Random Sampling Based Algorithms for Efficient Semi-Key Discovery

Ying Xu * Rajeev Motwani †

Abstract

We design efficient random sampling based algorithms for discovering keys and semi-keys in large tables. Given that these problems are provably hard, we adopt the approach of finding approximate solutions to save on time and space requirements. We first propose two natural measures for quantifying the approximation of a semi-key. For the problem of finding minimum keys, we develop efficient algorithms that find small semi-keys with provable size and key-approximation guarantees, and have space and time requirements sublinear in the number of tuples. We also design fast algorithms for finding all minimal exact and approximate keys. Finally, we provide an extensive set of experimental results on real world data sets which confirm the efficiency and accuracy of our algorithms.

1 Introduction

Keys and semi-keys play an important role in many aspects of database management, such as query optimization, indexing and data integration. Automatic discovery of keys and semi-keys is not only helpful to database management, but also interesting for knowledge discovery. In general, finding all or even just the minimum keys are provably hard problems with severe time and space requirements. In this paper, we focus on designing efficient *approximation* algorithms for key/semi-key discovery in large tables.

Before presenting the algorithms, we need to first define measures for quantifying the approximation of semi-keys. Keys are a special case of functional dependencies, and measures have been proposed previously for quantifying the approximation of functional dependency [5, 12]. We propose two natural measures for key-approximation – α-separation and α-distinct, adapted from the approximate functional dependency measures proposed earlier by Kivinen and Mannila [5].

We first consider the problem of finding the minimum key. Previous work shows that it is NP-hard to find the minimum exact key even for the special case where each attribute is boolean [1]. The best known approximation algorithm can only find a key whose size is within $\ln n$ of the minimum key size, where n is the number of tuples; even this algorithm requires multiple scans of the table, which is expensive for large databases that cannot reside in main memory and prohibitive for streaming databases. To enable more efficient algorithms, we sacrifice accuracy by allowing approximate answers (semi-keys). We develop efficient algorithms that find small semi-keys with provable size and key-approximation guarantees. Both space and time requirements of our algorithms are sublinear in the number of tuples, which is a desirable property for large tables.

In the second part of the paper, we design algorithms for identifying all minimal keys and semi-keys. This problem inherently requires worst case running time exponential in the number of attributes because of the output size. Our algorithms use small sample tables to efficiently prune the key space, and are shown by experiments to perform well on real data sets with a reasonable number of attributes. In our experiments the pruning is able to improve the running time by orders of magnitude.

1.1 Definitions and Overview of Results

A *key* is a subset of attributes that uniquely identifies each tuple in a table. A *semi-key* is a subset of attributes that can almost distinguish all tuples. We consider two measures of key-approximation:

> (1) An α-*distinct key* is a subset of columns such that it is possible to remove $1-\alpha$ fraction of tuples after which the subset of columns becomes a key in the remaining table. This definition conforms with the measure of approximate functional dependency in earlier work [5, 3].
>
> (2) We say that a subset of attributes *separates* a pair of tuples x and y if x and y have different values on at least one attribute in the subset. An α-*separation key* is a subset of columns which separates at least a fraction α of all distinct pairs of tuples. This definition is adapted

*Stanford University. E-mail: xuying@cs.stanford.edu. Supported in part by Stanford Graduate Fellowship and NSF Grant ITR-0331640.

†Stanford University. E-mail: rajeev@cs.stanford.edu. Supported in part by NSF Grant ITR-0331640, and a grant from Media-X.

		Exact	α-Distinct	α-Separation
Greedy	time	m^2n	$mn + m^2\sqrt{\frac{2\alpha n}{1-\alpha}\ln\frac{2^m}{\delta}}$ $= mn + O(m^2\sqrt{nm})$	$mn + m^2\log_{\frac{1}{\alpha}}\frac{2^m}{\delta}$ $= mn + O(m^3)$
	space	mn	$m\sqrt{\frac{2\alpha n}{1-\alpha}\ln\frac{2^m}{\delta}} = O(m\sqrt{nm})$	$m\log_{\frac{1}{\alpha}}\frac{2^m}{\delta} = O(m^2)$
	key size	$(1+2\ln n)k^*$	$(1+\ln(\frac{2\alpha n}{1-\alpha}\ln\frac{2^m}{\delta}))k^*$ $= (\ln m + \ln n + O(1))k^*$	$(1+\ln\log_{\frac{1}{\alpha}}\frac{2^m}{\delta})k^*$ $= (\ln m + O(1))k^*$
Random Deletion	time	m^2n	$mn + m^2\sqrt{\frac{2\alpha n}{1-\alpha}\ln\frac{m}{\delta}}$ $= mn + O(m^2\sqrt{n\ln m})$	$mn + 2m\log_{\frac{1}{\alpha}}\frac{m}{\delta}$ $= mn + O(m\ln m)$
	space	mn	$m\sqrt{\frac{2\alpha n}{1-\alpha}\ln\frac{m}{\delta}} = O(m\sqrt{n\ln m})$	$m\log_{\frac{1}{\alpha}}\frac{m}{\delta} = O(m\ln m)$

Table 1: **Performance of algorithms for finding minimum semi-keys.** The "α-distinct"("α-separation") column gives the complexities if we want the algorithms to output an α-distinct (separation) key with probability $1-\delta$, contrasted with applying the algorithms to find the minimum exact key (the "exact" column). In O-notation, α and δ are considered as constants. In the "key size" row, k^* is the size of the minimum exact key. Random Deletion provides no guarantee of key sizes.

from the self-join size measure for approximate functional dependency [5].

	age	sex	state
1	20	Female	CA
2	30	Female	CA
3	40	Female	TX
4	20	Male	NY
5	40	Male	CA

Table 2: **An example table.** The first column labels the tuples for future references and is not part of the table.

We illustrate the notions with an example (Table 2). The example table has 3 attributes. The attribute *age* is a 0.6-distinct key because it has 3 distinct values in a total of 5 tuples; it is a 0.8-separation key because there are 10 distinct pairs of tuples and 8 pairs can be separated by *age*. Readers can verify that the attribute set $\{sex, state\}$ is 0.8-distinct and 0.9-separation.

The problem of finding minimum keys is closely related to the classical minimum set cover problem. (We will show the connection between the two problems in Section 2.) We also define an approximate measure for partial set covers.

> *Minimum Set Cover Problem:* Given a finite set S (called the *ground set*) and a collection C of subsets of S, a *set cover* I is a subcollection of C such that every element in S belongs to at least one member of I. The *Minimum Set Cover* problem asks for a set cover with the smallest cardinality.
>
> An α-*set cover* is a subcollection of C that covers at least a fraction α of elements in S.

We summarize below the contributions of this paper. (In the minimum key problem, let n be the number of tuples and m be the number of columns; in set cover, let n be the size of the ground set S, and m be the number of subsets in C.)

1. We propose two algorithms, Greedy and Random Deletion, that find small semi-keys with provable size and key-approximation guarantees, with space and time requirements sublinear in n. The algorithms are particularly useful when $n \gg m$, which is typical of database applications where a large table may consist of millions of tuples, but only a relatively small number of attributes. The results are summarized in Table 1. (Section 3)

2. Our results on semi-keys are based on a novel technique for solving an approximation version of the minimum set cover problem. We design a randomized algorithm for the minimum set cover problem that uses $O(m^2)$ space and produces an α-set cover within $\ln m + O(1)$ of the minimum set cover size with constant probability. Minimum set cover is a classical problem in theoretical computer science and has important applications in various fields of computer science, so this result may be of independent interest. (Section 3.1)

3. We extend the algorithms to find the approximate minimum β-set covers and separation keys. (Section 3.4)

4. We design fast algorithms for finding all minimal exact, distinct, and separation keys of a given table. (Section 4)

5. We have implemented all the above algorithms and conducted experiments using real data sets. The experiment results validate the theoretical claims about the performance and accuracy of our approximate minimum key algorithms, and confirm the efficiency of algorithms for finding all keys (semi-keys). (Section 5)

2 Preliminaries

In this section, we briefly review the connection between the minimum exact key problem and the minimum set cover, as well as known approximation algorithms for the two problems. The reduction between the two problems and the exact key algorithm play an important role in designing algorithms for approximate keys in later sections.

A special case of the minimum exact key problem, where each attribute is boolean, has been studied under the name "Minimum Test Collection".

> *Minimum Test Collection:* Given a set S of elements and a collection C of subsets of S, a test collection is a subcollection of C such that for each pair of distinct elements there is some set that contains exactly one of the two elements. The Minimum Test Collection problem is to find a test collection with the smallest cardinality.

The Minimum Test Collection problem is known to be NP-hard [1], and approximable within a factor of $1 + 2\ln|S|$ [10].

2.1 Reducing Minimum Key Problem to Minimum Set Cover

The reduction from Minimum Test Collection to Minimum Set Cover has been known for a while, see for example [10, 2]. The reduction can be easily extended to the general minimum key problem where each attribute can be from an arbitrary domain, not just boolean. We describe below the reduction from the minimum key problem to minimum set cover.

Given an instance of the minimum key problem with n tuples and m attributes, reduce it to a set cover instance where the set S consists of all distinct unordered pairs of tuples ($|S| = \binom{n}{2}$). Each attribute c in the table is mapped to a subset containing all pairs of tuples separated by attribute c. Now a collection of subsets covers S if and only if the corresponding attributes can separate all pairs of tuples, i.e., those attributes form a key, therefore there is a one-to-one map between minimum set covers and minimum keys.

Consider the example of Table 2. The ground set of the corresponding set cover instance contains 10 elements where each element is a pair of tuples. The column *age* is mapped to a subset c_{age} with 8 pairs: $\{(1,2),(1,3),(1,5),(2,3),(2,4),(2,5),(3,4),(4,5)\}$; the column *sex* is mapped to a subset c_{sex} with 6 pairs, and *state* 7 pairs. The attribute set $\{age, sex\}$ is a key; correspondingly $\{c_{age}, c_{sex}\}$ is a set cover.

2.2 Approximation Algorithms for Minimum Set Cover and Minimum Key

The greedy algorithm for the minimum set cover problem starts with an empty collection (of subsets) and adds subsets one by one until every element in S has been covered; each time it chooses the subset covering the largest number of uncovered elements. It is well known that this greedy algorithm is a $1 + \ln|S|$ approximation algorithm to the minimum set cover problem.

Theorem 1 *[4] The greedy algorithm outputs a set cover of size at most $1 + \ln|S|$ times the size of the minimum set cover.*

Given a minimum key instance, we first reduce it to a set cover instance where $|S| = \binom{n}{2}$, and then use the greedy algorithm to get a $1+\ln|S|$ approximate solution to the minimum set cover, which maps back to a $1+2\ln n$ approximate solution to the minimum key. It is also known neither problem is approximable within $c\ln|S|$ for some $c > 0$ [2].

3 Finding Minimum Semi-Keys

As pointed out in Section 2, it is not only hard to find the exact minimum key, but also hard to find a good approximate solution. The best approximation algorithm known gives $O(\ln n)$-approximate solution and requires multiple scans of the table, which is expensive for large tables. In this section, we relax the minimum key problem by allowing semi-keys, and design efficient algorithms with approximate guarantees.

We introduce the approximate parameter α meaning that we allow an "error" of at most $1-\alpha$. For example, the *α-Approximate Minimum Set Cover problem* looks for an approximate set cover with small size and requires to output an α-set cover (an approximate set cover that covers at least $\alpha|S|$ elements) with probability at least $1-\delta$. The *α-Distinct (Separation) Minimum Key* problem is defined similarly.

Our algorithms are based on random sampling. We first randomly sample k elements (tuples), and reduce the input set cover (key) instance to a smaller set cover (key) instance containing only the sampled elements

(tuples). We then solve the exact minimum set cover (key) problem in the smaller instance (which is again a hard problem but has much smaller size now; we use the greedy algorithm and a random deletion heuristic to solve the exact problem), and output the solution as an approximate solution to the original problem. The number of samples k is carefully chosen so that the approximate parameter α is guaranteed. We present in detail the algorithms for the approximate minimum set cover in Section 3.1; the α-separation minimum key can be solved by reducing to approximate set cover (Section 3.2); we discuss distinct keys in Section 3.3.

Note that α indicates our "error tolerance", not our goal. Take α-separation minimum key as example. Its goal is to find a key as close to the minimum exact key as possible and our algorithms are likely to output semi-keys whose separation ratio are far greater than α. For example, suppose the minimum key of a given table consists of 100 columns, while the minimum 0.9-separation key has 10 columns, then our 0.9-separation minimum key algorithm may output a semi-key that has say 98 columns and is 0.999-separation. However, sometimes we may not need so high precision, and are interested in finding 0.9-separation keys which have much smaller sizes. For this purpose we consider the (β, α)-separation key problem in Section 3.4, which aims at finding the minimum β-separation key, and again allows an error of $1-\alpha$.

3.1 α-Approximate Minimum Set Cover

Before presenting the algorithms, we consider a fundamental operation indispensable to any set cover algorithm: checking whether a given collection of subsets is a set cover. The basic idea of our algorithms is that we only need to check some randomly sampled elements if we allow approximate solutions. If the collection only covers part of S, then it will fail the check after enough random samples. The idea is formalized as the following lemma.

Lemma 2 *$s_1, s_2, \ldots, s_k$ are k elements independently randomly chosen from S. If $|S'| < \alpha|S|$, then $Pr[s_i \in S', \forall i] < \alpha^k$.*

The proof is straightforward. The probability that a random element of S belongs to S' is $|S'|/|S| < \alpha$, therefore the probability of all k random elements belonging to S' is at most α^k.

3.1.1 Greedy α-Set Cover Algorithm

We combine the idea of random sample checking with the greedy algorithm for the exact set cover. Our greedy α-approximate minimum set cover algorithm works as follows:

1. Randomly choose k elements from S (k is defined later);
2. Reduce the problem to a smaller set cover instance where the ground set $\tilde{S}$ is the set of those k chosen elements and each subset in the original problem maps to a subset which is the intersection of $\tilde{S}$ and the original subset;
3. Apply the greedy algorithm in Section 2.2 to find an exact set cover for $\tilde{S}$, and output the solution as an approximate set cover to S.

Let n be the size of the ground set S, and m be the number of subsets.

Theorem 3 *With probability $1-\delta$, the above algorithm with $k = \log_{\frac{1}{\alpha}} \frac{2^m}{\delta}$ outputs an α-set cover whose cardinality is at most $(1 + \ln \log_{\frac{1}{\alpha}} \frac{2^m}{\delta})|I^*|$, where I^* is the optimal (exact) set cover.*

Proof: Denote by $\tilde{S}$ the ground set of the reduced instance ($|\tilde{S}| = k$); by $\tilde{I}^*$ the minimum set cover of $\tilde{S}$. The greedy algorithm outputs a subcollection of subsets covering all k elements of $\tilde{S}$, denoted by $\tilde{I}$. By Theorem 1, $|\tilde{I}| \le (1 + \ln |\tilde{S}|)|\tilde{I}^*|$. Note that I^*, the minimum set cover of the original set S, corresponds to a set cover of $\tilde{S}$, so $|\tilde{I}^*| \le |I^*|$, and hence $|\tilde{I}| \le (1 + \ln k)|I^*|$.

We map $\tilde{I}$ back to a subcollection I of the original problem. We always have $|I| = |\tilde{I}| \le (1 + \ln k)|I^*| = (1 + \ln \log_{\frac{1}{\alpha}} \frac{2^m}{\delta})|I^*|$.

Now bound the probability that I is not an α-set cover. By Lemma 2, the probability that a subcollection covering less than α of S covers all k chosen elements of $\tilde{S}$ is at most

$$\alpha^k = \alpha^{\log_{\frac{1}{\alpha}} \frac{2^m}{\delta}} = \alpha^{\log_\alpha \frac{\delta}{2^m}} = \frac{\delta}{2^m}.$$

There are 2^m possible subcollections; by union bound, the overall error probability, i.e. the probability that any subcollection is not an α-cover of S but is an exact cover of $\tilde{S}$, is at most δ. Hence, with probability at least $1-\delta$, I is an α-set cover for S. □

If we take α and δ as constants, the approximation ratio is essentially $\ln m + O(1)$, which is smaller than $1 + \ln n$ when $n \gg m$. The space requirement of the above algorithm is $mk = O(m^2)$. The greedy algorithm on $\tilde{S}$ takes time $m^2 k = O(m^3)$, and the random sampling takes time mn, therefore the total time is $O(mn + m^3)$.

3.1.2 Random Deletion Algorithm

We now propose a heuristics algorithm for the α-approximate minimum set cover problem. As in Section 3.1.1, we first randomly choose a set $\tilde{S}$ of k elements

from S and reduce the problem to a smaller set cover instance on $\tilde{S}$. Then we use the following procedure to find a set cover of $\tilde{S}$: start with the collection I of all subsets; randomly pick a subset c that has not been picked before, delete c from I if $I - c$ covers $\tilde{S}$, and otherwise keep I unchanged. After all subsets have been picked once, output I as an approximate set cover for S.

We call the algorithm "Random Deletion" because it assigns a random order on the subsets and tries to delete each subset in turn. It is easy to see that I is a minimal set cover for $\tilde{S}$ by the end of the algorithm, i.e. any proper subcollection of I is no longer a set cover.

Theorem 4 *With probability $1 - \delta$, Random Deletion algorithm with $k = \log_{\frac{1}{\alpha}} \frac{m}{\delta}$ outputs an α-set cover .*

Proof: By Lemma 2, the probability that a set covering less than α of S covers all k chosen elements of $\tilde{S}$ is at most $\alpha^k = \alpha^{\log_{\frac{1}{\alpha}} \frac{m}{\delta}} = \frac{\delta}{m}$. There are m collections checked, so the error probability is at most δ by union bound. □

The algorithm only takes $mk = m \log_{\frac{1}{\alpha}} \frac{m}{\delta}$ space, which is $O(m \ln m)$ taking α and δ as constants. The minimal set cover algorithm on $\tilde{S}$ takes at most $2mk$ time: there are m rounds; in each round, we need to check if $I - c$ covers S, which takes time k by keeping a counter for each element recording how many times the element is covered in I and subtracting 1 from the counter for each element in subset c; there is mk cost to initialize the counters. It takes additional mn time to sample from the input, hence the total time requirement is $mn + 2m \log_{\frac{1}{\alpha}} \frac{m}{\delta}$.

Although we cannot provide guarantee of the size of output set cover, experiments on real data sets show that the algorithm often outputs small set covers.

3.2 α-Separation Minimum Key

The reduction in Section 2.1 preserves the separation ratio: an α-separation key separates at least a fraction α of all pairs of tuples, so its corresponding subcollection is an α-set cover; and vice versa. Therefore, we can reduce the α-Separation Minimum Key problem to an α-approximate minimum set cover problem where $|S| = O(n^2)$. Replacing n in the results of Section 3.1 with $\binom{n}{2}$, we get the space and time complexity and key-approximation guarantee as listed in Table 1.

3.3 α-Distinct Minimum Key

The *α-Distinct Minimum Key* problem looks for small semi-keys, and requires to output an α-distinct key with probability at least $1 - \delta$.

Unfortunately, the reduction in Section 2.1 does not necessarily map an α-distinct key to an α-set cover. Let us consider an example of a 0.5-distinct key in a table of 100 tuples. One possible scenario is that projected on the key, there are 50 distinct values and each value corresponds to 2 tuples. This key can separate all but 50 pairs of tuples, hence it is a $1 - \frac{50}{\binom{100}{2}} \approx 0.99$-separation key, corresponding to a 0.99-set cover in the reduced set cover problem. The other possible scenario is that for 49 of the 50 distinct values, there is only one tuple for each value, and all the other 51 tuples have the same value. Then the 0.5-distinct key becomes a 0.75-set cover after reduction. Indeed, an α-distinct key can be an α'-separation key where α' can be as small as $2\alpha - \alpha^2$, or as large as $1 - \frac{2(1-\alpha)}{n}$. Therefore reducing directly to the set cover problem gives too loose bound, and a new algorithm is desired.

Our algorithms for finding α-distinct minimum keys are again based on random sampling. We reduce the input α-distinct key instance to a smaller minimum key instance by randomly choosing k tuples and keeping all m columns. The following lemma bounds the probability that a subset of columns is an (exact) key in the reduced table, but not an α-distinct key in the original table.

Lemma 5 *Randomly choose k tuples from input table T to form table T_1. Let p be the probability that an (exact) key of T_1 is not an α-distinct key in T. Then*

$$p < e^{-\frac{(\frac{1}{\alpha}-1)k^2}{2n}}$$

Proof: Suppose we have n balls distributed in $d = \alpha n$ distinct bins. Randomly choose k balls without replacement, and the probability that the k balls are all from different bins is exactly p. Let $x_1, x_2, \ldots, x_d$ be the number of balls in the d bins ($\sum_{i=1}^{d} x_i = n, x_i > 0$), then

$$p = \frac{\sum_{all\{i_1,i_2,\ldots,i_k\}} x_{i_1} x_{i_2} \ldots x_{i_k}}{\binom{n}{k}}.$$

p is maximized when all x_is are equal, i.e. each bin has $\frac{1}{\alpha}$ balls. Next we compute p for this case. The first ball can be from any bin; to choose the second ball, we have $n - 1$ choices, but it cannot be from the same bin as the first one, so $\frac{1}{\alpha} - 1$ of the $n - 1$ choices are infeasible; similar arguments hold for the remaining balls. Summing up, the probability that all k balls are from distinct bins is

$$\begin{aligned} p &= 1(1 - \frac{\frac{1}{\alpha}-1}{n-1})(1 - \frac{2(\frac{1}{\alpha}-1)}{n-2}) \ldots (1 - \frac{(k-1)(\frac{1}{\alpha}-1)}{n-(k-1)}) \\ &\leq e^{-(\frac{\frac{1}{\alpha}-1}{n-1} + \frac{2(\frac{1}{\alpha}-1)}{n-2} + \frac{(k-1)(\frac{1}{\alpha}-1)}{n-(k-1)})} \\ &< e^{-\frac{(\frac{1}{\alpha}-1)k^2}{2n}} \end{aligned}$$

□

Having converted the minimum distinct key problem in T to the minimum exact key problem in T_1, we can now use the greedy and random deletion algorithms to find small keys in T_1.

For the greedy algorithm, we choose k such that $p \leq \frac{\delta}{2^m}$ to guarantee that the overall error probability is less than δ, so $k = \sqrt{\frac{2\alpha}{1-\alpha} n \ln \frac{2^m}{\delta}}$. Now apply the greedy algorithm to the smaller table and get an exact key with size at most $1 + 2\ln k = 1 + \ln(\frac{2\alpha}{1-\alpha} n \ln \frac{2^m}{\delta})$ times the minimum key size; with probability $1 - \delta$, it is an α-distinct key for T. The approximation ratio is $\ln m + \ln n + O(1)$, which slightly improves the $1 + 2\ln n$ result for the exact key. Note that we can run the greedy set cover algorithm directly on the $k \times m$ table without expanding all the k^2 tuple pairs, so the space requirement is $mk = O(m\sqrt{mn})$. The time requirement is $m^2 k = O(m^2\sqrt{mn})$.

For the random deletion heuristic, we choose k such that $p \leq \frac{\delta}{m}$ to guarantee that the overall error probability is less than δ, hence $k = \sqrt{\frac{2\alpha}{1-\alpha} n \ln \frac{m}{\delta}}$. Now we look for a minimal exact key in the small table T_1, and with probability $1-\delta$ it is an α-distinct key for T. The space requirement is $mk = O(m\sqrt{n \ln m})$. The algorithm for finding a minimal exact key is similar to the minimal set cover algorithm in Section 3.1.2, but each time to check whether a subset is a key takes time $O(mk)$. The overall time requirement is $mn + m^2 k = O(mn + m^2\sqrt{n \ln m})$.

3.4 Minimum (β, α)-Separation Key

As pointed out at the beginning of this section, our α-distinct (α-separation) minimum key algorithms aim at finding keys close to the minimum exact key and are likely to output semi-keys whose key approximation ratios are far greater than α. However, sometimes we may be interested in finding say 0.9-separation keys which have much smaller sizes, and the α-separation minimum key algorithms in Section 3.2 cannot serve this purpose.

In this section, we consider the problems of finding β-set cover (key). However, such problems are even harder than finding exact solutions as the latter is actually a special case where $\beta = 1$. So again we introduce the relaxing parameter α to enable more efficient algorithms.

Minimum (β, α)-Set Cover problem looks for the minimum β-set cover, and allows an error of $1 - \alpha$: we require to output an $\alpha\beta$-set cover with probability at least $1 - \delta$. *Minimum (β, α)-Separation (Distinct) Key* problem is defined similarly.

We present the algorithm for set cover. The minimum (β, α)-separation key problem can be solved by reducing to (β, α)-set cover problem. Unfortunately, we cannot provide similar algorithms for (β, α)-distinct keys.

As before, we use random sampling to reduce the problem to a smaller set cover instance. We need enough samples to tell, with high probability, whether a subcollection covers more than β or less than $\alpha\beta$ fraction of the ground set.

Lemma 6 *If we choose k elements from the ground set S, then for any given set S', we can tell whether $|S'| \leq \alpha\beta|S|$ or $|S'| \geq \beta|S|$ with probability at least $1 - e^{-\frac{\beta k(1-\alpha)^2}{16}}$.*

The proof is attached in the appendix.

Note that the α-approximate minimum set cover problem is the special case of the (β, α)-set cover problem where $\beta = 1$, therefore Lemma 6 also applies to α-cover. However, Lemma 2 provides a tighter bound for the special case. Suppose we want to tell whether one subcollection is an exact cover or not an α-cover with error probability at most δ. We need $k = \log_\alpha \delta$ samples according to Lemma 2, while Lemma 6 asks for $\frac{16}{(1-\alpha)^2} \ln \frac{1}{\delta}$ samples. For example, when $\alpha = 0.9$, $\log_\alpha \delta \approx 10 \ln \frac{1}{\delta}$, while $\frac{16}{(1-\alpha)^2} \ln \frac{1}{\delta} \approx 1600 \ln \frac{1}{\delta}$.

Our Greedy Minimum (β, α)-Set Cover algorithm works as follows: first randomly sample $k = \frac{16}{\beta(1-\alpha)^2} \ln \frac{2^m}{\delta}$ elements from the ground set S, and construct a smaller set cover instance defined on the k chosen elements; run the greedy algorithm on the smaller set cover instance until get a subcollection covering at least $(1 + \alpha)\beta k/2$ elements (start with an empty subcollection; each time add to the subcollection a subset covering the largest number of uncovered elements).

Theorem 7 *With probability at least $1 - \delta$, the algorithm outputs an $\alpha\beta$-set cover with size at most $(1 + \ln \frac{(1+\alpha)\beta k}{2})|I^*|$, where I^* is the minimum β-set cover of S.*

The proof is attached in the appendix. The algorithm takes space $mk = \frac{16m}{\beta(1-\alpha)^2} \ln \frac{2^m}{\delta}$.

4 Finding All Minimal Keys

In this section, we consider the problem of finding all minimal exact, distinct and separation keys. This problem is inherently hard as the number of minimal keys can be exponential in the number of attributes m, so it is inevitable that the worst case running time is exponential in m. Nevertheless we want to design algorithms efficient in practice, at least for tables with a small number of attributes.

All algorithms in this section perform a search in the lattice of attribute subsets (we describe the basic search procedure in Section 4.1). The key idea for improvement is to use small sample tables to detect non-keys quickly. In order not to prune any key by mistake, we

need to find necessary conditions in the sample table for an attribute set to be a key (semi-key) in the entire table. The necessary conditions are different for exact, distinct and separation keys, and are addressed in Section 4.1, 4.2 and 4.3 respectively.

4.1 Finding All Minimal Exact Keys

We first describe a brute-force algorithm using levelwise algorithm to find all exact keys. Then we will improve upon the basic algorithm by pruning with random samples.

The collection of all possible attribute subsets form a set containment lattice, the bottom of which are all singleton sets and the top is the set of all attributes. Since we are only concerned with the minimal keys, once find a key, we discard all its superset in the lattice. Levelwise algorithm [8] is an efficient algorithm to perform such search on lattices and has been exploited in many data mining applications, see for example [9, 3].

The levelwise algorithm starts the search from the singleton sets and works its way up to the top of the lattice. The level L_{i+1} contains the attribute subsets of size $i+1$ whose size i subsets are all in L_i. Once find a key, we remove the key from its level, and any of its superset will thus never appear in higher levels. The procedure of generating L_{i+1} given L_i is as follows. Sort the attributes in each subset. If two subsets X and Y in L_i have the common prefix of length $i-1$, i.e. match all the attributes except the last one, then generate $Z = X \cup Y$ as a candidate set of L_{i+1}. We need to further check if all size i subsets of Z are in L_i and if Z is not a key; Z is added to L_{i+1} only if both conditions are satisfied. Note that all subsets in L_i sharing the common prefix are clustered together in the lexicographic ordering, so the procedure can be implemented efficiently. Please refer to [9, 3] for more detail of levelwise algorithm.

Now we improve upon the brute-force algorithm by introducing techniques to effectively prune the lattice. We make use of the following simple fact.

Fact *A key of the entire table is still a key in any sub-table; in other words, if a attribute set is not a key in some sub-table, it cannot be a key for the entire table.*

Our pruning algorithm first samples some random tuples to form a small table and find all minimal keys in the small table, which defines a lower border in the lattice. Then we use the levelwise algorithm to search keys in the original table, but start from the lower border instead of the bottom of lattice. [1]

[1]There are two obvious alternatives. One is that when check if an attribute set is a key, keep fetching the next tuple until detect a collision. However the check is hard to implement efficiently if the table cannot be fit in memory. The second alternative is that when check if an attribute set is a key, first check in a small sub-table, and check the original table only if it is a key in sub-table.

Furthermore, we can apply the pruning idea iteratively by constructing a series of tables with increasing tuple numbers. The minimal keys of table i defines a lower border in the lattice where we start the search for minimal keys of table $i+1$.

We implemented all three algorithms (brute-force, pruning with one sub-table, iterative pruning). Because the table is too large to fit in memory, we keep the table in database and check if an attribute set is a key by issuing the SQL query "select count(*) from (select distinct <list of attributes in the checked set> from table) " and comparing the count with the total number of tuples. Experiments show that pruning with small sub-tables improves the running time by orders of magnitude, especially for large tables.

4.2 Finding All Minimal Distinct Keys

The brute-force algorithm for finding all minimal β-distinct keys is similar to that of exact keys: use the levelwise algorithm to search the lattice except that now we check whether an attribute set is a β-distinct key.

However, pruning with sub-tables for distinct keys is not as straightforward as exact keys because the Fact in Section 4.1 does not extend trivially to distinct keys, i.e. a β-distinct key in the input table is not necessarily β-distinct in a sub-table. Fortunately, we have the following lemma analogous to the Fact.

Lemma 8 *Randomly sample k tuples from the input table T into a small table T_1 ($k \ll n$, where n is the number of tuples in T). A β-distinct key of T is an $\alpha\beta$-distinct key of T_1 with probability at least $1 - e^{-(1-\alpha)^2\beta k/2}$.*

Proof: By the definition of β-distinct key, the tuples has at least βn distinct values projected on the distinct key. Take (any) one tuple from each distinct value, and call those representing tuples "good tuples". There are at least βn good tuples in T.

Let k_1 be the number of distinct values in T_1 projected on the distinct key, and k' be the number of good tuples in T_1. We have $k_1 \geq k'$ because all good tuples are distinct. Next we bound the probability $Pr[k' \leq \alpha\beta k]$. Since each random tuple has a probability at least β of being good, and each sample are chosen independently, we can use Chernoff bound (see [7] Ch. 4) and get

$$Pr[k' \leq \alpha\beta k] \leq e^{-(1-\alpha)^2\beta k/2}$$

Since $k_1 \geq k'$, we have

$$Pr[k_1 \leq \alpha\beta k] \leq Pr[k' \leq \alpha\beta k] \leq e^{-(1-\alpha)^2\beta k/2}$$

We implemented this method and found the performance is worse, probably due to the redundance of checking sub-table once above the lower border defined by the minimal keys of the sub-table.

Hence with probability at least $1 - e^{-(1-\alpha)^2\beta k/2}$, the attribute set is an $\alpha\beta$-distinct key of T_1. □

When prune with a sub-table of size k, to guarantee that with probability $1-\delta$ no β-distinct key is pruned, we can set the parameter $\alpha = 1-\sqrt{\frac{2\ln(2^m/\delta)}{\beta k}}$, and prune all attribute sets that are not $\alpha\beta$ keys in the sub-table. Again we can construct a series of tables and conduct pruning iteratively.

We use Chernoff bound in the proof of Lemma 8 to get a clean formula. In fact, when all the parameters (n, k, β, α) are given, we can compute $Pr[k' \leq \alpha\beta k]$ accurately. There are βn good tuples and $(1-\beta)n$ bad tuples, so the probability that i of k chosen elements are bad is $\binom{\beta n}{k-i}\binom{(1-\beta)n}{i}/\binom{n}{k}$. Therefore,

$$Pr[k' \leq \alpha\beta k] = 1 - \sum_{i=0}^{k-\alpha\beta k} \frac{\binom{\beta n}{k-i}\binom{(1-\beta)n}{i}}{\binom{n}{k}}$$

Given a desired error probability δ and fixed n, k, β, we can set α to be the maximum value such that $Pr[k' \leq \alpha\beta k] \leq \delta$ is satisfied.

4.3 Finding All Separation Keys

The idea of pruning with sub-tables is also applicable to finding all β-separation keys. We can use Lemma 6 to set the pruning parameter α.

5 Experiments

We have implemented the algorithms in Section 3 and 4, and conducted extensive experiments using real data sets. All experiments were run on a 2.4GHz Pentium PC with 1GB memory.

5.1 Data Sets

We use two databases *adult* and *covtype* provided by UCI Machine Learning Repository [16]. The *covtype* table has 581012 rows and 54 attributes. *adult* has 15 attributes such as age, education level, marital status, and 32561 rows, among which only 32537 distinct rows. We discard one attribute *fnlwgt* because it is some weight without physical meaning and has too many distinct values; this single attribute is a 0.99-separation key/0.67-distinct key, and all algorithms perform extremely well in identifying it. The number of distinct rows after removing *fnlwgt* is 29096.

Another source of data sets is the census microdata "Public-Use Microdata Samples (PUMS)" [13], provided by US Census Bureau. We use 1 percent samples of state-level Census 2000 data containing individual records. To test the performance of our algorithms on tables with different sizes, we select 4 states with different population sizes. We extract 42 attributes including age, sex, race, education level, salary etc. It turns out that even with all 42 attributes, we cannot distinguish many children, so we only use adult records (age ≥ 20). Table 3 summarizes the sizes of the 4 census tables used in the experiment.

state	total	adult	distinct
Idaho	13112	8881	8867
Washington	59150	41959	41784
Texas	208074	142629	141130
California	338725	235374	233687

Table 3: **Census table sizes.** The "total" column shows the total number of records in the original files; the "adult" column is the number of adults; the "distinct" column is the number of distinct adults with 42 extracted attributes.

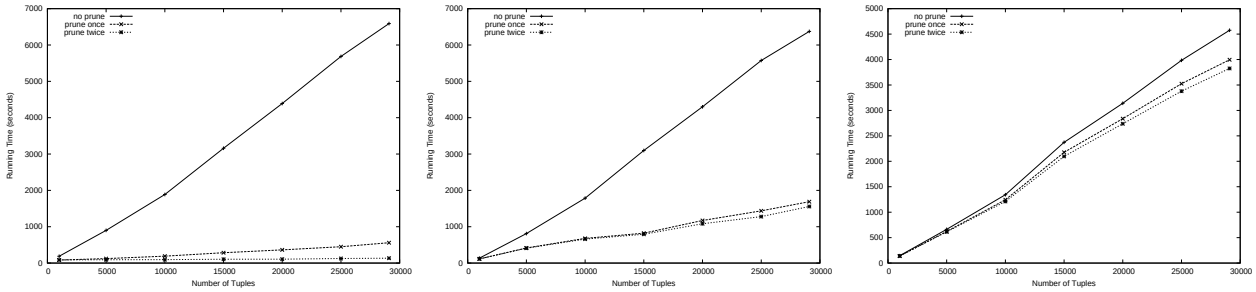
5.2 Performance of Finding All Minimal Keys

We implement algorithms for finding all minimal exact, distinct and separation keys. For each of them, we implement 3 algorithms: no pruning, pruning with one sub-table, iterative pruning.

The pruning algorithms for distinct and separation keys are randomized algorithms that guarantee to output the correct result with probability $1-\delta$. In all the experiments we set $\delta = 0.01$, and the algorithms are able to find all the minimal keys correctly throughout the experiments.

We measure the running time of the three algorithms on *adult* table, and the results are illustrated in Figure 1. The figures show that pruning is highly effective for exact and distinct keys while the improving for separation keys is only marginal; iterative pruning is effective for exact keys. For example, to find all minimal exact keys in the entire table, it takes 6590 seconds without pruning, 559 seconds if prune with one sub-table, and only 134 seconds if prune with two sub-tables; to find all 0.9-distinct keys, pruning improves the running time from 6373 seconds to around 1600 seconds. The improvement on separation keys are not significant, because there are a large number of small separation keys causing pruning ineffective in reducing the search space. We also generate input tables of different sizes by taking random samples from the original table; the running time of all algorithms increases almost linearly with the number of tuples.

We next study the influence of pruning levels and sub-table sizes on the running time. (We do not include separation keys in future experiments of this subsection because the pruning is not effective for separation keys.) From Figure 2, we can see that iterative pruning outperforms pruning with one sub-table in most cases, especially when the pruning table size is large; using three



(a) Running time of finding all keys

(b) Running time of finding all 0.9-Distinct keys

(c) Running time of finding all 0.999-Separation keys

Figure 1: **Comparison of running time of finding all minimal keys using different pruning methods.** The three figures show the running time of three algorithms for finding all exact, 0.9-distinct and 0.999-separation keys in *adult* table respectively. The "pruning once" algorithm uses a 10% sub-table; the iterative pruning algorithm uses two sub-tables, the first one with 1% samples and the second one 10%.

levels of pruning is slightly better than using two levels. Fixing the pruning level and increasing the sub-table sizes, the running time first decreases because larger sub-tables are more effective in pruning the search space, and then increases after a certain point because the gain cannot compensate the cost of sampling and processing the sub-tables. We do not have a general algorithm to compute the optimal pruning level and table sizes; they depend on the characteristics of the input table, such as the sizes and numbers of the keys. Our simulation on various tables show that two or three levels of pruning with each table size 10% of the next level usually provides reasonably good performance.

In Section 4.2, we mentioned that to set the pruning parameter α, we can either use Chernoff bound or compute the exact probability. Table 4 compares the performance of the two methods and shows that computing the exact probability can improve the running time by a factor of 0.2 to 0.3.

	Chernoff	Exact
100/600/3000	2036	1501
300/3000	2054	1556
3000	2107	1688
4500	2003	1651

Table 4: **Setting pruning parameters using Chernoff bound vs by computing exact probabilities.** The second and third columns show the running time (in seconds) of finding all 0.9-distinct keys in *adult* table using different techniques to compute pruning parameters. The first column is the sub-table sizes used for pruning; for example, 300/3000 means using a first table of 300 tuples and a second table of 3000 tuples.

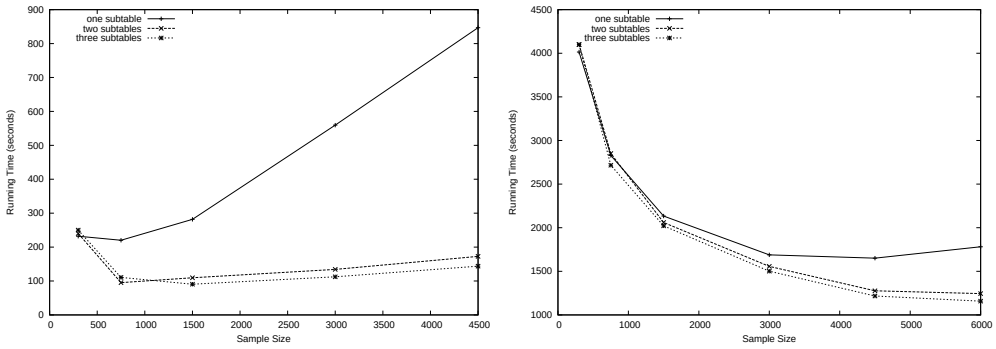
Even though our algorithms have effectively reduced the running time, they are yet not able to process larger data sets as the running time is still exponential in the number of attributes. When the number of attributes reaches 30, with our 2.4GHz PC it takes prohibitively long time to simply go through the attribute set lattice without checking any tuple.

5.3 Performance of Approximate Minimum Key Algorithms

Finding all keys in databases with a large number of attributes is expensive or even infeasible with current hardware. For those data sets, we have to settle with a less ambitious goal and fall back on our approximate algorithms to find the minimum keys. Table 5 and 6 show the experiment results of finding the minimum 0.9-distinct keys and 0.999-separation keys using greedy and random deletion algorithms.

Compared with applying the two algorithms to finding the minimum exact keys (columns "Greedy" and "RD" in Table 5), finding the approximate minimum keys is much faster and the gap in running time increases as the table size increases. For *Idaho* census table with 8867 tuples, all the algorithms take less than 1 minute; when the number of tuples increases to 233687 tuple (*California* table), the greedy algorithm for the minimum exact key takes almost one hour, while the 0.9-distinct minimum key takes two or three minutes, and 0.999-separation key merely seconds. The space and time requirements of our minimum semi-key algorithms are sublinear in the number of tuples, so we expect the algorithms to perform well on even larger data sets.

One observation is that the separation ratios of attribute sets are often quite high. Actually if an attribute only has 2 values and half tuples take on each value, a



(a) Finding all minimal exact keys

(b) Finding all minimal 0.9-Distinct keys

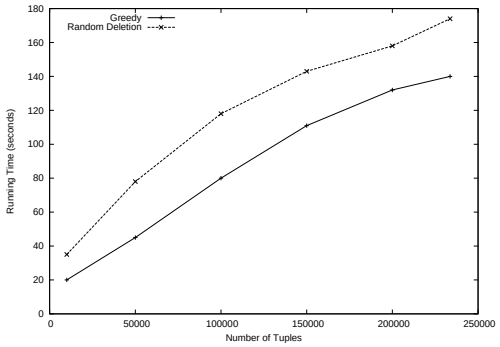
Figure 2: **Running time using different pruning levels and sub-table sizes.** Figures (a) and (b) show the running time (y axis) for finding all exact keys and 0.9-distinct keys in *adult* table respectively. The three curves use one, two, three pruning tables respectively. The x axis gives the pruning table size at the last level; if prune with more than one table, the table size at each level is 10% of the next level, or 100 tuples at minimum.

Data Sets	Greedy		RD		0.9-distinct Greedy			0.9-distinct RD		
	time	key size	time	key size	time	key size	distinct ratio	time	key size	distinct ratio
adult	35.5s	13	9.1s	13	8.8s	13	1.0	10.7s	13	1.0
idaho	50.4s	14	27.2s	14	15.2s	8	0.997	25.1s	8	0.982
wa	490s	22	159s	22	34.1s	8	0.995	62.0s	7	0.985
texas	2032s	29	490s	29	120s	14	0.995	128s	11	0.981
ca	3307s	29	960s	29	145s	13	0.994	174s	9	0.982
covtype	964s	5	450s	5	78.1s	3	0.9997	83.0s	3	0.997

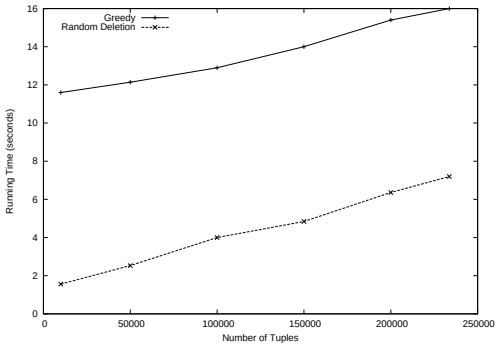
Table 5: **Running time and output key sizes of finding the** 0.9**-distinct minimum keys.** The last two columns show results of Greedy and Random Deletion (abbreviated as RD) algorithms for the 0.9-distinct minimum keys on various data sets, contrasted with the same algorithms for the minimum exact keys (columns "Greedy" and "RD").

Data Sets	0.999-Separation Greedy			0.999-Separation RD		
	time	key size	separation ratio	time	key size	separation ratio
adult	3.11s	5	0.99995	1.34s	5	0.9998
idaho	1.07s	3	0.9999	1.23s	6	0.9993
wa	7.14s	3	0.99993	2.56s	5	0.9991
texas	13.2s	4	0.99995	5.03s	7	0.9997
ca	16.3s	4	0.99998	7.76s	6	0.9999
covtype	27.1s	2	0.999996	20.0s	3	0.99998

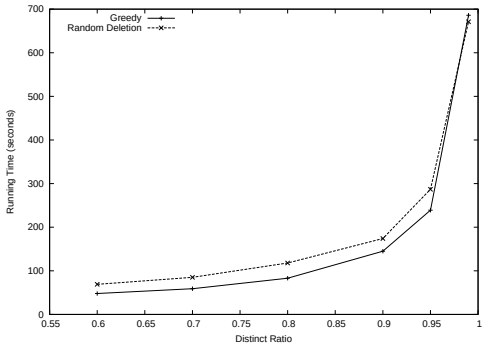
Table 6: **Running time and output key sizes of finding the** 0.999**-separation minimum keys using Greedy and Random Deletion (RD) algorithms.**



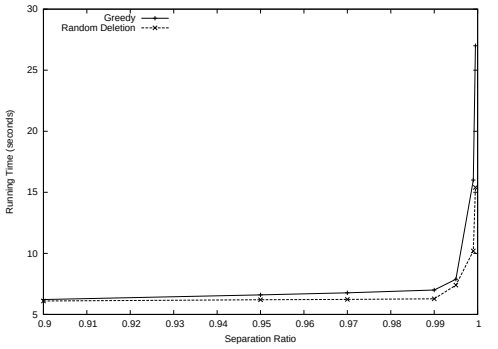
(a) Running time vs. table size (0.9-Distinct keys)



(b) Running time vs. table size (0.999-Separation keys)



(c) Running time vs. distinct ratio



(d) Running time vs. separation ratio

Figure 3: **The relationship of running time with table sizes and distinct (separation) ratios.** Figures (a) and (b) show how running time (y axis) changes as the number of tuples in the table (x axis) increases. We generate input tables of different sizes by taking random samples from the *California* census table. Figures (c) and (d) show how running time (y axis) changes with the distinct (separation) ratio, using the entire *California* census table. In each figure, two algorithms (greedy and random deletion) are measured.

simple calculation shows that the separation ratio of this single attribute is 0.5. From Table 6, we can see that a subset of 4 attributes can often have separation ratios as high as 0.9999.

To our surprise, although the random deletion algorithm does not provide bounded key sizes as the greedy algorithm, it performs well in our experiments. Especially in finding 0.9-distinct keys, it produces semi-keys of size equal to or smaller than the greedy algorithm on all data sets in almost all runs. However, it does not perform as well in separation keys and the output key size is instable (the tables show the average key size over multiple runs). For example, when run on *Idaho* table, it generates 0.999-separation keys whose sizes vary from 3 to 10. Since it only takes time $O(m \ln m)$ to run the random deletion algorithm for separation keys on the sample table, in practice we can afford to run it multiple times on the same sample table and output the smallest key found.

We also measure the distinct(separation) ratios of the output keys, and find the ratios much higher than the requirement. When required 0.9-distinct keys, the greedy algorithm usually outputs keys with distinct ratio > 0.99, and the random deletion outputs keys with distinct ratio > 0.98. The experiment results verify that our algorithms output semi-keys with key-approximation guarantee. On the other hand, if people are interested in finding 0.9-distinct keys, they should use the (β, α)-key algorithms in Section 3.4 instead.

We next study how the running time changes with the number of tuples n and distinct (separation) ratio, and the experiment results confirm our analysis (illustrated in Figure 3). In the greedy and random deletion algorithms for α-separation minimum keys, the sample size is independent on n but it takes linear time to read the input table, so the total running time increases linear

with n; for α-distinct keys, the sample size and the time to process the sample table are proportional to $\sqrt{n}$. The running time also increases as we increase the key approximation ratio α, and the curves get very steep when distinct (separation) ratio gets close to 1.

6 Related Work

A special case of the minimum exact key problem has been studied under the name of "Minimum Test Collection", and there exist a series of papers in theoretical computer science field studying its hardness and approximability [1, 10, 2]. It is also noticed that the minimum test collection problem can be reduced to the well-studied set cover problem [10].

As noted in Section 1, keys are special cases of functional dependencies, and (approximate) functional dependency has received considerable interests. [5, 12] propose measures for quantifying approximations of functional dependencies. Many algorithms and systems have been developed for discovering exact and/or approximate functional dependencies, for example [6, 5, 3, 11]. These algorithms can of course be used for key and semi-key discovery, but is not as efficient for this specific purpose since they are designed for a more general problem.

The idea of pruning with a smaller sample table has been exploited in other data mining applications such as associate rule mining [14, 15]. We are not aware of any work having used this idea for key and semi-key discovery.

7 Conclusions and Future Work

In this paper, we design efficient algorithms for discovering keys and semi-keys in large tables. we develop efficient algorithms that find small semi-keys with provable size and key-approximation guarantees, with space and time complexity sublinear in the number of tuples. We also design fast algorithms for finding all minimal exact, distinct and separation keys.

The idea of pruning the attribute set lattice with small sample tables can potentially be explored in other data mining applications. We plan to apply the technique to accelerate (approximate) functional dependency mining.

Another interesting topic of future research is to study whether our algorithms are space optimal. The analysis is tight for the current algorithms, but there may exist different sampling methods. For example, for the α-separation minimum key algorithm, we sample k random pairs of tuples, which requires $2k$ tuples; $2k$ tuples can produce almost $2k^2$ pairs, but we are not using most of them. There may exist other algorithms making better use of the sampled tuples and requiring fewer samples. Another example is finding all distinct keys. We have tried another pruning criterion alternative to Lemma 8: an β-distinct key in the entire table must be an β'-separation key for some β' in the sub-table. However, it turns out not an effective pruning criterion in practice. Our current algorithm uses the distinct ratio in the sub-table, but there may exist other quantities more effective in pruning.

8 Acknowledgements

The *Adult* and *Covtype* databases have been obtained from the UCI Machine Learning Repository [16].

References

[1] Michael R. Garey and David S. Johnson. Computers and Intractability. WH Freeman and Company, 1979.

[2] Bjarni V. Halldorsson, M. M. Halldorsson and R. Ravi. On the Approximability of the Minimum Test Collection Problem. *ESA* 2001.

[3] Y. Huhtala, J. Karkkainen, P. Porkka and H. Toivonen. Efficient Discovery of Functional and Approximate Dependencies Using Partitions. *Proceedings of the Fourteenth International Conference on Data Engineering,* pp. 392-401, 1998.

[4] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.* 9, 256-278, 1974.

[5] J. Kivinen and H. Mannila. Approximate dependency inference from relations. *Theoretical Computer Science*, 149(1):129-149, 1995.

[6] H. Mannila and K.J. Raiha. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering*, 12:83-99, 1994.

[7] R. Motwani and P. Raghavan. *Randomized Algorithm.* Cambridge University Press, 1995.

[8] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241-258, 1997.

[9] H.Mannila, H.Toivonen and A.I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259-289, 1997.

[10] B.M.E. Moret and H.D. Shapiro. On minimizing a set of tests. *SIAM Journal on Scientific and Statistical Computing*, 6:983-1003, 1985.

[11] N. Novelli and R. Cicchetti. Fun: an efficient algorithm for mining functional and embedded dependencies. *ICDT*, 2001.

[12] B. Pfahringer and S. Kramer. Compression-based evaluation of partial determinations. *SIGKDD*, 1995.

[13] Public-Use Microdata Samples (PUMS). http://www.census.gov/main/www/pums.html

[14] A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. *VLDB* 1995.

[15] H. Toivonen. Sampling Large Databases for Association Rules. *VLDB* 1996.

[16] D.J. Newman, S. Hettich, C.L. Blake and C.J. Merz. UCI Repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/ ML-Repository.html, 1998.

9 Appendix

Lemma 6 *If we choose k elements from the ground set S, then for any given set S', we can tell whether $|S'| \leq \alpha\beta|S|$ or $|S'| \geq \beta|S|$ with probability at least $1 - e^{-\frac{\beta k(1-\alpha)^2}{16}}$.*

Proof: Let k' be the number of elements covered by S' in the k chosen elements.

If $|S'| \geq \beta|S|$, then a random element of S is covered by S' with probability at least β. Let x_i be an indicator random variable which is set to 1 if the ith chosen element is covered by S', and 0 otherwise. It is easy to see

$E[x_i] = Pr[x_i = 1] = |S'|/|S| \geq \beta$, and

$E[k'] = \sum_{i=1}^{k} E[x_i] \geq \beta k$.

Since all x_is are independent, we can apply Chernoff bound (see for example [7] Ch. 4),

$Pr[k' \leq (1+\alpha)\beta k/2] < e^{-\frac{\beta k(1-\alpha)^2}{8}}$.

On the other hand, if $|S'| \leq \alpha\beta|S|$, similarly we have

$Pr[k' \geq (1+\alpha)\beta k/2] < e^{-\frac{\beta k(1-\alpha)^2}{16}}$.

Therefore, by checking if S' covers more than $(1+\alpha)\beta/2$ fraction of the chosen elements, we can tell with high probability if S' covers more than β or less than $\alpha\beta$ fraction of S. □

Theorem 7 *With probability at least $1-\delta$, the greedy algorithm for (β, α)-set cover outputs an $\alpha\beta$-set cover with size at most $(1 + \ln \frac{(1+\alpha)\beta k}{2})|I^*|$, where I^* is the minimum β-set cover of S.*

To prove the theorem, we need the following lemma about approximation ratio of the greedy algorithm on partial set covers. The proof is similar with the original proof for the exact set cover.

Lemma 9 *Apply the greedy algorithm for minimum set cover problem until get a γ-set cover. The size of result subcollection is within $1 + \ln \gamma n$ of the size of the minimum γ-set cover.*

Proof: For each element e, define $price(e) = \frac{1}{size(c)}$, where c is the first subset covering e in the greedy algorithm, and $size(c)$ is the number of elements first covered by subset c in the greedy algorithm.

Number the elements of S covered by the greedy algorithm in the order of which they were covered by the greedy algorithm, breaking ties arbitrarily. Let $e_1, \ldots, e_{\gamma n}$ be the numbering. Right before e_k is covered, at most $k-1$ elements have been covered, so OPT_γ covers at least $\gamma n - (k-1)$ uncovered elements. Since the greedy algorithm picks a subset c with the maximum $size(c)$, it follows that

$$price(e_k) \leq \frac{OPT_\gamma}{\gamma n - (k-1)}$$

The size of γ-set cover output by the greedy algorithm equals to

$$\sum_{k=1}^{\gamma n} e_k = OPT_\gamma(\frac{1}{\gamma n} + \frac{1}{\gamma n - 1} + \ldots + 1) = (1 + \ln \gamma n) OPT_\gamma$$

□

Proof of Theorem 7

We say a subcollection of subsets "good" if it covers at least $(1+\alpha)\beta k/2$ of the k chosen elements.

We first bound the error probability that the algorithm outputs a subcollection covering less than $\alpha\beta$ of S. According to Lemma 6, the probability that any such subcollection is good is less than

$$e^{-\frac{\beta k(1-\alpha)^2}{16}} = e^{-\ln \frac{2^m}{\delta}} = \frac{\delta}{2^m}.$$

Suppose there are x such small subcollections, then with probability at least $1 - \frac{x\delta}{2^m}$ none of them is good.

Similarly, any β-cover of S will be good with probability $\frac{\delta}{2^m}$. Suppose there are y β-covers, then with probability at least $1 - \frac{y\delta}{2^m}$ all β-covers are good. Under the condition that all β-covers are good, it holds that $|I^*| \geq |\tilde{I}^*|$, where $\tilde{I}^*$ is the minimum good subcollection.

By Lemma 9, the greedy algorithm outputs a good subcollection whose size is within $1 + \ln \frac{(1+\alpha)\beta k}{2}$ of the minimum good subcollection size $|\tilde{I}^*|$, thus also within $(1 + \ln \frac{(1+\alpha)\beta k}{2})|I^*|$ under the condition that all β-covers are good.

The overall error probability is at most $\frac{(x+y)\delta}{2^m}$. Since $x+y$ is at most the total number of subset 2^m, the error probability is at most δ.□