EFFICIENT PARALLEL ALGORITHMS FOR FUNCTIONAL DEPENDENCY MANIPULATIONS

Radhakrishnan Sridhar and Sitharama S. Iyengar

Department of Computer Science
Louisiana State University
Baton Rouge, LA 70903, USA.

ABSTRACT

Given a set of functional dependencies Σ and a single dependency σ, we show that the algorithm to test whether Σ implies σ is log-space complete in P. The functional dependencies Σ are represented as a directed hypergraph H_Σ [1]. We first present a parallel algorithm which solves the above implication problem using P processors on a EREW-PRAM in $O(e/P + n.logP)$ time and on a CRCW-PRAM in $O(e/P + n)$ time, where e and n are the number of arcs and nodes of the graph H_Σ. For graphs H_Σ with fixed degree and diameter, we show that the closure $H_\Sigma{}^+$ can be computed in NC. We present NC algorithms to obtain a non-redundant and a LR-Minimum cover for the set of functional dependencies Σ. All our algorithms on a n-node directed hypergraph with fixed degree and diameter can be implemented to run in $O(\log^2 n)$ time with $M(n)$ processors on a CREW-PRAM model, where $M(n)$ is the cost of multiplying two binary matrices. The algorithms are efficient based on the *transitive closure bottleneck* phenomenon [7] that is, any improvement in the time and processor complexity of the transitive closure algorithm will result in an improvement by the same amount for the algorithms presented here.

Keywords and Phrases: relational databases, functional dependency, non-redundant cover, minimal cover, minimum cover, parallel algorithm, time complexity, processor complexity, NC algorithm, log-space reduction, directed hypergraph

1. Introduction

Functional dependencies (FDs) and their manipulation plays a decisive role in the design, use, and maintenance of relational databases. The elimination of data redundancy and the enhancement of data reliability can be done by imposing restrictions on the data. Functional dependencies provide a way to impose restrictions on data and prior knowledge about them are useful in designing better relational databases [8, 10].

Given a set of attributes T: A_1, A_2, ..., A_k, a *relation scheme* $R(T_1)$ is a subset of attributes T_1 in T. A *relation* R over the scheme $R(T_1)$ is the subset of the cartesian product $\mathrm{DOM}(A_1) \times \mathrm{DOM}(A_2) \times \ldots \times \mathrm{DOM}(A_r)$, where A_1, ..., A_r are the attributes in T_1. An element of the cartesian product is called a *tuple*. A *functional dependency* $X \rightarrow Y$ (where $X, Y \subseteq T_1$) holds in R iff, given two tuples t_1 and t_2 of R, $t_1.X = t_2.X$ implies $t_1.Y = t_2.Y$. Given a set of FDs Σ, it is important to determine those functional dependencies which are not explicitly expressed but derived from those contained in Σ. Such a derivation is possible using Armstrong's sound and complete set of axioms (see [8, 10]). The Armstrong's axioms are as follows.

Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$.

Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.

The manipulation of Σ involves the following.

(i) (Membership-Test): Given a set of dependencies Σ and a dependency σ, find whether Σ implies σ using the Armstrong's axioms.

(ii) (Closure-Finding): Determine Σ^+ the closure of Σ consisting of all dependencies that can be derived from Σ using the Armstrong's axioms.

(iii) (Minimal Key-Finding):Finding a minimal set $X \subseteq T$ of attributes, such that $X \rightarrow T$ is a member of Σ^+. The attribute set X is called the *minimal key* of the relational scheme $R(T)$.

(iv) (LR-Minimal cover): Finding a set of dependencies Σ_r from Σ such that $\Sigma_r{}^+ = \Sigma^+$ with the following properties.

(a) For any dependency σ in Σ_r, $(\Sigma_r - \sigma)^+ \neq \Sigma^+$.

(b) We say an attribute A in X of the dependency $X \rightarrow Y$ as *extraneous* if $X - A \rightarrow Y$ is in Σ^+. No dependency in Σ_r has extraneous attributes on its left side as well as its right side.

The set Σ_r is called the LR-Minimal cover for Σ.

For discussion about LR-Minimum and the advantages of manipulating the given set of dependencies Σ (see [8, 10]).

Several datastructures and sequential algorithms for representation and manipulation of functional dependencies have been proposed in the past [1,4,6]. A new graph-theoretic approach which leads to efficient algorithms for manipulation and representation of FDs were introduced in Ausiello et. al [1]. In this approach, the given set of FDs were represented as a directed hypergraph and known graph algorithms like the transitive closure, transitive reduction, and finding *strong connected components*† were modified for manipulating FDs. Using the algorithms of Maier [8] an LR-Minimum is obtained and with the LR-Minimum set of FDs, the synthesis algorithm [8] can be applied to get the relational schemes. Several theoretical issues based on directed hypergraphs were discussed in [2]. The algorithms of Diederich and Milton [4] for computing minimal covers and synthesizing relations into third normal form do not try to achieve a reduction in the computational complexity of the algorithms in [8]. They present interesting insights into the manipulation algorithms of [8] and suggest techniques for enhancement of those algorithms. For example, in standard methods for synthesizing relations, most dependencies have to be checked a second time for redundancy after grouping dependencies with equivalent left-hand sides. Using the method of Diederich and Milton the dependencies can be characterized in such a way they are checked only once. At the present time we don't know of any parallel algorithm for manipulating functional dependencies.

The availability and the increase in the development of parallel architectures have rekindled the efforts to design efficient parallel algorithms for different problems which make use of the parallel hardware. There has been a growing interest in the development of parallel architectures and algorithms in the area of database systems [3, 9]. In fact, there is no reason as to why the design, manipulation, and use of databases has to be done sequentially given the availability of parallel architectures. Such interests have motivated us in the development of parallel algorithms for manipulation of functional dependencies. Research in parallel algorithms have focussed on developing algorithms which run in polynomial of the logarithm of the input size with processors whose number is bounded by a polynomial in the input size. Such algorithms belong to the class of *NC* (Nick's Class) [7]. An algorithm in NC tells us that they can be executed at high speeds using a "reasonable" amount of hardware. It is not yet known, whether all problems solvable in polynomial time (P-time) can be solved in NC. If such is the case, it would mean that every problem that is solvable in P-time can be solved very fast in parallel, using a polynomial-bounded number of processors. Certain difficult problems that can be solved sequentially in P-time have been identified as they are called as P-Complete problems [7]. Using reduction techniques it was shown that a P-Complete problem is in NC if and only if $P = NC$. Thus, P-complete problems can be viewed as the problems in P most resistant to parallelization.

We will show by a simple reduction technique that the FD-Membership problem is P-Complete (Section 2). Using the directed hypergraphs [1] as the representation scheme for the given set of FDs, we derive parallel manipulation algorithms. Our algorithms unlike the algorithms of Ausiello et. al [1], are highly suitable for parallelization. Our characterization of the FD-manipulations in terms of directed hypergraph representing the FDs are simpler compared to the ones presented in [1]. The algorithms for manipulating the functional dependencies use algorithms for computing transitive closure, transitive reduction, and strongly connected components. In order to construct efficient parallel algorithms for computing transitive reduction and strongly connected components it will be necessary to avoid the use of matrix powering or transitive closure as a subroutine; our inability to do so is sometimes called the *transitive closure bottleneck* [7]. The FD-manipulation algorithms have to necessarily use the transitive closure algorithm as a subroutine and hence, it is also affected by the transitive closure bottleneck phenomenon. We will show in this paper that our parallel FD-manipulation algorithms are efficient based on the transitive closure bottleneck phenomenon. This is done by showing that all operations other than those involving the transitive closure as a subroutine take $O(logn)$ time with processors at most equal to the size of the directed hypergraph H_Σ representing a set of functional dependencies Σ. First we present a parallel algorithm to obtain the closure H^+ of the directed hypergraph H. We show that our closure algorithm is in NC for fixed degree and diameter graph H (Section 3). Section 4. presents algorithms to obtain a non-redundant and a LR-Minimum cover and it is also in NC for fixed degree and diameter graph H. From the LR-Minimal cover a minimal key can be easily determined. Conclusions are presented in Section 5.

The model of computation used in this paper is the *uniform parallel random access machine (PRAM)* model. An EREW-PRAM model the weakest of all models does not allow neither concurrent reading or writing. A CREW-PRAM model which is used in our algorithms allows concurrent reading, but not concurrent writing. The CRCW-PRAM model allows concurrent reading and writing. The variants of the CRCW-PRAM are based either on "priority", where proces-

† A set of nodes are in a strongly connected component if there are paths from every node to every other node in the strong component.

sors are assigned priorities and the processor with the highest priority succeeds in writing, or "arbitrary" where among the set of processors which try to write only one is chosen to write. It is known that the CREW-PRAM and the CRCW-PRAM model can be simulated by an EREW-PRAM in $O(logP)$ time with $O(P)$ extra processors or with no extra processors in $O(\log^2 P)$ time. [5]. It was shown in [11], that all PRAM models with P processors can be simulated by an ultracomputer (bounded-degree network of processors with no global memory) in $O(logP\ (loglogP)^2)$ time per step and with no extra processors.

1.1 Definitions and Notations

Definition 1.1 (Directed Hypergraph): A directed hypergraph $H = (V, E)$ consists of nodes and arcs as follows.
nodes: The node set V consists of simple and compound nodes. A compound node j has components $j_1, j_2, ..., j_r, r > 1$ and each j_k is a simple node. A simple node is a node with only one component.
Arcs: The arc set E has the following arcs:

(i) arcs (i, j) from one simple node to another,

(ii) arcs $(j, j_1), ..., (j, j_r)$ from each compound node to its components.

(iii) arcs (i, j) from node i to compound node j if and only if there are arcs $(i, j_1), ..., (i, j_r)$, where $j_1, ..., j_r$ are the components of compound node j. If such an i exists we say that node j is *satisfied* by node i.■

We say that there is a *path* from node i to node j, written $<i, j>$, if and only if there are paths $<i, k>$ and $<k, j>$. Also, there is a path from node i to a compound node j, if and only if there are paths $<i, j_1>, ..., <i, j_r>$, where, $j_1, ..., j_r$ are all the components of compound node j.

Definition 1.2 (Hypergraph Accessibility Problem (HGAP)): Given a directed hypergraph $H = (V, E)$, and two distinguished nodes $i, j \in V$, does there exists a path $<i, j>$ in H.■

The above HGAP problem on a n-node directed graph containing only simple nodes, can be solved in $O(\log^2 n)$ time with $M(n)$ processors, where $M(n)$ is the cost of multiplying two binary matrices [7].

We will assume that the set Σ is in *reduced form* as follows

(a) there exist no two FDs $X \rightarrow Y$ and $X' \rightarrow Y'$ such that $X = X'$, and

(b) for all FDs $X \rightarrow Y, X \cap Y = \emptyset$.

Let the given set of FDs Σ be in reduced form and represented by a directed hypergraph H_Σ as follows. For each FD $X \rightarrow Y$ create a compound node X and simple nodes $X_1, ... X_r$ and arcs $(X, Y), (X, X_1), ..., (X, X_r)$ in H_Σ. The nodes $X_1, ..., X_r$ are components of node X. We will denote $n = |\Sigma| = |\Sigma_l| + |\Sigma_r|$ the sum of the length of the strings of attributes appearing on the left (right) side of the dependencies. Also, $e = ||\Sigma||$ will denote the number of FDs in Σ. We will use the notation H instead of H_Σ when the context is clear and call H as a graph instead of a directed hypergraph.

Proposition 1.0: Let the given set of functional dependencies (FDs) be represented by a directed hypergraph $H = (V, E)$. The FD-Membership test on the dependency $X \rightarrow Y$ is equivalent to the HGAP instance from node X to node Y.■

Example 1: See Figure 1. for a set of FDs and its corresponding directed hypergraph.

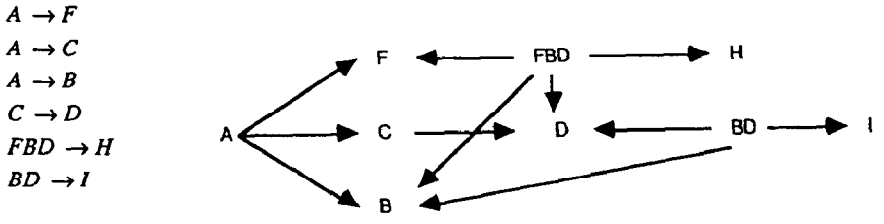

Figure 1: A set of FDs and the directed hypergraph corresponding to it (from Ausiello et. al [1]).

Definition 1.3: We say a directed hypergraph $H = (V, E)$ *generates* a set of functional dependencies Σ, when, for each arc (X, Y) in E a functional dependency $X \rightarrow Y$ is generated.■

2. The P-Completeness Result and a Parallel Algorithm

In this section, we show that the *monotone circuit value problem* is *log-space* reducible to HGAP and thus establish HGAP is P-Complete. The monotone circuit value problem is P-Complete (see [7]).

Definition 2.1 (Monotone Circuit Value Problem):
Given a finite set of g gates; for $1 \leq j \leq g$, gate j is either an input (0 or 1), an AND-gate AND($i_{j,1}, i_{j,2}, ..., i_{j,k(j)}$), or an OR-gate OR($i_{j,1}, i_{j,2}, ..., i_{j,k(j)}$), where $1 \leq i_{j,1}, i_{j,2}, ..., i_{j,k(j)} < j$, what is the value of the expression represented by gate g.

Lemma 2.1 (see [7]): The monotone circuit value problem is log-space complete in P.■

Theorem 2: The HGAP is log-space complete in P.

Proof: We show by the following construction that the monotone circuit value problem is log-space reducible to the HGAP. Consider the case where all the gates have two inputs for the sake of ease in presentation. We construct the following directed hypergraph H. For an AND-gate $g_i = g_j \wedge g_k$, create a compound node g_i with two components g_{i1} and g_{i2}. Add arcs (g_j, g_{i1}) and (g_k, g_{i2}). For an OR-gate $g_i = g_j \vee g_k$, add arcs (g_j, g_{i1}), (g_j, g_{i2}), (g_k, g_{i1}), and (g_k, g_{i2}). We can easily show by induction, that on an input 1 at gate g_i, an output of 1 is obtained at gate g; if and only if there exists a directed path from node g_i to node g in H. The construction of H can be done in log-space. Hence the theorem.■

Coro 2.1: The FD-membership test is log-space complete in P.

Proof: Follows directly from Proposition 1. and Theorem 2.■

The negative result in Theorem 2. only tells us that HGAP is resistant to high-degree parallelisms. We present a simple sequential algorithm for the HGAP which runs in time $O(e + n)$, where e and n are the number of edges and vertices of the graph H. A parallel version of the sequential algorithm runs in time $O(e/P + n.logP)$ with P processors on a EREW-PRAM and in time $O(e/P + n)$ with P processors on a CRCW-PRAM. The technique used in the following algorithm is similar in spirit to the one presented for the monotone circuit value problem by Vitter and Simmons [13].

(* Initially all vertices are marked "not visited." *)

Algorithm HGAP (x, y)

Begin

1. Starting from x determine all the k vertices that can be reached from x by using transitive closure; Mark all the k vertices "visited" including x.
2. If y is one of the k vertices, then, RETURN ('Found'); STOP.
3. If either $k = 0$ or there is no arc (x, p) such that vertex p is a component of some compound node, then, RETURN ('Nil').
4. For each "unvisited" compound node j, such that there is at least one arc (x, j_r), where j_r, is a component of the node j, Do
   Begin
5. If node j is *satisfied* by x, then,
   Begin
6. ADD arc (x, j)
7. HGAP (j, y)
   End
   End

End.

It can be easily seen, that if there should exist a path $<x, y>$ and has not been determined at the end of Step 3., then, there exists at least one compound node in H which is satisfied by x. The algorithm HGAP presented above can be parallelized in several ways. Each of the steps 1-7 can be parallelized. Step 4-7 is executed sequentially and the processors are assigned to keep track if node j is *satisfied* by node x. Each of the P processors are assigned to check the presence of the arc from x to the Pth component of j. Once each processor determines the presence/absence of arcs assigned to it, the time taken to check if node j is satisfied by x is $O(logP)$ using binary-tree communication scheme among P processors. Essentially, we are computing AND of P binary values. On a CRCW-PRAM, we can determine the AND of P binary values using P processors in constant time.

Theorem 3: The HGAP can be solved using P processors on a EREW-PRAM in $O(e/P + n.logP)$ time or on a CRCW-PRAM in $O(e/P + n)$ time.

Proof: Follows from the discussion above.■

3. Closure of a Directed Hypergraph

Computing the closure of a directed hypergraph H is finding all the possible arcs in the graph H. The closure of the graph H is the transitive closure on directed graphs when H contains only simple nodes. Since, finding whether there exists a path $<X, Y>$ is P-Complete, determining the closure is also P-Complete. In this section, we present a parallel algorithm whose execution time is dependent on the *diameter* and the *degree* of the graph H. The diameter of the graph H is the maximum distance between any two nodes in H. The degree of the graph H is the degree of a node having maximum number of arcs going out. For graphs with fixed diameter and degree algorithm, we show that the closure can be computed in NC. Having determined the closure the HGAP problem can be solved in constant time. In terms of the the functional dependencies Σ, the degree of a node X in H_Σ is the number of FDs in Σ whose left hand side is in X or equal to X. The distance between two nodes X and Y in H_Σ is the number of dependencies in Σ which have to be applied before X determines Y. In the worst case the maximum distance and degree can both be equal to the number of FDs in Σ.

Theorem 4: The algorithm H-Graph-Closure correctly determines the closure of directed hypergraph H in $O(\log^2 n + MAX(diameter(H), degree(H))*logn)$ with $O(M(n))$ processors on a CREW-PRAM.

Proof: It is straightforward to understand Steps 1-6 of the algorithm H-Graph-Closure. Step 8. performs the closure operation with respect to a node i. Let us assume that there should exists an arc (i, j) in the closure of H and not in H

Algorithm H-Graph-Closure

Begin

1. Perform transitive closure on H. Here we find all simple nodes that can be reached from node i and add arcs to nodes reachable from i.
2. **Do** Steps 3-9 **Until** no new arcs are added
3. For each node i **In-Parallel**
4. If there are arcs $(i, j_1), ..., (i, j_r)$, where $j_1, .., j_r$ are *all* the components of compound node j, then
   Begin
5. Add the arc (i, j) in H
6. For all arcs (j, k) add arcs (i, k)
   End
7. For each node i **In-Parallel**
   Begin
8. Let $\mathbf{J} = \{j^1, ..., j^k\}$ be the compound nodes such that there is an arc (i, j_p), where, j_p is a component of node $j \in \mathbf{J}$ AND for all components p of j, there are arcs only from compound nodes in $\mathbf{J}$.
9. Add arcs (i, j) for each $j \in \mathbf{J}$ and arcs (i, k) such that node k is adjacent to some compound node in $\mathbf{J}$
   End

End.

after the execution of Step 6. The absence of the arc (i, j) implies that there exists a compound node k in the directed path between node i to node j. Assume there is an arc (k, j), if it is absent, we have the case described above. The arc (i, k) is absent, otherwise, we would have the arc (i, j). For the arc (i, k) to be present, there should be arcs from node i to every component of k. In Steps 7-9 determines the arcs from node i to the components of k. It can be easily shown, that if there should exists an arc (i, j) in the closure and has not been determined at the end of Step 6., then, there exists at least one compound node k in **J** of Step 8.

In Figure 2. and Figure 3. we have depicted the worst-case scenario in terms of the number of iterations of Steps 3-9 before the arc (i, j) is determined. We can easily show that at most MAX(diameter(H), degree(H)) iterations of Steps 3-9 would be necessary to determine the closure. Step 1. takes $O(\log^2 n)$ time to determine the closure with $O(M(n))$ processors (see [7]). Each of the Steps 3-9 can be executed in $O(\log n)$ time with $O(n + e)$ processors using suitable matrix structures on a CRCW-PRAM.■

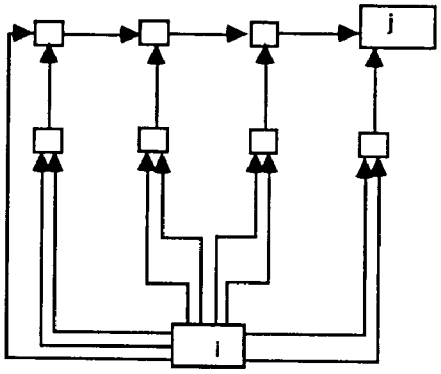

Figure 2: The number of iterations of steps 3-9 in algorithm H-Graph-Closure to determine the arc (i, j) is at most equal to the degree of the above graph. The graph above consists of compound nodes with two components each.

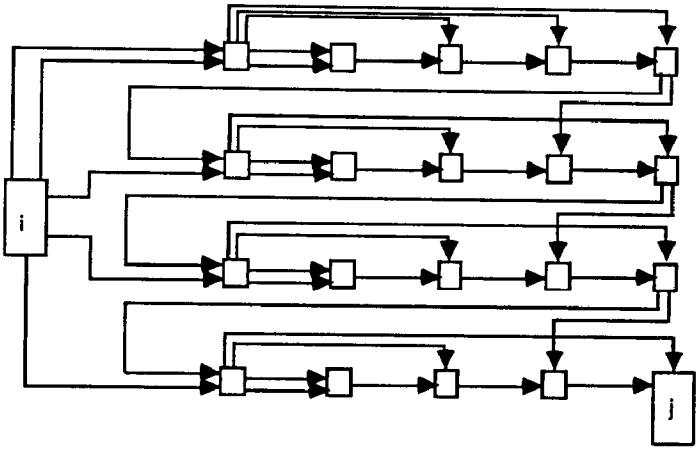

Figure 3: The number of iterations of steps 3-9 needed by algorithm H-Graph-Closure to determine the arc (i, j) is at most equal to the diameter of the above graph.

Example 2: The closure of the graph in Figure 1. is given in Figure 4.

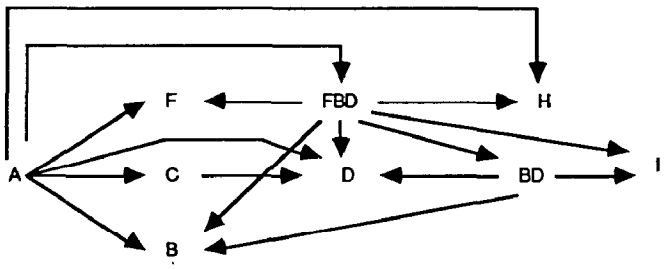

Figure 4: The closure of the graph in Figure 1.

4. Non-Redundant and Minimum Directed Hypergraphs

Given a set of functional dependencies Σ, we present algorithms to obtain a non-redundant and a minimum cover as defined by Maier [8]. Since, the FD-membership test is P-Complete, algorithms for determining a non-redundant and minimum cover are also P-Complete. In the previous section we presented the closure algorithm which was shown to be in NC for fixed diameter and degree graph H. We will now define several terminologies.

Left reduction involves removal of "extraneous" attributes from X in each of the dependencies $X \rightarrow Y$ in Σ. Given the set Σ, an attribute B is *extraneous* in $X \rightarrow Y$ if $X = ZB$, $X \neq Z$, and $A \in Z_{\Sigma}^{+}$. There are two kinds of extraneous attributes. If $X = ZB$, $X \neq Z$, and $B \in Z_{\Sigma}^{+}$, then B is called an *implied* extraneous attribute and all other extraneous attributes are *non-implied* extraneous attributes.

We say two attribute sets P and Q are equivalent in Σ written $P \equiv Q$, if $P \rightarrow Q$ and $Q \rightarrow P$ are in Σ^{+}. Let $X \rightarrow Y$ be a dependency with $X \cap Y = \emptyset$ and let $X_1, X_2, \ldots X_l$ be some subsets of X such that $(X_1 \equiv Y_1), (X_2 \equiv Y_2), \ldots, (X_l \equiv Y_m)$ with $Y_1 \cup Y_2 \cup \ldots \cup Y_m = Y$. The dependency $X \rightarrow Y$ above is trivial and Y is *textually contained* in X.

A *non-redundant* cover for Σ is the set Σ_r in which all dependencies σ in Σ_r, when removed is not in the closure Σ_r. If we assume that the right hand side of each dependency in Σ_r is a single attribute, then, a *minimal cover* for Σ_r can be obtained by removing both the implied and non-implied extraneous attributes from the left hand side of each dependency in Σ_r [10]. A *minimum cover* is a minimal cover with a minimum number of functional dependencies than any other equivalent set. A minimal cover is a minimum cover which does not contain some two dependencies $X \rightarrow A$ and $Y \rightarrow B$, such that X and Y are equivalent.

We will give definitions for non-redundant, minimal, and minimum directed hypergraph H.

A hypergraph H is non-redundant if it does not contain any *redundant arcs*. An arc (i, j) is *redundant* in H if,

(i) there are arcs (i, k) and (k, j) in H^{+}, or

(ii) the node j is textually contained in node i.

Condition (ii) identifies arcs which generate trivial dependencies. A non-redundant hypergraph is minimal if each compound node does not contain any implied extraneous attributes. The non-implied extraneous attributes are removed when redundant arcs satisfying condition (i) is removed. A minimal hypergraph is a minimum one if there are no two arcs (I_1, J) and (I_2, K) in H, where I_1 and I_2 are nodes in the same strongly connected component I with J and K in different strong components. From a minimum hypergraph a minimum set of FDs can be easily generated.

The following algorithm obtains a minimum directed hypergraph H_m from the graph H_{Σ} for a given set of dependencies Σ.

Algorithm Minimize

Begin

1. Let H_1 be the graph H_Σ with arcs to compound nodes present in H_Σ^+.
2. Compute the strongly connected components of the graph H_1.
3. For each component i **do in parallel**
   Begin
4. If more than two arcs from the component i to the component j, then, REMOVE all except one from H_1.
5. Choose a node X as a representative of component i.
6. For all arcs (Y, Z), such that Y is in i and Z not in i, REMOVE (Y, Z) and add (X, Z).
7. Remove nodes j from component i which is textually contained to some node k in component i.
   End.
8. Process the acyclic graph formed by the strong components as follows:
   Mark the arc (I, J) from strong component I to component J for deletion when representative node j of J is textually contained in representative node i of I.
9. Transitively reduce the acyclic graph formed by the strong components and mark arcs to be deleted.
10. Remove implied extraneous attributes from each compound node i.
11. Remove arcs marked for deletion in Step 8 and Step 9.
12. For each one of the strong-components form a Hamiltonian-Circuit with the nodes in the component.
13. Remove redundant arcs formed due to the Hamiltonian-Circuits by Transitively reducing the graph H_1.
14. For each arc (i, j), where j is a compound node, **do in parallel**
    Begin
16. Add arcs (i, j_1), ..., (i, j_r), where j_1, ..., j_r are components of compound node j.
17. Add arcs (j, j_1), ..., $(j, ..., j_r)$.
    End
18. Transitively reduce the resulting graph and remove the arcs to the compound nodes.

End.

The Steps 1-6 are easy to understand. In Step 7 node j is removed from strong component I if it is textually contained in node k in the same component I. Since j and k are in the same strong component, if j is textually contained in k, then k is also textually contained in j. Hence, to avoid deleting both j and k from component I, ranks are assigned to each node in component I and j is deleted from component I iff j is textually contained in k and rank(j) < rank(k). The textual containment of two nodes in the same component can be tested as follows. Let j and k be two nodes in the component I with rank(j) < rank(k). Let (I, I_1), ..., (I, I_l) be the arcs from strong component I to strong components I_1, ..., I_l, respectively. For all strong components I_m, $1 \le m \le l$, we do the following. If $k_m \subset k$ is in node I_m, then for all nodes j_m in I_m with $j_m \subset j$ remove j_m from j. If j becomes empty then, j is textually contained in k, otherwise it is not.

In Step 8 we delete the arc from strong component I to strong component J when some node j in J is textually contained in some node i in I. In fact, if j and i are representative nodes chosen in Step 5, arc (I, J) can be removed if j is textually contained in i. Now, the test is carried out as follows. Let I_1, ..., I_l be the components such that there are arcs (I, I_1), ..., (I, I_l), $1 \le m \le l$, and the arcs from strong component J are to only the strong components I_1, ..., I_l. We perform the operation described previously on the node $j \in J$, if j becomes empty, the arc (I, J) is redundant, i.e., j is textually contained in node $i \in I$, otherwise, arc (I, J) is not redundant. The arcs which are found redundant in this step are removed in Step 11 as they are required to determine and remove implied extraneous attributes.

The removal of implied extraneous attribute from every compound node i in component I is done as follows. Let I_1, I_2, ..., I_l be the strong components which have arcs from compound node I containing i with no I_k having an arc from any

of I_j, $1 \leq j \leq l$ and each I_k contains a node $z \subset i$. If each I_k, $1 \leq k \leq l$ is reduced, i.e., no compound node $z \subset i$ in I_k contains extraneous attributes, then, pick a node z from each one of the I_k's. The union of all z's gives the compound node i without any implied extraneous attributes. The arcs to the compound nodes are redundant and Steps 14-18 performs the right reduction operation.

The minimal key for a set of dependencies Σ by taking the union of representatives of the strong components with indegree zero in the minimum hypergraph H_m of the graph H_Σ.

Example 3: For an illustration of the algorithm Minimize see Figures 5(a) - 5(d).

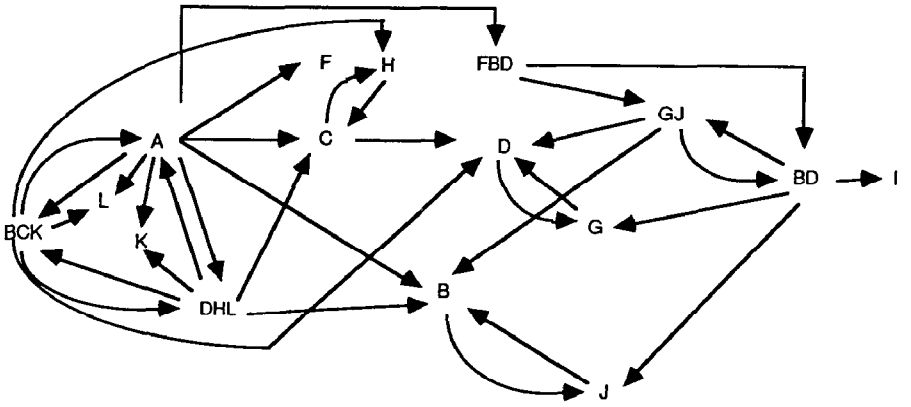

Figure 5(a): The graph H_1 formed in Step 1 of the algorithm Minimize. The arcs to the compound nodes have been added after the closure of the original graph is determined. The arcs from a compound node to its simple components have been removed to lessen the clustering of the Figure.

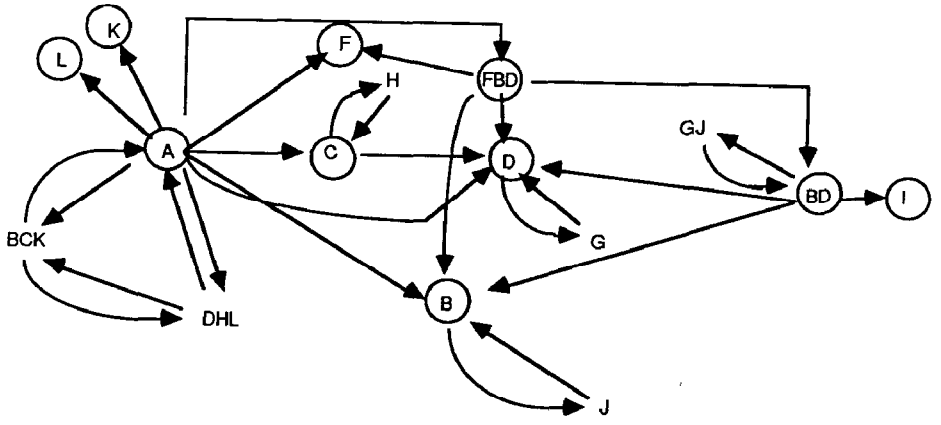

Figure 5(b): The graph H_1 after the execution of steps 1-6 of the algorithm Minimize on the graph in Figure 5(a). The strong components can be clearly seen and the reprentatives are marked with circles around them. In Step 7, node *GJ* would be determined to be textually contained in *BD* and would be removed.

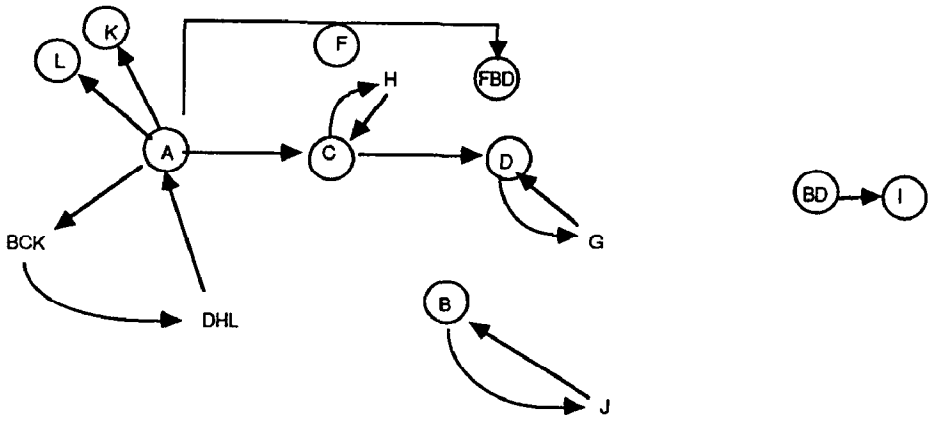

Figure 5(c): The graph H_1 at the end of Step 13. of the algorithm Minimize.

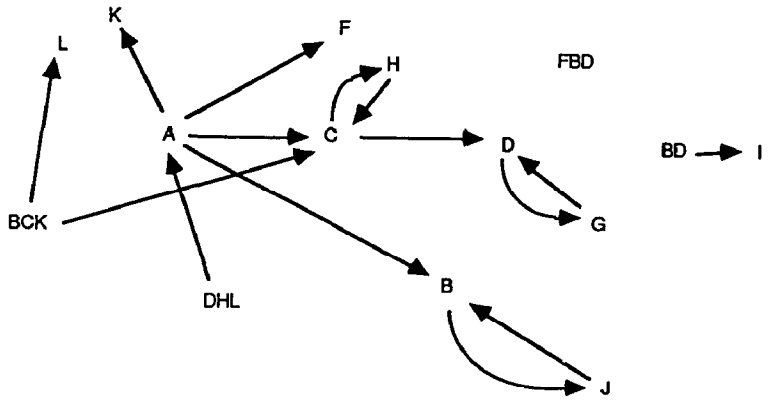

Figure 5(d): The minimum directed hypergraph of the graph H_1 in Figure 5(a) after the completion of the algorithm Minimize.

Lemma 4.1: Given directed hypergraph H the algorithm Minimize correctly obtains the minimum directed hypergraph H_m.

Proof: We will first show that the graph H_m does not contain any redundant arcs.

Case (i): Let (i, j) be the redundant arc and let i and j be in different strong components determined in Step 2. Since (i, j) is redundant there can exists a node k such that there are arcs (i, k) and (k, j). If node k is in a different strong component, then, the transitive reduction of the acyclic graph formed by the strong components would delete the arc (i, j) in Step 9. If node k is in the component of i or j, then, Steps 3-6 would delete the arc (i, j). Also the arc (i, j) is redundant if node j is textually contained in node i and it is removed in step 8. Hence the arc (i, j) is not redundant.

Case (ii): Let the arc (i, j) be redundant and let nodes i and j be in the same strong component determined in Step 2. Since (i, j) is redundant there can exists a node k and arcs (i, k) and (k, j). Node k cannot be in a different component, since all nodes reachable by the node i are reachable by k using arcs (i, k) and (k, j). If node k is in the same component as nodes i and j, then, the Hamiltonian circuit formed for each strong-component would delete arc (i, j) in Step 12. The arc (i, j) is redundant if node j is textually contained in node i. In Step 7. node j is removed. Hence the arc (i, j) is not redundant.

Case (iii): The construction of an Hamiltonian circuit in Step 12, creates redundant arcs as shown in Fig. 4.2(a). A transitive reduction on the entire graph H_1 in Step 13, removes redundant arcs.

Case (iv): The arc (i, j), where j is a compound node is redundant when there are arcs (i, j_1), ..., (i, j_l), where j_1, ..., j_l are *some* or *all* of the components of compound node j and each arc (i, j_k), where, $1 \leq k \leq l$, is redundant. In Steps 14-18 the arc (i, j) is removed and arcs (i, j_1), ..., (i, j_r) are added, where j_1, ..., j_r are *all* of the components of j. The resulting graph is reduced using transitive reduction.

From the above we can clearly infer that H_m is non-redundant. It is minimal since in Step 10 the implied extraneous attributes are removed from each compound node and there are no arcs to compound nodes. Also, for two equivalent nodes X and Y there is only one arc from the component containing both X and Y and hence H_m is a minimum one.■

The presence of the arcs to compound nodes helps to treat the graph H as a directed digraph which makes the application of parallel transitive closure algorithms possible. Given H_1 all the steps in the algorithm except for the Steps 7,8 and 10 can be implemented using parallel transitive closure algorithms in $O(M(n))$ time with $O(n+e)$ processors on a CREW-PRAM.

The implementations of Steps 7, 8, and 10 are given in the following. We would first present a method to determine textual containment of two nodes in H_1. We will assume that Steps 1-6 have been executed on the graph H_1. We will show how to test the textual containment of two nodes in the same strong component first. We construct the following data structure for the graph H_1. For each compound node J, let j_1, j_2, ..., j_J be the nodes such that each $j_p \subset J$. The set $S(J)$ consists of ordered pairs (j_c, c), where j_c is the union of all j_p's in strong component c. Keep all the $S(J)$ sets sorted first on the component number containing J and next the rank compound node J within the component. For each ordered pair (j_c, c) in $S(J)$ we have a list of pointers $L(j_c, c)$. Each pointer $l_{jc} \in L(j_c, c)$ points to an ordered pair in the set $S(K)$ and K is in the same component as J with rank(K) > rank(J). In order to check the textual containment of node J in node K collect the pointers $l_{j1}, l_{j2}, ..., l_{jr}$, where, each l_{ji} points to an ordered pair in $S(K)$. Now if $j_1 \cup j_2 \cup ... \cup j_r \supseteq J$, then J is textually contained in node K.

The sum of the sizes of the $S(J)$'s and the pointers in $L(j_c, c)$'s are both at most $O(n+e)$. The $S(J)$ sets can be ordered as required above in time $O(\log n)$ time using $O(n)$ processors on a CREW-PRAM. We will show that the $S(J)$ sets can be constructed in $O(\log n)$ time using $O(n+e)$ processors. The sets $S(J)$'s can be constructed by sorting the j_i's $\subset J$ based on the strong component number they are in and then taking a union of j_i's which are in the same component c to form the ordered pair (j_c, c). The sorting and union operation using recursive doubling techniques can be done in $O(\log n)$ time with $O(n+e)$ processors on a CREW-PRAM. Note that the j_i's $\subset J$ can be obtained during the construction of the graph H_1. The collecting of pointers which point to the ordered pair in $S(K)$ can all be done by sorting the pointers based on the list they point to and the test of textual containment can all be done in $O(\log n)$ total time for all compound nodes in the graph H_1 using $O(n+e)$ processors on a CREW-PRAM.

The Step 8 of the algorithm can be implemented as described above. The most time consuming step in the entire algorithm is the Step 10. Removal of implied extraneous attributes can be compared with the problem of finding the minimal key. In fact, given a minimum directed hypergraph H_m, the minimal key is the union of the representatives contained in the strong components whose indegree is zero. The problem of left-reducing a dependency $X \rightarrow Y$ is finding a minimal key in the graph $H_m(X)$, where $H_m(X)$ is the graph induced by nodes $Z \subset X$ in H_m. Now, the removal of implied extraneous attributes from a compound node X is done by finding all strong components containing a node $Z \subset X$ and assuming all such Z's do not have implied extraneous attributes we pick Z's from components which do not have an incoming arc. The union of all such Z's gives the node X without any implied extraneous attributes. The time taken to do this is dependent on the diameter of the graph H_1 and can be easily done in $O(\log n)$ time for a constant diameter graph H_1 with $O(n+e)$ processors on a CREW-PRAM. nodes.

Theorem 5: A minimum directed graph H_m of the directed hypergraph H_Σ for a given set of functional dependencies Σ can be obtained using algorithm Minimize in $O(\log^2 n + MAX(degree(H_\Sigma), diameter(H_\Sigma))*\log n)$ using $O(M(n))$ processors on a CREW-PRAM.

Proof: The correctness of the algorithm Minimize follows from Lemma 4.1. Step 1 takes $O(\log^2 n + MAX(degree(H_\Sigma), diameter(H_\Sigma))*\log n)$ using $O(M(n))$ processors (Theorem 4). Steps 7 and 8 take $O(\log n)$ time and uses $O(n+e)$ processors from the above discussion. Also, Step 10 can be implemented in $O(diameter(H_\Sigma).\log n)$ with $O(n+e)$ processors as discussed above. All other steps can be done in $O(\log^2 n)$ time using $O(M(n))$ processors, since they all use transitive closure algorithms as a subroutine. All the steps require the CREW-PRAM model.■

From Theorem 5 we can see that for fixed degree and diameter graphs H_Σ the complexity of the algorithm Minimize is $O(\log^2 n)$ and uses $O(M(n))$ processors. Hence it is optimal based on the transitive closure bottleneck phenomenon.

5. CONCLUSION

We have presented parallel algorithms for the manipulation of functional dependencies which arise in relational databases. The functional dependencies were represented as a directed hypergraph and parallel graph algorithms for finding transitive closure, transitive reduction and to compute strongly connected components were used for manipulating the directed hypergraph. The complexity of manipulating the given set of FDs is shown to be P-Complete in the general case. This implies that manipulation of FDs is resistant to high-degree parallelism. We have shown that efficient parallel manipulations can be derived for directed hypergraphs with fixed degree and diameter. From the LR-Minimum directed graph the design of relational schemes can be easily carried out. For more details (see [1]).

Manipulating functional dependencies were thought of a one-time-process, that is, the dependencies are manipulated during the design stage and never again processed. This was shown to be no longer true as dependencies could be introduced as the database system is being used and maintained. Such dependencies were called as *dynamic data dependencies* [12]. It would be interesting to see how the directed hypergraphs can be maintained in the context of dynamic data dependencies i.e., perform updates on the directed hypergraph in parallel.

ACKNOWLEDGEMENTS

We thank the anonymous refrees for their comments and Dr. Rakesh Agarwal for the prompt communications with respect to our submission to this conference.

References

[1] AUSIELLO, GIORGIO, ATRI, ALESSANDRO D', AND SACCA, DOMENICO, "Graph Algorithms for Functional Dependency Manipulation," *JACM*, vol. 30, no. 4, pp. 752-766, October 1983.

[2] AUSIELLO, GIORGIO, ATRI, ALESSANDRO D', AND SACCA, DOMENICO, "Minimal Representation of Directed Hypergraphs," *SIAM J. COMPUT.*, vol. 15, no. 2, pp. 418-431, May 1986.

[3] BARU, C. K. AND FRIEDER, O., "Database operations on Cube-Connected Multicomputer System," *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 920-927, June 1989.

[4] DIEDERICH JIM. AND MILTON JACK., "New Methods and Fast Algorithms for Database Normalization," *ACM TODS*, vol. 13, no. 3, pp. 339-365, September 1988.

[5] ECKSTEIN, D.M., "Simultaneous Memory Access," *Dept. Comput. Sci., Iowa State Univ., Iowa City, Tech. Rep. TR-79-6*, 1979.

[6] GALIL Z., "An almost linear time algorithm for computing dependency basis in a relational database," *JACM*, vol. 29, no. 1, pp. 96-102, January 1982.

[7] KARP, R.M. AND RAMACHANDRAN, V., "A Survey of Parallel Algorithms for Shared-Memory Machines," *Rept. No. UCB/CSD 88/408, Comp., Sci., Division, Berkeley, California*, 1988.

[8] MAIER DAVID., "The Theory of Relational Databases," *Computer Science Press, Rockville, Md.*, 1983..

[9] OMIECINSKI, E. AND TIEN, E., "A Hash-Based Join Algorithm for a Cube-Connected Parallel Computer," *IPL*, vol. 30, pp. 269-275, March 1989.

[10] ULLMAN, J.D., "Principles of Database Systems," *Computer Science Press, Rockville, Md.*, 1983..

[11] UPFAL, E. AND WIGDERSON, A., "How to Share Memory in a Distributed Environment," *Proc. 25th Annu. IEEE Symp. Foundations Comput. Sci., West Palm Beach, FL*, pp. 171-180, October 1984.

[12] VIANU, V., "Dynamic Functional Dependencies and Database Aging," *JACM*, vol. 34, no. 1, pp. 28-59, January 1987.

[13] VITTER, J.S. AND SIMONS, R.A., "New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems in P," *IEEE. Trans. On Comput.*, vol. c-35, no. 5, pp. 403-417, May 1986.