AN ALGORITHM FOR TREE-QUERY MEMBERSHIP OF A DISTRIBUTED QUERY*

C.T. Yu
Department of Information Engineering
University of Illinois, Chicago Circle
Chicago, Illinois 60680

ABSTRACT

The aim is to process distributed queries efficiently. The cost of communications between sites is dominant in processing such queries. It is assumed that the amount of data transferred determines the transmission cost to a large extent. Thus, it is desirable to minimize the amount of transmitted data.

Bernstein and Chiu [2] classified queries into two types: tree and cyclic queries. They defined an operation called semi-join which requires minimal transfer of data between sites. Then they showed that tree queries can always be answered by semi-joins but cyclic queries may not. An algorithm to decide whether a query is cyclic or not was presented in their paper. Their algorithm works when the number of domains in common between any two relations is no more than one. The aim of this paper is to generalize their algorithm. Specifically, we present a conceptually simple algorithm which decides the type of a query when the number of domains in common between two relations may exceed one. An implementation of the algorithm is outlined. The algorithm runs in O(max(e,e')) time and O(e) space complexity where e and e' are the number of edges in the transitive closure of the join graph and the query graph respectively.

1. Introduction

A distributed database management system (DBMS) allows data to be stored at multiple locations and to be accessed as a single unified database. In comparison to a centralized database, a distributed DBMS has the following advantages [11]: (1) it is more reliable, (2) provides faster access by storing data at locations where it is frequently used, and (3) the database capacity can be more easily adjusted to changing needs. The trend is toward development of general purpose DBMS. A survey of problems related with the distributed DBMS can be found in [1,3,11].

An important problem in the area is to find an efficient strategy to process queries referencing data in different sites. Answering such queries requires data movement over the communication lines among different sites. An example of a distributed query is a join of two relations residing in different sites. Usually the cost of communications between sites is the dominant factor in processing a distributed query. It is therefore desirable to minimize the communication cost. Recent development in teleprocessing technology indicates that communication cost can be mostly accounted for by the amount of data transferred.

It is well-known that the minimization of data transfer is a very complicated problem [11,14,10,2]. Wong proposed a greedy algorithm [14] for query processing in SDD-1 [12] system. The algorithm obtains a local optimum solution. Hevner and Yao [9,10] studied the problem under the independence assumption of domains in a relational distributed DBMS. Their approach resulted in an optimum strategy for single domain relations [9]. For more general situations, their algorithm serve as a heuristic [10]. Other distributed query processing strategies that have been proposed [13,7] are extensions of centralized query processing.

Recently, Bernstein and Chiu analyzed the problem with a different approach [2]. They classified queries into two disjoint classes: tree queries and cyclic queries. They showed that tree queries can be answered by transmitting only the data in the common domains of relations which reside at different sites. On the other hand, answering cyclic queries involves more elaborate data transfer. Hopefully, enough insight can be gained by designing optimal strategies for tree queries so that the approach can be generalized to cyclic queries.

Bernstein and Chiu presented an algorithm which decides whether a query is a tree query [2]. However, their algorithm works under the assumption that semi-joins are only capable of comparing a single domain of one relation with that of another. Naturally, semi-joins can be extended to compare multiple domains of relations simultaneously and this generalization results in a larger class of tree queries.

The aim of this paper is to present an algorithm which decides whether a given query is a tree query under the generalized definition of semi-joins. In section 2, the background concepts and notations are given. Section 3 provides an example for a tree query which is classified as a cyclic query by Bernstein and Chiu's Algorithm [2] but answerable by generalized semi-joins. In Section 4, we present an algorithm which decides
whether a given query is a tree query. The implementation of the algorithm together with its time and space complexities are discussed in the last section.

2. Notations and Previous Results

The relational database model [4,5,6] is assumed. The \textit{join} of two relations \(R_i(X,Y)\) and \(R_j(Y,Z)\), where \(X, Y\) and \(Z\) are disjoint sets of attributes, is given by
\[
R_i \ join \ R_j = \{(x,y,z) | (x,y) \in R_i \ and \ (y,z) \in R_j\}.
\]

A query in the present paper is assumed to be a conjunction of clauses of the form \((R_.d = R_.d')\) where \(d\) and \(d'\) are called the common joining domains of the relations \(R_i\) and \(R_j\). This type of queries were considered by Bernstein and Chiu [2].

The database is distributed in different sites. Following [9,10,14], it can be assumed that each relation resides in one site. Local processing refers to query processing operations that are performed in a site. Communication cost, i.e., the cost for transferring data from one site to another, is assumed to be dominant. The objective is to transfer as small amount of data between sites as possible even at the expense of more local processing.

Bernstein and Chiu defined an operation, called semi-join [2]. The data transfer required by this operation between two sites having two different relations is restricted to subtuples which are in the common domains of the relations. Clearly, the amount of transferred data is limited and the transmission cost is minimal. However, Bernstein and Chiu [2] showed that some queries may be answered by semi-joins while semi-joins may not answer other queries.

A query \(Q\) is represented by a join graph \([2], JG(V,E)\), whose vertices are domains of the relations involved in the query and whose edges represent the clauses of the query. More precisely,
\[
V = \{R_.a | a \ is \ a \ domain \ of \ R_i\}
\]
and
\[
E = \{(R_.a,R_.b) | (R_.a = R_.b) \ is \ a \ clause \ in \ Q\}.
\]

An example is given in figure 2.1.

Two queries are equivalent if their answers are the same, irrespective of the contents of the relations [2]. Let \(Q\) contain
\[
W = \bigwedge_{i=1}^{n} (R_.d_i = R_.d_i') \ where \ d_i's \ are \ domains.
\]
Suppose an additional clause \((R_.d_i = R_.d_j)\) for some \(1 \leq i,j \leq q+1\) is added to \(Q\). The new clause does not change the answer to \(Q\) since both \(W\) and \(W \land (R_.d_i \land R_.d_j)\) are equivalent to \((R_.d_i = R_.d_j)\) for some \(1 \leq i,j \leq q+1\), by the transitivity of equality.

W corresponds to a spanning tree of a connected components in the join graph \(JG\) containing the vertices \((R_.d_1 \ldots R_.d_n)\), while the additional clause corresponds to an additional edge in the same component. It is clear that adding additional edges within any connected component which already exists in the join graph of \(Q\) will not affect the answer to the query. In fact, two queries are equivalent if they have the same transitive closure in their join graphs, where the transitive closure of the connected components consists of all possible edges between any two distinct vertices within a connected component. In this and the next two sections, we shall restrict ourselves to join graphs which are spanning forests (A spanning forest is a set of spanning trees, each for one connected component in a join graph) since additional edges are redundant.

Since \((R_.a=constant)\) and \((R_.a=R_.a')\) can be performed by local processing, such clauses can be eliminated. Thus, if a domain of a relation is in a connected component, other domains of the same relation can be assumed to be absent from the connected component. For simplicity, we say a connected component contains \(R_i\) if a domain of \(R_i\) is in the component. We also say that \(R_i\) and \(R_j\) are adjacent in a connected component, if an edge \((R_.d_i,R_.d_j)\) exists in the component.

The join graph as defined above differs from that of Bernstein and Chiu [2] in that our definition limits the number of edges between any two vertices to be at most one while their definition allows multiple edges between any pair of vertices.

Given a join graph \(JG\) of \(Q\), we want to examine whether it is possible to answer the query by transferring data in the common joining domains of the relations, i.e., by using semi-joins. This is facilitated by a query graph \((QG(V_1,E_1))\), where
\[
V_1 = \set{R_i} \text{ referenced in } Q, \quad E_1 = \{(R_.a,R_.b) | R_i \ and \ R_j \ are \ adjacent \ in \ a \ connected \ component \ in \ JG\}.
\]

In essence, a mapping \(NQ\) is defined, mapping edges in the connected components of the join graph to edges in the query graph. An example of a query graph is given in figure 2.1. In the next section, we shall assume the given query has a connected query graph, otherwise the query is a conjunction of subqueries, each corresponding to a connected component in the query graph [2]. It is easy to verify that if the query graph of a query is connected then the query graph of any equivalent query is also connected.

It was shown in [2] that if the query graph of \(Q\) is a tree, then \(Q\) can be answered by semi-joins. On the other hand, if \(Q\) has a cyclic query graph, it may be possible to transform \(Q\) to an equivalent query (having a different spanning forest in its join graph) such that the equivalent query has a tree query graph. A query is a tree
query if either itself or an equivalent query has a tree query graph. If all equivalent queries of Q produce cyclic query graphs, then the query is cyclic and may not be answerable by semi-joins [2]. Thus a query can be one of the two types: a tree query or a cyclic query. The set of all tree queries are denoted by TQ and all cyclic queries by CQ.

3. An Example

Bernstein and Chiu [2] give an algorithm which decides whether a query is a tree query. In this section, their algorithm is presented in the notation defined in Section 2. An example taken from page 24 in [2] shows that it is possible that their algorithm decides a query to be cyclic, yet the query graph of an equivalent query is a tree.

Their algorithm presented in our notation:

1. Choose any spanning forest.
2. Map the spanning forest to a query graph by the mapping MJQ.
3. If the query graph is cyclic then the query is cyclic; otherwise it is a tree query.

The example given in figure 2.1 has a tree query graph, yet can be determined to be cyclic by the above algorithm.

4. A Simple Tree Query Membership Algorithm

In the last section, we saw that a query can have a cyclic query graph and yet is a tree query. This tends to indicate that certain edges and vertices are unimportant in deciding on the type of a query. Thus the elimination of edges and vertices can continue as long as the type of the query is preserved. It turns out that if all the vertices and edges are eliminated then we have a null join graph and a null query graph, indicating that the final query and therefore the initial query are tree queries. On the other hand, it will be shown that if some vertices and edges remain, then the resulting query and the initial query are cyclic queries.

Initially, there are at least two relations in each connected component (since each clause in the query refers to some domains of two relations) and we define PJ to be the set of connected components which contain Rj. If Pi intersect Pj, it will be shown (proposition 4.4) that we can restrict our consideration to a subset of join graphs denoting equivalent queries of the original query Q because Q has an equivalent query with a tree query graph iff there is a tree query graph corresponding to a join graph in the subset. Furthermore, the query graph of any join graph in the subset must contain the edge (Ri,Rj) and no other edge is incident on Rj.

Since there is a single edge incident on Rj, any cycle, if exists, in the query graph can not be incident on Rj. Thus deleting Rj will not affect the type of Q. After the elimination of Rj (and the edge (Rj,Rj)), there may be some connected components each having a single vertex. Those connected components should be removed as they do not contribute any edge to the query graph and therefore have no influence on the type of Q. However, their elimination may alter some P's, which may in turn cause some P's to be subsets of some other P's. This process continues until either all relations and edges of the form (Ri,Rj) are removed (giving a null join graph and a null query graph) or some relations remain such that for every pair of relations Ri, Rj in the remaining set, (iP,Pj) and each connected component has at least 2 relations. In the former situation, Q is a tree query since a null query graph is a tree query graph and each elimination preserves the type of the query. Furthermore, a tree query graph of an equivalent query of Q consists of the deleted edges (lemma 4.3). In the latter situation, proposition 4.5 shows that the resulting query must be cyclic.

We now proceed to show the above statements. Let SJGQ be the set of join graphs such that each join graph is a spanning forest and denotes a query equivalent to Q. A function M is now defined, mapping connected components in join graphs in SJGQ to connected components in join graphs of a subset of SJGQ, denoted by SJGQ, such that the query graph of a join graph in SJGQ contains the edge (Ri,Rj) but does not contain any edge of the form (Ri,Rk), k # j.

M is described as follows. A connected component in a join graph in SJGQ is one of the following three types: (1) it does not contain Rj, (2) it contains Ri and Rj and the edge (Ri,Rj), (3) it contains Rj but without the edge (Ri,Rj). Connected components containing Ri but not Rj do not exist in SJGQ since Pi intersect Pj. M leaves connected component of the first type unchanged. For the second type of connected components, edges of the form (Ri,Rj), k # j are mapped to (Ri,Rj), while other edges are unaffected. Let the edges incident on Rj in a connected component of the third type be (Rj,Rj),..., (Rj,Rj). Since the connected component is a spanning tree, there is a unique path from Rj to Rj passing exactly one vertex in (Rj,Rj). Without loss of generality, let Rj be that vertex. Then (Rj,Rj) is mapped to (Rj,Rj) and (Rj,Rj), 2 # i, are mapped to (Rj,Rj).

It will be shown in lemma 4.1 that if C is a connected component in SJGQ then M(C) has identical set of vertices and is connected. Thus,
$JC_0$ and $M(JC_0)$ represent queries which are equivalent to $Q$. Lemmas 4.2 & 4.3 show that a tree query graph is obtained from a join graph in $SJG_0$ iff a tree query graph is obtained from the corresponding join graph in $SJG_0$. Thus, in order to determine the type of a query, it is sufficient to consider only join graphs in $SJG_0$. By the construction of $M$, the degree of $R_i$ in the query graph corresponding to any join graph in $SJG_0$ is 1. Thus, any cycle, if exists in the query graph can not be incident on $R_i$. As a consequence, removing $R_i$ doesn’t affect the query type, as summarized by proposition 4.4. Finally, proposition 4.5 shows that if the elimination process leaves some relations, then the query must be cyclic.

**Lemma 4.1:** If $C$ is a connected component in $JC_0$, then $M(C)$ has identical set of vertices and is connected.

**Proof:** It is easy to verify that if $R_k$, $k \neq i$, $k \neq j$, and $R_j$ are in $C$, then $R_k$ is connected to $R_j$ in $M(C)$. Furthermore, if $R_i$ is in $C$, then $R_i$ is adjacent to $R_j$ in $M(C)$. Thus, $M(C)$ is connected.

**Lemma 4.2:** Let $JC_1$ be a join graph in $SJG_0$ such that $QC_1 = M(JC_1)$ is a tree. Then there exists a join graph $JC_0$ in $SJG_0$ such that $QC_0 = M(JC_0)$ is a tree.

**Proof:** Since $SJG_0 \subseteq SJG_1$, we can choose $JC_0 = JC_1$.

**Lemma 4.3:** Let $QC_0$ be a tree query graph such that a join graph $JC_0$, in $SJG_0$ is mapped to $QC_0$, i.e., $M(JC_0) = QC_0$. If $QC_0$ is a tree then $M(JC_0)$ produces a tree query graph, i.e., $M(JC_0) = QC_0$ is a tree.

**Proof:** Let $JC_1$ denote $M(JC_0)$ and $QC_1$ denote $M(JC_1)$. We show $QC_1$ is a tree by demonstrating that $QC_1$ is connected and it has the same number of edges as $QC_0$. $QC_0$ may or may not have the edge $(R_i, R_j)$.

**Case 1:** If $QC_0$ has the edge $(R_i, R_j)$, then all connected components in $JC_0$ must be of types (1) or (2), because any connected component of type (3), when mapped by $M$, to the query graph $QC_0$, would provide an alternate path from $R_i$ to $R_j$ via some vertex. This contradicts $QC_0$ being a tree.

Consider an edge $(R_i, R_k)$, $k \neq i$, $k \neq j$, in $QC_0$. By the mapping $M$, there exists a connected component containing the edge $(R_i, R_k)$. This edge is mapped by $M$ into $(R_i, R_j)$ in a connected component in $JC_1$. Thus, $QC_1$ contains $(R_i, R_j)$. $QC_0$ can not have the edge $(R_k, R_j)$, otherwise there is a cycle containing $(R_i, R_k, R_j)$. Thus $QC_1$, in comparison with $QC_0$, loses an edge $(R_i, R_k)$ but gains back $(R_j, R_k)$. Since the same argument is applicable for every edge $(R_i, R_j)$, $k \neq i$, $k \neq j$, $QC_1$ has the same number of edges as $QC_0$.

We now show that $R_j$ is connected to $R_j$, $k \neq i$, $k \neq j$, in $QC_1$. Since $QC_0$ is connected, there is a path from $R_k$ to $R_j$ in $QC_0$. This path may or may not pass through $R_i$. If the path does not pass through $R_i$, then this path remains unaffected by mapping $M$ and therefore such a path exists in $QC_1$. If the path goes through $R_i$, let the path be $R_k \cdots R_i R_j \cdots j$. Then the path $R_k' \cdots R_j'$ exists in $QC_1$, by definition of $M$. Since $R_k$, $k \neq i$ and $k \neq j$, is connected to $R_j$ and $R_i$ and $R_j$ are adjacent, $QC_1$ is connected.

**Case 2:** If $QC_0$ does not have the edge $(R_i, R_j)$, then similar arguments show that all connected components in $JC_0$ must be of types (1) or (3). Furthermore, if $R_i$ is adjacent to $R_j$ which is then connected to $R_j$ in a connected component in $JC_0$, then in any other connected component in $JC_0$, $R_i$ can not be connected to $R_j$ via a different vertex $R_p$, $p \neq k$, since $QC_0$ is a tree. The remaining arguments are similar to those of case 1.

**Proposition 4.4:** Let $Q$ be a query such that $P_i \subseteq P_j$ for some $1 \leq i, j \leq n$. There exists a tree query graph corresponding to a join graph in $SJG_0$ iff there exists a tree query graph corresponding to a join graph in $SJG_0$. Furthermore, any cycle in the query graph corresponding to a join graph in $SJG_0$ can not be incident on $R_i$.

**Proof:** Follows from lemmas 4.2 & 4.3 and the fact that $R_i$ has only one edge incident on it.

**Proposition 4.5:** Let $Q$ be a query involving the relations $(R_1, \ldots, R_n)$ such that $\gamma(P_i \subseteq P_j)$, $1 \leq i, j \leq n$, and each connected component has at least two relations. Let $Q$ be the query graph of any join graph, say $JC$, representing $Q$. Then $QC$ is cyclic.

**Proof:** Consider $R_i$ in $QC$. $R_i$ is adjacent to some vertex $R_j$ in a connected component in $JC$. Thus, they are also adjacent in $QC$. Since $\gamma(P_i \subseteq P_j)$, there is at least one connected component in $JC$. 

309
containing \( R_i \) but not \( R_j \). Let \( R_k \) be a vertex adjacent to \( R_i \) in that connected component. Then \( R_j \) must also be adjacent to \( R_k \) in \( Q \). This shows that the degree of \( R_j \) in \( Q \) is at least 2. Since the above argument is applicable to every vertex in \( Q \), each vertex has degree \( \geq 2 \) and therefore \( \lbrack 8 \rbrack \) \( Q \) is cyclic.

Below we give an algorithm which successively removes \( R_i \) from \( Q \) if \( P_i \subseteq P_j \) where \( R_i \) and \( R_j \) are in \( Q \), \( 1 \leq i, j \leq n \). After each elimination step, the procedure is repeated for the resulting query until either the resulting query is null or does not contain any \( P_i \subseteq P_j \).

**Algorithm (test if \( Q \in CQ \))**

1) From \( Q \), construct \( P=\{P_1, \ldots, P_n\} \) and connected components \( \{C_1, \ldots, C_m\} \). Initialize \( T \) to be an empty set of edges.

2) While check \( (P_i, P_j) \) for some \( i, j, 1 \leq i, j \leq n \), do

3) If \( \{P_i \subseteq P_j\} \) then return \( (Q \in CQ) \) else return \( (Q \not\in CQ) \).

**End**

Clearly, the algorithm terminates either when \( P \not= \emptyset \) or when there is no \( (P_i \subseteq P_j) \) among the remaining relations \( i \neq j, 1 \leq i, j \leq n \) and \( P_i, P_j \not= \emptyset \). The correctness of the algorithm follows directly from Propositions 4.4 and 4.5.

5. An Implementation of the Algorithm

It is clear that step 1 can be performed in time proportional to the number of vertices in a join graph of \( Q \), and step 3 takes constant time only. Thus, we concentrate on step 2. We first give the data structures which represent the \( P \)'s, the connected components and which facilitate the checking for \( (P_i \subseteq P_j) \) for some \( i, j, 1 \leq i, j \leq n \).

Then the algorithm will be analyzed for its time and space requirements.

As explained in section 2, a connected component can contain at most one occurrence of a relation if local processing is applied whenever possible. Thus, the set of join graphs of \( Q, S \bar{Q} \), may be represented by an \( n \) by \( m \) binary matrix, where the rows denote the relations, the columns denote the connected components and the \((i,j)\) entry of the matrix is 1 if the \( i \)th relation is contained in the \( j \)th connected component. In practice, the nonzero entries of this matrix is represented by two sets of intersecting linked lists \( \{LR_j\}_{1 \leq j \leq n} \) and \( \{LC_j\}_{1 \leq j \leq m} \). The singly-linked list \( LR_j \) represents \( P_i \) while the doubly-linked and circular list \( LC_j \) which consists of the set of relations contained in the \( j \)th connected component. Figure 5.1 gives an example of how the linked lists are used. A one-dimensional array, \( CC \), is used where \( CC(i) \) gives the number of the \( i \)th connected component, \( 1 \leq i \leq m \).

In order to check whether \( P_i \subseteq P_j \), \( 1 \leq i, j \leq n \), an \( n \) by \( n \) matrix, \( Count \), is constructed. Initially, \( Count(i,j)=|P_i\| \), \( 1 \leq i, j \leq n \). As executions (to be described) proceed, \( Count(i,j)=|P_i-P_j| \). Clearly, \( |P_i-P_j|=0 \) iff \( P_i \subseteq P_j \). To find the set of \( P \)'s contained in a given \( P_j \), the linked list \( LR_j \) is traversed. This list intersects with the lists of connected components, \( \{LC_{j'}\}_{1 \leq j' \leq m} \), contains \( R_j \), containing \( R_j \). Each occurrence of any relation \( R_u \), \( u \neq j \), in the list \( LC_{j'} \) indicates a connected component containing both \( R_u \) and \( R_j \). Thus, \( Count(u,j) \) should be decremented by the number of occurrences of \( R_u \) in such lists \( LC_{j'} \). This is accomplished by traversing the linked lists \( LC_{j'} \), while decrementing \( Count(u,j) \) for each occurrence of \( R_u \), \( u \neq j \). Each element in \( LR_j \) denotes an occurrence of \( R_j \) in a connected component \( C_u \). When this element is reached in \( LR_j \), each element in \( LC_u \) is visited once. The number of operations to visit the elements in \( LC_u \) (excluding \( R_j \)) is proportional to the number of edges incident \( R_j \) in \( C_u \). Since this process is carried out for each element in \( LR_j \), each list \( LR_j \), the total time required to check whether \( P_i \subseteq P_j \), \( 1 \leq i, j \leq n \), is \( O(e) \) where \( e \) is the number of edges in the transitive closure of a join graph of \( Q \).

Removing \( P_i \) from \( P \) consists of deleting \( LR_i \).
In order to remove \( P_i \) from each connected component containing \( R_j \), it is sufficient to traverse \( L_{R_j}^{i} \) once while removing the common elements from intersecting \( L_{C}^{i} \)'s. The removal is facilitated by having \( L_{C}^{i} \)'s as doubly linked lists. In the worst case, all \( L_{R_j}^{i} \)'s are removed. Thus, time required by step 2(i) is no more than \( O(e) \).

While traversing \( L_{R_j}^{i} \) to delete \( P_i \) from \( P_i \), the counts of the intersecting connected components, \( C_j \)'s can be decremented. Since the sum of all counts in the connected components are the number of vertices in the join graph, the maximum time required in step 2(ii) is no more than \( O(e) \).

The execution of step 2(ii) may cause some \( P_i \)'s to be subsets of other \( P_i \)'s. Thus, the matrix \( \text{Count} \) has to be updated. Let \( S = \{C_j, 1 \leq j \leq n\} \) be the set of connected components which contain only one relation, after removing \( P_i \). The relation contained in any connected component in \( S \) is \( R_j \) because each such connected component previously contained both \( R_j \) and \( R_k \), and \( R_j \) was just removed. Consequently, if there are \( f \) connected components in \( S \), \( 1 \leq j \leq n \), then all the counts associated with \( R_j \), \( \text{Count}(j, t) \), \( 1 \leq t \leq n \), has to be decremented by \( f \). The number of counts that has to be decreased due to the elimination of \( P_i \) is at most \( n-1 \). Since the number of relations to be removed is at most \( n \), the updating of the matrix takes at most \( O(n^2) \) operations.

Since the query graph is connected, \( O(e'') = O(e') \) where \( e' \) is the number of edges in the transitive closure of the query graph.

Since step 2(iv) can not take more than \( O(e_i) \) the overall time complexity of the algorithm is bounded by \( O(\text{max}(e,e'')) \).

For each vertex in the connected components of a join graph of \( Q \), there is a node in the linked lists \( \{L_{R_j}^{i}, 1 \leq j \leq n\} \). Thus, the storage requirement for the two sets of lists is proportional to the number of vertices in a join graph of query. This is no more than \( O(e) \).

The matrix \( \text{Count} \) takes \( n^2 \) storage. However, the \( (i,j)^{th} \) entry of the matrix is used iff \( P_i \cap P_j \neq \emptyset \). Thus, the number of entries in the matrix that are needed is \( 2e_i \) where \( e_i \) is the number of edges in the query graph corresponding to the transitive closure of join graph. Since the mapping from a join graph to a query graph is many (edges) to one (edge), \( e_i \leq e' \). The total storage requirement is therefore \( O(e) \). The matrix \( \text{Count} \) can be represented by linked lists storing the useful counts only, while preserving the time complexity.

Recently, Bernstein and Goodman [15] obtained similar results. Their approach was somewhat different from others and their report appeared two months later.

References

Figure 2.1: Two equivalent queries, one having a cyclic query graph and the other a tree query graph.