GORDIAN: Efficient and Scalable Discovery of Composite Keys

Yannis Sismanis Paul Brown Peter J. Haas Berthold Reinwald
IBM Almaden Research Center
San Jose, CA, USA
{syannis,pbrown1,phaas,reinwald}@us.ibm.com

ABSTRACT

Identification of (composite) key attributes is of fundamental importance for many different data management tasks such as data modeling, data integration, anomaly detection, query formulation, query optimization, and indexing. However, information about keys is often missing or incomplete in many real-world database scenarios. Surprisingly, the fundamental problem of automatic key discovery has received little attention in the existing literature. Existing solutions ignore composite keys, due to the complexity associated with their discovery. Even for simple keys, current algorithms take a brute-force approach; the resulting exponential CPU and memory requirements limit the applicability of these methods to small datasets. In this paper, we describe GORDIAN, a scalable algorithm for automatic discovery of keys in large datasets, including composite keys. GORDIAN can provide exact results very efficiently for both real-world and synthetic datasets. GORDIAN can be used to find (composite) key attributes in any collection of entities, e.g., key column-groups in relational data, or key leaf-node sets in a collection of XML documents with a common schema. We show empirically that GORDIAN can be combined with sampling to efficiently obtain high quality sets of approximate keys even in very large datasets.

1. INTRODUCTION

Keys play a fundamental role in understanding both the structure and properties of data. Given a collection of entities, a key is a set of attributes whose values uniquely identify an entity in the collection. For example, a key for a relational table is a set of one or more columns such that no two rows have matching values in each of the key columns. The notion of keys carries over into many other settings, such as XML repositories, document collections, and object databases. Identification of keys is a crucially important task in many areas of modern data management, including data modeling [1], query optimization, indexing, anomaly detection, and data integration. The knowledge of keys can be used to (1) provide better selectivity estimates in cost-based query optimization, (2) provide a query optimizer with new access paths that can lead to substantial speedups in query processing, (3) allow the database administrator (DBA) to improve the efficiency of data access via physical design techniques such as data partitioning or the creation of indexes and materialized views, (4) provide new insights into application data, and (5) automate the data-integration process [4].

Unfortunately, in real-world scenarios with large, complex databases, an explicit list of keys is often incomplete, if available at all. In the best case, some small number of keys are explicitly known to the database management system (DBMS) because they are an integral part of the schema, representing important logical relationships among entities. Such keys are often maintained explicitly by the use of referential-integrity constraints. Many other keys are often unknown to the DBMS, because

- the key represents a "constraint" or "dependency" that is inherent to the data domain but unknown to both the application developer and the database administrator (DBA);
- the key arises fortuitously from the statistical properties of the data, and hence is unknown to the application developer and DBA;
- the key is known and exploited by the application without the DBA explicitly knowing about it; or
- the DBA knows about the key but for reasons of cost chooses not to explicitly identify or enforce it.

Many of the unknown keys are *composite* keys, that is, keys consisting of two or more attributes. The unknown keys in a database represent a loss of valuable information.

This paper is concerned with methods for automated, data-driven key discovery. There has been a great demand on the part of industry for such methods, because they vastly simplify the job of the DBA and thereby decrease the overall cost of database ownership. Although there has been much renewed interest in the research and industrial communities [18, 23, 32] in autonomic and self-tuning database technology, the problem of automatic key discovery has received relatively little attention, perhaps due to the challenging nature of the problem. Discovery of composite keys is especially difficult, because the number of possible keys increases exponentially with the number of database attributes. The general problem of discovering a minimal composite key—i.e. a composite key with the fewest possible number of attributes—is NP-complete [14], and associated problems like discovering the exact number of minimal composite keys are even harder, indeed, #P-hard [14]. Because such functionality is nevertheless needed by industry, the goal is to provide practical algorithms that have good "typical case" behavior on real-world datasets. This relatively modest quest is less daunting than it may appear, because the datasets used to demonstrate theoretical worst-case performance tend to be highly artificial and unrepresentative of the type of data encountered in practice.

VLDB '06, September 12-15, 2006, Seoul, Korea.

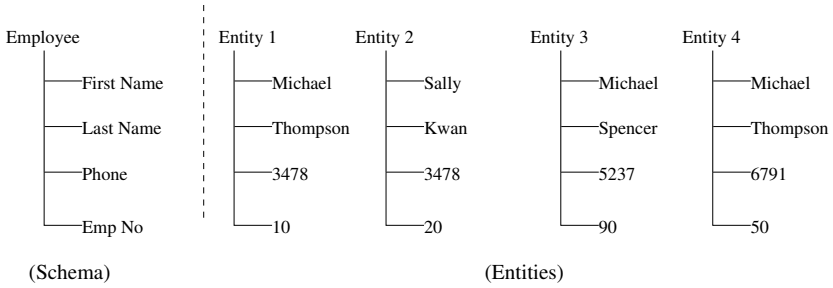

Figure 1: Example Dataset

A brute-force approach, as used in most commercial products, does not scale as the database size grows. For example, one of the databases that we use in our experiments contains entities having more than 50 attributes, so that the number of candidate keys is $2^{50}-1$. Even if we argue that most interesting keys are composed of at most three or four attributes, we would still be faced with $\binom{50}{1}+\binom{50}{2}+\binom{50}{3}+\binom{50}{4} = 251{,}175$ candidate keys. The existing research literature on key discovery [3, 17, 21, 24, 33] has not directly addressed the problem of composite keys.

In this paper, we introduce GORDIAN, a novel algorithm for efficiently discovering composite keys in a collection of entities. GORDIAN represents a vast improvement over currently used brute-force techniques. In our experiments, GORDIAN performed well on both real-world and synthetic databases with large numbers of both attributes and entities. As discussed below, it can even be shown that GORDIAN, when restricted to a certain class of generalized Zipfian datasets, has a time complexity that is polynomial in both the number of entities and attributes.

The basic idea behind GORDIAN is to formulate the problem as a cube computation problem [13] and then to interleave the cube computation with the discovery of all *non-keys*—sets of attributes that are *not* keys. Finally, GORDIAN efficiently computes the complement of this collection, yielding the desired set of keys. In our setting, the cube computation corresponds to the computation of the entity counts for all possible attribute projections of a given dataset. From such counts we can quickly identify whether or not a projection corresponds to a composite key. Many optimizations are possible during the cube computation, because we are not interested in storing, indexing, or even fully computing the cube. Working with non-keys instead of keys is advantageous for a couple of reasons. First, a non-key can often be identified after looking at only a subset of the entities—unlike keys, a discovered non-key cannot subsequently be "invalidated" as more entities are examined. Moreover, any subset of the attributes in a non-key is also a non-key, so that GORDIAN can apply pruning techniques reminiscent of those used in the Apriori algorithm for association-rule mining [28]. GORDIAN applies a number of other powerful pruning techniques that can further reduce the time and space requirements by orders of magnitude. Finally, experiments show that when GORDIAN is applied to a relatively small sample of the data, the algorithm discovers a high-quality set of "approximate" keys as in [21].

The remainder of the paper is organized as follows. In Section 2, we discuss several important concepts related to keys and non-keys, and develop intuition about keys and non-keys through an example. Section 3 contains the description and analysis of the GORDIAN algorithm. Section 4 contains the results of an empirical evaluation of GORDIAN on both synthetic and real-world datasets. We discuss related work in Section 5 and conclude in Section 6.

2. KEYS AND NON-KEYS

As discussed above, given a schema (i.e., set of attributes) R and a set of entities $\mathscr{R}$ over R, we define $K \subseteq R$ as a *key* if and only if for any $t,u \in \mathscr{R}$ we have $t[K] = u[K]$ only if $t = u$.[1] Similarly, we define $K \subseteq R$ as a *non-key* if and only if there exist $t,u \in \mathscr{R}$ such that $t[K] = u[K]$ but $t \neq u$. If K is a key, then the projection of $\mathscr{R}$ onto the attributes in K (with duplicate removal) results in an entity-set of the same size as $\mathscr{R}$; if K is a non-key, then the projection of $\mathscr{R}$ onto the attributes in K results in an entity-set strictly smaller than $\mathscr{R}$.

For example, consider the simple dataset $\mathscr{R}$ in Figure 1, which comprises four entities. Here ⟨EmpNo⟩ is a key, because `EmpNo` uniquely identifies an employee. Moreover, ⟨Last Name, Phone⟩ is a composite key. If we project on either of these keys, the result is a set containing four entities. Because three entities share the first name 'Michael', it follows that $K =$⟨First Name⟩ is a non-key. If we project on `First Name`, the result is a set containing only two entities. Because there is a 1-to-1 correspondence between a set of attributes and a projection of $\mathscr{R}$ onto the attributes, we sometimes use the term "projection" instead of "key", "non-key," "candidate non-key," and so forth.

If $K \subseteq R$ is a non-key with respect to a dataset $\mathscr{R}$ and $K' \subseteq K$, then K' is a non-key. For example, if ⟨First Name, Last Name⟩ is a non-key (because there are two Michael Thompsons), then ⟨First Name⟩ is a non-key (there are at least two Michaels) and ⟨Last Name⟩ is a non-key (there are at least two Thompsons). In this situation we say that K *covers* K' or, equivalently, that K' is *redundant* to K. A set of non-keys $\{K_1, K_2, \ldots\}$ is *non-redundant* or *minimal* if and only if $K_j \not\subseteq K_i$ for all $i \neq j$. The non-redundant non-keys for our running example are ⟨Phone⟩ and ⟨First Name, Last Name⟩. As part of its operation, GORDIAN maintains a *NonKeySet* container that holds a set of non-redundant non-keys; see Section 3.6.

As mentioned in the introduction, non-keys are easier to identify than keys. Suppose, for example, that we have examined the first three entities in the example dataset, and have determined that ⟨First Name⟩ is a non-key with respect to the entities processed so far. Then we know that ⟨First Name⟩ must be a non-key with respect to the entire dataset. Keys, on the other hand, do not have this nice property. Suppose that we have examined the first three entities in our example dataset and have determined that ⟨Last Name⟩ is a key with respect to the entities processed so far. Then, at any point in the future, this property might be invalidated by some entity that we haven't seen yet. Indeed, we will discover that ⟨Last Name⟩ is actually not a key after we examine the final entity.

GORDIAN converts non-keys to keys during the final stage of its

[1] See [1]; here, as usual, $t[X]$ denotes the specific values of the attributes in $X \subseteq R$ for the entity t.

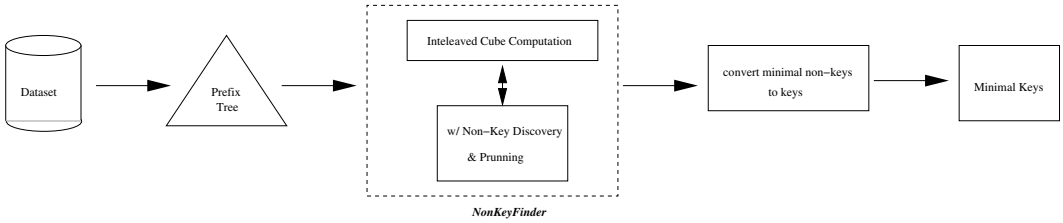

Figure 2: GORDIAN Overview

operation, and the following definitions are pertinent to this process. The *complement* of a non-key is the set of single-attribute keys that correspond to the attributes not appearing in K and provides the starting point for converting non-keys to keys. Formally, $C(K) = \{\langle a\rangle : a \in R \setminus K\}$. For example, the complement of the non-key ⟨First Name, Last Name⟩ is the set $\{\langle$Phone$\rangle, \langle$EmpNo$\rangle\}$. The covering relationship for keys is the reverse of the relationship for non-keys: a key K' is *redundant* to a key K if $K \subseteq K'$. We define a non-redundant set of keys analogously to our definition of a non-redundant set of non-keys.

3. GORDIAN ALGORITHM

In this section we present the complete GORDIAN algorithm. We first give an overview of GORDIAN, explaining the intuition behind our approach, and then we describe the details of the algorithm in subsequent subsections.

3.1 Overview of GORDIAN

3.1.1 Using the CUBE operator

The main idea behind GORDIAN is that the problem of discovering (composite) keys can be formulated in terms of the cube [13] operator. The cube operator encapsulates all possible projections of a dataset while computing aggregate functions on the projected entities. Figure 3 depicts some possible projections for the *count* aggregate function. We observe that a projection corresponds to a key if and only if all the count aggregates for a projection are equal to 1. For example, ⟨EmpNo⟩ and ⟨First Name, Phone⟩ are keys, while ⟨First Name, Last Name⟩ is a non-key. Since we are not dealing with the full complexity of the cube operator (i.e full computation, storing, or indexing), many optimizations are possible.

FirstName	LastName	Phone	EmpNo	COUNT
Michael	Thompson	3478	10	1
Sally	Kwan	3478	20	1
Michael	Spencer	5237	90	1
Michael	Thompson	6791	50	1

FirstName	LastName	COUNT
Michael	Thompson	2
Sally	Kwan	1
Michael	Spencer	1

FirstName	COUNT
Michael	3
Sally	1

Phone	COUNT
3478	2
5237	1
6791	1

FirstName	Phone	COUNT
Michael	3478	1
Sally	3478	1
Michael	5237	1
Michael	6791	1

LastName	COUNT
Thompson	2
Kwan	1
Spencer	1

EmpNo	COUNT
10	1
20	1
90	1
50	1

Figure 3: A subset of the cube operator for the Dataset in Fig. 1

FirstName	LastName	Phone	EmpNo	COUNT
Michael	Thompson	3478	10	1
Michael	Spencer	5237	90	1
Michael	Thompson	6791	50	1

FirstName	LastName	COUNT
Michael	Thompson	2
Michael	Spencer	1

Figure 4: Some segments of the slice `First Name`='Michael'

3.1.2 Singleton Pruning Overview

GORDIAN exploits a novel form of powerful pruning, called *singleton* pruning, that is based on a *slice-by-slice* computation of the cube. A *slice* of the cube is defined as a cube that is based on a subset of the entities; the subset is obtained via a selection operation on the dataset. Whereas a cube comprises all possible projections, a slice comprises the *segments* of the projections that correspond to the slice selection.

For example, consider the dataset depicted in Figure 1 and assume that we have computed the *slice* $\mathscr{F}$ of the cube that corresponds to `First Name` = Michael. In Figure 4 we depict some segments that correspond to that slice.

LastName	Phone	EmpNo	COUNT
Thompson	3478	10	1
Thompson	6791	50	1

LastName	COUNT
Thompson	2

Figure 5: Some segments of the slice `Last Name` = 'Thompson'

Now consider the slice $\mathscr{L}$ of the cube that corresponds to `Last Name` = Thompson. Because (in the full dataset) the value 'Thompson' appears only with the value 'Michael', it follows that the slice $\mathscr{L}$ is *subsumed* by the slice $\mathscr{F}$, in the sense that all the segments of $\mathscr{L}$ already appear in $\mathscr{F}$ with just the `First Name` attribute 'Michael' prepended to them. Thus all the aggregate counts of $\mathscr{L}$ appear in $\mathscr{F}$ (with 'Michael' prepended). It follows that any non-keys of $\mathscr{L}$ appear in $\mathscr{F}$ with the additional attribute `First Name`. Therefore, as discussed in Section 2, each non-key of $\mathscr{L}$ is redundant to some non-key of $\mathscr{F}$. Indeed in our example, the slice $\mathscr{F}$ contains the non-key ⟨First Name, Last Name⟩ and the slice $\mathscr{L}$ contains the (redundant) non-key ⟨Last Name⟩. This observation leads to the following lemma:

LEMMA 1. *If a slice $\mathscr{L}$ is subsumed by another slice $\mathscr{F}$ then each non-key of $\mathscr{L}$ is redundant to some non-key of $\mathscr{F}$.*

This simple yet powerful result allows GORDIAN to avoid all computation and traversal of subsumed slices, without the need to consult the *NonKeySet* container. See Section 3.4.1 for details.

3.1.3 Futility Pruning Overview

Futility pruning complements singleton pruning, using a repository of the non-keys discovered so far to avoid computing segments of future slices. For example, if at some time we determine that ⟨First Name, Last Name⟩ is a non-key by finding an aggregate count greater than 1, then we suppress computation of the ⟨First Name, Last Name⟩ segments as well as the ⟨First Name⟩ and ⟨Last Name⟩ segments when processing future slices. We refer to such prunings as *futile prunings*; see Section 3.4.2 for details.

3.1.4 Computing Projections using Prefix Trees

The general flow of the GORDIAN algorithm is shown in Figure 2. First, the dataset is compressed into a compact representation called a *prefix tree* during a single pass through the data. The prefix-tree representation (see Section 3.2) minimizes both space and processing requirements and, more importantly, facilitates efficient singleton pruning.[2] While GORDIAN is computing the cube aggregates, it keeps track of any discovered non-redundant non-keys, and finally, it converts the non-keys to a set of non-redundant keys.

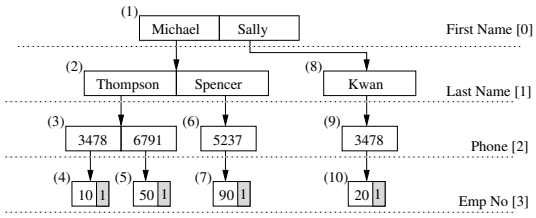

Figure 6: Prefix Tree for data in Figure 1

Figure 6 shows the prefix tree for the dataset of Figure 1. As can be seen, each level of the tree corresponds to an attribute in the schema. The "attribute number" for each attribute is displayed in brackets; the *attrNo* variable in the NonKeyFinder algorithm refers to these values. Each node contains a variable number of *cells*, where each cell contains a value in the domain of the attribute corresponding to the node's level; the values within the cells of a node are distinct. Each non-leaf cell has a pointer to a single child node. The idea is that there is a 1-to-1 correspondence between the set of root-to-leaf paths in the tree and the set of unique entities in the dataset. Therefore, each unique prefix of the entities is stored only once (hence the name prefix tree). Leaf-node cells have, in addition to a value, an associated counter—represented by a gray box in the figure—which records the number of times the entity corresponding to the root-to-leaf path appears in the dataset. Nodes are numbered in depth-first order; node numbers in Figure 6 appear in parentheses. Algorithms for creating and manipulating prefix trees are given in Section 3.2.1. Although not depicted here, each cell of the prefix tree also records the sum of the counters over all leaf nodes that are descended from the cell; this structural information is used for pruning (see Section 3.4).

A high-level, simplified representation of GORDIAN's method for finding non-keys is given as Algorithm 1; the full, detailed method is given as Algorithm 4 below. In our simplified version we have suppressed any explicit mention of pruning (Section 3.4); in the full algorithm, many of the traversal and merge steps are not actually executed. Moreover, the full algorithm combines the traversal and merge functionality—displayed separately in Algorithm 1—in a compact and efficient manner.

[2]A similar structure is used in [26, 27] to efficiently answer aggregation queries in a datacube.

Algorithm 1 GORDIAN's Method for Finding Non-Keys

```
 1: t ← root of prefix tree
 2: Traverse(t)                          // recursively explore all slices
 3: return
 4:
 5: Traverse(t):
 6: if at leaf level then
 7:    discover and store non-keys       // see Section 3.6
 8: else
 9:    for c in Children(t) do
10:       Traverse(c)                   // explore slices in depth-first order
11:    end for
12:    Merge(t)                         // explore segments of current slice
13: end if
14: return
15:
16: Merge(t):
17: m ← root of subtree obtained by merging children of t
                                        // see Section 3.2.2
18: Traverse(m)                         // recursive exploration and merging
19: Discard(m)                          // discard prefix tree rooted at m
20: return
```

The GORDIAN algorithm begins by performing a depth-first (DF) traversal of the prefix tree. When all the children of a node are traversed, we compute the slice of the cube that corresponds to the path from the root to the node by recursively *merging* the children of the node. Each merge operation corresponds to the computation of a different segment (i.e., projection) for the slice. For example, for the prefix tree in Figure 6, assume that we have visited all the children of node (3). The path from the root "Michael,Thompson, {3748,6791}" identifies the current slice. By recursively merging the children of (3), we compute all the segments for that slice; one such merge is depicted in Figure 7.

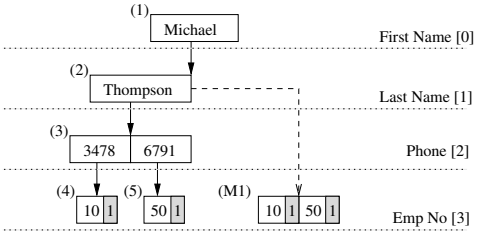

Figure 7: Computing segments for slice="Michael,Thompson"

The doubly recursive nature of GORDIAN (one recursion visits all the nodes and the other merges the children of a visited node) guarantees that—if no singleton or futility pruning is performed—all possible segments for all slices will be generated and traversed. This property provides also an informal sketch of the correctness of GORDIAN: all possible projections are processed and all non-keys are discovered. The details of the merging operation are given in Section 3.2.2, and the details of how GORDIAN performs the doubly recursive DF-traversal are explained in Section 3.3. The operation of the *NonKeySet* container, which maintains a non-redundant set of non-keys, is described in Section 3.6. The final step of the GORDIAN algorithm is to compute a non-redundant set of keys from the set of non-keys in the *NonKeySet* container; this procedure is described in Section 3.7.

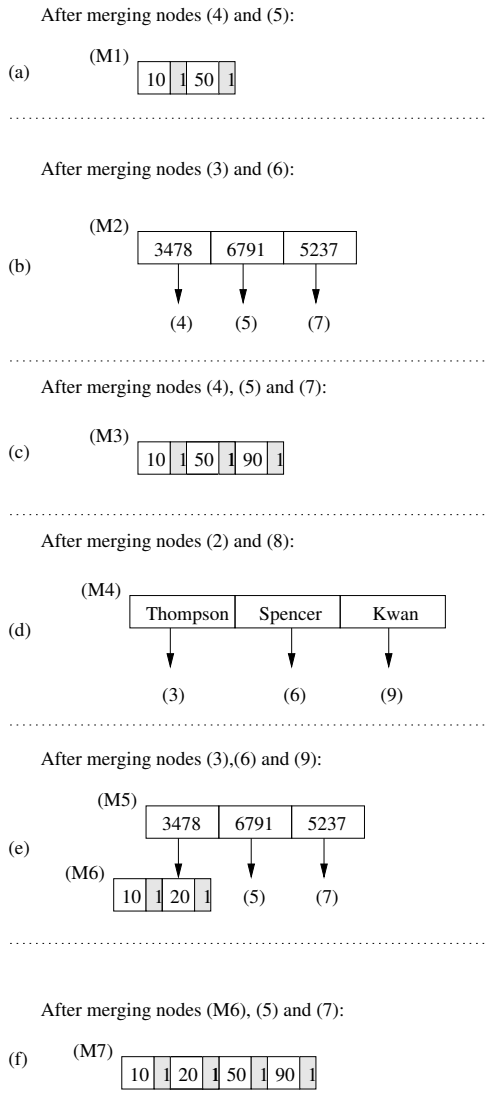

Figure 8: Merges performed for the data in Figure 6

3.2 Prefix-Tree Operations

We now describe how GORDIAN creates prefix trees and merges nodes.

3.2.1 Creating a prefix tree

GORDIAN uses Algorithm 2 to create a prefix-tree representation from an input set of entities. The algorithm requires only a single pass through the data. Variable *root* holds the root of the prefix tree, variable *node* holds the current node and variable *cell* the current cell in *node*.

The algorithm begins by creating the root node of the tree, i.e., node (1). Initially, node (1) is empty and contains no cells at all. For each entity processed, set the variable *node* to *root* (line 3). For each attribute value v_i of this entity, either create a new cell (line 9) and insert the value v_i, or locate the cell in *node* that has value equal to v_i (line 7); in either case, recursively populate the subtree rooted at the cell with the remaining attributes of the entity.

As discussed earlier, if the algorithm ever increases the count of a leaf-node cell to a value greater than 1, then it must be the case that the dataset has no keys at all. GORDIAN therefore aborts its processing and reports that the dataset has no keys (lines 17–18).

Algorithm 2 Prefix Tree Creation

```
Input: DataSet
 1: root ← new empty node
 2: while there are entities in DataSet do
 3:    node ← root
 4:    t ← next entity of DataSet
 5:    for each attribute v_i of t do
 6:       if node contains v_i then
 7:          cell ← cell in node containing v_i
 8:       else
 9:          cell ← create new cell in node
10:          cell.count ← 0
11:          cell.value ← v_i
12:          cell.child ← create new node
13:       end if
14:       if v_i is the last attribute of t then
15:          make cell a leaf
16:          increment cell.count by 1
17:          if cell.count > 1 then
18:             abort GORDIAN and report that no keys exist
19:          end if
20:       else
21:          node ← cell.child
22:       end if
23:    end for
24: end while
25: return root
```

Note that different prefix-tree representations are possible, depending upon the order in which attributes are scanned. GORDIAN finds all keys regardless of representation, and experiments indicate that GORDIAN's performance is relatively insensitive to the choice of representation. One heuristic is to processes attributes in descending order of their cardinality in the dataset, in order to maximize the amount of pruning at lower levels of the prefix tree.

3.2.2 Merging prefix trees

Algorithm 3 is an efficient algorithm for merging nodes to create a modified tree. The merge algorithm takes as input a set of nodes that are to be merged. If the set consists of a single node, then the algorithm immediately returns this node (line 2). The algorithm merges all of the nodes of the input, creating and returning only one node (*mergedNode*). If the nodes to be merged are leaves, then, for each distinct value v that appears in at least one cell, the algorithm creates a new cell in the merged node with value equal to v, and sets the counter equal to the sum of counter values over all input cells having value v (line 9). Otherwise, for non-leaf nodes, the algorithm proceeds by recursively merging the child nodes for each set of cells that share the same value (line 12).

For example, when merging nodes (2) and (8), the algorithm creates a new node that has enough cells to accommodate all the distinct values that appear in nodes (2) and (8). Specifically, it creates three cells with values 'Thompson', 'Spencer' and 'Kwan', respectively. Then the algorithm proceeds recursively for 'Thompson' to merge node (3). However, there is only one node to merge, so the algorithm immediately returns a reference to node (3). The same happens for the child pointers of 'Spencer' and 'Kwan', i.e., nodes (6) and (9) are returned, respectively. The result of merging nodes (2) and (8) is shown in Figure 8(d).

It is worth noting that the merging operation minimizes space consumption by avoiding unnecessary duplication of nodes. E.g.,

Algorithm 3 Prefix Tree Merging

Input: *toMerge*: set of Prefix-tree nodes

```
 1: if there is only one node in toMerge then
 2:    mergedNode ← (the only) node in toMerge
 3: else
 4:    mergedNode ← create new node
 5:    for each distinct value v_i in nodes in toMerge do
 6:       newCell ← create new cell in mergedNode
 7:       newCell.value ← v_i
 8:       if nodes in toMerge are leaves then
 9:          newCell.count ← sum of all counts of cells (in
             toMerge) with values equal to v_i
10:       else
11:          partialSet ← all the children of the cells (in toMerge)
             with values equal to v_i
12:          newCell.child ← Merge(partialSet)
13:       end if
14:    end for
15: end if
16: return mergedNode
```

in Figure 8(d) we see that the newly created node (M4) points to the *existing* nodes (3), (6), and (9)—these nodes are shared, rather than duplicated. As discussed in Section 3.3 below, care needs to be taken when discarding a node, because it might be a shared node, but this inconvenience is vastly outweighed by the space-saving advantages.

When running the merge algorithm on real datasets, it is often the case that most of the merge steps are degenerate, because there is only one node to be merged. This scenario holds especially for sparse datasets with a large number of attributes. Even in the toy dataset of Figure 8, we see that most merges generate a prefix tree with just a single node.

3.3 Finding Non-Keys

The NonKeyFinder routine (Algorithm 4) performs the modified DF-traversal—as introduced in Section 3.1.4—of the prefix tree, and appropriately merges nodes to discover non-keys. NonKeyFinder takes the *root* of a prefix tree and the corresponding level number (i.e., the attribute number as in Figure 6) as input. For the tree in Figure 6, the initial call to NonKeyFinder has *root* equal to node (1) and $attrNo = 0$, which corresponds to the first-level attribute `First Name`. The variable *curNonKey* is static and global, and is initialized to be empty prior to the first (topmost) call to NonKeyFinder. As mentioned earlier, NonKeyFinder consults and updates the *NonKeySet* container which records, in a compressed manner, a non-redundant set of the non-keys discovered so far. The algorithm is carefully designed to avoid producing redundant non-keys: a redundant non-key will either never be discovered, due to the invocation of one of the pruning methods, or it will be discovered but immediately eliminated upon insertion into the *NonKeySet* container.

The algorithm can be summarized as follows. The path from the root to the current node being visited specifies the current slice under consideration, and the variable *curNonKey* contains the current non-key candidate (equivalently, current segment) that NonKeyFinder is working on for the slice. When NonKeyFinder visits a node, it appends *attrNo* to *curNonKey* (Line 1) and then processes the contents of the node. Then it removes *attrNo* from *curNonKey* (Lines 9 and 22), merges the cells of the node using Algorithm 3, and recursively visits the root of the merged prefix tree.

Consider a data-set with three attributes X, Y, and Z. If we ignore for a moment the effects of the pruning mechanisms described in subsequent sections, then, for the current slice under consideration, the order in which NonKeyFinder would traverse all possible non-keys is as depicted in Figure 9; equivalently, the figure shows the order in which segments of the current slice are processed. This ordering makes the pruning mechanism described in Section 3.4 possible.

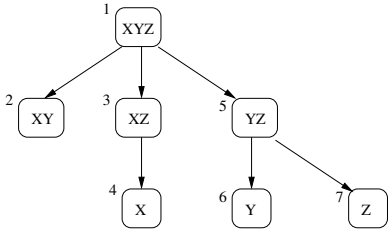

Figure 9: Segment processing order (three attributes)

In more detail, when NonKeyFinder processes a leaf node of the prefix tree, it first checks whether any of counters exceeds 1 and, if so, it adds *curNonKey* to the *NonKeySet* container (Line 5). Then NonKeyFinder removes *attrNo* from *curNonKey*, merging the cells of the leaf node, and then checking if the counter value exceeds 1. Actually, the foregoing operation can be optimized, as in Line 10; i.e. if there is more than one cell in the node or the count of the only cell in the leaf exceeds 1, then *curNonKey* is indeed a non-key, and we insert it into the *NonKeySet* container (Line 11).

When NonKeyFinder processes a non-leaf node, it first recursively visits all the children of the cells in the node (Line 19). Then—after removing *attrNo* from the current non-key candidate *curNonKey*—NonKeyFinder merges the cells in the node using Algorithm 3 and recursively visits the merged prefix tree (Line 28), which it discards afterwards (Line 29). Caution is required when discarding a merged prefix tree to ensure that any shared nodes are retained; in our implementation, a reference-counting scheme was used to this end.

3.4 Search Space Pruning

GORDIAN's pruning techniques speed up NonKeyFinder by orders of magnitude without affecting accuracy. As mentioned earlier, singleton pruning is based on relationships between slices, whereas futility pruning is based on previously discovered non-keys.

3.4.1 Singleton Pruning

We have seen that the sharing of prefix-tree nodes significantly reduces time and space requirements when computing slices of the cube. In this section, we describe an additional benefit of node sharing, namely, pruning of redundant searches.

When NonKeyFinder processes a node, the path from the root to the node specifies the current slice $\mathscr{L}$ under consideration. It may be the case that some cells of the node point to shared and previously traversed prefix (sub)trees, as in Figure 8(d) or, more generally, Figure 10(a). We claim that NonKeyFinder does not need to traverse these subtrees again. To see this, observe that the mere fact that the node points to a previously traversed subtree means that there exists a previously processed slice $\mathscr{F}$ that subsumes $\mathscr{L}$ in the sense of Section 3.1.2. As discussed in Section 3.1.2, this subsumption means that any non-key discovered in $\mathscr{L}$ will be redundant to a previously discovered non-key in $\mathscr{F}$. This observation is exploited in NonKeyFinder by pruning the search in line 18.

Figure 10(b) illustrates an extension of this pruning idea, when

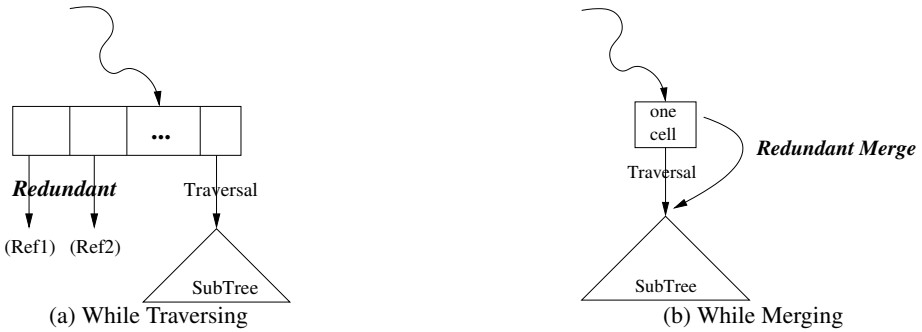

Figure 10: Singleton Pruning

a node with just one cell is being processed. It is obvious that the merging operation will return a shared prefix tree, and thus cannot provide any non-redundant non-keys. This extension is exploited in line 23.

As a final optimization, if NonKeyFinder encounters a prefix tree (i.e., a slice) that corresponds to just one entity, it does not search the tree (line 14). Such a search is unnecessary because no count can exceed 1, and hence the tree cannot yield any non-keys.

3.4.2 Futility Pruning

This pruning operation prevents NonKeyFinder from merging and searching trees that can generate only redundant non-keys. Futility pruning, unlike singleton pruning, uses the non-key container to discover if searching can be pruned.

Recall that if K is a non-key, then $K' \subset K$ implies that K' is a non-key. NonKeyFinder takes advantage of this property by checking for such *futile* trees before merging them. The non-key container holds all of the non-keys seen so far. Before creating a new prefix tree, NonKeyFinder checks (line 24) whether there exists a non-key in the non-key container that covers all of the possible non-keys that could be found. The coverage test for all such paths can be performed very efficiently using bitmaps; see Section 3.6.

3.5 An Example of NonKeyFinder Operation

In this section, we illustrate the NonKeyFinder by applying it to the prefix tree in Figure 6. Although the dataset contains only four entities with four attributes each, the dataset is sufficient to demonstrate all of the concepts discussed so far.

NonKeyFinder performs a DF-traversal on the prefix tree. It starts with the root node (1) and proceeds recursively to nodes (2) and (3) until it arrives at leaf node (4). The current slice therefore corresponds to the entity "Michael,Thompson,3478,10". During this recursive traversal, NonKeyFinder builds up the sequence of attributes in *curNonKey*, i.e. ⟨First Name, Last Name, Phone, EmpNo⟩. Because the count of the (only) cell in (4) equals 1, NonKeyFinder does not find a non-key. The next segment (i.e., non-key candidate) is *curNonKey* = ⟨First Name, Last Name, Phone⟩. Since cell '3478' has only one child, no non-key is found.

Recursively, NonKeyFinder now follows the child pointer of cell '6791' to node (5). The current slice now is "Michael,Thompson, 6791,50" and, just as at node (4), NonKeyFinder doesn't find any non-keys for ⟨First Name, Last Name, Phone, Emp No⟩ and ⟨First Name, Last Name, Phone⟩. NonKeyFinder backtracks to node (3), thereby increasing the slice to the two entities "Michael,Thompson, 3478,10" and "Michael,Thompson,6971,50". Because all the children of node (3) have been traversed, NonKeyFinder merges these children and creates a new prefix tree with a single node (M1); node (M1) is depicted in Figure 8(a). The merge operation essentially projects out the `Phone` attribute from the current slice. The next candidate non-key is now ⟨First Name, Last Name, EmpNo⟩.

NonKeyFinder now traverses (M1). Because all of the cells in the leaf node (M1) have counter values equal to 1, no non-keys are discovered. NonKeyFinder is now finished with (M1) and projects out the leaf attribute (`EmpNo`) to obtain the new candidate non-key ⟨First Name, Last Name⟩. Since (M1) has more than one cell, NonKeyFinder discovers the first non-key ⟨First Name, Last Name⟩ and inserts it into the non-key container. Node (M1) is then discarded.

The recursion backtracks to node (2), so that the current slice is based on all three 'Michael' entities. NonKeyFinder now follows the child pointer of the cell with value 'Spencer' and reaches node (6). As node (6) has only one cell, singleton pruning [as in Figure 10(b)] stops the traversal immediately. NonKeyFinder examines a new segment by merging the children of node (2), thereby creating a prefix tree with node (M2) [Figure 8(b)]. The traversal would now proceed recursively to nodes (4), (5) and (7). However, since all of these nodes have been traversed before, singleton pruning [as in Figure 10(a)] terminates the traversal immediately and NonKeyFinder merges nodes (4), (5) and (7) to create a prefix tree with node (M3) [Figure 8(c)], i.e., the current candidate non-key is set to *curNonKey* = ⟨First Name, EmpNo⟩. By traversing Node (M3) we see that all aggregate counts in the cells are equal to 1, and therefore no non-keys are found. NonKeyFinder now needs to see whether ⟨First Name⟩ is a key. Since we are at the leaf level, a naive procedure would scan node (M3) to see whether this node has more than one cell. But as we have already determined that ⟨First Name, Last Name⟩ is a non-key (this is checked via the *NonKeySet* container), we know that ⟨First Name⟩ would be a redundant non-key. Hence, futility pruning immediately aborts the search.

NonKeyFinder now backtracks to node (1), follows the child pointer of the cell with value 'Sally', and proceeds in a manner similar to that described above. The algorithm eventually discovers the only other non-key, namely ⟨Phone⟩, when it merges the children of the cells in node (1). The search ultimately terminates, having found the non-keys ⟨First Name, Last Name⟩ and ⟨Phone⟩.

3.6 Non-Key Container

As discussed previously, the *NonKeySet* container holds a current set of non-redundant non-keys during NonKeyFinder processing. Algorithm 5 is used to insert a non-key, denoted *NonKey*, into *NonKeySet*. The algorithm goes over the non-keys in the container to check if any of them cover *NonKey* (Lines 2 to 7). If no covering non-key can be found, then the algorithm removes any previously inserted non-keys that are now covered by *NonKey* during a second pass (Lines 8 to 15). The last step of the second pass inserts *NonKey* into the container (Line 14). We use a bitmap representation for non-keys—where each bit corresponds to an attribute of

Algorithm 4 NonKeyFinder

Input: *root*: node of the prefix tree, *attrNo*: attribute number

```
 1: add attrNo as part of curNonKey
 2: if root is leaf then
 3:    for each cell in root do
 4:       if cell.count ≠ 1 then
 5:          add curNonKey to NonKeySet
 6:          break
 7:       end if
 8:    end for
 9:    remove attrNo from curNonKey
10:    if root has more than one cell or the count of the only cell
       exceeds 1 then
11:       add curNonKey to NonKeySet
12:    end if
13: else
14:    if there is only one entity then
15:       return
16:    end if
17:    for each cell in root do
18:       if cell.child is not a shared prefix tree then
19:          NonKeyFinder(cell.child,attrNo +1)
20:       end if
21:    end for
22:    remove attrNo from curNonKey
23:    if there is more than one cell in root then
24:       if curNonKey is futile then
25:          return
26:       end if
27:       mergeTree ← Merge all the children of the cells in root
28:       NonKeyFinder(mergeTree,attrNo +1)
29:       discard mergeTree
30:    end if
31: end if
```

R—both for compactness and for efficiency when performing the redundancy test and other operations.

3.7 Computing keys from non-keys

The final step of the GORDIAN algorithm is to compute a non-redundant set of discovered keys from the set of discovered non-keys. The basic idea is that the set of keys corresponds to the cartesian product of the complement sets (see Section 2) of the *N* non-redundant non-keys. To see this, observe that an element of this cartesian product $K =< A_1, A_2, \ldots, A_N >$ has the property that it is not covered by any of the non-keys, because A_1 is not covered by the first non-key, A_2 is not covered by the second non-key, and so forth. If *K* were a non-key, then it would be covered by at least one of the non-redundant non-keys in the *NonKeySet* container, but it is not, and hence must be a key. Thus we can proceed for each non-key by computing its complement set, taking the cartesian product with the previously-seen complement sets, and pruning any redundant keys on the fly (where redundancy is defined as at the end of Section 2).

Algorithm 6 performs the conversion, using the variable *KeySet* to store the set of keys that will be returned and the variables *complementSet* and *newSet* to hold extra sets of keys for book-keeping purposes. The algorithm reads the first non-key and assigns the complement set of the non-key to *complementSet* (line 3). In our running example, the algorithm computes the complement of the non-key ⟨First Name, Last Name⟩ which is the set of candidate keys ⟨Phone⟩ and ⟨EmpNo⟩. The *KeySet* is currently empty, so *complementSet* is assigned to *KeySet* and we proceed to the next non-key. Again, *complementSet* gets the complement set of the non-key. In our example, the next non-key is ⟨Phone⟩ whose complement set is the candidate keys ⟨First Name⟩, ⟨Last Name⟩ and ⟨EmpNo⟩. Now, for each candidate key *pKey* in the *complementSet* and for each key *Key* already inserted in *KeySet* we insert the union (line 10) of *pKey* and *Key* into the set *newSet*. Then we remove all redundant keys from *newSet* and assign *newSet* to *KeySet*.

The final result is:

Key
⟨EmpNo⟩
⟨First Name, Phone⟩
⟨Last Name, Phone⟩

Algorithm 5 *NonKeySet* Insertion

Input: *NonKey*: non-key to insert, *NonKeySet*: container of non-keys

```
 1: toAdd ← true
 2: for each non-key nk in NonKeySet do
 3:    if nk covers NonKey then
 4:       toAdd ← false
 5:       break
 6:    end if
 7: end for
 8: if toAdd is true then
 9:    for each non-key nk in NonKeySet do
10:       if NonKey covers nk then
11:          remove nk from NonKeySet
12:       end if
13:    end for
14:    add NonKey in NonKeySet
15: end if
```

Algorithm 6 Obtaining the Keys from the Non-Keys

Input: *NonKeySet*: container of non-keys

```
 1: KeySet ← 0
 2: for each NonKey in NonKeySet do
 3:    complementSet ← complement of NonKey
 4:    if KeySet = 0 then
 5:       KeySet ← complementSet
 6:    else
 7:       newSet ← 0
 8:       for each pKey in complementSet do
 9:          for each Key in KeySet do
10:             insert ( Key union pKey ) into newSet
11:          end for
12:       end for
13:       simplify newSet
14:       KeySet ← newSet
15:    end if
16: end for
17: return KeySet
```

3.8 Complexity

Determining the complexity of any sophisticated data-driven algorithm is a challenging task, because it is hard to model all pertinent properties of the data distribution. In our setting, where we are dealing with multi-dimensional datasets, attribute correlations make the problem even harder. In the general case we know that the problem of finding a minimal composite key is NP-complete [14] and indeed we can construct (highly artificial) datasets on which the

behavior of our algorithm is exponential. However, as described in Section 4, GORDIAN performs well on a wide variety of real-world and synthetic datasets.

The following result helps explain GORDIAN's good empirical performance. Due to lack of space, we omit the proof, which is rather long and uses arguments similar to those in [27]. In the following, suppose that

1. The frequencies for each attribute follow a generalized Zipfian distribution with parameter θ, so that the frequency of the ith most frequent value is proportional to $i^{-\theta}$.

2. The only pruning employed by GORDIAN is the sub-case of singleton pruning in which the subsumed slice $\mathscr{L}$ is based on a single entity. This assumption is conservative in that, in actuality, GORDIAN will apply the other available pruning methods, and hence be much more efficient.

3. There are no correlations among the attributes. Note that real data tends to have many complex correlation patterns. Such patterns greatly benefit GORDIAN because they lead to a lot of pruning; thus this assumption is also conservative.

THEOREM 1. *Under Assumptions 1–3 above, the time complexity of* GORDIAN *is*

$$O\left(s \cdot d \cdot T^{1+\frac{1+\theta}{\log_d C}} + s^2\right)$$

and the memory complexity is $O(d \cdot T)$, where s is the number of mutually non-redundant non-keys, d is the number of attributes, C is the average cardinality (number of distinct values) of the attributes, and T is the number of entities.

For uniform data ($\theta = 0$) in which each entity has 30 attributes and 5,000 distinct values per attribute, we have $1 + (\log_d C)^{-1} \approx 1.4$, which implies that the time complexity scales almost linearly with the number of entities. The s^2 term in the complexity expression reflects the cost of computing the keys from the non-keys, and uses the fact that the number of keys is $O(s)$. Although the statistical assumptions of the theorem rarely hold exactly in the real world, our experiments show that GORDIAN's actual performance is clearly superior to the exponential time and polynomial (at best) space requirements of the brute-force approach.

3.9 Sampling

Instead of processing every entity in a dataset of size T, we can process a sample of the entities, with the goal of making GORDIAN scalable to very large datasets. GORDIAN, when applied to a sample, will discover all of the keys in the dataset, but will also discover *false keys*, i.e., sets of attributes that are keys for the sample but not for the entire dataset. Some false keys can be useful, however, if their *strength*—defined as the number of distinct key values in the dataset divided by the number of entities—is sufficiently high. A set of attributes whose strength is close to 1 is called an *approximate key*. (Of course, a true key has strength equal to 1.) Kivinen and Mannila [21] show that, in general, a minimum sample size of $O\left(T^{1/2}\varepsilon^{-1}(d + \log \delta^{-1})\right)$ is needed to ensure that, with probability $(1-\delta)$, the strength of each key discovered in a sample exceeds $1-\varepsilon$. Here, as before, T is the number of entities and d is the number of attributes. This sample size can be large for typical values of the parameters. As with the algorithmic complexity results cited previously, however, the datasets used to establish this theoretical result are rather artificial. For the more realistic datasets that we considered in our experiments, we found that GORDIAN can use a relatively small sample and still produce a high quality set of true and approximate keys.

Precise assessment and control of the strength of the discovered keys is an extremely challenging problem. Indeed, estimation of the strength of a set of attributes is closely related to the notoriously difficult problem of sampling-based estimation of the number of distinct values in a population [16]. The state-of-the-art estimation algorithms are quite expensive to apply in the current setting, so we do not pursue this topic further. Interestingly, we found in our experiments that, with fairly high probability, the quantity

$$\mathscr{T}(K) = 1 - \prod_{v \in K} \frac{N - D_v + 1}{N + 2},$$

is a reasonably tight lower bound on the strength of a sample-based discovered key K, where N is the sample size and D_v is the number of distinct values of attribute v in the sample. This quantity is derived via an approximate Bayesian argument similar to the derivation of Laplace's "rule of succession" [9, Sec. 7.10].

4. EXPERIMENTS

We implemented GORDIAN on top of DB2 V8.2 and applied this prototype to several synthetic, real-world, and benchmark datasets. First, we validated GORDIAN over these datasets and compared GORDIAN to other key-discovery algorithms. We then examined the impact of sample size on GORDIAN's accuracy and speed, as well as the overall impact of GORDIAN on query execution times.

4.1 Experimental Setup

We evaluated GORDIAN on a number of real and synthetic datasets. The TPC-H dataset corresponds to the synthetic database described in [31]. The OPIC dataset is a real-world database containing product information for a large computer company. The BASEBALL dataset contains real data about baseball players, teams, awards, hall-of-fame membership, and game/player statistics for the baseball championship in Australia. Table 1 displays some summary characteristics of the datasets. All experiments were performed on a UNIX machine with one 2.4 GHz processor and 1 GB of RAM. Unless stated otherwise, results are reported for experiments on the OPIC dataset; results for the other datasets are similar.

4.2 Performance Comparison

Figure 11 and Table 2 compare GORDIAN's processing time and memory requirements to (1) a brute-force algorithm that finds all composite keys by checking all possible combinations (2) the brute force algorithm, but limited to finding composite keys with at most four attributes and (3) same as (2), but limited to single-attribute keys only. As can be seen, for roughly the same time and memory needed by a brute-force algorithm to find single-attribute keys, GORDIAN can find *all* composite strict keys, as well as approximate keys.

To study how the number of dimensions affects the relative performance of the foregoing algorithms, we ran GORDIAN on a sequence of datasets having increasingly many attributes. To obtain this sequence, we selected a relation in the OPIC dataset that has 50 attributes, and projected the relation onto 5 attributes, then 10 attributes, and so forth. Figure 12 displays our results. We see that GORDIAN scales almost linearly with the number of attributes and that, although it finds all composite keys, its performance is very close to the approach that just checks for single-attribute keys. For readability, we do not display results for the brute-force approach that checks all possible combinations of attributes—these times were orders of magnitude slower than the rest.

Dataset	Number of Tables	Average #Attributes	Maximum #Attributes	# Tuples (Entities)
TPC-H	8	9	17	866,602
OPIC	106	17	66	27,757,807
BASEBALL	12	16	40	262,432

Table 1: Dataset Characteristics

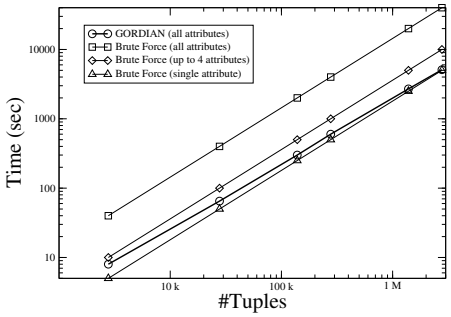

Figure 11: Time comparison

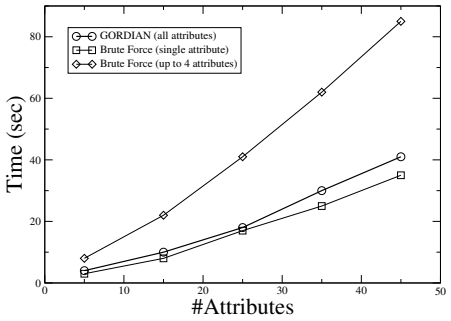

Figure 12: Performance vs. #Attributes

In Figure 13 we show the effect of GORDIAN's pruning methods. We see that singleton pruning and futility pruning together speed up processing by orders of magnitude.

4.3 Effect of Sample Size

Because, as shown above, GORDIAN's execution time scales almost linearly with the number of entities, it follows that the execution time is an almost linear function of the sample size. Thus GORDIANis applicable even to very large datasets.

Of course, GORDIAN identifies not only strict keys but also approximate keys when it operates on a sample of the data; see Section 3.9. Figure 14 shows the minimum strength found for each of the datasets at various sample sizes. We computed the strength exactly, by performing the projection of the full dataset on the key attributes (eliminating duplicates) and dividing by the total number of tuples. We observe that even with fairly small sample sizes, GORDIAN finds a useful set of approximate keys, i.e., having high strength.

Dataset	GORDIAN	Brute force ($\leq$ 4 attrib's)	Brute force (single attrib)
TPC-H	12MB	240MB	6MB
OPIC	100MB	600MB	77MB
BASEBALL	6MB	30MB	4MB

Table 2: Maximum Memory Usage

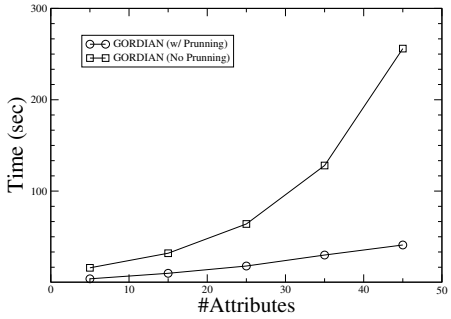

Figure 13: Pruning Effect

To further study the effect of sample size on accuracy, we defined a *false key* as a key with a strength $< 80\%$, and examined the ratio of false keys to true (strict) keys as the sample size varied. Our results are displayed in Figure 15. Again, GORDIAN yields acceptable results even at fairly small sample sizes.

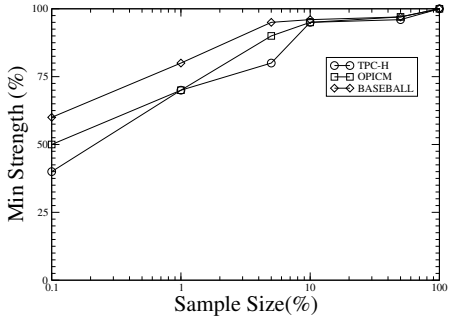

Figure 14: Minimum Strength vs. Sample Size

4.4 Application to Query Execution

As discussed in the introduction, there are many possible uses for the keys discovered by GORDIAN. In this section we discuss one interesting and important use for such keys in the context of query optimization. In this setting, GORDIAN proposes a set of indexes that correspond to the discovered keys. Such a set serves as the search space for an "index wizard" that tries to speed up query processing by selecting the most appropriate indexes based on available storage, workload characteristics, maintenance considerations, and so forth. We explored the applicability of GORDIAN for index recommendation using a synthetic database with a schema similar to TPC-H. The largest table had 1,800,000 rows and 17 columns. GORDIAN required only 2 minutes to discover the candidate indexes. Because we had sufficient storage available, we were "naive" in that we simply built all of the candidate indexes. Figure 16 displays the resulting speedups obtained for a workload

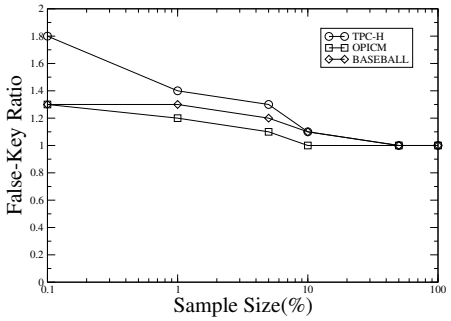

Figure 15: Quality comparison

of 20 typical warehouse queries. For query 4 the speedup was dramatic ($\approx$ 6 times) because the index covered all of the attributes in the query, so that the query was processed using only index pages.

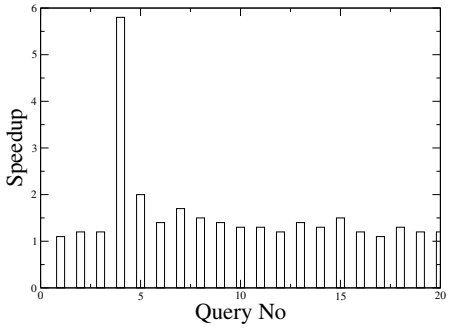

Figure 16: Effects of GORDIAN on Query Execution Time

5. RELATED WORK

Many researchers have focused on the problem of automated metadata discovery, especially in the context of query optimization. The two main approaches can be characterized as either *query driven* or *data driven*. The query-driven, or feedback, approach extracts information from the answers to user queries. An advantage of this approach is that it directs system resources toward the users' needs and interests. For example, the LEO learning optimizer [30] corrects its cardinality estimates based on actual cardinalities observed while processing user queries. Another set of examples example is given by the work in [6, 29], where query feedback is used to maintain self-tuning histogram approximations to the underlying data distribution. Query-driven techniques scale well, and yield immediate gains by focusing on real production queries. These techniques, however, require a "burn-in" period of initial learning. Moreover, these techniques may not be robust when faced with previously-unseen queries or significant changes to the underlying data; in the latter case, the feedback from queries executed at different times can be mutually inconsistent. A variant type of query-driven technique uses information about a query workload, rather than the actual results of executing the queries [5, 7].

Data-driven techniques look directly at the base data, without reference to a query workload. These techniques form an important complement to query-driven methods: while perhaps less accurate, data-driven techniques tend to be more robust. Indeed, the two techniques can be fruitfully combined; see [2] for some work in this direction. Well known data-driven techniques include methods for producing "summary" or "synopsis" data structures such as histograms [20], wavelets [22] and graphical statistical models [12]. These techniques typically do not scale well to high-dimensional data (the so-called "curse of dimensionality"), and the user usually has to select which (few) dimensions to include in the summary. In an attempt to combat the curse of dimensionality, Cheng, et al. [8] construct a Bayesian network model from the base data by using conditional independence tests and a mutual information measure. Application of the method is limited to discrete attributes without any missing values, and has processing complexity of $O(d^4)$ where d is the number of attributes. In [11] a technique is provided that combines a set of low-dimensional histograms with a Markov Network model by searching through the space of possible models and sorting them according to a scoring function. However, the search space is exponential on the number of attributes.

The foregoing techniques focus on discovering metadata related to the joint frequency distribution; numerous other techniques are oriented directly toward relationship discovery. BHUNT [15] is a data-driven technique that can discover algebraic constraints in the data, e.g., SHIPDATE $-$ ORDERDATE $=$ 1 week. CORDS [19] builds upon BHUNT and develops an efficient technique for identifying soft functional dependencies and correlations between pairs of columns. The Bellman system [10] uses a variety of techniques for discovering similarities between (multi)sets of values. There has also been a great deal of work related to mining strict and soft functional dependencies. TANE [17], for example, uses a partitioning of the tuples with respect to their attribute values to quickly test the validity of functional dependencies. TANE can also discover approximate FDs and it implicitly identifies keys (including composite keys) that are used to prune the search space. Our approach can bootstrap TANE in that respect, identifying all keys before executing TANE to find FDs. Kivinen and Mannila [21] provide a sampling-based technique for discovering approximate FDs. In [24], inclusion and functional dependencies are extracted by analyzing equi-join queries embedded in an application program, for the purpose of organizing the database in third-normal form. See [15] and [19] for further references in this area, as well as a discussion of the difference between strict, approximate, and probabilistic dependencies. The literature on mining semantic integrity constraints and association rules—see, for example, [25] and [28]—is a related body of work that, however, focuses on relations between values of attributes rather than the attributes themselves.

With respect to the above taxonomy, GORDIAN is a data-driven method which works directly on the base data. However, GORDIAN can be enhanced to exploit workload information or other DBA knowledge in order to further prune the search space. As in [15, 19, 21], the use of sampling reduces significantly the overhead of processing and makes GORDIAN applicable to real-world environments with thousands of datasets, hundreds of attributes and millions of entities. GORDIAN also works well with updates, since usual referential constraints or triggers can be set to check for the continuing validity of a key.

6. CONCLUSIONS AND FUTURE WORK

We have described GORDIAN, a novel technique for efficiently identifying all composite keys in a dataset. This capability is crucial for many different data management tasks such as data modeling, data integration, query formulation, query optimization, and indexing. Our solution is the first to allow the discovery of composite keys while avoiding the exponential processing and memory requirements that have limited the applicability of previous brute-force methods to very small data sets. Our technique can be used to find keys in any collection of entities, e.g., relational tables or XML repositories. Our empirical study has demonstrated that GOR-

DIAN has excellent real-world performance, discovering all composite keys in the time that previous approaches required to find single-attribute keys. Our study also shows that GORDIAN, when combined with sampling, can quickly find high quality sets of approximate keys in very large datasets. GORDIAN can be effectively used for index recommendation, resulting in dramatic speedups in query processing. We plan to extend our approach to permit identification of foreign-key relationships, thereby automating the discovery of full entity-relationship diagrams.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Aboulnaga, P. J. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, , and V. Raman. Automated statistics collection in DB2 UDB. In *Proc. VLDB*, pages 1146–1157, 2004.

[3] S. Bell and P. Brockhausen. Discovery of constraints and data dependencies in databases. In *Proc. ECML*, pages 267–270, 1995.

[4] P. Brown, P. Haas, J. Myllymaki, H. Pirahesh, B. Reinwald, and Y. Sismanis. Toward automated large-scale information integration and discovery. In T. Härder and W. Lehner, editors, *Data Management in a Connected World*. Springer, 2005.

[5] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. ACM SIGMOD*, pages 263–274, 2002.

[6] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proc. ACM SIGMOD*, 2001.

[7] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. *IEEE Trans. Knowl. Data Engrg.*, 13:7–20, 2001.

[8] J. Cheng, D. A. Bell, and W. Liu. Learning belief networks from data: An information theory based approach. In *Proc. CIKM*, pages 263–274, 1997.

[9] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, 1991.

[10] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proc. SIGMOD*, pages 240–251, 2002.

[11] A. Deshpande, M. Garofalakis, and R. Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proc. ACM SIGMOD*, pages 199–210, 2001.

[12] L. Getoor, T. B, and K. D. Selectivity estimation using probabilistic models. In *Proc. ACM SIGMOD*, 2001.

[13] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M.Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizsing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[14] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, 2003.

[15] P. J. Haas and P. G. Brown. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proc. VLDB*, pages 668–679, 2003.

[16] P. J. Haas and L. Stokes. Estimating the number of classes in a finite population. *J. Amer. Statist. Assoc.*, 93:1475–1487, 1998.

[17] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Tiovonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42:100–111, 1999.

[18] IBM Research. Autonomic computing. In *http://www.research.ibm.com/autonomic*, 2003.

[19] I. Ilyas, V. Markl, P. J. Haas, P. G. Brown, and A. Aboulnaga. CORDS: Automatic generation of correlation statistics in DB2. In *Proc. VLDB*, pages 1341–1344, 2004.

[20] Y. E. Ioannidis. The history of histograms (abridged). In *Proc. VLDB*, pages 19–30, 2003.

[21] J. Kivinen and H. Mannila. Approximate dependency inference from relations. *Theoret. Comput. Sci.*, 149:129–149, 1995.

[22] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-Based histograms for selectivity estimation. In *Proc. ACM SIGMOD*, pages 448–459, 1998.

[23] Microsoft Research. The Autoadmin project. In *http://research.microsoft.com/autoadmin*, 2003.

[24] J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *Proc. ICDE*, pages 218–227, 1996.

[25] M. Siegelan, E. Sciore, and S. Salveter. A method for automatic rule derivation to support semantic query optimization. *ACM Trans. Database Sys.*, 17:563–600, 1992.

[26] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the Petacube. In *Proc. ACM SIGMOD*, 2002.

[27] Y. Sismanis and N. Roussopoulos. The Polynomial Complexity of Fully Materialized Coalesced Cubes. In *Proc. VLDB*, 2004.

[28] R. Srikant and R. Agrawal. Mining quantitative associations rules in large relational tables. In *Proc. ACM SIGMOD*, pages 1–12, 1996.

[29] U. Srivastava, P. J. Haas, V. Markl, and N. Megiddo. ISOMER: Consistent histogram construction using query feedback. In *Proc. ICDE*, 2006.

[30] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. VLDB*, pages 19–28, 2001.

[31] http://www.tpc.org/tpch/default.asp.

[32] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proc. VLDB*, pages 20–34, 2002.

[33] S. K. M. Wong, C. J. Butz, and Y. Xiang. Automated database schema design using mined data dependencies. *J. Amer. Soc. Inform. Sci.*, 49:455–470, 1998.