TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies

YKÄ HUHTALA[1], JUHA KÄRKKÄINEN[1], PASI PORKKA[1] AND
HANNU TOIVONEN[2]

[1]*Department of Computer Science, PO Box 26, FIN-00014 University of Helsinki, Finland*
[2]*Rolf Nevanlinna Institute, PO Box 4, FIN-00014 University of Helsinki, Finland*
Email: juha.karkkainen@cs.helsinki.fi

The discovery of functional dependencies from relations is an important database analysis technique. We present TANE, an efficient algorithm for finding functional dependencies from large databases. TANE is based on partitioning the set of rows with respect to their attribute values, which makes testing the validity of functional dependencies fast even for a large number of tuples. The use of partitions also makes the discovery of approximate functional dependencies easy and efficient and the erroneous or exceptional rows can be identified easily. Experiments show that TANE is fast in practice. For benchmark databases the running times are improved by several orders of magnitude over previously published results. The algorithm is also applicable to much larger datasets than the previous methods.

Received September 17, 1998; revised March 31, 1999

1. FUNCTIONAL AND APPROXIMATE DEPENDENCIES

Functional dependencies are relationships between attributes of a database relation: a functional dependency states that the value of an attribute is uniquely determined by the values of some other attributes. For example, in an address database, zip code is determined by city and street address. The discovery of functional dependencies from relations has received considerable interest (e.g. [1, 2, 3, 4, 5, 6, 7, 8]).

Automated database analysis is, of course, interesting for knowledge discovery and data mining (KDD) purposes. For instance, consider a database of chemical compounds and their outcomes on various bioassays. Discovering that an essential quality, such as carcinogenicity, of a compound depends functionally from certain structural attributes can be invaluable. Functional dependencies also have well-known applications in the areas of database management, reverse engineering [9] and query optimization [10].

Formally, a *functional dependency* over a relation schema R is an expression $X \to A$, where $X \subseteq R$ and $A \in R$. The dependency *holds* or is *valid* in a given relation r over R if for all pairs of tuples $t, u \in r$ we have: if $t[B] = u[B]$ for all $B \in X$, then $t[A] = u[A]$ (we also say that t and u *agree* on X and A). A functional dependency $X \to A$ is *minimal* (in r) if A is not functionally dependent on any proper subset of X, i.e. if $Y \to A$ does not hold in r for any $Y \subset X$. The dependency $X \to A$ is *trivial* if $A \in X$. The central task we consider is the following: given a relation r, find all minimal non-trivial dependencies that hold in r.

An approximate functional dependency is a functional dependency that almost holds. For example, gender is approximately determined by first name. Such dependencies arise in many databases when there is a natural dependency between attributes, but some tuples contain errors or represent exceptions to the rule. The discovery of unexpected but meaningful approximate dependencies seems to be an interesting and realistic goal in many data mining applications. Consider, again, a database of chemical compounds. An approximate dependency from a set of structural attributes to the carcinogenicity could be as valuable as a functional dependency: both could provide valuable hints to biochemists for potential causes of cancer but neither can be taken as a fact without further analysis by domain specialists. Approximate functional dependencies also have applications in database design [11].

There are many possible ways of defining the approximateness of a dependency $X \to A$. The definition we use is based on the minimum number of tuples that need to be removed from the relation r for $X \to A$ to hold in r: the *error* $e(X \to A)$ is defined as $e(X \to A) = \min\{|s| \mid s \subseteq r \text{ and } X \to A \text{ holds in } r \setminus s\}/|r|$. The measure e has a natural interpretation as the fraction of tuples with exceptions or errors affecting the dependency. Given an error threshold ε, $0 \leq \varepsilon \leq 1$, we say that $X \to A$ is an *approximate (functional) dependency* if and only if $e(X \to A)$ is at most ε. In this paper, we also consider the approximate dependency inference task: given a relation r and a threshold ε, find all minimal non-trivial approximate dependencies.

We describe a new approach to the discovery of both functional and approximate dependencies, and we present TANE, an algorithm that implements the ideas. The main innovation is a new method for determining whether a dependency holds or not. The method is based on representing attribute sets by equivalence class partitions of the set of tuples. TANE also has an improved method for searching the space of functional dependencies.

The worst case time complexity of the algorithm with respect to the number of attributes is exponential, but this is inevitable since the number of minimal dependencies can be exponential in the number of attributes [2, 12]. However, with respect to the number of tuples, the time complexity is only linear (provided that the set of dependencies does not change as the number of tuples increases). To our knowledge, only one previous algorithm can claim this [13]. Previous algorithms have almost invariably been based on either repeatedly sorting the tuples of the relation or comparing every tuple to all other tuples and this can obviously be inefficient for large relations. The linearity makes TANE especially suitable for relations with a large number of tuples.

Experimental results show that the algorithm is effective in practice and that it makes the discovery of functional and approximate dependencies feasible for relations with even hundreds of thousands of tuples. Dependency discovery tasks that have been reported to take minutes or even hours are solved with the new algorithm in seconds or fractions of a second on a PC.

1.1. Related work

Several algorithms for the discovery of functional dependencies have been presented [1, 3, 5, 6, 12, 13, 14]. We review these algorithms and compare them with our method in Section 5.3. The complexity of discovering functional dependencies has been studied in [2, 12, 15].

Approximate functional dependencies have been considered in [7, 8, 16, 17]. Kivinen and Mannila [16] define several measures for the error of a dependency and derive bounds for discovering dependencies with errors; they denote the measure e by g_3.

The use of partitions to describe and define functional and approximate dependencies has been suggested in [8] parallel to our work. There the emphasis is on a conceptual viewpoint and no algorithms are given. Partition semantics for relations have been considered in [18], and a rough set approach in [19].

Our search strategy is, on an abstract level, similar to the search of association rules [20]: one first computes some non-trivial information about attribute sets (partitions in our case as opposed to frequent itemsets in the case of association rules), from which the dependencies (versus association rules) can be computed easily. The levelwise method for the computation of dependencies is an instance of the generic data mining algorithm [21], also used successfully in the *a priori* algorithm for association rule mining [20].

1.2. Paper organization

We start in Section 2 by formulating the dependency discovery task in terms of equivalence classes and partitions. In Section 3 we lay out the principles of searching the space of functional dependencies. Detailed algorithms are given in Section 4 and analysed in Section 5. We give experimental results in Section 6 and conclude in Section 7.

An earlier and shorter version of this paper appeared as [22]. Proofs of non-trivial lemmata in this article can be found in [23]. An implementation of the TANE algorithm can be obtained via the WWW page at *http://www.cs.helsinki.fi/research/fdk/datamining/tane/*.

2. PARTITIONS AND DEPENDENCIES

A dependency $X \rightarrow A$ holds if all tuples that agree on X also agree on A. Our approach to the discovery of dependencies is based on considering sets of tuples that agree on some set of attributes. Determining whether a dependency holds or not can be done by checking whether the tuples agree on the right-hand side whenever they agree on the left-hand side. Furthermore, when this is not the case, we can easily identify the tuples that do not agree on the right-hand side. Thus the approach extends naturally to approximate dependencies. Formally, the approach can be described using equivalence classes and partitions.

2.1. Partitions

Two tuples t and u are *equivalent* with respect to a given set X of attributes if $t[A] = u[A]$ for all A in X. Any attribute set X partitions the tuples of the relation into equivalence classes. We denote the *equivalence class* of a tuple $t \in r$ with respect to a given set $X \subseteq R$ by $[t]_X$, i.e. $[t]_X = \{u \in r \mid t[A] = u[A] \text{ for all } A \in X\}$. The set $\pi_X = \{[t]_X \mid t \in r\}$ of equivalence classes is a *partition* of r under X. That is, π_X is a collection of disjoint sets (equivalence classes) of tuples, such that each set has a unique value for the attribute set X and the union of the sets equals the relation r. The *rank* $|\pi|$ of a partition π is the number of equivalence classes in π.

EXAMPLE 1. Consider the relation in Figure 1. Attribute A has value 1 only in tuples 1 and 2, so they form an equivalence class $[1]_{\{A\}} = [2]_{\{A\}} = \{1, 2\}$ (we use here tuple identifiers to denote tuples). The whole partition with respect to A is $\pi_{\{A\}} = \{\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}\}$. The partition with respect to $\{B, C\}$ is $\pi_{\{B,C\}} = \{\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6\}, \{7\}, \{8\}\}$.

2.2. Partition refinement

The concept of partition refinement gives almost directly functional dependencies. A partition π *refines* another partition π' if every equivalence class in π is a subset of some equivalence class of π'. We have the following lemma.

LEMMA 2.1. *A functional dependency $X \rightarrow A$ holds if and only if π_X refines $\pi_{\{A\}}$.*

Tuple ID	A	B	C	D
1	1	a	$	Flower
2	1	A		Tulip
3	2	A	$	Daffodil
4	2	A	$	Flower
5	2	b		Lily
6	3	b	$	Orchid
7	3	C		Flower
8	3	C	#	Rose

Partitions of attributes:

$$\pi_{\{A\}} = \{\{1,2\},\{3,4,5\},\{6,7,8\}\}$$
$$\pi_{\{B\}} = \{\{1\},\{2,3,4\},\{5,6\},\{7,8\}\}$$
$$\pi_{\{C\}} = \{\{1,3,4,6\},\{2,5,7\},\{8\}\}$$
$$\pi_{\{D\}} = \{\{1,4,7\},\{2\},\{3\},\{5\},\{6\},\{8\}\}$$

FIGURE 1. An example relation and its partitions with respect to all attributes.

There is an even simpler test for whether $X \to A$ holds or not: check if $|\pi_X| = |\pi_{X\cup\{A\}}|$. If π_X refines $\pi_{\{A\}}$, then $\pi_{X\cup\{A\}}$ equals π_X. On the other hand, since $\pi_{X\cup\{A\}}$ always refines π_X, $\pi_{X\cup\{A\}}$ cannot have the same number of equivalence classes as π_X unless $\pi_{X\cup\{A\}}$ and π_X are equal. We have shown the following lemma.

LEMMA 2.2. *A functional dependency $X \to A$ holds if and only if $|\pi_X| = |\pi_{X\cup\{A\}}|$.*

2.3. Approximate dependencies

Recall that the error $e(X \to A)$ of a dependency $X \to A$ is the minimum fraction of tuples that must be removed from the relation for $X \to A$ to hold. The error $e(X \to A)$ can be computed from the partitions π_X and $\pi_{X\cup\{A\}}$ in the following way. Any equivalence class c of π_X is the union of one or more equivalence classes $c'_1, c'_2, \ldots$ of $\pi_{X\cup\{A\}}$, and the tuples in all but one of the c'_is must be removed for $X \to A$ to hold. The minimum number of tuples to remove is thus the size of c minus the size of the largest of the c'_is. Summing that over all equivalence classes c of π_X gives the total number of tuples to remove. Thus we have

$$e(X \to A) = 1 - \sum_{c\in\pi_X} \max\{|c'| \mid c' \in \pi_{X\cup\{A\}} \text{ and } c' \subseteq c\}/|r|.$$

An algorithm with which to compute $e(X \to A)$ given the partitions π_X and $\pi_{X\cup\{A\}}$ is described in Section 4.

3. SEARCH

3.1. Search strategy

To find all minimal non-trivial dependencies, TANE works as follows. It starts the search from singleton sets of attributes and works its way to larger attribute sets through the set containment lattice level by level. When the algorithm is processing a set X, it tests dependencies of the form $X \setminus \{A\} \to A$, where $A \in X$. This guarantees that only non-trivial dependencies are considered. The small-to-large direction of the algorithm can be used to guarantee that only minimal dependencies are output. It can also be used to prune the search space efficiently (see Figure 2).

A similar small-to-large search strategy, the levelwise algorithm, has been used successfully in many data mining applications [21]. In addition to effective pruning, the efficiency of the levelwise algorithm is based on reducing the computation on each level by using results from previous levels.

In this section we consider different aspects of the search, including effective pruning criteria for the levelwise algorithm in TANE, as well as fast computation of partitions. Both tasks can be solved efficiently in the levelwise strategy by using information from the previous levels. Based on the material presented in this section, exact algorithms are given in Section 4.

3.2. Pruning the search space

3.2.1. *Rhs candidate pruning*

TANE works through the lattice until the minimal dependencies that hold are found. To test the minimality of a potential dependency $X \setminus \{A\} \to A$, we need to know whether $Y \setminus \{A\} \to A$ holds for some proper subset Y of X. We store this information in the set $\mathcal{C}(Y)$ of right-hand side candidates of Y.

If $A \in \mathcal{C}(X)$ for a given set X, then A has not been found to depend on any proper subset of X. More precisely, the collection of *initial rhs candidates* of a set $X \subseteq R$ is $\mathcal{C}(X) = R \setminus \overline{\mathcal{C}(X)}$, where $\overline{\mathcal{C}(X)} = \{A \in X \mid X \setminus \{A\} \to A \text{ holds}\}$. To find minimal dependencies, it suffices to test dependencies $X \setminus \{A\} \to A$, where $A \in X$ and $A \in \mathcal{C}(X \setminus \{B\})$ for all $B \in X$.

EXAMPLE 2. To illustrate the initial rhs candidate set, assume that TANE is considering the set $X = \{A, B, C\}$ and that $\{C\} \to A$ is a valid dependency. Since $\{C\} \to A$ holds, we have that $A \notin \mathcal{C}(\{A, C\}) = \mathcal{C}(X\setminus\{B\})$, which tells TANE that $\{B, C\} \to A$ is not minimal.

Pruning the search space in TANE is based on the fact that if $\mathcal{C}(X) = \emptyset$, then $\mathcal{C}(Y) = \emptyset$ for all supersets Y of X. Thus no dependency of the form $Y \setminus \{A\} \to A$ can be minimal and the set Y need not be processed at all. The breadth-first search in the set containment lattice can use this information effectively, as illustrated in Figure 2.

3.2.2. *Rhs$^+$ candidates*

While the initial rhs candidates are sufficient to guarantee the minimality of discovered dependencies, we will use improved *rhs$^+$ candidates* $\mathcal{C}^+(X)$ that prune the search space more effectively:

$$\mathcal{C}^+(X) = \{A \in R \mid \forall B \in X : X \setminus \{A, B\} \to \{B\} \text{ does not hold}\}.$$

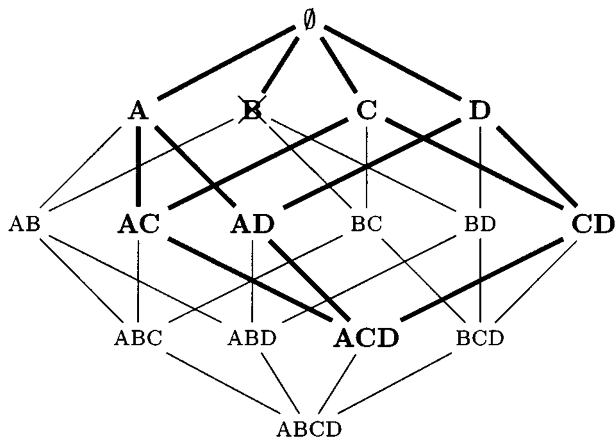

FIGURE 2. A pruned set containment lattice for $\{A, B, C, D\}$. Due to the deletion of B, only the bold parts are accessed by the levelwise algorithm.

Note that A can equal B. The following lemma shows that we can use the rhs$^+$ candidates to test the minimality of a dependency just as we would use the initial rhs candidates.

LEMMA 3.1. *Let $A \in X$ and let $X \setminus \{A\} \to A$ be a valid dependency. The dependency $X \setminus \{A\} \to A$ is minimal if and only if, for all $B \in X$, we have $A \in \mathcal{C}^+(X \setminus \{B\})$.*

The lemma would hold if we replaced $\mathcal{C}^+(X \setminus \{B\})$ with $\mathcal{C}(X \setminus \{B\})$, but rhs$^+$ candidates have two advantages over initial rhs candidates. First, we may encounter a B for which $A \notin \mathcal{C}^+(X \setminus \{B\})$ and stop checking earlier, saving some time. Second and more importantly, for some B, $\mathcal{C}^+(X \setminus \{B\})$ can be empty while $\mathcal{C}(X \setminus \{B\})$ is not. Then, with rhs$^+$ candidates, the set X is never processed due to the pruning.

The definition of $\mathcal{C}^+(X)$ is based on a fundamental property of functional dependencies, stated in the following lemma.

LEMMA 3.2. *Let $B \in X$ and let $X \setminus \{B\} \to B$ be a valid dependency. If $X \to A$ holds, then $X \setminus \{B\} \to A$ holds.*

The lemma allows us to remove additional attributes from the initial rhs candidate sets $\mathcal{C}(X)$. Assume that $X \setminus \{B\} \to B$ holds for some $B \in X$. Then, by the lemma, a dependency with X on the left-hand side cannot be minimal because we can remove B from the left-hand side without changing the validity of the dependency. Hence, we can safely remove from $\mathcal{C}(X)$ the following set:

$$\overline{\mathcal{C}'(X)} = \begin{cases} R \setminus X & \text{if } \exists B \in X : X \setminus \{B\} \to B \text{ holds} \\ \emptyset & \text{otherwise.} \end{cases}$$

EXAMPLE 3. Assume that TANE is considering the set $X = \{A, B, C\}$ and that $\{C\} \to B$ is a valid dependency. Then $A \in \overline{\mathcal{C}'(\{B, C\})} = \overline{\mathcal{C}'(X \setminus \{A\})}$ which tells TANE that $X \setminus \{A\} \to A$ is not minimal. Note that TANE does not need to know whether $X \setminus \{A\} \to A$ holds or not.

Furthermore, assume that X has a proper subset Y such that $Y \setminus \{B\} \to B$ holds for some $B \in Y$. Then we can also remove from $\mathcal{C}(X)$ all $A \in X \setminus Y$. The set of attributes removed by this rule is the following:

$$\overline{\mathcal{C}''(X)} = \{A \in X \mid \exists B \in X \setminus \{A\} : X \setminus \{A, B\} \to B \text{ holds}\}.$$

EXAMPLE 4. Assume that TANE is considering the set $X = \{A, B, C, D\}$, and that $\{C\} \to B$ is a valid dependency. Then $A \in \overline{\mathcal{C}''(\{A, B, C\})} = \overline{\mathcal{C}''(X \setminus \{D\})}$ which tells TANE that $X \setminus \{A\} \to A$ is not minimal.

Finally, the following lemma shows that the sufficient but optimized set of rhs$^+$ candidates $\mathcal{C}^+(X)$ can be also defined in terms of $\overline{\mathcal{C}(X)}$, $\overline{\mathcal{C}'(X)}$, and $\overline{\mathcal{C}''(X)}$.

LEMMA 3.3.

$$\begin{aligned} \mathcal{C}^+(X) &= \{A \in R \mid \forall B \in X : \\ &\qquad X \setminus \{A, B\} \to \{B\} \textit{ does not hold}\} \\ &= ((R \setminus \overline{\mathcal{C}(X)}) \setminus \overline{\mathcal{C}'(X)}) \setminus \overline{\mathcal{C}''(X)} \end{aligned}$$

3.2.3. Key pruning

An attribute set X is a *superkey* if no two tuples agree on X, i.e. partition π_X consists of singleton equivalence classes only. The set X is a *key* if it is a superkey and no proper subset of it is a superkey. When a key is found during the search of dependencies, additional pruning methods can be applied.

LEMMA 3.4. *Let $B \in X$ and let $X \setminus \{B\} \to B$ be a valid dependency. If X is a superkey, then $X \setminus \{B\}$ is a superkey.*

Normally, a dependency $X \to A$, $A \notin X$, is tested when $X \cup \{A\}$ is processed because we need $\pi_{X \cup \{A\}}$ for validity testing. However, if X is a superkey then $X \to A$ is always valid and we do not need $X \cup \{A\}$.

Now, consider a superkey X that is not a key. Obviously, a dependency $X \to A$ is not minimal for any $A \notin X$. Furthermore, if $A \in X$ and $X \setminus \{A\} \to A$ holds, then, by Lemma 3.4, $X \setminus \{A\}$ is a superkey and we do not need π_X for testing the validity of $X \setminus \{A\} \to A$. In other words, we have no use for X or π_X in finding minimal dependencies. Hence, we can prune all keys and their supersets, i.e. the superkeys that are not keys.

3.3. Computing with partitions

We next introduce two ways to reduce the time and space requirement of working with partitions. The first one replaces partitions with a more compact representation, 'stripped partitions'. The second one is a method to quickly approximate the e error. These methods optimize the algorithms described in the following section. We then describe how to compute partitions efficiently in the levelwise TANE algorithm.

For both optimizations we need the concept of approximate superkey. The e error measure can be extended to other properties of a relation [24]; in particular, it can be extended to the property of an attribute set being a superkey. We define $e(X)$ to be the minimum fraction of tuples that need to be removed from the relation r for X to be a superkey. If $e(X)$

is small, then X is an *approximate superkey*. The error $e(X)$ is easy to compute from the partition π_X using the equation $e(X) = 1 - |\pi_X|/|r|$.

3.3.1. Stripped partitions

A *stripped partition* is a partition with equivalence classes of size one removed. The stripped version of a partition π is denoted by $\widehat{\pi}$. For example, $\widehat{\pi_{\{D\}}} = \{\{1, 4, 7\}\}$ in the relation of Figure 1. An intuitive explanation for discarding singleton equivalence classes is that a singleton equivalence class (of the left-hand side) cannot break any dependency.

Stripped partitions contain the same information as full partitions. For example, the value $e(X)$ is easy to compute from stripped partitions using the equation

$$e(X) = (||\widehat{\pi_X}|| - |\widehat{\pi_X}|)/|r|, \quad (1)$$

where $||\widehat{\pi_X}||$ is the sum of the sizes of the equivalence classes in $\widehat{\pi_X}$. Also, the refinement relations of partitions are the same, and Lemma 2.1 thus holds for stripped partitions as well.

Lemma 2.2 does not hold for stripped partitions, because $|\widehat{\pi_X}|$ can be the same as $|\widehat{\pi_{X\cup\{A\}}}|$ even if $\pi_X \neq \pi_{X\cup\{A\}}$. However, since $e(X) = e(Y)$ if and only if $|\pi_X| = |\pi_Y|$, we can replace Lemma 2.2 with the following lemma.

LEMMA 3.5. *A functional dependency $X \to A$ holds if and only if $e(X) = e(X \cup \{A\})$.*

3.3.2. Bounding e

Computing the error $e(X \to A)$ from partitions needs $O(|r|)$ time. It is often possible to avoid this computation by using the following bounds.

$$e(X) - e(X \cup \{A\}) \leq e(X \to A) \leq e(X). \quad (2)$$

If $e(X) - e(X \cup \{A\}) > \varepsilon$ or $e(X) < \varepsilon$, TANE does not need to compute $e(X \to A)$ to find whether $X \to A$ holds approximately or not. The time saving by this optimization can be significant, because the number of functional dependencies considered can be as much as $|R|/2$ times the number of attribute sets processed.

3.3.3. Computing partitions

The partitions are not computed from scratch for each attribute set. Instead, when TANE works its way through the lattice, it computes a partition as a product of two previously computed partitions: the *product* of two partitions π' and π'', denoted by $\pi' \cdot \pi''$, is the least refined partition π that refines both π' and π''. We have the following result.

LEMMA 3.6. *For all $X, Y \subseteq R$, $\pi_X \cdot \pi_Y = \pi_{X\cup Y}$.*

TANE computes the partitions $\pi_{\{A\}}$, for each $A \in R$, directly from the database. Partitions π_X, for $|X| \geq 2$, are computed as a product of partitions with respect to two subsets of X. Any two different subsets of size $|X| - 1$ will do, which is convenient for the levelwise algorithm since only partitions from the previous level are needed.

Once TANE has the partition π_X, it computes the error $e(X)$, to be used in validity testing based on Lemma 3.5. The full partition is needed only for the computation of partitions on the next level.

After the initial setup of the first partitions $\pi_{\{A\}}$ for all $A \in R$, TANE deals with tuple identifiers only. This gives two advantages. First, the different attribute types and values can be discarded and the computation is conducted, in effect, on integers. The operations on partitions are thus simple and fast. Second, when computing approximate dependencies, the identifiers of the exceptional tuples are readily available.

4. ALGORITHMS

4.1. TANE main algorithm

To find all valid minimal non-trivial dependencies, TANE searches the set containment lattice in a levelwise manner. A *level* L_ℓ is the collection of attribute sets of size ℓ such that the sets in L_ℓ can potentially be used to construct dependencies based on the considerations of the previous sections. TANE starts with $L_1 = \{\{A\} \mid A \in R\}$, and computes L_2 from L_1, L_3 from L_2, and so on, according to the information obtained during the algorithm.

ALGORITHM. TANE
Input: relation r over schema R
Output: minimal non-trivial functional dependencies that hold in r

1 $L_0 := \{\emptyset\}$
2 $\mathcal{C}^+(\emptyset) := R$
3 $L_1 := \{\{A\} \mid A \in R\}$
4 $\ell := 1$
5 **while** $L_\ell \neq \emptyset$
6 COMPUTE_DEPENDENCIES(L_ℓ)
7 PRUNE(L_ℓ)
8 $L_{\ell+1}$:= GENERATE_NEXT_LEVEL(L_ℓ)
9 $\ell := \ell + 1$

The procedure COMPUTE_DEPENDENCIES(L_ℓ) finds the minimal dependencies with the left-hand side in $L_{\ell-1}$. The procedure PRUNE(L_ℓ) prunes the search space by deleting sets from L_ℓ as described in Section 3. The procedure GENERATE_NEXT_LEVEL(L_ℓ) forms the next level from the current level. These procedures are described in the following subsections.

4.2. Generating levels

The procedure GENERATE_NEXT_LEVEL computes the level $L_{\ell+1}$ from L_ℓ. The level $L_{\ell+1}$ will contain only those attribute sets of size $\ell + 1$ which have all their subsets of size ℓ in L_ℓ. The pruning methods guarantee that no dependencies are lost. The specification of GENERATE_NEXT_LEVEL is

$$L_{\ell+1} = \{X \mid |X| = \ell + 1 \text{ and for all } Y \text{ with } Y \subset X \text{ and } |Y| = \ell \text{ we have } Y \in L_\ell\}.$$

The algorithm is given below.

Procedure GENERATE_NEXT_LEVEL(L_ℓ)

1 $L_{\ell+1} := \emptyset$
2 **for each** $K \in$ PREFIX_BLOCKS(L_ℓ) **do**
3 **for each** $\{Y, Z\} \subseteq K, Y \neq Z$ **do**
4 $X := Y \cup Z$
5 **if** for all $A \in X, X \setminus \{A\} \in L_\ell$ **then**
6 $L_{\ell+1} := L_{\ell+1} \cup \{X\}$
7 **return** $L_{\ell+1}$

The procedure PREFIX_BLOCKS(L_ℓ) partitions L_ℓ into disjoint blocks as follows. Consider a set $X \in L_\ell$ to be a sorted list of attributes. Two sets $X, Y \in L_\ell$ belong to the same prefix block if they have a common prefix of length $\ell - 1$, i.e. they differ in only one attribute and the non-matching attribute is the last attribute in both X and Y. Each prefix block forms a consecutive block in lexicographic ordering of L_ℓ. The prefix blocks are thus easy to compute from lexicographically ordered L_ℓ. The idea of this procedure is from [20] and is explained in detail in [25, Algorithm 3].

4.3. Computing dependencies

Below is the procedure COMPUTE_DEPENDENCIES of Algorithm TANE.

Procedure COMPUTE_DEPENDENCIES(L_ℓ)

1 **for each** $X \in L_\ell$ **do**
2 $\mathcal{C}^+(X) := \bigcap_{A \in X} \mathcal{C}^+(X \setminus \{A\})$
3 **for each** $X \in L_\ell$ **do**
4 **for each** $A \in X \cap \mathcal{C}^+(X)$ **do**
5 **if** $X \setminus \{A\} \to A$ is valid **then**
6 output $X \setminus \{A\} \to A$
7 remove A from $\mathcal{C}^+(X)$
8 remove all B in $R \setminus X$ from $\mathcal{C}^+(X)$

By Lemma 3.1, steps 2, 4 and 5 guarantee that the procedure outputs exactly the minimal dependencies of the form $X \setminus \{A\} \to A$, where $X \in L_\ell$ and $A \in X$. The validity testing on line 5 is based on Lemma 3.5.

COMPUTE_DEPENDENCIES(L_ℓ) also computes the sets $\mathcal{C}^+(X)$ for all $X \in L_\ell$. The following lemma shows that this is done correctly.

LEMMA 4.1. *For all $Y \in L_{\ell-1}$, let $\mathcal{C}^+(Y)$ be correctly computed. After executing the procedure* COMPUTE_DEPENDENCIES(L_ℓ), *$\mathcal{C}^+(X)$ is correctly computed for all $X \in L_\ell$.*

Line 8 implements the difference between $\mathcal{C}^+(X)$ and $\mathcal{C}(X)$. If that line was removed, the algorithm would work correctly, but pruning might be less effective.

4.4. Pruning the lattice

The pruning procedure of Algorithm TANE is given below.

Procedure PRUNE(L_ℓ)

1 **for each** $X \in L_\ell$ **do**
2 **if** $\mathcal{C}^+(X) = \emptyset$ **do**
3 delete X from L_ℓ
4 **if** X is a (super)key **do**
5 **for each** $A \in \mathcal{C}^+(X) \setminus X$ **do**
6 **if** $A \in \bigcap_{B \in X} \mathcal{C}^+(X \cup \{A\} \setminus \{B\})$ **then**
7 output $X \to A$
8 delete X from L_ℓ

The procedure PRUNE implements the two pruning rules described in Section 3. By the first rule, X is deleted if $\mathcal{C}^+(X) = \emptyset$. By the second rule, X is deleted if X is a key. In the latter case, the algorithm may also output some dependencies. We will show that the pruning does not cause the algorithm to miss any dependencies.

Let us first consider pruning by empty $\mathcal{C}^+(X)$. If $\mathcal{C}^+(X) = \emptyset$, the loop on lines 4–8 in the procedure COMPUTE_DEPENDENCIES and the loop on lines 5–7 in the procedure PRUNE will not be executed at all. Since $\mathcal{C}^+(Y) = \emptyset$ also for all $Y \supset X$, deleting X will have no effect on the output of the algorithm.

Let us now consider the pruning of keys. The correctness of the pruning is based on the following lemma.

LEMMA 4.2. *Let X be a superkey and let $A \in X$. The dependency $X \setminus \{A\} \to A$ is valid and minimal if and only if $X \setminus \{A\}$ is a key and, for all $B \in X$, $A \in \mathcal{C}^+(X \setminus \{B\})$.*

A dependency $X \to A$ is output on line 7 of the procedure PRUNE if and only if X is a key, $A \in \mathcal{C}^+(X) \setminus X$, and $A \in \mathcal{C}^+(X \cup \{A\} \setminus \{B\})$, for all $B \in X$. Lemma 4.2 shows that such a dependency is valid and minimal. The lemma also shows that if a minimal dependency $X \setminus \{A\} \to A$ is not output in the procedure COMPUTE_DEPENDENCIES because of the pruning, it is output in the procedure PRUNE. Therefore, the pruning works correctly.

4.5. Computing partitions

The above algorithm contains no references to partitions. However, the implementation of the central test on line 5 of COMPUTE_DEPENDENCIES requires knowing $e(X)$ and $e(X \setminus \{A\})$. Also, the superkey test on line 4 of PRUNE is based on $e(X)$. In TANE, the e values are computed from stripped partitions by Equation (1). The partitions are computed as follows.

In the beginning, partitions with respect to the singleton attribute sets are computed straight from the relation r. A partition $\pi_{\{A\}}$ is computed from the column $r[A]$ as follows. First, the values of the column are replaced with integers $1, 2, 3, \ldots$ so that the equivalence relations do not change, i.e. same values are replaced by same integers and different values with different integers. This can be done in linear time using a data structure such as a trie or a hash table

to map the original values to integers. After this, the value $t[A]$ is the identifier of the equivalence class $[t]_{\{A\}}$ of $\pi_{\{A\}}$, and $\pi_{\{A\}}$ is then easy to construct. Finally, the singleton equivalence classes in $\pi_{\{A\}}$ are stripped off to form the stripped partition $\widehat{\pi_{\{A\}}}$.

A partition with respect to a larger attribute set X is computed when X is added to its level on line 6 of GENERATE_NEXT_LEVEL. The set X was formed as $Y \cup Z$ and the partition π_X is computed as the product $\pi_Y \cdot \pi_Z$. The product is computed with the following procedure in linear time.

Procedure STRIPPED_PRODUCT
Input: Stripped partitions $\widehat{\pi'} = \{c'_1, \dots, c'_{|\widehat{\pi'}|}\}$ and $\widehat{\pi''} = \{c''_1, \dots, c''_{|\widehat{\pi''}|}\}$.
Output: Stripped partition $\widehat{\pi} = \widehat{\pi' \cdot \pi''}$.

1 $\widehat{\pi} := \emptyset$
2 **for** $i := 1$ **to** $|\widehat{\pi'}|$ **do**
3 **for each** $t \in c'_i$ **do** $T[t] := i$
4 $S[i] := \emptyset$
5 **for** $i := 1$ **to** $|\widehat{\pi''}|$ **do**
6 **for each** $t \in c''_i$ **do**
7 **if** $T[t] \neq$ NULL **then** $S[T[t]] := S[T[t]] \cup \{t\}$
8 **for each** $t \in c''_i$ **do**
9 **if** $|S[T[t]]| \geq 2$ **then** $\widehat{\pi} := \widehat{\pi} \cup \{S[T[t]]\}$
10 $S[T[t]] := \emptyset$
11 **for** $i := 1$ **to** $|\widehat{\pi'}|$ **do**
12 **for each** $t \in c'_i$ **do** $T[t] :=$ NULL
13 **return** $\widehat{\pi}$

The procedure assumes that the table T has been initialized to all NULL. Since the procedure resets T to all NULL before exit, the same table can used repeatedly without re-initialization.

4.6. Approximate dependencies

Algorithm TANE can be modified to compute all minimal approximate dependencies $X \rightarrow A$ with $e(X \rightarrow A) \leq \varepsilon$, for a given threshold value ε. The key modification is to change the validity test on line 5 of procedure COMPUTE_DEPENDENCIES to

5′ **if** $e(X \setminus \{A\} \rightarrow A) \leq \varepsilon$ **then**

In addition, the pruning has to be slightly weakened by replacing line 8 of COMPUTE_DEPENDENCIES with

8′ **if** $X \setminus \{A\} \rightarrow A$ holds exactly **then**
9′ remove all B in $R \setminus X$ from $\mathcal{C}^+(X)$

The above algorithm returns only minimal approximate dependencies. In some applications, it might also be useful to know approximate dependencies that are not minimal but have smaller error. We leave the necessary modifications as an exercise to the reader.

TANE tries to resolve the test on line 5′ first by using the bounds in (2). If that fails the exact value of $e(X \setminus \{A\} \rightarrow A)$ is computed from partitions using the following procedure.

Procedure e
Input: Stripped partitions $\widehat{\pi_X}$ and $\widehat{\pi_{X \cup \{A\}}}$.
Output: $e(X \rightarrow A)$.

1 $e := 0$
2 **for each** $c \in \widehat{\pi_{X \cup \{A\}}}$ **do**
3 choose (arbitrary) $t \in c$
4 $T[t] := |c|$
5 **for each** $c \in \widehat{\pi_X}$ **do**
6 $m := 1$
7 **for each** $t \in c$ **do** $m := \max\{m, T[t]\}$
8 $e := e + |c| - m$
9 **for each** $c \in \widehat{\pi_{X \cup \{A\}}}$ **do**
10 choose $t \in c$ (same t as on line 3)
11 $T[t] := 0$
12 **return** $e/|r|$

Note the similarity to the procedure STRIPPED_PRODUCT. Here too, the table T must be initialized to all 0 once, but needs no re-initialization after that.

5. ANALYSIS

5.1. Worst case analysis

The time and space complexities of the TANE algorithm depend on the number of sets in the levels L_ℓ, called the sizes of the levels. Let $s_{\max}$ be the size of the largest level and s the sum of the sizes of all levels. In the worst case, $s = O(2^{|R|})$ and $s_{\max} = O(2^{|R|}/\sqrt{|R|})$. Another factor is the number of keys, denoted by k. In the worst case, $k = O(s_{\max}) = O(2^{|R|}/\sqrt{|R|})$.

During the computation, s partitions are formed. The time complexity for computing the partitions is $O(s\,|r|)$. Not counting the handling of partitions, the execution time of Algorithm TANE is dominated by random (nonlinear) accesses to the levels L_ℓ. During the whole computation, procedure COMPUTE_DEPENDENCIES makes $O(s\,|R|)$ random accesses on line 2, procedure PRUNE $O(k\,|R|^2)$ random accesses on line 6 and procedure GENERATE_NEXT_LEVEL $O(s\,|R|)$ random accesses on line 5. No operation is executed more often during the computation. The access time depends on the implementation of the levels L_ℓ. Using suffix arrays [26] gives $O(|R| + \log|L_\ell|)$ access time, which is $O(|R|)$ because $|L_\ell| \leq 2^{|R|}$. The suffix array for L_ℓ can be constructed in $O(|L_\ell||R|)$ time.

In summary, the algorithm has time complexity $O(s(|r| + |R|^2) + k\,|R|^3)$. The algorithm needs to maintain at most two levels at a time. Hence, the space complexity is $O(s_{\max}(|r| + |R|))$. The following theorem gives upper bounds for the time and space complexities in terms of the size of the input.

THEOREM 5.1. *The time complexity of Algorithm* TANE *is bounded by* $O((|r| + |R|^{2.5})2^{|R|})$ *and the space complexity by* $O((|r| + |R|)2^{|R|}/\sqrt{|R|})$.

Approximate validity testing needs $O(|r|)$ time in contrast to the $O(1)$ time of exact validity testing. Thus, the time complexity of finding approximate dependencies with TANE

is $O(v\,|r| + s\,|R|^2 + k\,|R|^3)$, where v is the number of validity tests done. In the worst case, $v = s\,|R|/2 = O(|R|\,2^{|R|})$ and thus the time in terms of the size of the input is $O((|r||R| + |R|^{2.5})2^{|R|})$.

5.2. Practical analysis

Due to the structure of the dependency set and pruning, s and $s_{\max}$ can be significantly smaller than the worst case analysis shows. The number k of keys is almost always much smaller than $s_{\max}$. In addition, the average size of stripped partitions can be much less than $|r|$. There are also some implementation details that further reduce the practical time and space complexities.

We have implemented the attribute sets as bit vectors of $O(1)$ words and the random access with hashing. This means, in practice, that set operations and random access take constant time. The time complexity is then reduced to $O(s(|r|+|R|)+k\,|R|^2)$ and space complexity to $O(s_{\max}|r|)$. Limiting the bit vectors to $O(1)$ words is not a severe restriction because the number of attributes is typically small and, due to exponential time and space complexities, the algorithm could not handle a very large number of attributes anyway.

To reduce the main memory requirement of the algorithm, the partitions can be stored on disk. The algorithm can be organized so that at most $|R|$ partitions at a time are in the main memory and each partition is written to disk and read from disk only once. Then, the main memory requirement is $O(|r||R| + s_{\max})$ and the algorithm makes $O(s)$ disk accesses of size $O(|r|)$. These modifications do not change the time complexity of the algorithm.

The practical properties of the modified algorithm are summarized below:

- CPU time: $O(s(|r| + |R|) + k\,|R|^2)$
- disk accesses: $O(s)$ accesses of size $O(|r|)$
- main memory requirement: $O(|r||R| + s_{\max})$
- disk space requirement: $O(s_{\max}|r|)$

To compute $e(X \to A)$, we need the partitions π_X and $\pi_{X\cup\{A\}}$, and not just $e(X)$ and $e(X \cup \{A\})$. This has two negative effects on the approximate dependency version of TANE. First, approximate validity testing is slower, by a factor $O(|r|)$ in the worst case, but somewhat less in practice due to stripped partitions and the bounds for e described in Section 3.3. Second, partitions are needed much more often and, therefore, storing partitions to disk does not work as well. The approximate dependency algorithm works in $O(v\,|r| + s\,|R| + k\,|R|^2)$ time and $O(s_{\max}|r|)$ space. However, because there are more approximately valid dependencies, pruning can be much more effective in reducing s, $s_{\max}$ and v.

5.3. Comparison to other algorithms

One of the main advantages of the new algorithm is the linear dependency on the number of tuples in the relation (for a fixed set of dependencies). To our knowledge, the only previously published practical algorithm achieving this is by Schlimmer [4, 13], who uses decision trees for validity tests. The decision tree approach is roughly equivalent to computing each partition from partitions with respect to singletons. It is slower by a factor $O(|R|)$ than using partitions the way we do. All other algorithms have $\Omega(|r|^2)$ or $\Omega(|r|\log|r|)$ dependency on the number of tuples. Some of these could actually be implemented to run in linear time as well by using, e.g. radix sorting.

Schlimmer also used the levelwise search strategy, as did Bell and Brockhausen [6]. Both use less effective pruning criteria than we do, i.e. their algorithms may end up computing a larger part of the lattice. In addition, our implementation of the pruning based on the procedure GENERATE_NEXT_LEVEL is more efficient than what Schlimmer, and Bell and Brockhausen use.

There are also algorithms that search the lattice in a more depth-first like manner [5, 12]. Such a search allows criteria for the pruning of the search space that are different from the breath-first search of the levelwise algorithm. A comparison of the effectiveness of pruning in the two approaches is difficult. However, validity and minimality testing, and the mechanisms of pruning are less efficient in the depth-first algorithms.

Still another approach is to first compute all maximal invalid dependencies by a pairwise comparison of all tuples and then compute the minimal valid dependencies from the maximal invalid dependencies [1, 3, 12, 14]. The first part of such algorithms requires $\Omega(|r|^2)$ time with respect to the number of tuples but is polynomial both in the number of tuples and the number of attributes, while the second part requires exponential time in the number of attributes but has no dependency on the number of tuples. The algorithm by Savnik and Flach [3] implements the second part with a depth-first search. During the search, the maximal invalid dependencies are used both for testing validity of dependencies and for pruning the search space. In Section 6 we present results of an experimental comparison between our algorithm and the algorithm of Savnik and Flach.

6. PERFORMANCE

We have implemented the TANE algorithm described in this paper and experimented with it to find out how it performs in practice. We have two implementations of the algorithm. The first, scalable version, denoted simply as TANE, keeps most of the partitions on disk as described in Section 5.2. The other version, TANE/MEM, works completely in main memory.

To provide perspective, we performed the same experiments with the FDEP program of Savnik and Flach. The FDEP implementation is based on the algorithm described in [3] and is available at [27].

All algorithms, including FDEP, are written in C and were compiled with a GNU C compiler with full optimizations. All experiments were run on the same 233 MHz Pentium PC with 64 MB of memory running the Linux operating system.

TABLE 1. Performance of the algorithms on real life databases.

Database				Time (s)		
Name	$\|r\|$	$\|R\|$	N	TANE	TANE/MEM	FDEP
Lymphography	148	19	2730	68	24	88
Hepatitis	155	20	8250	30	14	663
Wisconsin breast cancer	699	11	46	1	0†	15
Wisconsin breast cancer × 64	44,736	11	46	81	23	17,521
Wisconsin breast cancer × 128	89,472	11	46	173	247	*
Wisconsin breast cancer × 512	357,888	11	46	884	*	*
Adult	48,842	15	85	1451	*	*
Chess	28,056	7	1	4	2	6685

†Time is 0.25 s.

TABLE 2. Performance of TANE/MEM on approximate dependency discovery. Times are given in seconds.

		$\varepsilon = 0.0$		$\varepsilon = 0.01$		$\varepsilon = 0.05$		$\varepsilon = 0.1$		$\varepsilon = 0.5$	
Database	$\|r\|$	N	Time	N	Time	N	Time	N	Time	N	Time
Lymphography	148	2730	89.1	3388	22.2	7031	4.9	6383	3.7	21	0.0†
Hepatitis	155	8250	16.6	9666	14.6	6617	9.3	2630	4.2	160	0.0†
W. breast cancer × 1	699	46	0.3	113	0.3	126	0.2	141	0.2	18	0.0†
W. breast cancer × 2	1398	46	0.5	113	0.5	126	0.5	141	0.4	18	0.1
W. breast cancer × 4	2796	46	1.1	113	1.1	126	1.0	141	0.9	18	0.2
W. breast cancer × 8	5592	46	2.4	113	2.3	126	2.0	141	1.9	18	0.4
W. breast cancer × 16	11,184	46	5.1	113	4.9	126	4.4	141	4.3	18	0.8
W. breast cancer × 32	22,368	46	11.0	113	10.6	126	9.3	141	8.9	18	1.8
W. breast cancer × 64	44,736	46	25.5	113	26.7	126	20.3	141	19.2	18	3.9
Chess	28,056	1	2.0	1	2.6	1	3.1	1	3.5	17	3.6

†Times are 0.01–0.02 s.

The times below are real times elapsed in the experiments as reported by the Unix `time` command. We report 'wall clock' times rather than CPU times in order to make the cost of I/O processing better visible and to give a fair account of the cost of swapping of TANE/MEM with large databases.

We ran the algorithms on a number of real life databases. The databases and their descriptions are available on the UCI Machine Learning Repository [28]. The number of tuples, attributes and minimal dependencies found (N) in each database are shown in the left half of Table 1. The datasets labeled 'Wisconsin breast cancer $\times n$' are concatenations of n copies of the Wisconsin breast cancer data. To avoid duplicate tuples, all values in each copy were appended with a unique string specific to that copy. Since the set of dependencies is the same in all these datasets, we were able to test how the algorithms scale with respect to the number of database tuples only.

The top three rows of Table 1 show the performance of the algorithms on three small databases. Running times are in the right half of the table; they are rounded to the nearest second. Our algorithms perform competitively in all cases. The Lymphography and Hepatitis databases are apparently very similar. However, our algorithms are much faster on Hepatitis than on Lymphography while FDEP is an order of magnitude faster on Lymphography than on Hepatitis. This is a good demonstration of how different approaches to pruning the search space have different effects in different databases.

The bottom part of Table 1 reports the performance of TANE on five larger databases. For TANE/MEM and FDEP, some experiments are marked with an asterisk (*) as infeasible; for TANE/MEM because of the lack of main memory, and for FDEP if it did not finish within 5 h. TANE, on the other hand, found the dependencies in seconds or minutes and was never in danger of running out of memory.

Table 2 shows some performance results for TANE/MEM in the approximate dependency discovery task for different thresholds ε. Results for the Hepatitis, Wisconsin breast cancer and Chess data sets are also presented graphically in Figure 3: N_ε/N_0 stands for the number of approximate dependencies found relative to the case for functional dependencies; similarly, $\text{Time}_\varepsilon/\text{Time}_0$ denotes the relative discovery time. Approximate dependencies could not be discovered in the Adult data set with TANE/MEM due to the lack of main memory.

Overall, approximate dependencies are found efficiently. The number of dependencies found varies differently for each dataset. Within a reasonable range $0 \leq \varepsilon \leq$

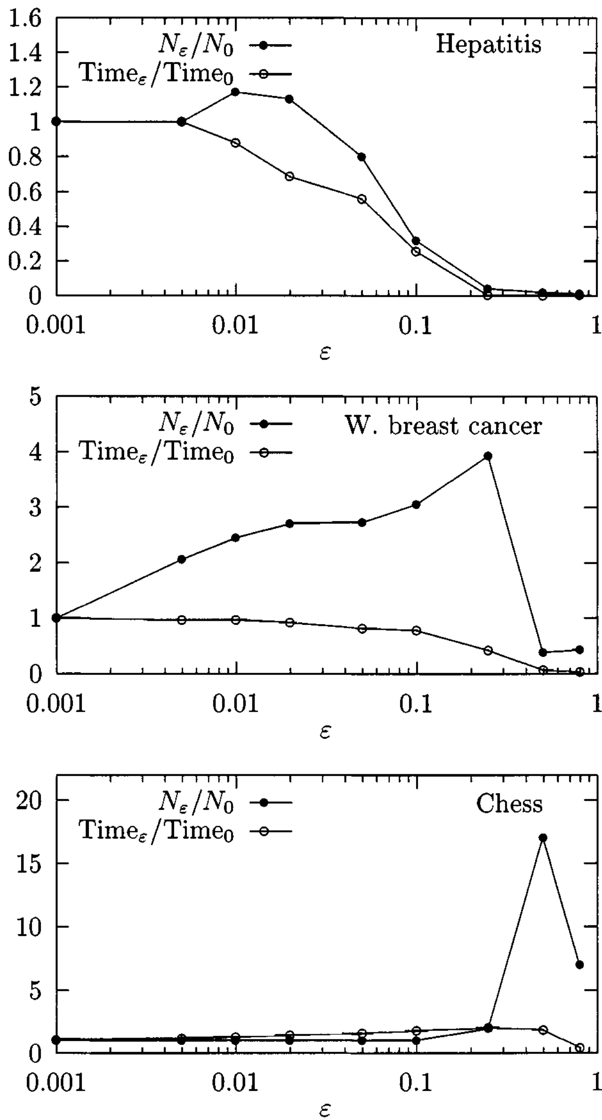

FIGURE 3. Performance of TANE/MEM for approximate dependencies in the Hepatitis (top), Wisconsin breast cancer (middle), and Chess (bottom) data sets.

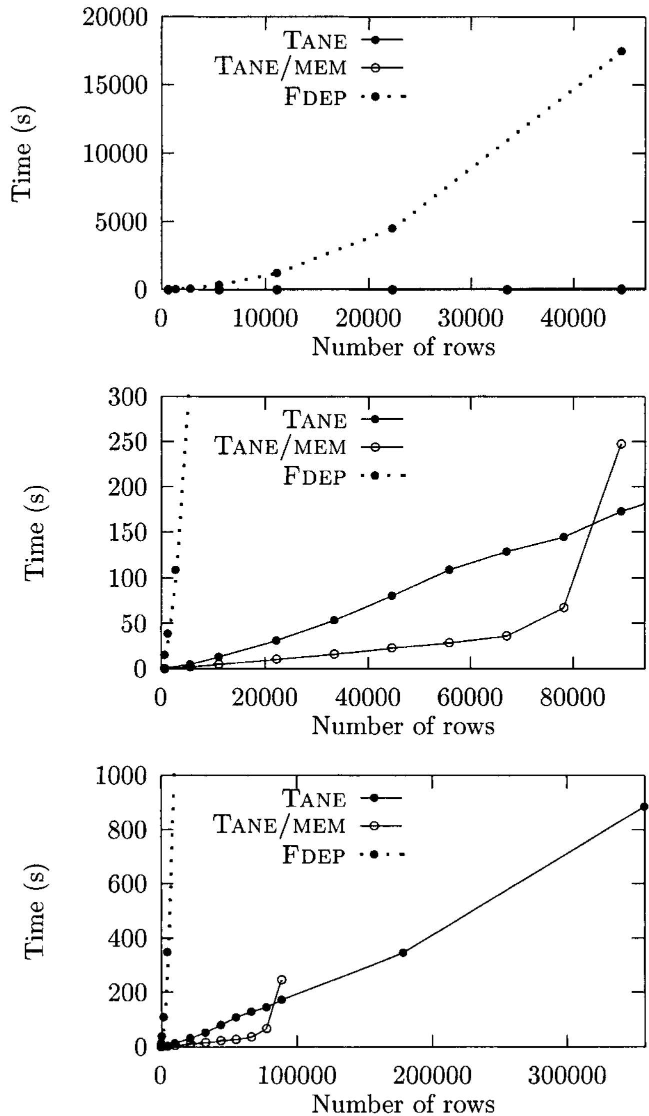

FIGURE 4. Performance of the algorithms when the number of tuples increases. The three graphs show the same data on different scales.

0.1, the time either increases slightly (Chess dataset), decreases slightly (Wisconsin breast cancer), or drops significantly (Hepatitis). The drop is even stronger with the Lymphography dataset (shown only in the table).

To find out how the number of tuples affects the algorithms, we ran a series of experiments with increasing number of tuples. The relations were formed by concatenating multiple copies of the Wisconsin breast cancer data as described earlier; recall that the set of dependencies remains the same. The results are illustrated in Figure 4. FDEP performs almost quadratically in the number of tuples while our algorithms are very near linear. The sharp turn in the curve of TANE/MEM is caused by the algorithm running out of main memory and starting to use swap space. With the largest relation (357,888 tuples, 512 times Wisconsin breast cancer), TANE used about 22 MB of main memory and about 450 MB of temporary disk space. The scaling properties of TANE/MEM on approximate dependency discovery can be read from Table 2; again, the performance is near linear in the number of tuples.

Our algorithms have not been optimized for memory and disk space consumption. With data compression, the feasible range of our algorithms can be extended. Even in their current form our algorithms can handle much larger databases than FDEP. Previously reported results for other algorithms are even worse [1, 3, 4, 6].

For a perspective on the size of problems considered before, consider Table 3. It contains results published in previous articles, marked with a 'dagger' (†), and results we obtained using TANE and the publicly available version of FDEP. Many of the databases used in previous articles are not publicly available, so results are missing altogether. The Lymphography data set marked with an asterisk (*) is reported by Bell and Brockhausen [6] as well as by Savnik

TABLE 3. Previously reported performance results and the new results. Numbers taken from other articles are marked with a 'dagger' (†); the source is given at the top of the column.

Database					Bell	Bitton	FDEP	Schlimmer	
Name	$\|r\|$	$\|R\|$	$\|X\|$	N	*et al.* [6]	*et al.* [1]	[3]	[4]	TANE
Lymphography*	150	19	7	641	>33 h†	—	540 s†	—	—
Lymphography	148	19	19	2730	—	—	88 s	—	68.2 s
Rel1	7	7	7	8	—	0.02 s†	—	—	—
Rel6	236	60	60	56	—	994 s†	—	—	—
W. breast cancer	699	11	4	35	259 s†	—	15 s	4440 s†	0.34 s
W. breast cancer	699	11	11	46	533 s†	—	15 s	—	0.76 s
W. breast cancer × 128	89,472	11	11	46	—	—	*	—	173 s
Books	9931	9	9	25	17,040 s†	—	—	—	—

and Flach [3] to have 150 tuples while the one available at the UCI repository has 148 tuples.

The column $|X|$ gives an upper limit for the number of attributes in the left-hand side of a dependency. Limiting the maximum size makes the task easier. The column N gives the size of the results, i.e. the number of dependencies output. The outputs are, however, different: some algorithms only output a (minimal) cover of the dependencies that hold.

Since the tests were run in different environments direct comparisons are not possible. The results are, however, indicative. For an overview, consider the Wisconsin breast cancer data set with the left-hand side limit $|X| = 4$. Although small and restricted, it is the only case for which there are results for four algorithms. TANE discovers dependencies in 0.34 seconds, FDEP in 15 s (slower by a factor of 44), Bell and Brockhausen [6] in 259 s (760 times slower), and Schlimmer [4] in 4440 s (13,000 times slower). It should be noted that Bell and Brockhausen [6] were the only ones to report results obtained on top of a commercial database management system, whereas all others used flat files and specialized access methods.

7. CONCLUDING REMARKS

We have presented a new algorithm, TANE, for the discovery of functional and approximate dependencies from relations. The approach is based on considering partitions of the relation and deriving valid dependencies from the partitions. The algorithm searches for dependencies in a breadth-first or levelwise manner. We showed how the search space can be pruned effectively and how the partitions and dependencies can be computed efficiently. Experimental results and comparisons demonstrate that the algorithm is fast in practice and that its scale-up properties are superior to previous methods. The method works well with relations of up to hundreds of thousands of tuples.

The method is at its best when the dependencies are relatively small. When the size of the (minimal) dependencies is roughly one half of the number of attributes, the number of dependencies is exponential in the number of attributes and the situation is more or less equally bad for any algorithm. When the dependencies are larger than that, the levelwise method that starts the search from small dependencies is obviously further from the optimum. The levelwise search can, in principle, be altered to start from the large dependencies. Then, however, the partitions could not be computed as efficiently.

There are also other interesting data mining applications for partitions. Association rules between attribute–value pairs can be computed with a small modification of the present algorithm. An equivalence class corresponds then to a particular value combination of the attribute set. By comparing equivalence classes instead of full partitions, we can find association rules. A possible future research direction is to use the unified view that partitions provide to functional dependencies and association rules, independently observed also in [8], to find an apt generalization of both and to develop an algorithm for discovering such rules.

ACKNOWLEDGEMENTS

The Adult, Hepatitis, Lymphography and Wisconsin breast cancer data have been obtained from the UCI Machine Learning Repository [28]. The Lymphography domain originates from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia. Thanks go to M. Zwitter and M. Soklic for providing the data. The FDEP program was obtained from I. Savnik's FDEP Home Page [27].

We thank Heikki Mannila, Jean-François Boulicaut, and the anonymous reviewers for constructive comments.

This research has been supported by the Academy of Finland.

REFERENCES

[1] Bitton, D., Millman, J. and Torgersen, S. (1989) A feasibility and performance study of dependency inference. In *Proc. 5th Int. Conf. on Data Engineering*, Los Angeles, CA, pp. 635–641. IEEE Computer Society Press.

[2] Mannila, H. and Räihä, K.-J. (1992) On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, **40**, 237–243.

[3] Savnik, I. and Flach, P. (1993) Bottom-up induction of functional dependencies from relations. In Piatetsky-Shapiro, G. (ed.), *Knowledge Discovery in Databases, Papers from the 1993 AAAI Workshop (KDD'93)*, Washington, DC, pp. 174–185. AAAI Press.

[4] Schlimmer, J. C. (1993) Using learned dependencies to automatically construct sufficient and sensible editing views. In Piatetsky-Shapiro, G. (ed.), *Knowledge Discovery in Databases, Papers from the 1993 AAAI Workshop (KDD'93)*, Washington, DC, pp. 186–196. AAAI Press.

[5] Mannila, H. and Räihä, K.-J. (1994) Algorithms for inferring functional dependencies. *Data & Knowledge Engineering*, **12**, 83–99.

[6] Bell, S. and Brockhausen, P. (1995) *Discovery of Data Dependencies in Relational Databases*. Technical Report LS-8 Report-14, University of Dortmund.

[7] Kramer, S. and Pfahringer, B. (1996) Efficient search of strong partial determinations. In Simoudis, E., Han, J. and Fayyad, U. (eds), *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD'96)*, Portland, OR, pp. 371–378. AAAI Press.

[8] Dalkilic, M. M., Gucht, D. V. and Robertson, E. L. (1997) CE: the classifier-estimator framework for data mining. In *Proc. 7th IFIP 2.6 Working Conf. on Database Semantics (DS-7)*, Leysin, Switzerland. Chapman & Hall.

[9] Petit, J.-M., Toumani, F., Boulicaut, J.-F. and Kouloumdjian, J. (1996) Towards the reverse engineering of denormalized relational databases. In *Proc. 12th Int. Conf. on Data Engineering*, New Orleans, Louisiana, pp. 218–227. IEEE Computer Society.

[10] Weddell, G. E. (1992) Reasoning about functional dependencies generalized for semantic data models. *ACM Trans. Database Systems*, **17**, 32–64.

[11] Bra, P. D. and Paredaens, J. (1984) Horizontal decompositions for handling exceptions to functional dependencies. In Gallaire, H., Minker, J. and Nicolas, J.-M. (eds), *Advances in Database Theory*, vol. 2, pp. 123–141. Plenum Publishing Company, New York.

[12] Mannila, H. and Räihä, K.-J. (1992) *The Design of Relational Databases*. Addison-Wesley, Menlo Park, CA.

[13] Schlimmer, J. C. (1993) Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In Piatetsky-Shapiro, G. (ed.), *Proc. 10th Int. Conf. on Machine Learning*, Amhearst, MA, pp. 284–290. Morgan Kaufmann.

[14] Mannila, H. and Räihä, K.-J. (1986) Design by example: An application of Armstrong relations. *J. Comput. Syst. Sci.*, **33**, 126–141.

[15] Mannila, H. and Räihä, K.-J. (1987) Dependency inference. In *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB'87)*, Los Altos, CA, pp. 155–158. Morgan Kaufmann.

[16] Kivinen, J. and Mannila, H. (1995) Approximate dependency inference from relations. *Theor. Comp. Sci.*, **149**, 129–149.

[17] Pfahringer, B. and Kramer, S. (1995) Compression-based evaluation of partial determinations. In Fayyad, U. M. and Uthurusamy, R. (eds), *Proc. 1st Int. Conf. on Knowledge Discovery and Data Mining (KDD'95)*, Montréal, Canada, pp. 234–239. AAAI Press.

[18] Cosmadakis, S. S., Kanellakis, P. C. and Spyratos, N. (1986) Partition semantics for relations. *J. Comp. Syst. Sci.*, **33**, 203–233.

[19] Ziarko, W. (1991) The discovery, analysis, and representation of data dependencies in databases. In Piatetsky-Shapiro, G. and Frawley, W. J. (eds), *Knowledge Discovery in Databases*, pp. 195–209. AAAI Press, Menlo Park, CA.

[20] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H. and Verkamo, A. I. (1996) Fast discovery of association rules. In Fayyad U. M., Piatetsky-Shapiro, G., Smyth, P. and Uthurusamy, R. (eds), *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. AAAI Press, Menlo Park, CA.

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

[22] Huhtala, Y., Kärkkäinen, J., Porkka, P. and Toivonen, H. (1998) Efficient discovery of functional and approximate dependencies using partitions. In *Proc. 14th Int. Conf. on Data Engineering (ICDE'98)*, pp. 392–401. IEEE Computer Society Press.

[23] Huhtala, Y., Kärkkäinen, J., Porkka, P. and Toivonen, H. (1997) *Efficient Discovery of Functional and Approximate Dependencies using Partitions (Extended Version)*. Technical Report C-1997-79, Department of Computer Science, University of Helsinki.

[24] Kivinen, J. and Mannila, H. (1994) The power of sampling in knowledge discovery. In *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS'94)*, Minneapolis, MN, pp. 77–85.

[25] Mannila, H., Toivonen, H. and Verkamo, A. I. (1997) Discovery of frequency episodes in event sequences. *Data Mining and Knowledge Discovery*, **1**, 259–289.

[26] Manber, U. and Myers, G. (1993) Suffix arrays: A new method for on-line string searches. *SIAM J. Comp.*, **22**, 935–948.

[27] Savnik, I. (1996) FDEP home page [http://martin.ijs.si/savnik/fdep.html].

[28] Merz, C. J. and Murphy, P. M. (1996) UCI repository of machine learning databases [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.