Optimization of Multiple-Relation Multiple-Disjunct Queries

by
M. Muralikrishna
David J. DeWitt

Computer Sciences Technical Report #740
January 1988

Optimization of Multiple-Relation Multiple-Disjunct Queries

M. Muralikrishna

David J. DeWitt

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

This research was partially supported by the Department of Energy under contract DE-AC02-81ER10920, by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grants DCR-8512862, MCS82-01870, and MCS81-05904, and by a Digital Equipment Corporation External Research Grant.

1. Abstract

In this paper we discuss the optimization of **multiple-relation multiple-disjunct queries** in a relational database system. Since optimization techniques for **conjunctive (single disjunct)** queries in relational databases are well known [Smith75, Wong76, Selinger79, Yao79, Youssefi79], the natural way to evaluate a multiple-disjunct query was to execute each disjunct independently [Bernstein81, Kerschberg82]. However, evaluating each disjunct independently may be very inefficient. In this paper, we develop methods that merge two or more disjuncts to form a **term**. The advantage of merging disjuncts to form terms lies in the fact that each term can be evaluated with a single scan of each relation that is present in the term. In addition, the number of times a join is performed will also be reduced when two or more disjuncts are merged. The criteria for merging a set of disjuncts will be presented. As we will see, the number of times each relation in the query is scanned will be equal to the number of terms. Thus, minimizing the number of terms will minimize the number of scans for each relation. We will formulate the problem of minimizing the number of scans as one of covering a **merge graph** by a minimum number of **complete merge graphs** which are a restricted class of **cartesian product graphs**. In general, the problem of minimizing the number of scans is NP-complete. We present polynomial time algorithms for special classes of merge graphs. We also present a hueristic for general merge graphs.

Throughout this paper, we will assume that no relations have any indices on them and that we are only concerned with reducing the number of scans for each relation present in the query. What about relations that have indices on them? It turns out that our performance metric of reducing the number of scans is beneficial even in the case that there are indices. In [Muralikrishna88] we demonstrate that when optimizing single-relation multiple-disjunct queries, the cost (measured in terms of disk accesses) may be reduced if all the disjuncts are optimized together rather than individually. Thus, our algorithm for minimizing the number of terms is also very beneficial in cases where indices exist.

2. Background and Previous Work

Previously all query optimization research has concentrated on the optimization of conjunctive (single disjunct) queries only. In fact, even the optimization of conjunctive queries is known to be a hard problem [Chandra77]. Tableaus [Aho79] have been used to represent a subset of relational calculus queries. These queries involve only equality based selections, projections, natural-joins, and only AND connectors. Thus, tableau queries are a subset of conjunctive queries [Chandra77]. Most query optimizers start by transforming the user query into some standard canonical form. This standard form is then rearranged/simplified into an equivalent but cheaper expression. Two standard forms that have been used are the disjunctive normal form (DNF) and the conjunctive normal form (CNF). Both Ingres [Wong76] and System R [Selinger79] seem to use CNF but do not explain how they deal with queries that have ORs in their qualification list. All of their examples (as in other query optimization papers) deal only with queries that have no ORs. As pointed out earlier, since the general optimization problem is computationally intractable, cheaper expressions are obtained by applying heuristics. Single disjunct queries can be augmented with additional selects using transitive rules [Youssefi79]. A series of projections can be combined into a single projection and similarly a sequence of restrictions can be combined into a single restriction [Smith75]. Minimizing the sizes of intermediate results is achieved by performing selections and projections before joins and cartesian products [Smith75, Wong76]. Experimental results [Youssefi79] have shown that performing selections before joins is a very good heuristic. DNF has been used in [Bernstein81, Kerschberg82] to optimize and evaluate the query disjuncts separately. An excellent survey of query optimization is presented in [Jarke84].

In this paper, we demonstrate that optimizing queries involving multiple disjuncts by optimizing each disjunct separately can be very inefficient. Considerable savings in cost may be achieved by optimizing the disjuncts together.

3. The Problem

Definition: A **term** is an expression of the form[1] $\prod_{i=1}^{n} P_i$, $n > 0$, where each P_i is either a join clause or a boolean expression (in disjunctive normal form) of selection clauses on exactly one relation. □

We will motivate the problem associated with optimizing each disjunct separately with a couple of examples. Throughout this chapter, S_i, T_j, and U_k will denote single-relation selection clauses on the relations S, T, and U respectively. J will denote a join clause.

To illustrate the effect of optimizing each disjunct separately, consider the following query with four disjuncts[2]:

$$S_1 \cdot T_1 \cdot J + S_1 \cdot T_2 \cdot J + S_2 \cdot T_1 \cdot J + S_2 \cdot T_2 \cdot J \text{ (Example 1)}$$

[1] $\prod_{i=1}^{n} P_i$ is equivalent to P_1 AND P_2 AND ... AND P_n.

If the four disjuncts in this query were optimized and run separately, the S and the T relations would each be scanned four times and the join J would be executed four times, each time with different inputs. On the other hand, if the four disjuncts are transformed into the following term:

$$(S_1+S_2)\cdot(T_1+T_2)\cdot J$$

S and T will each be scanned once and J will be executed only once.

As another example, the query,

$$S_1\cdot J + T_2\cdot J + S_2\cdot J + T_1\cdot J \text{ (Example 2)}$$

can be transformed to the equivalent query

$$(S_1+S_2)\cdot J + (T_1+T_2)\cdot J$$

The effect of performing this transformation is to reduce the number of scans of both S and T from 4 to 2 and the number of joins that must be executed from 4 to 2.

4. Covering by Complete Merge Graphs

In this section, we show how the problem of transforming a query in disjunctive normal form into a form that minimizes the number of terms can be formulated in terms of covering a **merge graph** with a minimum number of **complete merge graphs**. Each complete merge graph will correspond to a term in the result and vice-versa.

Definition: Given a query in disjunctive normal form, there is a one to one correspondence between the vertices of the **merge graph** and the disjuncts in the query. An edge of color χ is drawn between two vertices in the merge graph if and only if the two vertices satisfy each of the following three conditions:

Condition 1: The two vertices (disjuncts) have selection clauses on the same set of relations.

Condition 2: The two vertices have the same set of join clauses.

Condition 3: The two vertices differ in the selection clauses of exactly one relation, namely relation χ. □

A complete merge graph is defined in terms of a cartesian product graph that we will define first.

Definition: A **cartesian product** graph $G = G_1 \times G_2$ is defined as follows: the vertex set $V(G)$ is $V(G_1) \times V(G_2)$ and $(x_1, x_2) -- (y_1, y_2)$ exists if and only if $x_2 = y_2$ and $x_1 -- y_1$ is an edge in G_1, or $x_1 = y_1$ and $x_2 -- y_2$ is an edge in G_2. □

Thus, if $G = G_1 \times G_2 \times \ldots \times G_m$, each pair $(v_1, \ldots, v_m)$, $(w_1, \ldots, w_m)$ of adjacent nodes in G differs on exactly one

[2]Following standard boolean notation, we use '+' to denote the boolean OR while '·' is used to denote the boolean AND.

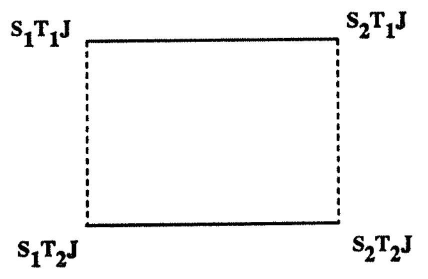
coordinate.

Definition: A complete merge graph $G = G_1 \times G_2 \times \ldots \times G_m$ is a cartesian product graph where each G_i is a clique. □

In drawing complete merge graphs, we will adopt the following convention: An edge in a complete merge graph between two adjacent nodes, $(v_1, \ldots, v_m)$ and $(w_1, \ldots, w_m)$, has color i where $v_i \neq w_i$ and is drawn parallel to the ith coordinate axis.

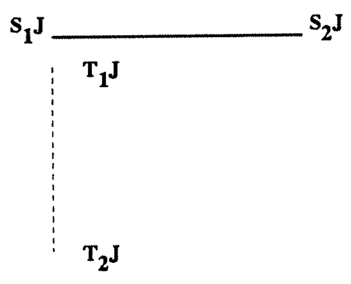
Figure 1 illustrates the merge graphs for the two examples described in Section 3. By definition, the disjuncts correspond to the vertices in the merge graphs. The dotted edges connect vertices that differ in selection clauses on relation T only, while the solid edges connect vertices that differ in selection clauses of the S relation only. The merge graph of Example 1 is also a complete merge graph, while the merge graph of Example 2 consists of two complete merge graphs.

Figure 2 shows some examples of complete merge graphs and the corresponding terms. Figure 2(A) shows a one-dimensional complete merge graph for the query given by $S_1 + S_2 + S_3 + S_4$. Figure 2(B) shows a two-dimensional complete merge graph for the query given by $S_1 \cdot T_1 + S_1 \cdot T_2 + S_2 \cdot T_1 + S_2 \cdot T_2 + S_3 \cdot T_1 + S_3 \cdot T_2$. Similarly, Figure 2(C) shows a three-dimensional complete merge graph for the query with twelve disjuncts. The figures show that the selection clauses of the various relations serve as coordinates in drawing merge graphs. Since all the vertices in a connected component of a merge graph have the same set of join clauses, we choose to drop them from



Merge Graph for Example 1.

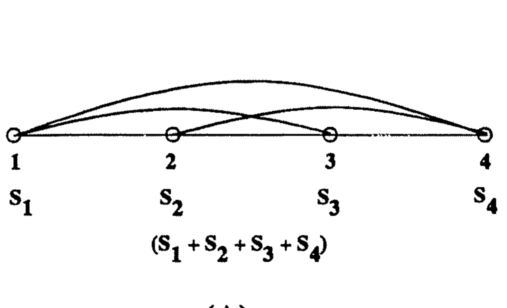
(Single Component)



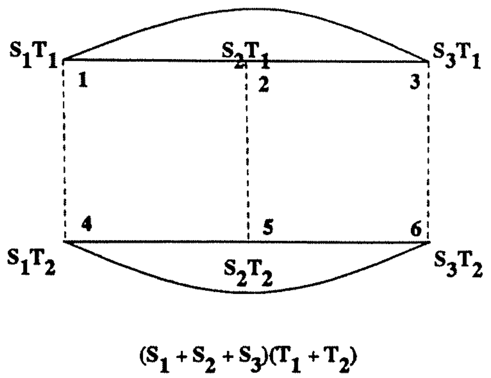
Merge Graph for Example 2.

(Two Components)

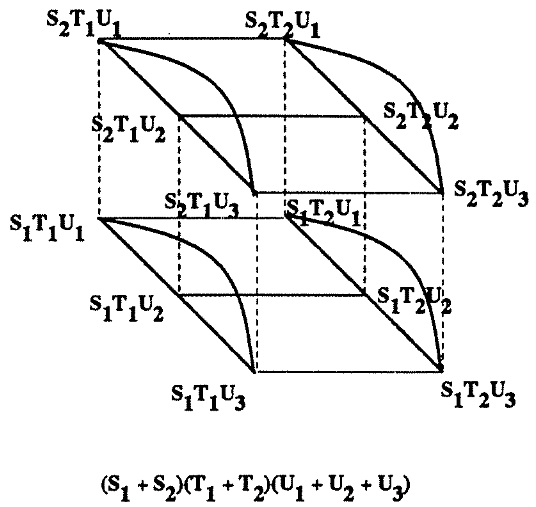
Figure 1



(A)



(B)



(C)

Figure 2

the disjuncts whenever it is convenient to do so. Clearly there is a 1-1 correspondence between a term and a complete merge graph. Given a term $\prod_{i=1}^{n} P_i$, the corresponding complete merge graph is the cartesian product of the complete graphs G_i, i = 1, n. If $P_i = p_1 + p_2 + ... + p_{|P_i|}$, the vertex set of the corresponding G_i is $(p_1, p_2, ..., p_{|P_i|})$. Similarly, given a complete merge graph, we can obtain the corresponding term uniquely. For example, given the complete merge graph $(S_1, S_2, S_3) \times (T_1, T_2)$ (shown in Figure 2(B)), the corresponding unique term is $(S_1 + S_2 + S_3) \cdot (T_1 + T_2)$.

Thus, given an arbitrary set of disjuncts, the problem of minimizing the number of terms is equivalent to covering the vertices of the corresponding merge graph with a minimum number of complete merge graphs. A vertex is said to be covered if it is part of at least one complete merge graph in the cover. A cover that consists of a minimum number of complete merge graphs is called a **minimum cover**.

5. Covering by the Minimum Number of Complete Merge Graphs is NP-complete

The problem P_1, of finding a minimum cover of complete merge graphs in a merge graph is NP-complete. In fact, this is true even when the corresponding set of disjuncts involve only two relations. The proof is simple [Pruhs87]. We state problem P_1 formally.

Instance: A two dimensional merge graph $G' = (V', E')$, positive integer $K \leq |V'|$.

Question: Are there $k \leq K$ complete merge graphs that cover G'?

Theorem 1: Problem P_1 is NP-complete.

Proof: It is easy to see that P_1 is in NP, since a nondeterministic algorithm need only guess k complete merge graphs and check in polynomial time that the merge graph is covered by them.
It is known that the problem P_2, of covering a given bipartite graph with the minimum number of complete bipartite subgraphs is NP-complete [Garey79]. We state problem P_2 formally.

Instance: Bipartite graph $G = (X, E, Y)$, $(V = X \cup Y)$, positive integer $K \leq |E|$.

Question: Are there $k \leq K$ subsets $V_1, V_2, ..., V_k$ of V such that each V_i induces a complete bipartite subgraph of G and such that for each edge $x\text{--}y \in E$ there is some V_i that contains both x and y?
We will polynomially reduce an instance of problem P_2 to an instance of problem P_1 as follows:
For every edge x--y in G, associate a disjunct (x', y') in the two dimensional merge graph, G'. Therefore, we have $|E| = |V'|$.

We now claim that G can be covered with $k \leq K$ complete bipartite subgraphs if and only if G' can be covered with k complete merge graphs. By the reduction, every point in the merge graph corresponds to an edge in the bipartite graph and vice-versa. Therefore, with every complete bipartite subgraph, (X_i, E_i, Y_i) in G, we can associate the complete merge graph $X'_i \times Y'_i$ in G' in a 1-1 fashion. □

In the following sections, we will develop polynomial time algorithms for special classes of merge graphs. We present a heuristic for general merge graphs in Section 6.

5.1. A Quadratic Algorithm for Simple Merge Graphs

We begin this section with a couple of definitions.

Definition: A **corner vertex** cv in a merge graph G is a vertex such that all edges incident on cv are part of one complete merge graph H. We say that H is rooted at cv and denote it by Hcv. □

Definition: A merge graph G is said to be **simple** if $V(G) = V(Hcv_1) \cup V(Hcv_2) \cup ... \cup V(Hcv_n)$. □

We will show that we can cover a simple merge graph G_s with a minimum number of complete merge graphs in $O(|V(G_s)|^2)$ time.

Definition: The **dimension** of a vertex v, denoted by dim(v), is equal to the number of distinct colors of the edges incident on v. □

Definition: The **degree_vector** of a vertex v, denoted by deg_vec(v), is given by: deg_vec(v) = $(n_1, n_2, ..., n_k)$ where k = dim(v), n_i is the number of edges with color i incident on v, i = 1, ..., k. □

Finding if a vertex v (dim (v) = k, deg_vec(v) = $(n_1, n_2, ..., n_k)$) is a corner vertex can be done in $O(|V(G_s)|)$ time. The vertices of a complete merge graph, rooted at v, form a rectilinearly oriented, complete k-dimensional grid. The coordinate of each vertex in this grid can be represented as a k-tuple. Each component of this k-tuple is the set of selection clauses on a particular relation. The number of vertices adjacent to v is $(n_1 + n_2 + ... + n_k) < |V(G_s)|$. By inspecting the components of the coordinates of the vertices adjacent[3] to v, we can deduce the coordinates of the remaining vertices that must be present in the merge graph in order for v to be a corner vertex. This can be done in $O((n_1 + n_2 + ... + n_k) < |V(G_s)|)$ time. We now need to check if the remaining $(n_1 * n_2 * ... * n_k)$ vertices are present in the merge graph. Either $(n_1 * n_2 * ... * n_k) \leq |V(G_s)|$ or $(n_1 * n_2 * ... * n_k) > |V(G_s)|$. In the latter case we know that there is no complete merge graph rooted at v. Checking for the existence of a vertex with a given coordinate can, in practice, be done in O(1) time by hashing on the coordinate. Only if all the $(n_1 * n_2 * ... * n_k)$ vertices exist, then is v a corner vertex. For example, in Figure 2(B), if the input vertex was $S_1 \cdot T_1$, we know that its adjacent vertices are $S_2 \cdot T_1$ and $S_3 \cdot T_1$ along the edges with label S and $S_1 \cdot T_2$ along the edge with label T. Thus the S-components of the coordinates of the $S_2 \cdot T_1$ vertex and the $S_3 \cdot T_1$ vertex are S_2 and S_3 respectively. Similarly, the T component of the $S_1 \cdot T_2$ vertex is T_2. Now all we need to do is to check the presence of vertices with labels

[3]The merge graph is stored as adjacency lists.

$S_2 \cdot T_2$ and $S_3 \cdot T_2$.

We now present the $O(|V(G_S)|^2)$ time algorithm for covering a simple merge graph with the minimum number of complete merge graphs.

Input: A simple merge graph G_S.

Output: The (minimum number of) complete merge graphs that cover G_S.

```
Algorithm A
V_I = ∅; /* V_I represents the set of Vertices that have been included in a complete merge graph */
while (corner vertices remain in (V - V_I)) do
begin
    Step 1: Find a corner vertex cv such that cv ∈ (V - V_I).
    Step 2: Find the complete merge graph $H_{cv}$ rooted at cv.  V_I = V_I ∪ V(Hcv).
end while
End Algorithm A
```

For reasons of efficiency, we will always search the vertices of a merge graph for a corner vertex in order of increasing degree. The proof of correctness of Algorithm A is presented in the following theorem.

Theorem 2: At the end of Algorithm A, V_I = V, and Algorithm A produces the minimum cover.

Proof: The proof is simple and is based on the following two observations:

1. Every corner vertex belongs to one and only one complete merge graph.
2. Two corner vertices with different corresponding complete merge graphs cannot simultaneously be part of any one particular complete merge graph.

The first observation follows directly from the definition of a corner vertex. All the edges adjacent to a corner vertex belong to the complete merge graph rooted at that corner vertex. Therefore, every corner vertex can belong to one and only one complete merge graph.

We will prove the second observation by contradiction. Assume that the two different complete merge graphs are Hcv_1 and Hcv_2. Assume that cv_1 and cv_2 are part of a third complete merge graph Hcv_3. By the first observation, cv_1 can belong to only one complete merge graph. This implies that Hcv_1 is identical to Hcv_3. Similarly, Hcv_2 is identical to Hcv_3. Therefore Hcv_1 is the same as Hcv_2.

The number of complete merge graphs in the minimum cover is equal to the number of iterations of the while loop in Algorithm A. This is because at each iteration exactly one complete merge graph is obtained. Hence the theorem. □

C, we have $cv_2 \in V_1 \times V_2 \times ... \times (V_k - W_k) \times V_{k+1} \times ... \times V_n$. Assume that there is more than one edge incident on v. Delete_Edge(e_i) is true for every edge of the form e_i = v--cv_2 as it will be adjacent to edge v--v_1, $v_1 \in V(Hcv_2)$ such that color(v--cv_2) = color(v--v_1) = k. If $v_1 \notin V(Hcv_2)$, then edge v--cv_2 and edge v--v_1 cannot be edges of the same chordless four cycle as this will contradict the fact that cv_2 is a corner vertex. Thus, all vertices in $V_1 \times V_2 \times ... \times (V_k - W_k) \times V_{k+1} \times ... \times V_n$ that were corner vertices before will remain corner vertices. The complete merge graph, $H'cv_2$, rooted at cv_2 is given by $V_1 \times V_2 \times ... \times (V_k - W_k) \times V_{k+1} \times ... \times V_n$. Clearly, $V(Hcv_1) \cup V(Hcv_2) = V(Hcv_1) \cup V(H'cv_2)$.

Case 2: Let T = {k, m, ...}. $W_i \subset V_i, i \in T; W_j = V_j, j \notin T$.

In other words, more than one W is a proper subset of the corresponding V. Consider v ∈ C. There exist edges e_k, $e_m \notin E(Hcv_1)$ such that e_k = v--v_k, $v_k \in V_1 \times V_2 \times ... \times (V_k - W_k) \times V_{k+1} \times ... \times V_n$. and e_m = v--v_m, $v_m \in V_1 \times V_2 \times ... \times (V_m - W_m) \times V_{m+1} \times ... \times V_n$. Clearly, edges e_k and e_m are the adjacent edges of a chordless four cycle as they belong to the complete merge graph, rooted at cv_2. In this case, none of the vertices in Hcv_2 will be deleted and cv_2 will still be a corner vertex. The complete merge graph, $H'cv_2$, is the same as Hcv_2. Again, $V(Hcv_1) \cup V(Hcv_2) = V(Hcv_1) \cup V(H'cv_2)$. □

The above theorem applies to any pair of 'intersecting' complete merge graphs that are in turn embedded in a larger merge graph. We now present an augmented version of Algorithm A which we will call Algorithm B. Algorithm B removes qualifying edges and vertices.

Input: A merge graph G.

Output: A set of complete merge graphs that cover a subset of V(G).

```
Algorithm B
V_I = ∅; Orig_vertex_set = V;
while(corner vertices remain in (V - V_I))do
begin
    Step 1: Find a corner vertex $cv_1$, $cv_1 \in$ (V - V_I). /* If dim(v) = 1, v is a corner vertex by definition. */
    Step 2: Find the complete merge graph $Hcv_1$ rooted at $cv_1$. V_I = V_I ∪ V($Hcv_1$).
    Step 3: Delete edge e = v--u if Delete_Edge(e) is true, v ∈ V($Hcv_1$).
    Step 4: Delete v ∈ V($Hcv_1$) if Delete_Vertex(v) is true[5].
    Step 5: Undelete all edges $v_a$--$v_b$ ($v_a$, $v_b \in V(Hcv_1)$) such that Undelete_Edge ($v_a$, $v_b$) is true.
end while
End Algorithm B
```

[5]It is possible that G may have broken up into more than one component after step 4. This can happen if more than one vertex of Hcv_1 is an articulation point of G. If G has more than one component, each component must be dealt with individually.

Step 5 has been introduced strictly for reasons of correctness. We will elaborate on this point at the end of the section.

If, at the end of the algorithm, V_I = Orig_vertex_set, by Theorems 2 and 3, we know that the algorithm has produced the optimal number of terms.

A note on the complexity of Algorithm B. Steps 1 and 2 are identical in algorithms A and B. Step 3, the step with the greatest time complexity, determines if an edge is part of a chordless four cycle. Checking if two adjacent edges are part of a chordless four cycle can be done in O(1) time (by hashing) by checking for the presence of the fourth vertex. A given edge is adjacent to at most $|E(G)| - 1$ edges of the merge graph. Thus, the time complexity of Step 3 in Algorithm B is $O(|E(G)|^2)$. The number of times the while loop is executed is at most $O(|V(G)|)$. Therefore, the time complexity of Algorithm B is $O(|V(G)| * |E(G)|^2)$.

We will conclude this section with an example that illustrates the operation of Algorithm B.

Example: Consider the merge graph shown in Figure 4(A). Each vertex in the merge graph represents the disjunct that is a product of its coordinates. For example, vertex 1 denotes the disjunct $S_1T_4U_1$ while vertex 12 denotes $S_2T_1U_2$. There are a total of four corner vertices in the merge graph. They are {1, 10, 11, 12}. Assuming $cv_1 = 10$, we can see that the set of vertices in the complete merge graph H_{10} is = {4, 5, 7, 8, 9, 10}. The vertices in $V(H_{10})$ can be merged into the single term $(S_2+S_3)\cdot(T_1+T_2+T_3)\cdot U_1$. Notice that H_{10} intersects with H_{11} as described in case 1 (in the proof) while H_{10} intersects with H_1 as described in case 2. The set of edges adjacent to vertices in $V(H_{10})$ is E_1 = {4--2, 4--3, 4--5, 4--7, 4--9, 4--11, 5--3, 5--8, 5--10, 7--2, 7--6, 7--8, 7--9, 8--6, 8--10, 9--2, 9--10, 9--12}. Figure 4(B) shows the merge graph after the edges in $E(H_{10})$ have been deleted. We calculate dim'(v) for $v \in V(H_{10})$. We find dim'(10) = 0; dim'(5) = dim'(8) = 1; dim'(9) = dim'(7) = 2; dim'(4) = 3. We remove vertices 5, 8, 10 and the incident edges 3--5, 6--8. The set of edges adjacent to vertices in $V(H_{10})$ is now E_1 = {4--2, 4--3, 4--11, 7--2, 7--6, 9--2, 9--12}. Of these, edges 4--11, 9--12, and 9--2 can be removed as they are not part of any chordless four cycle. Since vertices 4 and 7 are not removed, we put back edge 4--7. Figure 4(C) shows the 2 component merge graph at this stage. Notice that vertices 1, 11, and 12 are still corner vertices.

In the next two iterations, H_1 and H_{11} will be found. The algorithm terminates after the third iteration. □

We pointed out earlier that Step 5 was introduced strictly for reasons of correctness. Step 5 caused edge 4--7 to be put back into the merge graph in the above example. By definition of a complete merge graph, H_1 would not be a complete merge graph without the edge 4--7. However, as described in section 3.6, we would still have been

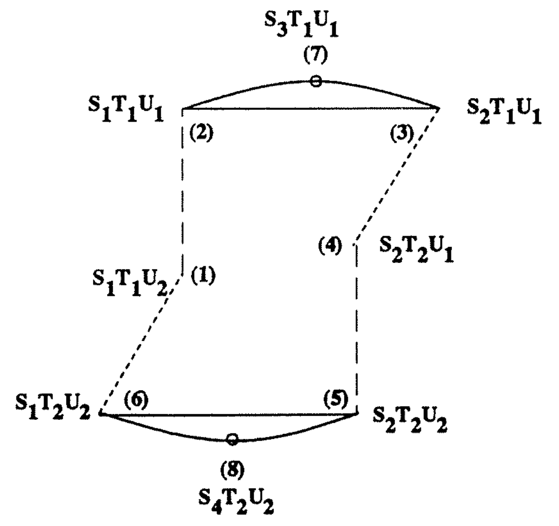
5.2. An Improved Algorithm for a Larger Class of Merge Graphs

Algorithm A assumes that a corner vertex would be found at every iteration and works optimally only for simple merge graphs. Figure 3(A) shows a merge graph G that is not simple. Vertices 7 and 8 are the only corner vertices[4] in G and $V(H_7) \cup V(H_8) \neq V(G)$. After finding corner vertex 7, if we had removed the vertices of the corresponding complete merge graph and edges adjacent to them, both vertices 1 and 4 would have become corner vertices. However, in general, after finding a corner vertex, we cannot remove all the vertices in the corresponding complete merge graph along with the adjacent edges without affecting the optimality of the result. The merge graph in Figure 3(B) can be reduced to two terms, viz., $(S_1 + S_2)\cdot(T_1 + T_2)\cdot U_1$ and $S_2\cdot T_2\cdot U_2$. On the other hand, if vertex 5 is identified first as a corner vertex and we remove vertices 3 and 5, we would finally get three result terms, viz., $S_2\cdot T_2\cdot(U_1 + U_2)$, $(S_1 + S_2)\cdot T_1\cdot U_1$, and $S_1\cdot T_2\cdot U_1$. Clearly, that would not be optimal.

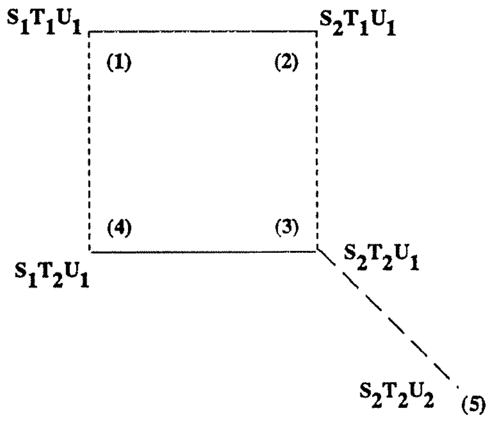
Let Hcv_1 be a complete merge graph that was identified in some iteration of Algorithm A. Before proceeding to the next iteration, we would like to:

(1) Remove certain qualifying edges in E_1, $E_1 = \{e = v\text{--}u \mid v \in V(Hcv_1)\}$, and

(2) Remove certain qualifying vertices from $V(Hcv_1)$.



(A)



(B)

Figure 3

[4]Vertex 4 would have been a corner vertex if the disjunct $S_2\cdot T_1\cdot U_2$ was in the query. Similarly, vertex 3 would have been a corner vertex if the disjuncts $S_1\cdot T_2\cdot U_1$ and $S_3\cdot T_2\cdot U_1$ were in the query.

Therefore, we need some criteria that must be met by the qualifying edges and vertices before they can be deleted. The criteria must ensure that the optimality of the result is not affected after qualifying edges and vertices are deleted. We must ensure that every vertex that was a corner before isolating Hcv_1 remains a corner vertex after removing the qualifying edges and vertices. We will first present the criteria and then an algorithm that we will call Algorithm B. The class of merge graphs for which Algorithm B will work optimally is the class of merge graphs in which a corner vertex will be found at every iteration after qualifying edges and vertices are removed. This class includes the class of simple merge graphs.

Let Hcv_1 be a complete merge graph identified at some iteration. Let $Hcv_2 = V_1 \times V_2 \times \ldots \times V_n$ be some other complete merge graph such that $C = V(Hcv_1) \cap V(Hcv_2)$ is a non-empty set. The subgraph induced by the vertices of C, denoted by G[C], is a complete merge graph. This is because complete merge graphs are cartesian product graphs and the intersection of two or more cartesian product graphs is also a cartesian product graph. $G[C] = W_1 \times W_2 \times \ldots \times W_n$, where $W_i \subseteq V_i$, $i = 1, n$. We define three boolean functions **Delete_Edge(e):** E_1 --> {false, true}, **Delete_Vertex(v):** $V(Hcv_1)$ --> {false, true}, and **Undelete_Edge(v_a, v_b):** $V(Hcv_1) \times V(Hcv_1)$ --> {false, true} which are true only in the following cases:

For $v \in V(Hcv_1)$ and e = v--u,

If $e \in E(Hcv_1)$, Delete_Edge(e) is true.

If $e \notin E(Hcv_1)$, then Delete_Edge(e) is true if and only if there is no edge e' adjacent to v, $e' \notin E(Hcv_1)$, such that e and e' are the adjacent edges of the same chordless four cycle. Clearly the adjacent edges of a chordless four cycle will be of different colors.

Delete_Vertex(v) is true if Delete_Edge(e) is true for every edge e adjacent to v. Clearly, Delete_Vertex(v) is true if $\dim'(v) \le 1$. dim'(v) is calculated using edges that do not belong to $E(Hcv_1)$.

Undelete_Edge(v_a, v_b) is true if Delete_Vertex(v_a) and Delete_Vertex(v_b) are both false, where $v_a, v_b \in V(Hcv_1)$.

Theorem 3: Let Hcv_1 and Hcv_2 be two complete merge graphs as described above. After

(1) removing all edges e such that Delete_Edge(e),
(2) removing all vertices v such that Delete_Vertex(v), and
(3) undeleting (putting back) all edges v_a--v_b such that Undelete_Edge (v_a, v_b)

cv_2 will still be a corner vertex of a complete merge graph $H'cv_2$ and $V(Hcv_1) \cup V(Hcv_2) = V(Hcv_1) \cup V(H'cv_2)$.

Proof: All edges in $E(Hcv_1)$ are first dropped. We divide the proof into two cases:

Case 1: $W_k \subset V_k$ for some k, $1 \le k \le n$; $W_j = V_j$, $j \ne k$, $1 \le j \le n$.

Consider $v \in C$. Every edge adjacent to $v \notin E(Hcv_1)$ as edges in $E(Hcv_1)$ have already been removed. Since $cv_2 \notin$

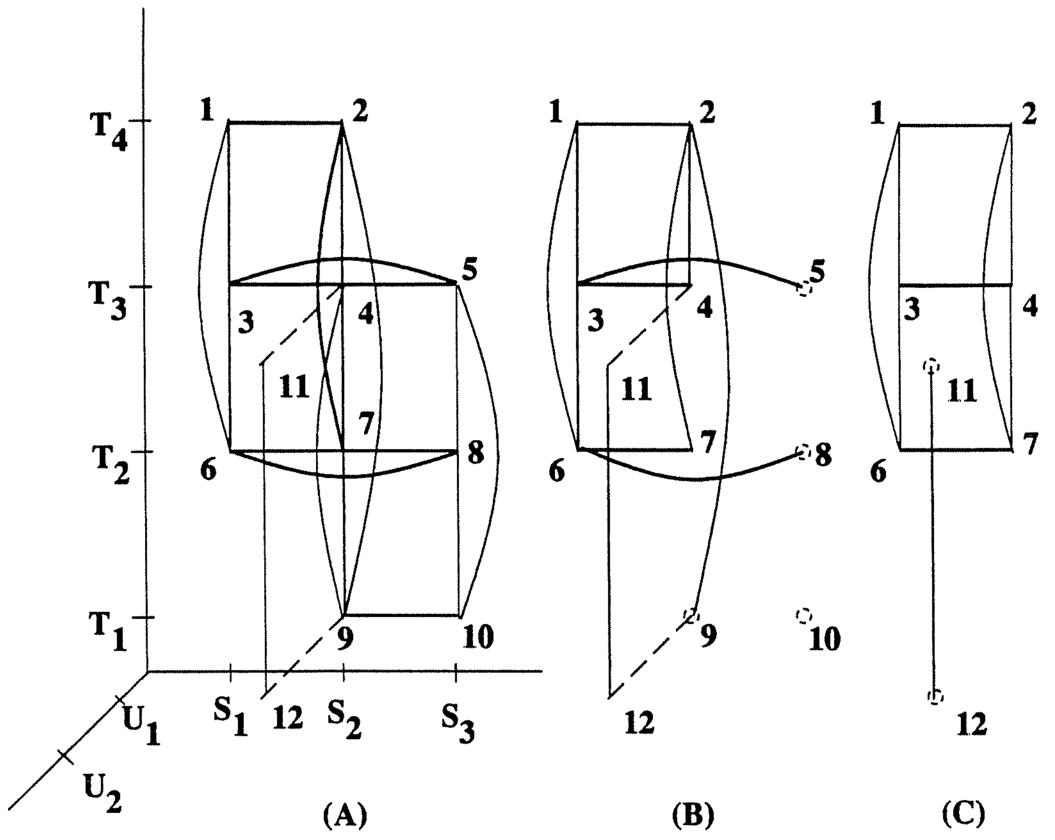


Figure 4

able to recognize that vertex 1 is a corner vertex because the edges adjacent to vertex 1 were not removed. In order to determine if a set of vertices are the vertices of a complete merge graph rooted at a given vertex v, our algorithm requires that only those edges adjacent to v be present along with all the vertices of the complete merge graph. The presence of the edges that are not adjacent to a corner vertex but that belong to the corresponding complete merge graph need not be present for finding all the vertices of the complete merge graph and the associated term.

6. Maximum and Maximal Merge Graphs

Algorithm B assumed that a corner vertex would be found at every iteration. Unfortunately, it is easy to find merge graphs that have no corner vertices. Figure 5 shows an example of a merge graph with no corner vertex that can be covered optimally with six complete merge graphs as shown. In the absence of corner vertices, it seems natural to start by finding a maximum merge graph rooted at some vertex.

Definition: Let E be the edges adjacent to a vertex v. A **maximum (maximal) merge graph** H' = (V',

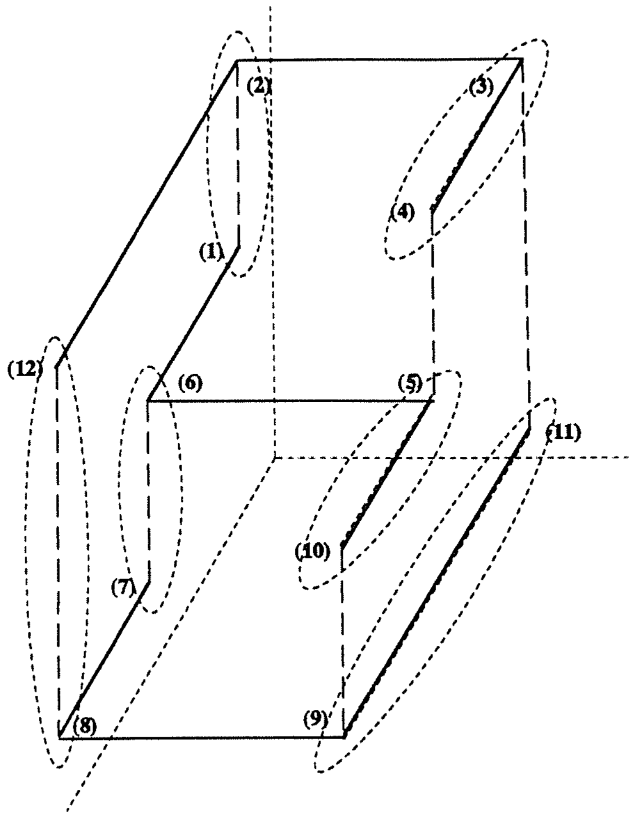


Figure 5

E') rooted at a vertex v, E' $\subseteq$ E, is defined to be a complete merge graph rooted at v such that the set of the vertices, V', of the complete merge graph form a maximum (maximal[6]) set. □

However, finding a maximum merge graph rooted at a vertex is NP-complete. In fact, the problem P_1 of finding a maximum merge graph rooted at any vertex in the (2-dimensional) merge graph is NP-complete. Formally, the problem P_1 may be stated as follows:

Instance: Given a two dimensional merge graph G' = (V', E') and a positive integer $M = K*K \leq |V'|$.

Question: Is there a complete merge graph with $\geq$ M vertices in the given merge graph?

Theorem 4: Problem P_1 is NP-complete.

Proof: Clearly P_1 is in NP, as a nondeterministic algorithm need only pick $m \geq M$ vertices and verify in polynomial time that these m vertices form a complete merge graph. The rest of the proof that P_1 is NP-complete

[6]The set of vertices in V' form a maximal set if no more vertices in the merge graph can be added to V' to find another complete merge graph. A maximum set is the largest maximal set.

involves two reductions and is presented below. We start with the following NP-complete problem in [Garey79].

Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.

Question: Does G contain a clique of size K or more, i.e., a subset $V' \subseteq V$ with $|V'| \geq K$ such that every two vertices in V' are joined by an edge in E?

We polynomially reduce G to a bipartite graph $B(V_1, E_B, V_2)$ as follows:

(1) For every vertex i in V, introduce two vertices $i_1 \in V_1$ and $i_2 \in V_2$, and the corresponding edge $i_1\text{--}i_2 \in E_B$.

(2) For every edge $i\text{--}j \in E$, add edges $i_1\text{--}j_2$, $i_2\text{--}j_1 \in E_B$.

We now claim that G has a clique of size $\geq K$ if and only if B has a complete bipartite subgraph of order $\geq K$ induced by the vertex partitions $V'_1 \subseteq V_1$ and $V'_2 \subseteq V_2$ where $|V'_1| = |V'_2| \geq K$. By the reduction, a clique of size k in G results in a complete bipartite subgraph of order k in B. This is because, there is a one to one correspondence between a clique in G and a complete bipartite graph in B.

The other polynomial reduction consists of reducing an instance of a bipartite graph B to a merge graph G' (as shown in Section 5). By this reduction we have $|E_B| = |V'|$. Therefore, B has a complete bipartite subgraph of order $\geq K$ if and only if the merge graph has in it a complete merge graph with $\geq K{*}K$ number of vertices. This is because there is a 1-1 correspondence between an edge in B and a vertex in G'. □

Unfortunately, finding a maximum merge graph rooted at a vertex does not guarantee an optimal solution. As an example consider the merge graph in three dimensions with 12 vertices shown in Figure 5. The optimal solution consists of a cover with six complete merge graphs as shown. There are no chordless four cycles in this merge graph. Vertex 2 has three edges incident on it, each of which is a maximum merge graph. If we start with edge 2--3 as the first complete merge graph, Algorithm B will yield a cover that consists of seven complete merge graphs. However, if we start either with edge 2--12 or 2--1, Algorithm B will yield an optimal cover.

Since the problem of finding a maximum merge graph is NP-complete, and does not necessarily lead to an optimal solution, we propose the following heuristic:

> In the absence of corner vertices in step 1 of Algorithm B, choose a maximal merge graph rooted at a vertex of minimum dimension. If two or more vertices have the minimum dimension, find a maximal merge graph rooted at a vertex with the smallest degree.

The complexity of finding a maximal merge graph, rooted at a vertex v (in a merge graph G), with degree_vector(v) = $(n_1, n_2, \ldots, n_k)$ is $O((n_1 * n_2 * \ldots * n_k))$. At the present time, we have not been able to derive any analytical bound on the performance of the heuristic. However, it is very likely that after a maximal merge graph has been found, and the qualifying vertices and edges in this maximal merge graph have been removed, there will be corner vertices in the subsequent iterations. On small examples, we have found that the heuristic gives a solution that is close to the optimal solution. As discussed above, the heuristic performs optimally for the merge graph in Figure 5. It seems difficult to be able to derive quantitatively how well the heuristic performs in general. The proof of NP-completeness for the problem of finding a minimum cover (in 2-dimensions) shows that the problem is equivalent to covering the edges of an arbitrary bipartite graph with complete bipartite graphs. There seem to be no good approximation algorithms in literature for the latter problem. Empirical methods for judging the quality of the heuristic also seem infeasible. Consider for example that we generate all possible merge graphs in three dimensions, with four vertices in each direction. The number of such complete merge graphs is equal to $2^{4*4*4} = 2^{64}$.

7. Acknowledgements

The authors would like to acknowledge the many comments by Eric Bach and Yannis Iaonnidis that have considerably improved the presentation. We also thank Giri Narasimhan for many fruitful discussions at various stages of the writing of this paper.

8. References

[Aho79] A.V. Aho, Y. Sagiv, and J.D. Ullman, "Equivalence of Relational Expressions," *SIAM Journal of Computing* 8(2) pp. 218-246 (1979).

[Bernstein81] P.A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J.B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems* 6(4) (Dec. 1981).

[Chandra77] A.K. Chandra and P.M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Data Bases," *Proc. of the 9th Annual ACM symposium on Theory of Computation*, pp. 77-90 (May 1977).

[Jarke84] M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys* 16(2) pp. 111-152 (June 1984).

[Kerschberg82] L. Kerschberg, P.D. Ting, and S.B. Yao, "Query optimization in a star computer network," *ACM Trans. on Database Systems* 7 pp. 678-711 (Dec. 1982).

[Muralikrishna88] M. Muralikrishna, "Optimization of Multiple Disjunct Queries in a Relational Database System," *Ph.D. Thesis*, University of Wisconsin, (Jan 1988).

[Pruhs87] Kirk Pruhs, "Finding a Minimum Cover of Complete Merge Graphs in a Merge Graph is NP-complete," *Private Communication*, University of Wisconsin, (Feb. 1987).

[Selinger79] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. price, "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD Conf.*, (June 1979).

[Smith75] J.M. Smith and P.Y.T. Chang, "Optimizing the performance of a relational database interface," *Comm. of the ACM* 18(10) pp. 568-579 (Oct. 1975).

[Wong76] E. Wong and K. Youssefi, "Decomposition - A Strategy for Query Processing," *ACM Trans. on Database Systems* 1(3) pp. 223-241 (Sept. 1976).

[Yao79] S.B. Yao, "Optimization of query evaluation algorithms," *ACM Trans. on Database Systems* 4(2) pp. 133-155 (June 1979).

[Youssefi79] K. Youssefi and E. Wong, "Query processing in a relational database system," *Proc. of the 5th International Conf. on Very Large Data Bases*, pp. 409-417 (Oct. 1979).