AN IMPROVED ALGORITHM FOR FINDING A KEY OF A RELATION

Sukhamay Kundu

Computer Science Dept.
Louisiana State University
Baton Rouge, LA 70803, USA

ABSTRACT

We present here an improved algorithm to find a key of a relation R(**A**) on the attributes **A**. The algorithm requires $O(|K|.||F||)$ time, where $|K|$ is the size of the key obtained and $||F||$ is the length of the input specification for the functional dependencies **F**. The previously known algorithms require $O(|A|.||F||)$ time, which can be an order of magnitude larger if $|K|$ is small compared to $|A|$.

1. INTRODUCTION

The relational model of data introduced by Codd [5] visualizes the data organized as a table or a (universal) relation. The rows of the table correspond to the facts about the universe of discourse which is being modeled. The columns of the table correspond to the attributes in terms of which the facts are described. An important concept in specifying a database is the notion of functional dependency (fd). Fd's are used to define the database semantics and other constraints that a database must satisfy. The presence of the fd's imply certain potential anomalies (inconsistencies) in the relational representation of data when one or more attributes of a row are updated, or new rows are inserted, etc. [6]. This problem is resolved by the normalization procedure in which the original relation is replaced by a set of "smaller" relations, each using only a subset of the full set of columns of R [6]. The database design problem consists of creating a set of normalized relations which is equivalent to the original universal relation. Bernstein [4] gives an algorithm for computing a normalized set of relations in 3rd normal form. The algorithm is subsequently refined in [3]. Ling and others [7] gives an improved notion of a 3rd normal form relation which removes certain potentially undesirable properties of the relations derived by Bernstein's method.

One of the basic steps in the normalization process is the construction of a key for the relation from the given functional dependencies. We present here an efficient algorithm for computing a key. Our algorithm is a more direct construction compared to the algorithm given in [8]. The latter starts with the whole attribute set **A**, and the attributes that are non-essential are eliminated one at a time, until the remaining attributes are all essential and form a key. In our method, one attribute which can be added to the present subkey in each step, and the process continues until a key is formed. This method is superior when the keys of the relation are small compared to the size $|A|$ of the full attribute set, which is often the case. In other cases, the performance of the our algorithm is same as that in [8].

2. DEFINITIONS

Let R = R(**A**) be a relation on the attributes **A**. For subsets $X, Y \subseteq A$, we say that X functionally determines Y, or equivalently, there is a *functional dependency* (fd) from X to Y, if for any two tuples (rows) t and t' of R, we have t[Y] = t'[Y] whenever t[X] = t'[X]. Here t[X] denotes the sub-tuple of values of t in the columns X. The functional dependency means two rows with the same X-values must have the same Y-values. This property must hold at all times as the relation is modified by adding new rows, or modifying values in the existing rows. (Deletion of a row cannot destroy functional dependency; this is not the case for multi-valued dependencies. The multivalued dependencies do not affect the key.) The fd from X to Y is denoted by $X \rightarrow Y$; we refer to X and Y respectively as the left and the right side of the fd. The following properties (1)-(3) of fd's are easily verified and are known as Armstrong's inference rules [2]. The property (1) allows one to reduce the right side of an fd, and by property (2) one can enlarge the right side. Note that if $Y = \{Y_1, Y_2, \ldots, Y_k\}$, then $X \rightarrow Y$ is simply an abbreviation for the set of fd's $\{X \rightarrow Y_i,\ 1 \leq i \leq k\}$. One can view (1)-(3) as rules that can be used to derive all functional dependencies that are implied by a given set of fd's.

(1) $X \rightarrow X'$ for all $X' \subseteq X$ (Projection)

(2) If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow Y \cup Z$ (Union)

(3) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ (Transitivity)

In view of the rule (1), one can rewrite (2) as [$X \rightarrow Y$ and $X \rightarrow Z$] if and only if [$X \rightarrow Y \cup Z$]. It is possible to have two subsets of **A** which functionally determine each other and yet no single attribute in one set determines any attribute in the other set. For example, let I =

Income, E = Expenditure, and N = Net-income; then any two of the attributes {I, E, N} functionally determine the other but no single attribute determines any other attribute.

$$\{I, E\} \rightarrow N \quad (N = I - E)$$
$$\{I, N\} \rightarrow E \quad (E = I - N)$$
$$\{E, N\} \rightarrow I \quad (I = E + N).$$

In an fd $X \rightarrow Y$, smaller the left side, more is the semantic information contained in that fd. If $X' \rightarrow Y$ and $X' \subseteq X$, then from (1) and (3) we get $X \rightarrow Y$. Thus $X' \rightarrow Y$ has more information than $X \rightarrow Y$. Similarly, a larger right side means more semantic information. This leads to the following definition. An fd $X \rightarrow Y$ is said to be *full* if Y is not functionally dependent on any proper subset of X; in that case we write $X \mid\!\rightarrow Y$ [7]. (Note that $X \mid\!\rightarrow Y$ does not say that for every attribute Y_i in Y, $X \rightarrow Y_i$ is full.) A subset X of the attributes is said to be a *key* if $X \mid\!\rightarrow \mathbf{A}$, i.e., X must determine all attributes and no subset of X must suffice for this purpose. A *subkey* is a subset of a key, i.e., a set of attributes that can be extended to a key. Note that if X, Y are two keys, then $X \mid\!\rightarrow Y$ and $Y \mid\!\rightarrow X$. Two subsets X, Y satisfying the mutual full functional dependency are sometimes said to be *equivalent*. For the I-E-N example above, the keys {I, E}, {I, N}, and {E, N} are mutually equivalent. An attribute set X is called a *superkey* if it contains a key, or equivalently, $X \rightarrow \mathbf{A}$.

3. ALGORITHMS FOR COMPUTING A KEY

We give below an efficient method for finding a key for an arbitrary set of fd's **F**. We assume that each fd has the reduced form $X \rightarrow Y$, where X and Y are disjoint; otherwise, we replace $X \rightarrow Y$ by $X \rightarrow Y-X$. The algorithm first finds a subkey K_0 which is contained in every key. K_0 consists of the attributes which do not belong to the right side of any fd. The subkey K_0 is then extended by adding more attributes until it becomes a key. The algorithm proceeds in two basic phases. An expansion phase extends the current subkey K (initially, $K = K_0$) to a superkey K', and this is then followed by a reduction phase in which a subkey K'', $K \subset K'' \subseteq K'$, is obtained. The cycle is repeated now with K'' in place of K, etc. until no further expansion is possible, at which point K is a key. Let *span(X)* denote the set of all attributes that are functionally dependent on X directly or transitively (i.e., can be obtained via one or more applications of the rule (3)); by convention, span(X) includes the attributes X themselves. The expansion phase of the subkey K consists of choosing a sequence of attributes $a_1, a_2, \ldots, a_k$ such that

(i) each a_{i+1} is outside the span of the previous elements, $a_{i+1} \notin \text{span}(K \cup \{a_1, a_2, \ldots, a_i\})$.

(ii) $\text{span}(K \cup \{a_1, a_2, \ldots, a_k\}) = \mathbf{A}$.

The following Lemma, which is easily proved, is the basis of our algorithm.

Lemma 1. If K is a subkey and $(a_1, a_2, \ldots, a_k)$ are chosen as above, then $K \cup \{a_k\}$ is a subkey.

Proof. By (ii), $K \cup \{a_1, a_2, \ldots, a_k\}$ contains a key $K'' \supset K$. However, K'' is not contained in $K \cup \{a_1, a_2, \ldots, a_{k-1}\}$, because its span does not contain a_k in particular. Thus K'' must contain $K \cup \{a_k\}$. (In general, we cannot say that K'' contains any other a_i, $i \leq i \leq k-1$. It is worth noting that if we scan the attributes **A** always in the same order, then each subsequent expansion phase will result in a subsequence of the previous expansion sequence, excluding the last term. Thus the 2nd expansion sequence is a subsequence of $(a_1, a_2, \ldots, a_{k-1})$.) QED.

Example 1. Consider the following fd's.

$$AB \rightarrow C, \quad BD \rightarrow A, \quad EF \rightarrow BD,$$
$$C \rightarrow E, \quad D \rightarrow F$$

The computation of successive spans and the attribute sequences $(a_1, a_2, \ldots)$ are shown below. The final key computed is BD using 2 expansion phases and 2 reduction phases. Note that the attribute A which was constructed in the superkey in the first expansion phase has been ultimately eliminated and is not in the final key.

1. $K = \phi$, $\text{span}(K) = \phi$

   $a_1 = A$, $\text{span}(A) = A$
   $a_2 = B$, $\text{span}(AB) = ABCE$
   $a_3 = D$, $\text{span}(ABD) = ABCDEF$

   The attribute D is added to K.

2. $K = D$, $\text{span}(D) = DF$

   $a_1 = A$, $\text{span}(DA) = ADF$
   $a_2 = B$, $\text{span}(DAB) = ABCDEF$

   The attribute B is added to K.

3. $K = BD$, $\text{span}(BD) = ABCDEF$.

To compare, the computation by the Lucchesi and Osborne's algorithm for the above fd's would be as shown below; the final key computed is EF.

1. ABCDEF is a superkey;
   A is not essential because span(BCDEF) contains A.
2. BCDEF is a superkey;
   B is not essential because span(CDEF) contains B.
3. CDEF is a superkey;
   C is not essential because span(DEF) contains C.
4. DEF is a superkey;
   D is not essential because span(EF) contains D.
5. EF is a superkey;
   E is essential because span(F) does not contain E.
6. EF is a superkey;
   F is essential because span(E) does not contain F.

In general, our algorithm computes larger number of span's than that by Lucchesi and Osborne's algorithm. But as can be seen form Example 1, the attribute sets for which we compute span are mostly increasing and this allows significant optimization (Lemma 2), ultimately resulting in the superior performance. In Lucchesi and Osborne's algorithm, the sets for which the span is computed are decreasing, and the smaller sets are not known until the discovery of non-essential elements. This requires each span to be computed essentially from scratch.

3.1. The Procedures INIT() and SPAN()

We now present the procedures to compute K_0, the span(X) for an arbitrary attribute set X, and finally the procedure to compute a key. For the efficiency consideration, we begin by forming for each attribute x in **A** the list list(x) which consists of all fd's whose left side contains x. We use an array size[], where for each fd f: X → Y we have size[f] = |X|, the cardinality or the number of attributes in X.

```
proc INIT(); {builds the list(x)'s and computes K0}
  begin
  for each fd f: X → Y do
    begin
    for each x in X do
      begin
      size[f] := size[f]+1; {initially, all size[] = 0}
      add f to the list(x);
      end;
    for each y in Y do {initially, all count[] = 0}
      count[y] := count[y]+1;
    end;
  K0 := empty set;
  for each attribute y in A do
    if  count[y] = 0
    then add y to K0;
  end; {of INIT}
```

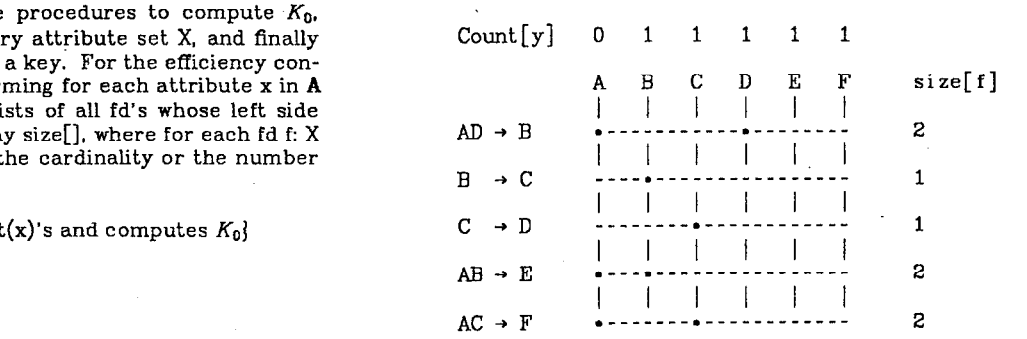
For the procedure SPAN(), we use an array mark[], where mark[x] = 1 indicates that the attribute x is marked. We sometimes write "mark x" to mean "mark[x] := 1." Q is a queue of the attributes which are presently known to be in span(X) but which have not been processed yet [1]. On termination of SPAN(X), the attributes that are marked constitute the set span(X). Initially all attributes are unmarked. (The procedure SPAN is very similar to Algorithm 2 in [3].)

```
proc SPAN(X); {computes span(X)}
  begin
  Q := empty queue;
  for each attribute x in X do
    begin
    add x to Q;
    mark x; {marking also prevents an x
             entering Q more than once}
    end;
  while Q not empty do
    begin
    x := head of Q;
    delete x from Q;
    for each fd f: X → Y in list(x) do  {x is in span(X)}
      begin
      size[f] := size[f]−1;
      if  size[f] = 0
      then for each y in the right side of f do
             if  y is not marked
             then mark y and add it to Q;
      end;
    end;
  end; {of SPAN}
```

Example 2. The following example illustrates the algorithms INIT() and SPAN(). Assume that the fd's are:

AD → B, AB → E, C → D
B → C, AC → F

The list(x) contains those fd's for which the column of x in the diagram below contains a dot (•) in the corresponding row. The value of each size[f] is shown on the right side and equals the number of dots in the row.



The life history of the queue Q in the call SPAN(A) is

Q = A
Q = empty

The behavior of Q in the call SPAN(AB) is more interesting as shown below; span(AB) = ABCDEF.

Q = AB (head of Q on the left)
Q = B
Q = CE
Q = EDF
Q = DF
Q = F (B is already marked and not added to Q)
Q = empty

3.2. The Procedure KEY()

Let K_0 be the attributes computed by INIT(), and S = span(K_0). Then it is not hard to see that each key of R(**A**) has the form K = $K_0 \cup$ K', where K' is a key of (**A**−S) for the reduced fd's on **A**−S:

replace each X → Y by X−S → Y−S

(Note that the reduced fd's are valid only for the purpose of computing the key. We need to consider now only those reduced fd's whose right side is non-empty; if X−S = ϕ, then because S equals its own span we necessarily have Y−S = ϕ.) The characteristic property of the reduced fd's on **A**−S is that every attribute of **A**−S is on the left side of some fd and also on the right side of some fd. We focus below on computing a key of such a family of fd's. To use it for the general case, simply compute K_0 by the procedure INIT() first and then execute SPAN(K_0). This will set the initial values of mark[] and size[] properly (see lines 1 and 2 in the procedure below). We use the following observation in computing the various spans during each expansion phase from a subkey to a superkey. (The proof of the lemma is omitted; the lines 3-5 in procedure KEY are based on the lemma.)

Lemma 2. If y is not in span(X) and the arrays mark[] and size[] are initialized to the values resulting from the call SPAN(X), then the array mark[] following the call SPAN(y) gives span(X $\cup$ {y}).

We use two additional arrays svmark[] and svsize[] in the procedure KEY() to save the current values of mark[] and size[], respectively, before proceeding with the each expansion phase. We assume that the arrays mark[] and save[] used by SPAN() are global and the successive application of SPAN() uses the previous mark[] and size[] values. Also, we assume that all

unmarked attributes are connected in a doubly linked list U; this would allow efficient selection of an unmarked attribute and deletion of an attribute as it gets marked. Initially, every attribute in **A** is in U (or, more generally, if K_0 is non-empty then U contains the attributes in **A**–span(K_0)).

```
proc KEY(); {compute a key}
   begin
1: call INIT();
2: call SPAN(K0);
   svmark[] := mark[]; {copy the whole array}
   svsize[] := size[]; {copy the whole array}
   {compute the initial expansion list L = (a1, a2, ..., ak)}
   L := empty list;
   last := nil; {last points to the last item of L}
   while U not empty do
      begin
      x := the first item of U;
      add x to the end of L and adjust the pointer last;
      delete x from U;
3:    call SPAN(x); {the array mark[] now
                     gives span(L ∪ {x})}
      end;
   K := empty list;
   repeat
      if last ≠ nil
      then begin {reduction phase}
         x := the last item of L;
         delete x from L and adjust pointer last;
         add x to K;
         mark[] := svmark[];
         size[] := svsize[];
4:       call SPAN(x);
         svmark[] := mark[];
         svsize[] := size[];
         last := nil;
      end;
      {compute the expansion list for the new subkey
            K ∪ {x} as a sublist of L}
      for each item y in L do
         if mark[y] = 0 {mark[] contains values set
                         from lines 4 or 5}
5:       then call SPAN(y)
         else delete y from L and reset last if necessary;
   until last is nil;
   end; {of KEY}
```

4. TIME COMPLEXITY OF THE PROCEDURE KEY()

Let us assume that each functional dependency X → Y is represented as a list of elements in X followed by a separator like "→" and then followed by the elements in Y. We denote by ||**F**|| the sum of the lengths of all these lists. Clearly, the sum of the lengths of list(x)'s is proportional to ||**F**||. The procedure INIT() scans the list for each fd once, and the computation of K_0 requires scanning the array count[] once. Therefore it requires $O(|\mathbf{A}| + ||\mathbf{F}||) = O(||\mathbf{F}||)$ time, assuming that ||**F**|| is non-trivial. As for the procedure SPAN(X), X is scanned once and each list(x) is scanned at most once in the while-loop. It too then requires at most O(||**F**||) time.

Now we analyze the procedure KEY(). All calls to SPAN in line 3 of the while-loop will together scan the list(x)'s at most once because of our use of the global arrays mark[] and size[]. The time requirement for the while-loop is then O(||**F**||). The repeat-until loop is iterated |K|+1 times, where K is the computed key. The total time for all copy operations between mark[] and svmark[], and those between size[] and svsize[] is O(|K|.|**A**|). The calls to SPAN in line 4 and those in line 5 of the for-loop together will scan the list(x)'s at most once in each iteration of repeat-until. The total time complexity of KEY() is thus O(|K|.||**F**||), as desired.

5. CONCLUSION

We have given here an improved algorithm for computing a key of a relation R(**A**) from a given set of functional dependencies **F**. The algorithm is more efficient than the previously best known algorithm for the cases when the key size is small compared to the number of attributes |**A**|, which is often the case. In the other cases, our algorithm has the same efficiency as before. It is possible that the ideas similar to those given here can be used to improve the algorithm for computing all keys of R(**A**). The algorithm given here is easily generalized for computing a key which contains a given subkey K' and/or contained in a given superkey K". Our algorithm has, however, no impact on the problem of determining if an attribute is prime (i.e., belongs to some key), which is known to be NP-complete [8].

REFERENCE

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., "*The design and analysis of computer algorithms*," Addison-Wesley Publ. Co., Reading, Massachussetts, 1974.
2. Armstrong, W. W., "Dependency structures of database relationships," in *Information Processing 74*, North-Holland Pub. Co., Amsterdam, 1974, pp. 580-583.
3. Beeri, C. and Bernstein, P. A., "Computational problems related to the design of normal form relational schema," *ACM Trans. on Database Systems*, 4(1979), pp. 30-59.
4. Bernstein, P. A., "Synthesizing third normal form relations from functional dependencies," *ACM Trans. on Database Systems*, 1(1976), pp. 272-298.
5. Codd, E. F., "A relational model for data for large shared data banks," *Comm. of ACM*, 13(1970), pp. 377-387.
6. Date, C. J., "*An introduction to database systems*," Vol 1 (3rd ed.), Addison-Wesley Pub. Co., Reading, Massachussetts, 1981.
7. Ling, T.-K., Tompa, F. W., and Kameda, T., "An improved third normal for relational database," *ACM Trans. on Database Systems*, 6(1981), pp. 329-346.
8. Lucchesi, C. L. and Osborne, S. L., "Candidate keys for relations," *J. of Computer and System Sc.*, 17(1978), pp. 270-279.