Implementation of a Linear Transitive Closure Algorithm in Relational Database Design

Roy Ladner
Graduate Student
Department of Computer Science
University of New Orleans, New Orleans, LA 70148
email: rladner@cs.uno.edu

Abstract - This paper describes a correction to an existing linear transitive closure algorithm found in [1], explains how the algorithm works and provides an example of how it works. Relational database design involves the generation of relational schemes that avoid unnecessary repetition of information. At the same time the ability to accurately retrieve all data stored in the database must be preserved. A transitive closure algorithm can be used in the design process to identify the correct attributes that can be removed without jeopardizing the accuracy of data retrieval [2]. Linear run time can be achieved with the corrected algorithm described in this paper.

1 Introduction

Relational database design involves the generation of relational schemes that avoid unnecessary repetition of information. At the same time the ability to accurately retrieve all data stored in the database must be preserved. A transitive closure algorithm can be used in the design process to identify the correct attributes that can be removed without jeopardizing the accuracy of data retrieval [2]. In a data models class I was assigned the project of implementing the linear transitive closure algorithm provided in [1]. The original algorithm follows as Figure 1 with line numbers added.

```
 1  result := α
 2  /*fdcount is an array whose ith element contains the
 3     number of attributes on the left side of the ith FD
 4     that are not yet known to be in α+ */
 5  for i := 1 to |F| do
 6    begin
 7      fdcount[i] := size of left side of ith FD;
 8    end
 9  /*appears is an array with one entry for each attribute.
10    The entry for attribute A is a list of integers. Each
11    integer i on the list indicates that A appears on the
12    leftside of the ith FD */
13  for each attribute A do
14  begin
15     appears[A] := NIL;
16       for i := 1 to |F| do
17     begin
18        let β → γ denote the ith FD;
19        if A ∈ β then add i to appears[A];
20     end
21  end
22 addin (α);
23 return (result);
24 procedure addin (α);
25 result := result ∪ {A};
26 for each attribute A in α do
27   begin
28     if A is not an element of result then
29       begin
30          for each element i of appears [A] do
31             begin
32               fdcount [i] := fdcount[i] - 1;
33                 if fdcount [i] := 0 then
34                    begin
35                       let β → γ  denote the ith FD;
36                          addin (γ );
37                    end
38               end
39        end
40   end
```

Figure 1 - The Original Algorithm

I implemented the algorithm and found that it would not achieve linear run time and would not even successfully compute closure. A brief explanation of the role of functional dependencies will clarify why this algorithm will not compute closure and will clarify the changes necessary for closure to be computed in linear run time.

2 The Use of Functional Dependencies

The algorithm explained in this paper takes advantage of the use of functional dependencies. A functional dependency expresses facts about the real world that are being modeled with the database and represents a functional relationahip of database attributes [3]. It also serves as a constraint on the database tables to impose consistency and nonredundancy [4].

As part of the design process before the closure algorithm is implemented, functional dependencies of the form $\beta \rightarrow \gamma$ are identified. The closure algorithm then computes the clo-

sure set as the set of all database attributes functionally determined by one or more attributes under the given set of functional dependencies [1]. Each functional dependency is examined. When all elements of the left hand side of a functional dependency are also in the current closure set, the elements of the current closure set functionally determine the right hand side of that functional dependency (i.e. - for $\beta \rightarrow \gamma$, $\beta \in$ {current closure set}). That right hand side (γ) is then added to the closure set. The set of functional dependencies must be successively examined with each new addition to the closure set; however, there is no need to reexamine any functional dependency that has previously yielded its right hand side to the closure set [2].

If the right hand side of a dependency has been added to the closure set, then it can be eliminated. Thus during successive examinations of the functional dependency set, it will not be considered for providing a candidate for addition to the closure set.

The flaw in Figure 1 is its treatment of *fdcount*. The data in that array can identify which dependencies will provide an addition to the closure set and can also identify those which are eliminated from further consideration. However, at line 7, the algorithm assigns to *fdcount* the "size of the left side of the i^{th} FD." This by itself provides no meaningful information to determine when to add the right hand side of the dependency to the closure set or when to eliminate it from further consideration.

The comment beginning at line 2 of Figure 1 suggests assigning to *fdcount[i]* "the number of attributes on the left side of the ith FD that are not yet known to be in" the closure set. This is not accurate because some attributes on the left side of functional dependencies could be in *result* but not be in the closure set.

On the other hand, assigning to *fdcount[i]* the number of attributes on the left side of the *ith* functional dependency that are not in the result provides the correct information. That will be "0" when all attributes in the functional dependency are also in the result. It is under those circumstances that the right hand side of the functional dependency can be added to *result* and the functional dependency can be eliminated.

Additionally, the assignment of *fdcount[i]* -1 to *fdcount[i]* at line 32 of Figure 1 will prevent a correct result in some circumstances. Values in *fdcount* will be altered prematurely. When *fdcount[i]* holds the value "0" so that the left hand side of the FD should be added to the result, that value will be modified to "-1" so that the left hand side will not be added to *result* at line 36. On the other hand, when *fdcount[i]* holds the value "1", it will be modified to "0" and *result* will be updated with invalid attributes. Instead, *fdcount[i]* should be assigned *fdcount[i]* -1 only when *fdcount[i]* equals 0 and only after the assignment to the result has been made.

For the algorithm to successfully work with array operations similar to those in lines 7 and 32 of Figure 1 it would be necessary to add and maintain a separate list. Attributes not currently in the closure set would be added to this list as each is added to the closure set [2]. Each attribute would then be removed from this list as *appears* is traversed while seeking new additions to the closure set. This would assure that *appears* is not traversed more than once for each attribute added to the closure set. The algorithm explained below eliminates the need for this extra list.

3 A Working Version

The algorithm in pseudo code with the necessary corrections follows as Figure 2.

```
 1  /*appears is an array with one entry for each attribute.
 2    The entry for attribute A is a list of integers. Each
 3    integer i on the list indicates that A appears on the
 4    left side of the ith FD */
 5  initialize all elements of appears to 0
 6  for each functional dependency i loop
 7  /*let β → γ denote the ith FD*/
 8    for each attribute A in β  loop
 9     add i to appears[A];
10    end loop;
11  end loop;
12  procedure closure_of ( α, Δ, {FD}, appears )
13    /*fdcount is an array whose ith element contains
14       the number of attributes on the left side of the
15      ith FD that are not yet known to be in α */
16    initialize all elements of fdcount to 1
17    AppearsCopy := appears;
18    procedure addin (α, Δ, fdcount, {FD}, AppearsCopy)
19      match := false;
20    begin  /* addin */
21      if Δ is not an element of α then
22        for J in 1.. | α | loop
23          /*let β → γ denote the Kth FD in the list
24           AppearsCopy(J) */
25          for K in 1..AppearsCopy(J)'Length loop
26            temp := AppearsCopy(J)(K);
27            if temp /= 0 and then fdcount (temp) /= -1
28               and then | β − α | = 0 then
29               α := α ∪ γ ;
30               fdcount ( temp ) := -1;
31               match := true;
32            end if;
33            AppearsCopy(J)(K) := 0;
34          end loop;
35        end loop;
36      end if;
37      if match then
38        addin ( α, Δ, fdcount, {FD}, AppearsCopy );
39      end if;
40    end addin;
41  begin  /* closure_of */
42    addin ( α, Δ, fdcount, {FD}, AppearsCopy );
43  end closure_of;
```

Figure 2 - The Working Agorithm

In the algorithm a functional dependency (FD) is denoted $\beta \rightarrow \gamma$. The left side of the arrow represents a set of one or more attributes. The right side of the arrow represents a set composed of exactly one attribute. Before parameters are passed to the algorithm, a functional dependency that may have looked like $\beta \rightarrow \gamma_1, \gamma_2, \gamma_3$ is first split into multiple functional dependencies of the form $\beta \rightarrow \gamma_1$, $\beta \rightarrow \gamma_2$, $\beta \rightarrow \gamma_3$.

Appears indexes all attributes to functional dependencies. *Appears* is an array indexed by the set of all attributes. Each component is a list of integers. Each integer *i* in each list indicates that the attribute appears on the left hand side of the *ith* functional dependency. *Appears* is initialized outside of closure_of and a copy is provided to addin. This will eliminate the need to reinitialize *appears* with each call of closure_of.

The main procedure closure_of has four parameters: α, Δ, {FD}, and *appears*. The set of attributes on which closure is being determined is represented by α. For a single functional dependency of the form $\beta \rightarrow \gamma$, α is selected from β, Δ represents the single attribute comprising γ. The variable *result* used in the original algorithm is omitted. Instead, the variable α is expanded to include the values that would otherwise have been added to *result*. The set of all functional dependencies is denoted {FD}.

The subprocedure addin modifies the values of *fdcount* as attributes are added to the closure set. *Fdcount* is an array that is indexed by the identifying numbers of each functional dependency. *Fdcount* is used as a value holder to flag each functional dependency that has been eliminated through the addition of its right hand side to the closure set.

At line 21 of Figure 2, the input is tested. If the closure set is complete, all processing stops. Otherwise, beginning at line 22, the algorithm examines each attribute in α in conjunction with the information stored in *AppearsCopy* and *fdcount* to identify the relevant functional dependencies on which set computations will be made during each call of addin. More concisely, for the *Jth* attribute in α, *AppearsCopy(J)* is an integer list of all functional dependencies in which the attribute appears on the left hand side. For each integer *K* in *AppearsCopy(J)*, *AppearsCopy(J)(K)* identifies the *Kth* functional dependency, $\beta \rightarrow \gamma$.

Beginning at line 27, three conditions must be met for an attribute to be added to the closure set. For clarity, let *temp := AppearsCopy(J)(K)*. As a first condition, *temp* /= 0. This establishes that there is a functional dependency in *AppearsCopy(J)* for consideration.

Second, *fdcount(temp)* /= -1. All values of fdcount are initialized outside of addin to 1. A value of -1 is assigned at line 30 only when the right hand side of the functional dependency is added to the closure set.

The third condition is that $|\beta - \alpha| = 0$. This means that all values in β are also in the current closure set α. The closure set α therefore functionally determines the attribute on the right hand side of the dependency, and the right hand side is added to α. The value "-1" is then assigned to *fdcount(temp)* at line 30.

At line 33 a "0" is assigned to *AppearsCopy(J)(K)*. During recursive calls of addin, this will prevent repetitive considerations of attributes previously evaluated.

A boolean variable *match* is initialized to false at line 19. If attributes are added to the closure set at line 29, then *match* becomes true and recursion will occur at line 38. Should Δ not be found to be an element of the closure set (line 21) during the next call of addin, then processing will continue. Otherwise, processing stops and α holds all elements of the closure set.

4 An Example

The following example illustrates how the algorithm works. Let {FD} = { CG -->B, AC --> B, C --> A, CG -->D}. Closure of {C} under {FD} will be computed. This will determine whether attribute "A" is extraneous to the dependency AC --> B. If B is found to be in the closure set, then the program implementing the closure algorithm will eliminate "A" from that functional dependency leaving only C --> B for further consideration.

Following the algorithm, closure_of is called with the input parameters holding the following values: α = {C}, Δ = B, {FD} = {CG --> B, AC --> B, C --> A, CG --> D}, and *appears* having the component values shown in Figure 3.

APPEARS

	(A)	(B)	(C)	(D)	(G)
1	2	0	1	0	1
2	0	0	2	0	4
3	0	0	3	0	0
4	0	0	4	0	0

Figure 3 - *Appears* Initialized

The set of attributes is {A, B, C, D, G}. Figure 3 indicates, for example, that attribute "C" appears in all four functional dependencies, but attribute "A" appears in only the second. A copy of *appears* is provided to addin.

Since there are four functional dependencies, *fdcount*'s index will be 1, 2, 3, 4, corresponding to the given ordering of the functional dependencies, and the component of each is initialized to "1". See Figure 4.

FDCOUNT

index -	(1)	(2)	(3)	(4)
component -	1	1	1	1

Figure 4 - *Fdcount* Initialized

4.1 The Original Call of Addin

Addin is then called. *Match* is assigned "false". The algorithm continues because Δ = "B" is not an element of {C}.

The loop at line 22 is entered. The sole attribute in {C} is

considered. *AppearsCopy(C)* indicates that attribute "C" appears in all four functional dependencies. The conditions at line 27 are tested for each. *AppearsCopy(C)(1)* is "1", identifying the first functional dependency, CG --> B, and satisfying the first condition. Second, *fdcount(1)* is not equal to "-1", satisfying the second condition. Finally, {CG} - {C} is not equal to "0". The third condition is not satisfied. Since *AppearsCopy(C)(2)* = "2", the second functional dependency AC --> B, is tested for all three conditions. Again, {AC} - {C} is not equal to "0". Next, *AppearsCopy(C)(3)* = "3". The third functional dependency is C --> A. This time all three conditions are satisfied. Particularly, {C} - {C} = "0". Attribute "A" is therefore added to {C}, and the closure set is expanded to {CA}. *Fdcount(3)* is assigned "-1", and match is assigned "true". *AppearsCopy(C)(4)* will not yield an addition to the closure set. *AppearsCopy(C)(1)* through *AppearsCopy(C)(4)* are assigned a "0". This will prevent attribute "C" from being considered during the next call of addin. Only the one new attribute added during this call of addin will be evaluated in connection with *fdcount* and *AppearsCopy*.

At the conclusion of the loop in lines 22 - 35, α is comprised of {CA}. *Fdcount* holds the values shown in Figure 5, and *AppearsCopy* holds the values shown in Figure 6.

FDCOUNT

index -	(1)	(2)	(3)	(4)
component -	1	1	-1	1

Figure 5 - *Fdcount* at the End of the First Call of Addin

APPEARSCOPY

	(A)	(B)	(C)	(D)	(G)
1	2	0	0	0	1
2	0	0	0	0	4
3	0	0	0	0	0
4	0	0	0	0	0

Figure 6 - *AppearsCopy* at the End of the First Call of Addin

4.2 The Second Call of Addin

Since a there was an addition to the closure set, addin is called again. During this recursive call, α , *fdcount*, and *AppearsCopy* hold the new values described above. All other parameters remain unchanged. *Match* is initialized to "false" at line 19. Processing continues since Δ = "B" is not an element of the closure set {CA} (line 21).

This time only attribute "A" will be considered in conjunction with *AppearsCopy* and *fdcount* since all values of AppearsCopy(C) are equal to "0". In particular, *AppearsCopy(A)(1)* = "2", identifying the second functional dependency, AC --> B, and *fdcount(2)* = "1", satisfying the first two conditions at line 27. The third condition is also satisfied since {C } - {CA} = 0. Attribute "B" is therefore added to {CA}. The closure set is expanded to {CAB}. *Fdcount(2)* is assigned a "-1", and *match* is assigned "true". *AppearsCopy(A)(1)* is also assigned "0".

At the conclusion of the loop in lines 22 - 35, α is comprised of {CAB}. *Fdcount* and *appears* hold the values shown in Figures 7 and 8 respectively.

FDCOUNT

index -	(1)	(2)	(3)	(4)
component -	1	-1	-1	1

Figure 7 - *Fdcount* at the End of the Second Call of Addin

APPEARSCOPY

	(A)	(B)	(C)	(D)	(G)
1	0	0	0	0	1
2	0	0	0	0	4
3	0	0	0	0	0
4	0	0	0	0	0

Figure 8 - *AppearsCopy* at the End of the Second Call of Addin

4.3 The Third Call of Addin

Since there was an addition to the closure set during the second call of addin, there is another recursive call of addin with the values of the closure set and *fdcount* updated as noted above. The algorithm terminates at line 21 because Δ which is equal to "B" is an element of {CAB}. However, had the algorithm not terminated, only the first and fourth functional dependencies would have been examined for additions to the closure set because the only values of *fdcount* not equal to "-1" are the first and fourth. Additionally, these two functional dependencies would have been examined only in connection with the addition of attribute "G" to the closure set immediately prior to the current call of addin.

The objective was to determine whether attribute "A" was extraneous to the dependency AC --> B. The algorithm computed the set { α } = {CAB} as the closure set of {CD} over the functional dependency set {FD}. Since "B" was found to be an element of the closure set, attribute "A" is considered extraneous, and attribute "A" may then be removed from the original functional dependency "AC --> B". The functional dependency set {FD} would then consist of {CG --> B, C --> B, C-->A, CG -->D}. This set would then be used for computing closure for the remainder of the attributes and dependencies [2].

5 Conclusion

The computation of closure is an integral part of relational database design. Efficient determination of closure is especially warranted as the complexity of design increases with greater numbers of attributes and relationships among the attributes. This efficiency can be achieved with the algorithm described in this paper.

References

[1] Korth, Henry F. and Abraham Silberschatz, *Database Systems Concepts*, pages 161-170, McGraw-Hill, Inc., New York, NY, 1991.

[2] Maier, David, *The Theory of Relational Databases*, pages 65-74, Computer Science Press, Rockville, MD, 1983.

[3] Salzberg, Betty Joan, *An Introduction to Database Design*, pages 18-20, Academic Press, Inc., Orlando, FL, 1986.

[4] Simovici, Dan A. and Richard L. Tenney, *Relational Database Systems*, pages 237-240, Academic Press, Inc., San Diego, CA, 1995.