10:         $CS \leftarrow P$
11:     else
12:         $CS \leftarrow CS - P$
13: return $CS$
```

CS of candidate sets based on the user feedback until a single set is left or the halt condition $\mathbf{\Gamma}$ (e.g., the user does not want to answer more questions) is met (Lines 5-12). Finally, it returns the remaining sets that are consistent with the user's answers (Line 13). The runtime of Algorithm 2 depends on the number of questions required to discover the desired set and the strategy used. In the worst case, the number of questions can be $n - 1$ for a collection of n sets.

Offline tree construction Our tree construction may be done offline for static collections, for example, when the initial query sets are known in advance or are always empty. An offline construction may be useful when the same decision tree is constructed multiple times or is used by multiple queries. Algorithm 3 provides the steps for precomputing a decision tree on a collection of sets. With the decision tree constructed offline, a set discovery can be efficiently performed by asking questions and following only a single path through the tree in real-time.

Algorithm 3 takes a collection C of unique sets as input. If the collection has only one set, then it constructs a tree T consisting of a single node with the only set G (Lines 1-3). Otherwise, the algorithm selects the best entity e using the entity selection strategy denoted by $\mathbf{Y}$ (Line 5). It recursively constructs the subtrees T^+ and T^- for the two sub-collections $C+$ and $C-$ respectively (Lines 6-9). Finally, a tree T, consisting of a root node e and two child subtrees (T^+, T^-), is constructed and returned (Lines 10-11).

There are $n - 1$ internal nodes in a full binary decision tree representing a collection of n sets and m entities, and each internal node requires a k-steps lookahead, which costs $O(m^k n)$. Hence the runtime of Algorithm 3 is $O(m^k n^2)$.

5 EXPERIMENTS

This section reports an experimental evaluation of our algorithms and pruning strategies on both real and synthetic data and under different parameter settings.

5.1 Evaluation Setup

As our evaluation measures, we study (a) the effectiveness of our algorithms in finding a "good" solution for the problem of set discovery, (b) the effectiveness of our pruning strategy in reducing the size of the search space, (c) the efficiency of our algorithms in terms of the running time, and (d) the scalability of our algorithms with both the number and the size of sets. Our

Algorithm 3 Tree Construction

Input: collection C of unique sets
Output: a decision tree representation of the input collection
Parameters: entity selection strategy $\mathbf{Y}$

```
1:  if $|C| = 1$ then
2:      let $G$ be the only element of $C$
3:      $T \leftarrow$ Tree $(G, null, null)$
4:  else
5:      $e \leftarrow \mathbf{Y}(C)$
6:      let $CS^+$ be the collection $\{S_i \in C \mid e \in S_i\}$
7:      $CS^- \leftarrow C - CS^+$
8:      $T^+ \leftarrow$ Tree-Construction $(CS^+)$
9:      $T^- \leftarrow$ Tree-Construction $(CS^-)$
10:     $T \leftarrow$ Tree $(e, T^+, T^-)$
11: return $T$
```

results are compared to the relevant algorithms in the literature (when applicable).

The effectiveness of our set discovery is measured in terms of the *number of questions* to be answered by a user looking for a target set. Without knowing much about the target set of a user, we assume all sets that contain an initially provided set are equally likely. With this, the effectiveness may be defined in terms of the *average number of questions* or the *maximum number of questions* to be answered by a user. These quantities also represent the average depth of the leafs (AD) and the height (H) of a decision tree that is constructed.

The efficiency of an entity selection algorithm is measured in terms of the *tree construction time*, which is the time needed to construct a decision tree using the selection strategy. It can be noted that the tree construction time is different from the time spent when searching for a specific set (*discovery time*). For the former, Algorithm 3 constructs a whole tree with all sets placed at the leaves and the internal nodes giving the paths to all sets at the leaves, whereas for the latter, Algorithm 2 only constructs a path from the root to the target set. The latter is much less if the wait time for user responses is excluded.

The algorithms being evaluated include entity selection using k-LP, k-LPLE, and k-LPLVE strategies. For a comparison with entity selection strategies from the literature, our evaluation also includes *information gain* (InfoGain) [29] and *gain-k* [14]. Our reported result for *information gain* holds for *indistinguishable pairs* [7] and 1-step lookahead, i.e. gain-k and k-LP with $k = 1$, since they all select the same entity, as shown earlier (Lemma 4.3).

Our algorithms were implemented in Python 3 and our experiments were run on a 64-bit machine with Intel(R) Core i5-9300H @2.40 GHz processor and 8 GB RAM.

5.2 Datasets and Queries

We conduct our experiments on two datasets for set discovery, including *web tables*, which consists of a collection of entity sets extracted from the columns of various web tables, and *synthetic datasets*, where large collections of sets are generated following some distributions. The former evaluates our algorithms on a real dataset, whereas the latter assesses the scalability of our strategies under different collection sizes and parameter settings.

We also evaluate our algorithms on the task of query discovery, based on a *baseball* database.

5.2.1 *Web tables.* Our web table dataset is collected from a 2014 snapshot of Wikipedia. We extract tables in document text and treat each of their columns as a set based on the observation that each column has a domain and the values are drawn from that domain. As an example, one set includes 58 NBA players including Steve Nash, Kobe Bryant, and Tracy McGrady. The sets are diverse, covering many domains of interest, but also noisy. We remove any set that has less than three distinct elements and sets that consist of all numbers. We further remove duplicate entries, making each list a pure set, and a few frequent keywords such as unknown, tba, total. After those cleanings, we obtain 1,407,178 unique sets containing in total 6,312,409 distinct entities. Examples of sets and queries are given elsewhere [15] For our experiments, we considered each combination of two entities as a possible initial example set and the sets that contained the two example entities as candidates. This resulted in 14,491 initial sets, giving us the same number of sub-collections with at least 100 sets in each. The choice of two example entities was based on the observation that at least two entities from a semantic class are required to unambiguously represent the class. As an example, Liverpool may represent both a "City" and a "Football Club" whereas Liverpool and Arsenal together do not represent the "Cities" semantic class. The number of sets in the selected collections was in the range of $[100, 11219]$ with an average of 390 and a standard deviation of 478, and the number of distinct entities was in the range of $[15, 15186]$ with an average of 3,112 and a standard deviation of 2,379. We constructed decision trees for all the selected sub-collections to evaluate our strategies.

5.2.2 *Synthetic data.* To study the performance of our entity selection strategies under different data distributions as well as the scalability with the number of entities and sets in the collection, we generated a few synthetic set collections. The set generation follows a copy-add preferential mechanism where some elements are copied from an existing set and the rest of the elements are added from a universe of elements. Similar copying models are used in other domains (e.g., the dynamics of the web graph [21], the copying and publishing relationships between data sources [13], etc.). Each set has two parameters: a set size s, chosen randomly from a range of values (e.g., $[50, 100]$), and an overlap ratio $\alpha \in [0, 1)$. For each set, we choose a size s from the range of possible sizes randomly and an overlap ratio α. Then $\alpha * s$ elements are copied from a previously generated set and $(1 - \alpha) * s$ elements are added from the entity universe. If a previously generated set does not exist or does not have enough elements, then additional elements are selected from the entity universe to bump up the set size to s. We generated 19 synthetic collections by varying the overlap ratio α, the range of set sizes d, and the number of sets n. Table 1 gives some information about these collections including the number of distinct entities in each collection. For this dataset, no entities were selected as query entities (i.e., the user-provided initial set is considered empty), and all the sets in each collection are considered as possible target sets.

5.2.3 *Baseball database.* The baseball database [22] is a complex, multi-relation database that contains batting, pitching, and fielding statistics plus standings, team stats, player information, and more for Major League Baseball (MLB) covering the years between 1871 and 2020. Our experiment is based on the *People* table which contains information about name, birth, death, height, weight, batting and throwing hand, etc., of 20,185 baseball players. For our experiment, we considered only CNF (conjunctive normal form) queries with conditions on columns *birthCountry*, *birthState*, *birthCity*, *birthYear*, *birthMonth*, *birthDay*, *height*, *weight*,

Overlap ratio α	Number of distinct entities
0.99	23k
0.95	36k
0.90	59k
0.85	83k
0.80	108k
0.75	132k
0.70	156k
0.65	178k

(a) $n = 10$**k**, $d = 50 - 60$

Number of sets n	Number of distinct entities
10k	59k
20k	125k
40k	216k
80k	385k
160k	622k

(b) $\alpha = 0.9$, $d = 50 - 60$

Set size range d	Number of distinct entities
50-100	119k
100-150	150k
150-200	180k
200-250	214k
250-300	249k
300-350	283k

(c) $n = 10$**k**, $\alpha = 0.9$

Table 1: Synthetic data by varying (a) overlap ratio α, (b) number of sets n and (c) set size range d

bats, and *throws* of the *People* table. At first, we constructed 7 target queries that could be interesting to a user. Table 2 describes the target queries and the number of tuples in their outputs. Then, for each target query, we randomly selected 2 output tuples as the example tuples and generated candidate CNF queries that contain the example tuples in their output. The candidate queries are generated using the following simple steps:

(1) The columns are grouped into categorical and numerical with columns *birthCountry*, *birthState*, *birthCity*, *birthMonth*, *birthDay*, *bats*, and *throws* treated as categorical and *birthYear*, *height*, and *weight* treated as numerical in our experiments.
(2) A few reference values are defined for each numerical column. For examples, *height*: {60, 65, 70, 75, 80}, *weight*: {120, 140, 160, 180, 200, 220, 240, 260, 280, 300}, and *birthYear*: {1850, 1870, 1890, 1910, 1930, 1950, 1970, 1990}.
(3) A selection condition on each categorical column is constructed as the disjunctions of the unique values of the example tuples for that column. For example, if the birth city of an example player is Chicago and that of another player is Seattle, then the selection condition is $birthCity = \text{“}Chicago\text{”} \vee birthCity = \text{“}Seattle\text{”}$, whereas if the birth city of all example players is Chicago, then the selection condition is $birthCity = \text{“}Chicago\text{”}$.
(4) A few selection conditions on each numerical column are constructed using the possible intervals of the reference values that contain the values of all example tuples. For example, if the height of an example player is 62 and that of another player is 73, then the possible selection conditions on *height* are $height > 60 \wedge height < 75$, $height > 60 \wedge height < 80$, $height > 60$, $height < 75$, and $height < 80$.
(5) Each selection condition on a column yields a candidate query, and the conjunction of any two selections on different columns provide additional candidate queries. For example, $\sigma_{birthCity=\text{“}LosAngeles\text{”}}(People)$ is a query with selection condition on a single column, whereas $\sigma_{birthCity=\text{“}LosAngeles\text{”} \wedge height>70 \wedge height<80}(People)$ is a query with selection conditions on two columns. Similarly, candidate queries with selection conditions on more columns can be generated. Our experiments consider queries with selection conditions on up to two columns.

Once the candidate queries were generated, we applied our set discovery strategy to discover the target query. The user answers about the membership of the presented tuples were simulated by verifying them against the output of the target query. Table 3 provides information about the selected example tuples for each target query, the number of generated candidate queries from the example tuples, and the average number of tuples in the output of those candidate queries.

Target query	Query description	Number of output tuples
T1	$\sigma_{birthCountry=\text{“}USA\text{”} \wedge birthYear>1990}(People)$	892
T2	$\sigma_{birthCity=\text{“}LosAngeles\text{”} \wedge height>70 \wedge height<80}(People)$	201
T3	$\sigma_{bats=\text{“}L\text{”} \wedge throws=\text{“}R\text{”}}(People)$	2179
T4	$\sigma_{birthCountry=\text{“}USA\text{”} \wedge bats=\text{“}B\text{”}}(People)$	939
T5	$\sigma_{birthMonth=12 \wedge birthDay=25}(People)$	65
T6	$\sigma_{height>75 \wedge weight>260}(People)$	49
T7	$\sigma_{height<65 \wedge weight<160}(People)$	26

Table 2: Target queries for the baseball database

Target query	Player ids of example tuples	# of candidate queries	Average number of output tuples
T1	baragca01, phillev01	776	9404.24
T2	ryanbr01, edwarda01	987	11254.35
T3	ellioal01, drumrke01	940	10612.07
T4	dashnle01, craigro02	916	10957.30
T5	brownll01, ellerfr01	1339	9772.70
T6	evansde01, fulchje01	600	7187.00
T7	emmerbo01, gearidi01	1189	7795.78

Table 3: Information about selected example tuples and generated candidate queries on baseball database

5.3 Evaluation Results

5.3.1 Choosing the parameters k and q. As k increases, the generated trees are expected to be closer to an optimal tree and when k is set to the height of an optimal tree or greater, our algorithm finds an optimal tree. However, as k increases, the running time increases dramatically. Parameter q acts similar to *beam size* in deep learning, and setting this parameter allows us to increase k without too much affecting the running time. To set the parameters k and q for our algorithms, we did run some experiments on our web tables dataset. Fig. 3 shows that the runtime of k-LP increases by one to two orders of magnitude when the number of lookahead steps k is increased from 2 to 3. At the same time, the average number of questions usually becomes less with higher k. To balance the runtime with the quality of the trees that are constructed, we set $k = 2$ for our experiments with the k-LP strategy. The runtime may also be kept low, while increasing k, using the k-LPLE strategy, which limits the number of entities in each step. For our experiments with k-LPLE and k-LPLVE strategies, we set $k = 3$ and experiment with different values (up to 50) of the number of entities q. The average number of questions that are required remains almost the same when the value of q exceeds 10, but the runtime increases significantly. Therefore, we set $q = 10$ for the k-LPLE and k-LPLVE strategies. The average numbers of questions for larger values of q are almost the same hence are not reported here.

5.3.2 Comparison to strategies in the literature. A strong baseline for comparison is *information gain* [28], which is also equivalent to *indistinguishable pairs*, *gain-k* with $k = 1$, and our k-LP

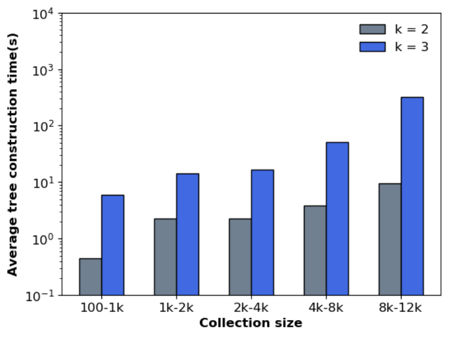


Figure 3: Tree construction time (seconds) for k-LP varying k on web tables dataset

	T1	T2	T3	T4	T5	T6	T7
Avg	97.3%	99.4%	99.1%	99.7%	88.5%	99.7%	99.9%
Min	90.1%	94.6%	96.5%	98.0%	30.6%	98.1%	99.5%

Table 4: Average and minimum number of entities pruned at all nodes for the Baseball dataset

with $k = 1$; they all select the same entity as discussed in Section 4.2. Our evaluation shows improvements over InfoGain in the average number of questions with the cost metric AD and the maximum number of questions with the cost metric H. The mean improvement in the maximum number of questions (H) is close to one, whereas the mean improvement for the average number of questions (AD) is less due to the facts that the improvement is averaged over all sets in each sub-collection and that the average number of questions for InfoGain is already very close to the optimal (the average difference in the average number of questions with optimal solution for InfoGain is only about 0.048) with little room for improvement. The improvements in the number of questions over Info-Gain for all our reported methods (k-LP with k=2 and k-LPLE and k-LPLVE with k=3 and q=10) under both AD and H are all statistically significant at $\alpha = 0.01$ using one-tailed t-test. It should be noted that Info-Gain is a pretty strong baseline, and any improvement in the number of questions is important. For example, if the questions are medical tests required to identify a disease, then a small reduction even in the average number of tests could save the patients a large amount of money and time to complete the tests.

5.3.3 *Effectiveness of our pruning.*

The pruning proposed in this paper makes a huge difference in the tree construction time of all our strategies. On our Web tables dataset, more than 99% of candidate entities are pruned at the root level (for both k-LP with $k = 2$ and $k = 3$), meaning no tree is constructed for those entities as the root. The pruning also happens at the nodes under the root. Table 4 shows the average and the minimum number of entities pruned at each node for the Baseball dataset at $k = 2$. The results are almost the same for $k = 3$. In most cases, more than 90% of the entities are pruned, demonstrating the effectiveness of our lower bounds and the choice of questions. Fig. 4a shows the speedup on the web tables dataset, and Fig. 4b shows the same on the synthetic datasets. The average speedup in runtime on the web tables dataset is in the range of two to three orders of magnitude for $k = 2$ and up to five orders of magnitude when $k = 3$. Since the runtime of gain-k increases polynomially with the number of entities and exponentially with k, the speedup is more for larger values of k and on datasets with a large number of entities and sets. This can be seen in Fig. 4a for the web tables dataset where k is varied from 2 to 3 and in Fig. 4b for the synthetic dataset with a fixed k and varying the number of sets.

5.3.4 *Performance varying the overlap between sets.*

One factor that affects the performance of a set discovery is the amount of overlap between sets. Consider an extreme case where there is no overlap between sets. With n sets, one needs to ask roughly $n/2$ questions on average ($n - 1$ questions in the worst case) to find a target set. As the overlap between sets increases, there is more chance to filter more than one set with each question. To better understand this relationship between the overlap and the search performance, we varied the overlap ratio as in Table 1a for our synthetic dataset and measured the number of questions that were needed to discover each set. Fig. 5 shows the average number of questions that were needed as the overlap ratio varied from 0.65 to 0.99. As the overlap ratio increases, both the average number of questions and the tree construction time decrease. When the overlap ratio becomes less than 0.90, the average number of questions starts showing an upward trend. This upward trend is expected to continue to the point where one needs to ask roughly $n/2$ questions on average ($n - 1$ questions in the worst case) to find a target set. This happens, for example, when all sets have the same elements except at least one more element that distinguishes each set from the rest.

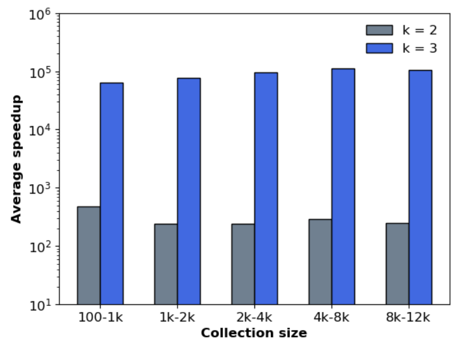
5.3.5 *Scalability with the number of entities and the collection size.*

To evaluate the scalability of our algorithms on larger datasets, we conducted some experiments using our synthetic data. In one experiment, we varied the number of distinct entities in a collection, while keeping the number of sets and the overlap ratio fixed at 10k and 0.9 respectively. The number of distinct entities changes (as shown in Table 1c) with the set size varied. As can be seen in Fig. 6, the average number of questions is not affected much, but the tree construction time increases because of the larger number of candidate entities that are considered during the lower bounds calculation. The increase in running time is linear for k-LPLE and k-LPLVE, and that of k-LP is quadratic with $k = 2$.

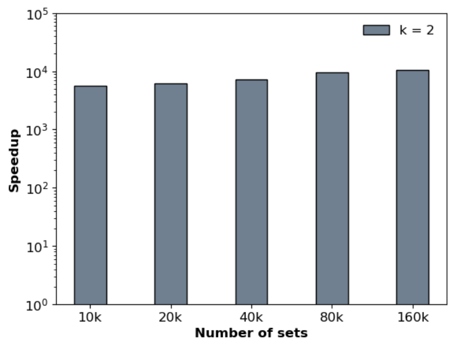
In another experiment, we varied the number of sets in the collection while keeping the set size in $[50, 60]$ and the overlap ratio fixed at 0.9. The number of distinct entities m increases as well (as shown in Table 1b), when we increase the number of sets n. As shown in Fig. 7, with each doubling of the input size, the average number of questions increases roughly by 1. The tree construction time is expected to increase linearly with the number of sets if the number of distinct entities is fixed. In our experiment, the tree construction time looks a bit far from linear (and more quadratic) because of the increase in m as n increases.

5.3.6 *Evaluation results on query discovery.*

Fig. 8 shows both the number of questions and the query discovery time to discover the target queries on the baseball database for the baseline InfoGain and our lookahead strategies. It can be seen that the number of questions for k-LP, k-LPLE, and k-LPLVE is less than or equal to InfoGain (except T7 for k-LP). Since none of the strategies are optimal, our strategies may sometimes require more questions than InfoGain, but that probability is very low as discussed in Section 5.3. Moreover, although the query discovery time of our strategies is higher than InfoGain, it is relatively small when the candidate queries have large result sets (on average 7000 to



(a) k**-LP vs Gain-**k **on web tables data**



(b) k**-LP vs Gain-**k **on synthetic data**

Figure 4: Speedup of our strategies because of pruning

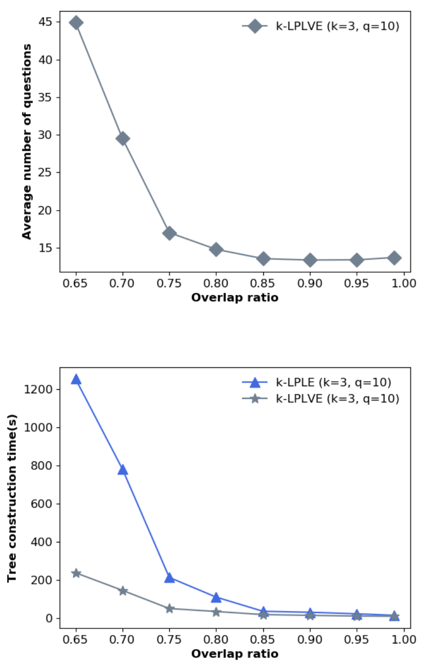


Figure 5: Effects of set overlaps on average number of questions (top) and tree construction time in seconds (bottom)

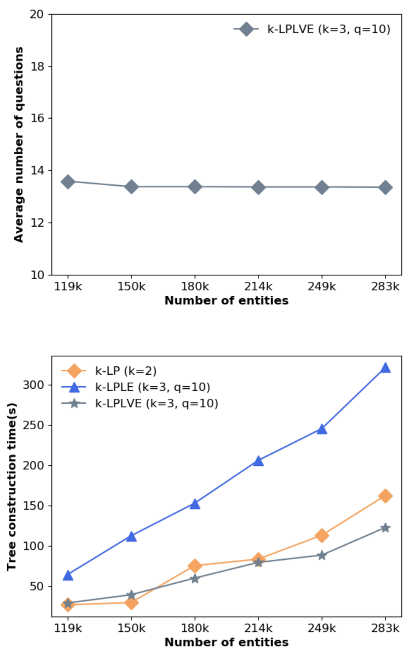


Figure 6: Effects of increasing the number of distinct entities in a collection on average number of questions (top) and tree construction time in seconds (bottom)

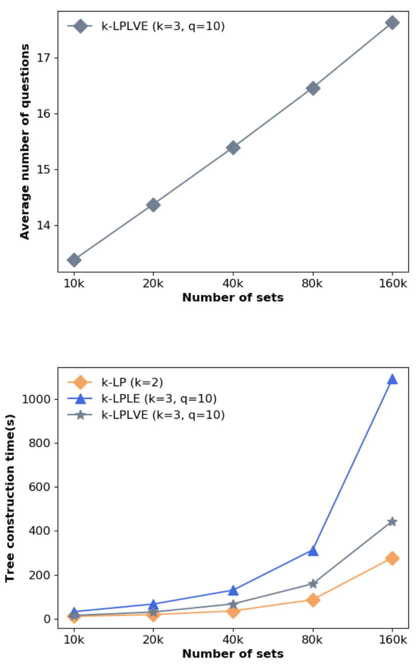


Figure 7: Effects of increasing the number of sets on average number of questions (top) and tree construction time in seconds (bottom)

12000), as shown in Table 3. Finally, an important observation can be made about our query discovery strategy. The user is required to confirm the membership of only a few tuples (9 to 11) to find the target query among a large number of candidate queries (600 to 1200) which is more convenient than listing all the possible output tuples (close to 2000 for some of our target queries) of a target query.

6 DISCUSSIONS

Our work is focused on reducing the number of interactions by selecting examples that are most informative, effectively reducing the number of candidates, but that is only one factor affecting the user experience. There are multiple other factors that need to be considered when applying our work in real settings.

Multiple-choice examples Sometimes it is more desirable to offer a set of examples (instead of one) and asking if one or more of those examples belong to the target set. For example, this can be more effective if the user is not sure about some examples. A interesting question is how those examples should be selected. One approach is to aim for maximizing the expected gain, as done in a multi-armed bandit setting. This can dramatically increases the size of the search space though, and using effective pruning strategies is essential. An alternative is to find some strategies

Target query	InfoGain	k-LP ($k=2$)	k-LPLE ($k=3, q=10$)	k-LPLVE ($k=3, q=10$)
T1	10	10	10	10
T2	10	9	10	10
T3	10	10	9	9
T4	10	10	9	9
T5	11	11	10	10
T6	10	9	9	9
T7	10	11	10	10

(a) Number of questions

Target query	InfoGain	k-LP ($k=2$)	k-LPLE ($k=3, q=10$)	k-LPLVE ($k=3, q=10$)
T1	1.798	163.097	11.662	7.999
T2	3.234	17.880	37.867	26.060
T3	2.921	31.499	31.589	19.453
T4	2.796	20.548	20.944	15.894
T5	3.687	19.124	23.314	18.690
T6	0.906	10.747	10.395	4.806
T7	2.187	7.108	16.257	17.685

(b) Query discovery time (seconds)

Figure 8: Number of questions and query discovery time to find the target queries on baseball database

for selecting the nodes of a decision tree that provide 'good' sets of examples but not computationally intensive.

Possibility of errors in answers It is possible that users make mistakes in their answers, and this can introduce another interesting challenge in detecting that a mistake is made and recovering from them. One approach is to backtrack when no target set satisfies all constraints and revisit those constraints. An alternative is to assign a level of certainty, and make the optimization process aware of the uncertainties.

Unanswered questions Sometimes the user is uncertain about the membership of an entity in the target set and may reply "don't know" to the membership question. In such cases, the entity selection strategy can be called again using the same collection of candidate sets but excluding the entities that the user is not sure about. With unanswered questions, the search may not resolve to a single set.

7 CONCLUSIONS

We have studied the problem of set discovery using an interactive approach, where example entities from candidate sets are presented and the search is narrowed down based on the feedback about the presence of those entities in the target set. We have formulated the search as a tree optimization and have developed both effective and efficient k-step lookahead algorithms to construct a tree which results in near-optimal number of questions needed to discover a set. Our evaluation on both real and synthetic data shows the efficiency and scalability of our algorithms.

Our work can be extended or improved in a few directions, in addition to those highlighted in Section 6. One direction is to further study the distribution of entities in a collection. Better understanding the distribution may provide some insight to develop other strategies. Another direction is to study scenarios where the sets to be discovered are not equally likely. Extending our algorithms to the cases where the sets are noisy or have errors is another direction.

ACKNOWLEDGMENTS

This research was funded by the Natural Sciences and Engineering Research Council of Canada and through a grant from Servus Credit Union.

REFERENCES

[1] [n.d.]. Sloan Digital Sky Survey. https://www.sdss.org/
[2] [n.d.]. SQLShare: Database-as-a-Service for Science. https://uwescience.github.io/sqlshare/
[3] Azza Abouzied, Dana Angluin, Christos Papadimitriou, Joseph Hellerstein, and Avi Silberschatz. 2013. Learning and Verifying Quantified Boolean Queries by Example. *Proceedings of the PODS Conference* (04 2013). https://doi.org/10.1145/2463664.2465220
[4] Micah Adler and Brent Heeringa. 2008. Approximating Optimal Binary Decision Trees. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, Ashish Goel, Klaus Jansen, José D. P. Rolim, and Ronitt Rubinfeld (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–9.
[5] Dana Angluin. 1988. Queries and concept learning. *Machine learning* 2, 4 (1988), 319–342.
[6] Marcelo Arenas, Gonzalo I Diaz, and Egor V Kostylev. 2016. Reverse engineering SPARQL queries. In *Proceedings of the WWW Conference.* 239–249.
[7] Senjuti Basu Roy, Haidong Wang, Gautam Das, Ullas Nambiar, and Mukesh Mohania. 2008. Minimum-Effort Driven Dynamic Faceted Search in Structured Databases. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management* (Napa Valley, California, USA) *(CIKM '08)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/1458082.1458088
[8] Angela Bonifati, Radu Ciucanu, and Slawomir Stawork. 2014. Interactive inference of join queries. In *In EDBT.* 451–462.
[9] Angela Bonifati, Radu Ciucanu, and Sławek Staworko. 2016. Learning Join Queries from User Examples. *ACM Trans. Database Syst.* 40, 4, Article 24 (Jan. 2016), 38 pages. https://doi.org/10.1145/2818637
[10] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. 2009. Query recommendations for interactive database exploration. In *International Conference on Scientific and Statistical Database Management.* Springer, 3–18.
[11] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-Example: An Automatic Query Steering Framework for Interactive Data Exploration. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 517–528. https://doi.org/10.1145/2588555.2610523
[12] Irit Dinur and David Steurer. 2014. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing.* 624–633.
[13] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. 2009. Truth discovery and copying detection in a dynamic world. *Proceedings of the VLDB Endowment* 2, 1 (2009), 562–573.
[14] Saher Esmeir and Shaul Markovitch. 2004. Lookahead-Based Algorithms for Anytime Induction of Decision Trees. In *Proceedings of the Twenty-First International Conference on Machine Learning* (Banff, Alberta, Canada) *(ICML '04)*. Association for Computing Machinery, New York, NY, USA, 33. https://doi.org/10.1145/1015330.1015373
[15] Arif Hasnat. 2021. *Interactive set discovery.* Master's thesis. University of Alberta. https://doi.org/10.7939/r3-k9jr-am91
[16] Bill Howe, Garret Cole, Emad Souroush, Paraschos Koutris, Alicia Key, Nodira Khoussainova, and Leilani Battle. 2011. Database-as-a-service for long-tail science. In *International Conference on Scientific and Statistical Database Management.* Springer, 480–489.
[17] Laurent Hyafil and Ronald L. Rivest. 1976. Constructing optimal binary decision trees is NP-complete. *Inform. Process. Lett.* 5, 1 (1976), 15 – 17. https://doi.org/10.1016/0020-0190(76)90095-8
[18] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. Sqlshare: Results from a multi-year sql-as-a-service experiment. In *Proceedings of the 2016 International Conference on Management of Data.* 281–293.
[19] Dmitri V Kalashnikov, Laks VS Lakshmanan, and Divesh Srivastava. 2018. Fastqre: Fast query reverse engineering. In *Proceedings of the 2018 International Conference on Management of Data.* 337–350.
[20] Nodira Khoussainova, YongChul Kwon, Wei-Ting Liao, Magdalena Balazinska, Wolfgang Gatterbauer, and Dan Suciu. 2011. Session-based browsing for more effective query reuse. In *International Conference on Scientific and Statistical Database Management.* Springer, 583–585.
[21] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D Sivakumar, Andrew Tomkins, and Eli Upfal. 2000. Stochastic models for the web graph. In

Proceedings 41st Annual Symposium on Foundations of Computer Science. IEEE, 57–65.

[22] Sean Lahman. 2020. Baseball database. http://www.seanlahman.com/baseball-archive/statistics/

[23] Hyafil Laurent and Ronald L Rivest. 1976. Constructing optimal binary decision trees is NP-complete. *Information processing letters* 5, 1 (1976), 15–17.

[24] Hao Li, Chee-Yong Chan, and David Maier. 2015. Query from Examples: An Iterative, Data-Driven Approach to Query Construction. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2158–2169. https://doi.org/10.14778/2831360.2831369

[25] Tova Milo and Amit Somech. 2018. Next-step suggestions for modern interactive data analysis platforms. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 576–585.

[26] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2014. Exemplar queries: Give me an example of what you need. *Proceedings of the VLDB Endowment* 7, 5 (2014), 365–376.

[27] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2017. New trends on exploratory methods for data analytics. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1977–1980.

[28] J. R. Quinlan. 1993. *C4.5: Programs for machine learning*. Morgan Kaufmann, San Mateo, CA.

[29] J. Ross Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (March 1986), 81–106. https://doi.org/10.1023/A:1022643204877

[30] Burr Settles. 2009. Active learning literature survey. (2009).

[31] Detlef Sieling. 2008. Minimization of decision trees is hard to approximate. *J. Comput. System Sci.* 74, 3 (2008), 394–403.

[32] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by Output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 535–548. https://doi.org/10.1145/1559845.1559902

[33] Quoc Trung Tran, CheeYong Chan, and Srinivasan Parthasarathy. 2014. Query Reverse Engineering. *The VLDB Journal* 23, 5 (Oct. 2014), 721–746. https://doi.org/10.1007/s00778-013-0349-3

[34] Yaacov Y. Weiss and Sara Cohen. 2017. Reverse Engineering SPJ-Queries from Examples. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 151–166. https://doi.org/10.1145/3034786.3056112

[35] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. 2021. Dive into deep learning. *arXiv preprint arXiv:2106.11342* (2021).

[36] Meihui Zhang, Hazem Elmeleegy, Cecilia Procopiuc, and Divesh Srivastava. 2013. Reverse engineering complex join queries. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 809–820. https://doi.org/10.1145/2463676.2465320

[37] Qianrui Zhang, Haoci Zhang, Thibault Sellam, and Eugene Wu. 2019. Mining precision interfaces from query logs. In *Proceedings of the 2019 International Conference on Management of Data*. 988–1005.

[38] Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 396–411.

[39] Moshé M Zloof. 1975. Query by example. In *Proceedings of the national computer conference and exposition*. 431–438.