Avoiding Cartesian Products in Programs for Multiple Joins

(Extended Abstract)

Shinichi Morishita *

Stanford University and IBM Research†

Abstract

Avoiding Cartesian products is a common heuristic to reduce the search space of join expressions (orderings) over some set of relations. However, this heuristic cannot guarantee optimal join expressions in its search space because the cheapest Cartesian-product-free (CPF, for short) join expression could be significantly worse than an optimal non-CPF join expression. In a recent PODS, Tay [9] gave some conditions on actual relations that ensure the existence of an optimal CPF join expression; however, the conditions turn out to be applicable only in special cases. In this paper, we do not put any restrictions on actual relations, and we introduce a novel technique that derives *programs* consisting of joins, semijoins, and projections from CPF join expressions. Our main result is that for every join expression, there exists an equivalent CPF join expression from which we can derive a program whose cost is within a constant factor of the cost of an optimal join expression.

*Department of Computer Science, Stanford University, Stanford, CA 94305, Email: morisita@cs.stanford.edu

†The author belongs to Almaden Research Center and Tokyo Research Laboratory

11th Principles of Database Systems/6/92/San Diego, CA
© 1992 ACM 0-89791-520-8/92/0006/0368...$1.50

1 Introduction

Computing the natural join of a set of relations plays an important role in relational and deductive database systems. In general, to solve this problem efficiently we need to find a way to reduce the number of intermediate tuples to compute the final result.

If the scheme of these relations is acyclic (tree), the cost of taking the join is polynomial in the size of the input relations and the output (See [11], Chapter 11). The method first applies to the relations a full reducer [2], a sequence of semijoins, which makes the relations globally consistent by eliminating dangling tuples. Then it takes the join by using a monotone join expression [1], which guarantees that no intermediate join has more tuples than the final join. Yannakakis [13] extended this idea to compute a project-join expression with an acyclic database scheme. For cyclic schemes the problem is NP-complete and those methods above cannot be generalized. Many attempts have been made to reduce the cyclic problem to the acyclic problem. For example, it was shown [3, 4] that if we have a program for solving the join by applying joins, semijoins, and projections, the program creates an embedded acyclic database scheme consisting of the input relations, the result relation, and subsets of the new relations generated by the program. Then once the embedded acyclic database scheme is found, it can be solved polynomially. However, there remains the question of how to make an efficient program to compute the join at the beginning.

Another approach to the problem is to reduce the search space of join expressions (orderings of joins), which can be exponential in the number of relations, and to pick an optimal join expression. An optimal join expression is one that has the least number of intermediate tuples generated among all join expressions. The cost of a join expression is the sum of the tuples appearing in the input relations or the relations generated during the evaluation of the join expression. One heuristic commonly used in the evaluation of joins is to use linear join expressions, which have the form $(\ldots(R_1 \bowtie R_2) \bowtie \ldots) \bowtie R_n$. This heuristic is of practical interest because it allows us to keep only one temporary relation at any time. Another heuristic is to avoid Cartesian products because Cartesian products tend to be expensive.

Smith and Genesereth [6] considered linear join expressions (conjunctions, to them), and gave an "adjacency restriction rule" which improves the cost of a join by locally swapping two adjacent relations. Query optimizers in many well-known systems, for example INGRES [12] and System R [5], use one or both heuristics. Swami and Gupta [7, 8] used both heuristics to reduce the search space and compared several statistical techniques, such as iterative improvement and simulated annealing, based on the criterion that a query optimizer is good if it performs well in the average and very rarely performs poorly.

From the theoretical viewpoint, however, restricting the search space by each heuristic may result in losing all optimal expressions from the search space. For Example, the cheapest linear join expression could be much worse than an optimal nonlinear join expression, and the cheapest CPF join expression could be significantly more expensive than an optimal non-CPF join expression. We will give such an example in Section 2. In a recent PODS, Tay [9] gave some conditions on actual data ensuring that there exists an optimal linear join expression or an optimal CPF join expression. The conditions turned out to be rather restrictive and applicable only in special cases but showed the difficulty of the problem.

Let us relax the problem a little. We define a *quasi-optimal join expression* to be one whose cost is bounded by the cost of an optimal join expression times the size of the database scheme. The size of the database scheme depends on the number of attributes and the number of relation schemes in the database scheme, and therefore it is independent of the size of the actual data (the number of the tuples in the actual relations). In practice the size of the database scheme is much smaller than the size of the actual data, and it can be thought as a constant. Thus the cost of a quasi-optimal join expression is within a constant factor of the cost of an optimal join expression. Now consider the following question;

> Among all CPF join expressions
> does a quasi-optimal one exist?

However, the answer to the question is "No." We will give an counter example in section 2.

To compute the join of the given relations, join expressions have been used so far. Now as facilities to compute the join we employ programs that consist of joins, semijoins, and projections. There could be, however, an infinite number of programs to compute the join of the given relations. In order to reduce the number of programs to consider we present a novel algorithm that derives a program from any CPF join expression, and we will use only those programs derived from CPF join expressions by the algorithm. Observe that the number of programs that we might use is equal to the number of all CPF join expressions. Furthermore, the cost of deriving a program from any CPF join expression is bounded by the size of the given database scheme instead of the size of actual relations.

Now we call a program a *quasi-optimal program* if the number of tuples appearing in the input relations and the intermediate relations generated by the program is bounded by the cost of an optimal join expression times the size of the database scheme. Again, if we consider the size of the database scheme as a constant, the cost of a quasi-optimal program is within a constant factor of the cost of an optimal join expression.

A program derived from any CPF join expression by the algorithm is not necessarily quasi-optimal. However, we can prove the following interesting property.

> Among all CPF join expressions there exists
> one from which a quasi-optimal program can
> be derived.

After giving some terminology in Section 2, we present the algorithm in Section 3.

2 Preliminaries

2.1 Relational Databases

We use the terminology given in [10]. A *relation scheme* is a finite set of *attributes*. We use bold letters, say $\mathbf{R}$, to denote relation schemes. A relation over a relation scheme $\mathbf{R}$ is represened by $R(\mathbf{R})$ or R. A *database scheme* is a multiset of relation schemes. We use calligraphic letters, for example $\mathcal{D}$, to denote database schemes.

Let D be a database over a database scheme $\mathcal{D} = \{\mathbf{R}_1, \ldots, \mathbf{R}_n\}$. Suppose that D assigns R_i to $\mathbf{R}_i$. $\bowtie D$ denotes the join of D, $R_1 \bowtie \ldots \bowtie R_n$. Observe that $\bowtie D$ is a relation over the relation scheme $\cup\mathcal{D}$. Let $\mathcal{D}'$ be a subset of $\mathcal{D}$. $D[\mathcal{D}']$ denotes the restriction of the database D to $\mathcal{D}'$; that is $D[\mathcal{D}']$ assigns to $\mathbf{R} \in \mathcal{D}'$ the same relation as $\mathcal{D}$ does. Note that $\bowtie D[\mathcal{D}']$ is a relation over the relation scheme $\cup\mathcal{D}'$.

We can represent a database scheme $\mathcal{D}$ by a *hypergraph* that has all attributes appearing in $\mathcal{D}$ as its nodes and all relation schemes in $\mathcal{D}$ as its hyperedges. In a hypergraph a *path* from edge $\mathbf{R}_1$ to $\mathbf{R}_k$ is a sequence of k (≥ 1) edges $\mathbf{R}_1, \ldots, \mathbf{R}_k$ such that $\mathbf{R}_i \cap \mathbf{R}_{i+1}$ is nonempty for $1 \leq i < k$. Two edges are *connected* if there is a path from one to the other. A set of edges is *connected* if every pair is connected. A *(connected) component* $\mathcal{C}$ of $\mathcal{D}$ is a connected subset of $\mathcal{D}$ that for any $\mathbf{R} \in \mathcal{C}$ and any $\mathbf{S} \in \mathcal{D} - \mathcal{C}$, $\mathbf{R}$ and $\mathbf{S}$ are not connected.

Example 1: Let $\mathcal{D}$ be the database scheme

$$\{ABC, CDE, EFG, GHA\}.$$

$\mathcal{D}$ is connected. Let D be a database

$$\{R_1, R_2, R_3, R_4\}$$

over $\mathcal{D}$. Suppose that D assigns R_1, R_2, R_3 and R_4 to ABC, CDE, EFG and GHA respectively. $D[\{ABC, CDE\}]$ is a database $\{R_1, R_2\}$ that assigns R_1 and R_2 to ABC and CDE respectively. $\square$

2.2 Join Expressions and Programs

A *join expression* is an expression with relation schemes as operands, join ($\bowtie$) as a binary operator, and parentheses. We call E a join expression *over* a database scheme $\mathcal{D}$ if the set of relation schemes in E is $\mathcal{D}$.

Join $E_1 \bowtie E_2$ is a *Cartesian product* if E_1 and E_2 do not share any attributes. A join expression is *Cartesian-product-free* (or *CPF*, for short) if no join is a Cartesian product. A join expression is non-CPF if there exists a join that is a Cartesian product.

Let D be a database over $\mathcal{D}$. Suppose that D assigns R_i to $\mathbf{R}_i$. $E(D)$ means the evaluation of E after replacing $\mathbf{R}_i$ by R_i. $E(D)$ computes the join of D. For example let E be

$$(\mathbf{R}_1 \bowtie \mathbf{R}_2) \bowtie \mathbf{R}_3.$$

Then $E(D)$ is $(R_1 \bowtie R_2) \bowtie R_3$. To evaluate this expression we first compute $R_1 \bowtie R_2$ and then $(R_1 \bowtie R_2) \bowtie R_3$. In this way join expressions specify the orders of joins to be executed.

E needs to have at least one occurrence of each relation scheme in $\mathcal{D}$ so that $E(D)$ computes the join of D. Furthermore only one occurrence is enough to compute the join. We define a join expression to be *exactly* over a database scheme $\mathcal{D}$ if the join expression has only one occurrence of each relation scheme in $\mathcal{D}$. In order to compute the join of D throughout this paper we only consider join expressions exactly over $\mathcal{D}$.

A *program* P over a database scheme $\mathcal{D} = \{\mathbf{R}_1, \ldots, \mathbf{R}_n\}$ is a finite sequence of statements $s_1 \ldots s_m$. In programs a bold letter is either a relation scheme in $\mathcal{D}$ or a *relation scheme variable* that represents a relation scheme. Each statement in a program is one of the following forms:

Project statement:	$R(\mathbf{R}) := \pi_{\mathbf{U}} R(\mathbf{S})$
Join statement:	$R(\mathbf{R}) := R(\mathbf{S}) \bowtie R(\mathbf{T})$
Semijoin statement:	$R(\mathbf{R}) := R(\mathbf{R}) \ltimes R(\mathbf{S})$

In the project statement we require that $\mathbf{U} \subseteq \mathbf{S}$ and $\mathbf{R}$ is a relation scheme variable. $\mathbf{R}$ in the head of the join statement must be a relation scheme variable. We will call the left side of the assignment operator ($:=$) the *head* and the right side the *body*. Each statement destructively assigns the relation computed in the body to the head. Thus the project statement assigns $\pi_{\mathbf{U}}R(\mathbf{S})$ to the head, the join statement assigns $R(\mathbf{S}) \bowtie R(\mathbf{T})$ to the head, and the semijoin statement assigns $R(\mathbf{R}) \ltimes R(\mathbf{S})$ to the head. The assignment also may assign a relation scheme to a relation scheme variable in the head. Thus in the project statement the value of $\mathbf{R}$ in the head becomes $\mathbf{U}$, and in the join statement the value of $\mathbf{R}$ becomes $\mathbf{S} \cup \mathbf{T}$.

Suppose that $R(\mathbf{V})$ appears in the body of a statement. If $\mathbf{V}$ is a relation scheme variable, $R(\mathbf{V})$ must be *defined earlier* by a join statement or a project statement; that is, it must appear in the head of an earlier statement s_j $(j < i)$ that is either a join statement or a project statement. If $\mathbf{V}$ is a relation scheme in $\mathcal{D}$, $R(\mathbf{V})$ may appear in the head of an earlier semijoin statement or may not appear in the head of any earlier statement.

P does not depend on any particular database over $\mathcal{D}$. Let D be a database over $\mathcal{D}$ that sets $\mathbf{R}_i$ to R_i. P is *applied to* a database D over $\mathcal{D}$ if we assign R_i to each occurrence of $R(\mathbf{R}_i)$ that is in the body of a statement and does not appear in the head of any earlier statement, and then we execute the statements in order. We denote the application of P to D by $P(D)$.

Example 2:

$$(ABC \bowtie EFG) \bowtie (CDE \bowtie GHA)$$

is a nonlinear and non-CPF join expression exactly over $\mathcal{D} = \{ABC, CDE, EFG, GHA\}$. The following sequence of statements is a program over $\mathcal{D}$. When applied to a database D over $\mathcal{D}$, this program computes $\bowtie D$ in the head of the final statement.

$$\begin{aligned} &R(\mathbf{X}) := R(ABC) \bowtie R(EFG) \\ &R(\mathbf{Y}) := R(CDE) \bowtie R(GHA) \\ &R(\mathbf{X}) := R(\mathbf{X}) \bowtie R(\mathbf{Y}) \end{aligned}$$

□

2.3 Cost Model

We consider the amount of computation in executing $E(D)$ or $P(D)$. In order to compute joins, semi-joins and projections there are many methods that take advantage of indices, block sizes and main memory sizes. We, however, simply use as the cost measure the number of tuples that appear in the input relations and the relations generated. This cost measure is reasonable because when this cost is n the cost of the actual best possible method is no more than $O(n \log n)$ [11].

Let $|R|$ denote the number of tuples in a relation R. The *cost* of solving $E(D)$, denoted $\text{cost}(E(D))$, is recursively defined as follows;

1. If E is a relation scheme $\mathbf{R}_i$ and D assigns relation $R(\mathbf{R}_i)$ to $\mathbf{R}_i$, $\text{cost}(E(D)) = |R(\mathbf{R}_i)|$.

2. If E is a join expression $E_1 \bowtie E_2$, $\text{cost}(E(D)) = |E(D)| + \text{cost}(E_1(D)) + \text{cost}(E_2(D))$.

For example, $\text{cost}((R_1 \bowtie R_2) \bowtie R_3) = |R_1| + |R_2| + |R_3| + |R_1 \bowtie R_2| + |(R_1 \bowtie R_2) \bowtie R_3|$.

Let P be a program consisting of m statements over a database scheme $\mathcal{D} = \{\mathbf{R}_1, \ldots, \mathbf{R}_n\}$. When we apply P to a database $D = \{R_1, \ldots, R_n\}$, let R_{n+k} be the relation appearing in the head of the kth statement $(1 \leq k \leq m)$. The *cost* of solving $P(D)$, denoted $\text{cost}(P(D))$, is defined as $\sum_{i=1}^{n+m} |R_i|$.

To compute the join of a database D over $\mathcal{D}$ we use join expressions exactly over the database scheme $\mathcal{D}$ for D. We, however, should keep in mind that a join expression having more than one occurrence of a relation scheme in $\mathcal{D}$ can cost less than any join expression exactly over $\mathcal{D}$. If we allow such join expressions, however, the number of join expressions to consider will explode, thus such cases will not be taken into account.

Example 3: Let D be the following database over $\mathcal{D} = \{ABC, CDE, EFG, GHA\}$.

$R(ABC)$	$R(CDE)$	$R(EFG)$	$R(GHA)$
$(a, 1, a)$	$(a, 1, a)$	$(a, 1, a)$	$(a, 1, b)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(a, 10^{2k}, a)$	$(a, 10^{k}, a)$	$(a, 10^{2k}, a)$	$(a, 10^{3k}, b)$
$(b, 1, b)$	$(b, 1, b)$	$(b, 1, b)$	$(b, 1, a)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(b, 10^{2k}, b)$	$(b, 10^{k}, b)$	$(b, 10^{2k}, b)$	$(b, 10^{3k}, a)$
(c, c, c)	(c, c, c)	(c, c, c)	(c, c, c)

We assume that $k \geq 1$. Observe that D is *locally (pairwise) consistent*; that is, for any pair of relations $R(\mathbf{X})$ and $R(\mathbf{Y})$ in D, $\pi_{\mathbf{X}}(R(\mathbf{X}) \bowtie R(\mathbf{Y}))$ is equal to $R(\mathbf{X})$. But D is not *globally consistent*; that is, it is not true that for any relation $R(\mathbf{X})$ in D $\pi_{\mathbf{X}}(\bowtie D)$ is equal to $R(\mathbf{X})$. Actually $\bowtie D$ has only one tuple. This fact implies that it is useless to apply a semijoin program [11] to this database. Among all join expressions exactly over $\mathcal{D}$

$$E = (ABC \bowtie EFG) \bowtie (CDE \bowtie GHA)$$

is optimal. $\text{cost}(E(D))$ is less than 10^{4k+1}. Note that E is a non-linear and non-CPF join expression. If we apply to D any CPF join expression exactly over $\mathcal{D}$, the cost exceeds $2 \cdot 10^{5k}$. The cost of any linear join expression applied to D also becomes greater than $2 \cdot 10^{5k}$. □

This example shows that the cheapest linear join expression and the cheapest CPF join expression could be much worse than the optimal nonlinear and non-CPF join expression. This is because in the example above we can increase k as much as we want. Thus in general there does not necessarily exist a quasi-optimal CPF join expression.

2.4 Join Expression Trees

In order to explain the algorithms given in the next section clearly we introduce a tree structure that represents a join expression. Given a join expression E exactly over $\mathcal{D}$ we recursively construct the *join expression tree* representing E, denoted T_E, as follows;

1. Each node is a database scheme. The root is $\mathcal{D}$.
2. If E consists of only one relation scheme $\mathbf{R}$, let T_E be $\{\mathbf{R}\}$.
3. If E is $E_1 \bowtie E_2$, let T_{E_1} and T_{E_2} be the join expression trees representing E_1 and E_2 respectively. T_E is the join expression tree that whose root $\mathcal{D}$ has T_{E_1} and T_{E_2} as its left and right subtrees respectively.

One-to-one correspondence exists between join expressions exactly over $\mathcal{D}$ and join expression trees over $\mathcal{D}$.

A join expression tree is called *Cartesian-product-free* (CPF) if it represents a CPF join expression. Every node in a join expression tree is a connected database scheme if and only if the tree is CPF.

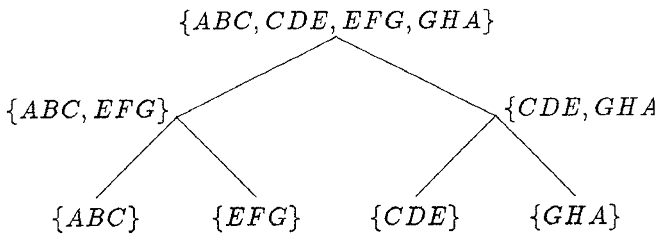

Figure 1: The join expression tree that represents $(ABC \bowtie EFG) \bowtie (CDE \bowtie GHA)$

Let T be a join expression tree over $\mathcal{D}$ and D be a database over $\mathcal{D}$. $T(D)$ means the *application* of T to D. $T(D)$ is obtained by replacing each database scheme $\mathcal{V}$ with $\bowtie D[\mathcal{V}]$. Let $\text{cost}(T(D))$ denote the sum of tuples in all relations in $T(D)$. If T represents a join expression E exactly over $\mathcal{D}$ then $\text{cost}(T(D))$ is equal to $\text{cost}(E(D))$.

Example 4: Figure 1 shows the join expression tree that represents

$$(ABC \bowtie EFG) \bowtie (CDE \bowtie GHA). \quad \square$$

3 Main Results

In this section first we present two algorithms Algorithm 1 creates a CPF join expression tree from any join expression tree over any connected database scheme. Algorithm 2 derives a program from any CPF join expression tree.

Suppose that by using Algorithm 1 and 2 sequentially we derive a program from an arbitrary join expression tree over any connected database scheme. We will prove that for an arbitrary actual database over the database scheme the cost of the program is bounded by the cost of the given join expression tree times the size of the database scheme.

If the given join expression tree is optimal to compute the join of the database, then the program is quasi-optimal. Observe that the program is derived by Algorithm 2 from the CPF join expression tree that is created from the given join expression tree by Algorithm 1. Thus we have;

> Among all CPF join expressions there exists one from which a quasi-optimal program can be derived by Algorithm 2.

Now we give Algorithm 1 and Algorithm 2.

Algorithm 1: Given a join expression tree over a connected database scheme $\mathcal{D}$, output a CPF join expression tree over $\mathcal{D}$ by performing the following method.

We will make a table that has a CPF join expression tree over each component of the database schemes at all nodes. Proceed up the given join expression tree by beginning at the leaves. Since each leaf is the CPF join expression tree over itself, put all leaves into the table. Then visit an internal node, say $\mathcal{U}$, whose children have been visited. Let $\mathcal{L}$ and $\mathcal{R}$ be the left and right children of $\mathcal{U}$ respectively. Since $\mathcal{U}$ is the union of $\mathcal{L}$ and $\mathcal{R}$, each component $\mathcal{C}$ of $\mathcal{U}$ is either a component of $\mathcal{L}$, a component of $\mathcal{R}$, or the union of a set Γ consisting of some components of $\mathcal{L}$ and some components of $\mathcal{R}$. Since both children have been visited, in the former two cases the table must have a CPF join expression tree over $\mathcal{C}$ by the inductive hypothesis. In the last case create a CPF join expression tree over $\mathcal{C}$ by performing the following procedure, and put the tree into the table.

```
1  delete from $\Gamma$ an arbitrary database scheme $\mathcal{X}$
   and let $T$ be the CPF join expression tree
   over $\mathcal{X}$ in the table;
2  while $\Gamma$ is not empty do begin
3      delete from $\Gamma$ such a database scheme $\mathcal{W}$
       that $\mathcal{X} \cup \mathcal{W}$ is connected;
4      make a new node $\mathcal{X} \cup \mathcal{W}$ whose left and
       right subtrees are respectively $T$ and the
       join expression tree over $\mathcal{W}$ in the table;
5      let $T$ be the tree at rooted the new node
       and set $\mathcal{X}$ to $\mathcal{X} \cup \mathcal{W}$;
   end
```

In Step 1 we set $\mathcal{X}$ to a connected database scheme arbitrarily chosen from Γ. In the while-loop we enlarge $\mathcal{X}$ by repeatedly adding a database scheme in Γ to $\mathcal{X}$ and make a CPF join expression tree denoted T over $\mathcal{X}$. In Step 3 we can always find such a database scheme $\mathcal{W}$ that $\mathcal{X} \cup \mathcal{W}$ is connected, otherwise $\cup\Gamma$, that is $\mathcal{C}$, cannot be connected. By induction we see that after Step 5 T becomes a CPF join expression tree over $\mathcal{X}$. When the procedure terminates $\mathcal{X}$ becomes $\cup\Gamma$, which implies that T is a CPF join expression tree over $\cup\Gamma$ $(= \mathcal{C})$.

After the root is processed a CPF join expression tree over $\mathcal{D}$ is put into the table, because $\mathcal{D}$ is connected and is the only one component in $\mathcal{D}$. □

Example 5: Let T_1 be the join expression tree presented in Figure 1. Algorithm 1 produces a CPF join expression tree from T_1 as follows;

- First all leaves in T_1, $\{ABC\}, \{EFG\}, \{CDE\}$ and $\{GHA\}$, are put into the table.
- Then intermediate nodes in T_1, $\{ABC, EFG\}$ and $\{CDE, GHA\}$, are visited. $\{ABC, EFG\}$ has two components, $\{ABC\}$ and $\{EFG\}$, which have been already put in the table. Similarly the two components of $\{CDE, GHA\}$, $\{CDE\}$ and $\{GHA\}$, have also been registered in the table. Thus no new components are found, and therefore no join expression trees are put into the table at this point.
- Finally the root of T_1 is visited. The root $\{ABC, CDE, EFG, GHA\}$ is the unique component of itself, and it is the union of the set $\Gamma = \{\{ABC\}, \{CDE\}, \{EFG\}, \{GHA\}\}$, where $\{ABC\}$ and $\{EFG\}$ are components of the left child of the root and $\{CDE\}$ and $\{GHA\}$ are components of the right child of the root. Subsequently Steps 1~5 are invoked. Steps 1 and 3 may have many choices to select a database scheme from Γ. Because of this non-deterministic nature we can produce 16 different CPF join expression trees. Suppose that at Step 1 we select $\{ABC\}$ from Γ, and at Step 3 we select $\{CDE\}$,$\{EFG\}$, and $\{GHA\}$ in order. Then we have the CPF join expression tree shown in Figure 2. □

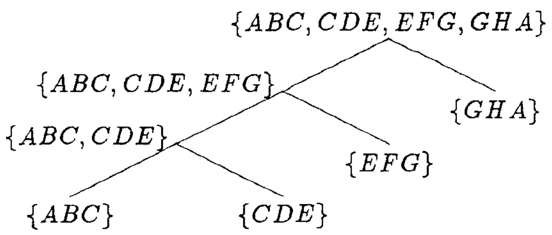

Figure 2: A join expression tree produced from the tree in Figure 1 by Algorithm 1

Algorithm 2: Given a CPF join expression tree over a connected database scheme $\mathcal{D}$, output a program P for computing $\bowtie D$ where D is an arbitrary database over $\mathcal{D}$.

Initialize P to be an empty program. First visit all leaves. When we visit a leaf $\{\mathbf{V}\}$, attach $R(\mathbf{V})$ to the leaf. Next let S be the set of the root and all internal nodes each of which is the right child of its parent. Visit S in some bottom up order; that is, when we visit $\mathcal{V} \in S$ we must have visited all nodes in S that belong to the subtree rooted at $\mathcal{V}$. From $\mathcal{V}$ go down the left branch until a leaf is reached, and let $\mathcal{V}_0$ be the leaf node (See Figure 3). Let $\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_n = \mathcal{V}$ be the sequence of nodes appearing on the branch from $\mathcal{V}_0$ to $\mathcal{V}$. Let $\mathcal{W}_i$ $(1 \leq i \leq n)$ be the right child of $\mathcal{V}_i$. In the procedure below, we append some statements to P, and then we attach $R(\mathbf{V})$ to $\mathcal{V}$. Since $\mathcal{V}_0$ and $\mathcal{W}_i$ have been already visited, they have been attached relation schemes $R(\mathbf{V}_0)$ and $R(\mathbf{W}_i)$ respectively. When we append a statement to P in the procedure below, we compute the value of the relation scheme variable in the head. For example if we append "$R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(\mathbf{W})$" to P, we set $\mathbf{V}$ to $\mathbf{V} \cup \mathbf{W}$.

```
1   create a new relation scheme variable named $\mathbf{V}$
    and set $R(\mathbf{V})$ to $R(\mathbf{V}_0)$;
2   for each $i := 1$ to $n$ do begin
3     let $\mathcal{F}$ be $\{\mathbf{W}_j \mid 1 \leq j < i, \mathbf{W}_j \cap \mathbf{W}_i \not\subseteq \mathbf{V}\}$;
4     if $\mathbf{V} \cap \mathbf{W}_i \neq \phi$ then begin
5       for each $\mathbf{W} \in \mathcal{F}$ do
          append "$R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(\mathbf{W})$" to $P$;
6       append "$R(\mathbf{V}) := R(\mathbf{V}) \ltimes R(\mathbf{W}_i)$" to $P$;
7     end
8     else begin
9       create a new relation scheme variable
        named $\mathbf{F}$;
10      append "$R(\mathbf{F}) := \pi_{(\cup\mathcal{F})\cap\mathbf{V}} R(\mathbf{V})$" to $P$;
11      for each $\mathbf{W} \in \mathcal{F}$ do
          append "$R(\mathbf{F}) := R(\mathbf{F}) \bowtie R(\mathbf{W})$" to $P$;
12      append "$R(\mathbf{F}) := \pi_{(\mathbf{V}\cup\mathbf{W}_i)\cap(\cup\mathcal{F})} R(\mathbf{F})$" to $P$;
13      append "$R(\mathbf{F}) := R(\mathbf{F}) \ltimes R(\mathbf{W}_i)$" to $P$;
14      append "$R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(\mathbf{F})$" to $P$;
15    end
16  end
17  for each $\mathbf{W}_i \not\subseteq \mathbf{V}$ do
      append "$R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(\mathbf{W}_i)$" to $P$;
18  attach $R(\mathbf{V})$ to $\mathcal{V}$;
```

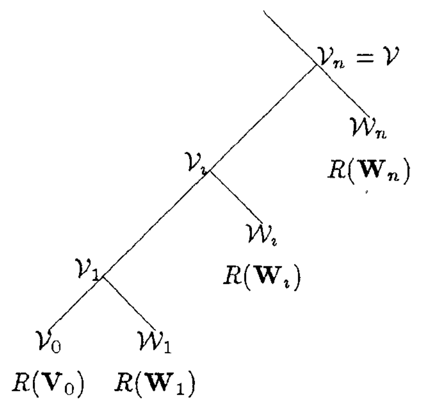

Figure 3:

If P is applied to an actual database D, $R(\mathbf{V})$ will be proved to compute $\bowtie D[\mathcal{V}]$ in the final statement appended by Steps 1~18. Since $\mathcal{V}_0$ and $\mathcal{W}_j (1 \leq j \leq n)$ have been visited before $\mathcal{V}$, $R(\mathbf{V}_0)$ and $R(\mathbf{W}_j)$ are respectively $\bowtie D[\mathcal{V}_0]$ and $\bowtie D[\mathcal{W}_j]$. Thus the join of $R(\mathbf{V}_0), R(\mathbf{W}_1), \ldots, R(\mathbf{W}_n)$ gives $\bowtie D[\mathcal{V}]$, but the statements generated by Steps 1~18 compute $R(\mathbf{V})$ from $R(\mathbf{V}_0), R(\mathbf{W}_1), \ldots, R(\mathbf{W}_n)$ in a rather complicated manner. This complex technique is crucial in order to bound the size of the relation computed in the head of each statement in the following sense. Let T_1 be an arbitrary join expression tree from which the given CPF join expression tree can be created by Algorithm 1. For any statement in P there exists a database scheme $\mathcal{U}$ in T_1 such that the size of the relation in the head of the statement is less than or equal to $|\bowtie D[\mathcal{U}]|$. □

Example 6: Suppose that we apply Algorithm 2 to the CPF join expression tree T_2 in Figure 2. Algorithm 2 first initializes P to be an empty program. Then Algorithm 2 visits all leaves and attaches $R(\mathbf{V})$ to a leaf $\{\mathbf{V}\}$. See Figure 4. Then Algorithm 2 visits the root, because every internal node is the left child of its parent. Subsequently Algorithm 2 performs Steps 1~18. Step 1 creates a new relation scheme variable $\mathbf{V}$ and sets $R(\mathbf{V})$ to $R(ABC)$. Then the outer for-loop is executed three times. When $i = 1$, $\mathbf{W}_1 = CDE$ and $\mathcal{F}$ is empty at Step 3. Since $\mathbf{V} \cap \mathbf{W}_1 \neq \phi$, the procedure goes through Steps 5 and 6. Step 5 does nothing because $\mathcal{F} = \phi$. Step 6 appends

$$R(\mathbf{V}) := R(ABC) \ltimes R(CDE)$$

to P. $\mathbf{V}$ is still ABC. Next when $i = 2$, $\mathbf{W}_2$ is EFG and $\mathcal{F}$ is $\{CDE\}$ in step 3. Since $\mathbf{V} \cap \mathbf{W}_2 = \phi$, the procedure goes through Steps 9~14. Step 9 generates a new relation scheme variable $\mathbf{F}$ and Steps 10~14 append the following statements to P in order.

$$\begin{aligned}
&R(\mathbf{F}) := \pi_C R(\mathbf{V})\\
&R(\mathbf{F}) := R(\mathbf{F}) \bowtie R(CDE)\\
&R(\mathbf{F}) := \pi_{CE} R(\mathbf{F})\\
&R(\mathbf{F}) := R(\mathbf{F}) \ltimes R(EFG)\\
&R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(\mathbf{F})
\end{aligned}$$

$\mathbf{V}$ is $ABCE$ at this point. When $i = 3$, $\mathbf{W}_3 = GHA$. In step 3 $\mathcal{F}$ is $\{EFG\}$. Since $\mathbf{V} \cap \mathbf{W}_3 \neq \phi$, the procedure goes through steps 5 and 6. Steps 5 and 6 append the following statements to P.

$$\begin{aligned}
&R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(EFG)\\
&R(\mathbf{V}) := R(\mathbf{V}) \ltimes R(GHA)
\end{aligned}$$

$\mathbf{V}$ is $ABCEFG$ at this point. The procedure leaves the outer for-loop. Step 17 appends the following two statements for CDE and GHA that are not included in $\mathbf{V}$.

$$\begin{aligned}
&R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(CDE)\\
&R(\mathbf{V}) := R(\mathbf{V}) \bowtie R(GHA)
\end{aligned}$$

Now Algorithm 2 terminates, and we obtain P. If we apply P to the database D given in Example 3, cost($P(D)$) is less than $2 \cdot 10^{4k}$. □

We present the following two main theorems. Their complete proofs will be given in the full version of this paper.

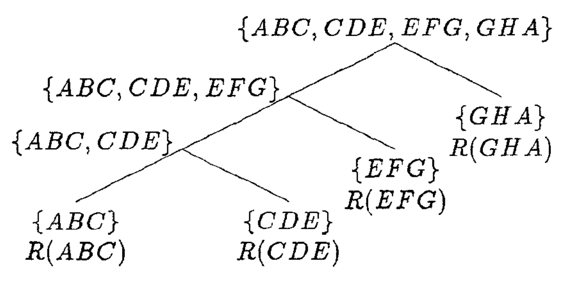

Figure 4: Applying Algorithm 2 to the join expression tree in Figure 2

Theorem 1: Suppose that Algorithm 2 derives a program P from a CPF join expression tree over a connected database $\mathcal{D}$. If we apply P to a database D over $\mathcal{D}$, $P(D)$ computes $\bowtie D$ in the last statement.

Theorem 2: Let T_1 be a join expression tree over a connected database scheme $\mathcal{D}$. Suppose that Algorithm 1 creates a CPF join expression tree T_2 over $\mathcal{D}$ from T_1, and Algorithm 2 derives a program P over $\mathcal{D}$ from T_2. Then for an arbitrary database D over $\mathcal{D}$ such that $\bowtie D \neq \phi$,

$$\text{cost}(P(D)) < r(a+5) \times \text{cost}(T_1(D)),$$

where a is the number of attributes in $\mathcal{D}$ and r is the number of relation schemes in $\mathcal{D}$.

Proof of Theorem 1: (Sketch) We employ the notations used in Algorithm 2 and Figure 3.

Suppose that we apply P to D while P is being generated. Precisely when we visit each leaf $\mathcal{V} = \{\mathbf{V}\}$ and attach $R(\mathbf{V})$ to it, we initialize $R(\mathbf{V})$ to be $D[\{\mathbf{V}\}](=\bowtie D[\mathcal{V}])$. Next when we visit internal nodes and generate statements, we evaluate each statement when it is created.

Suppose that we visit a node $\mathcal{V}$ in S, where S is defined to be the set of the root and all internal nodes each of which is the right child of its parent. Then Steps 1 ~ 18 will append statements to P. Suppose that we have executed all those statements generated while visiting $\mathcal{V}$. We will prove that

$$R(\mathbf{V}) = \bowtie D[\mathcal{V}].$$

This implies that if $\mathcal{V}$ is the root, $\mathcal{V}$ is equal to $\mathcal{D}$, and therefore $R(\mathbf{V})$ becomes $\bowtie D[\mathcal{D}](=\bowtie D)$. We will prove the above claim by an induction on the number of nodes in S visited.

When we visit $\mathcal{V}$, which is displayed in Figure 3, we have already had

$$\begin{aligned}
&R(\mathbf{V}_0) = \bowtie D[\mathcal{V}_0] \text{ and}\\
&R(\mathbf{W}_j) = \bowtie D[\mathcal{W}_j] \text{ for } 1 \leq j \leq n,
\end{aligned}$$

because $\mathcal{V}_0$ is a leaf, and $\mathcal{W}_j$ is a leaf or an internal node that has been already visited.

Let $R(\mathbf{V}_i)$ denote the relation $R(\mathbf{V})$ after the outer for-loop (Steps 2~16) has been executed i-times. We will show the following properties;

1. $R(\mathbf{V}_i) = \pi_{\mathbf{V}_i}(\bowtie D[\mathcal{V}_i])$
2. $\mathbf{W}_h \cap \mathbf{W}_j \subseteq \mathbf{V}_i$ for $1 \leq h < j \leq i$.
3. $\mathbf{V}_{i-1} \subseteq \mathbf{V}_i$ for $1 \leq j \leq n$.

The proof of the above properties 1, 2 and 3 is an induction on i. When $i = 1$ the proof is trivial. When $i > 1$ we can prove the following properties, though we omit the proof.

Throughout Step 5
$R(\mathbf{V}) = \pi_{\mathbf{V}}(\bowtie D[\mathcal{V}_{i-1}]),\ \mathbf{V} \subseteq \mathbf{V}_{i-1} \cup (\cup\mathcal{F})$.
After Step 6
$R(\mathbf{V}_i) = \pi_{\mathbf{V}_i}(\bowtie D[\mathcal{V}_i]),\ \ \mathbf{V}_i = \mathbf{V}_{i-1} \cup (\cup\mathcal{F})$.
After Step 10
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_{i-1}]),\ \ \mathbf{F} = (\cup\mathcal{F}) \cap \mathbf{V}_{i-1}$.
Throughout Step 11
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_{i-1}]),\ \ \mathbf{F} \subseteq \cup\mathcal{F}$.
After Step 12
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_{i-1}]),\ \ \mathbf{F} = (\mathbf{V}_{i-1} \cup \mathbf{W}_i) \cap (\cup\mathcal{F})$.
After Step 13
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_i]),\ \ \ \mathbf{F} = (\mathbf{V}_{i-1} \cup \mathbf{W}_i) \cap (\cup\mathcal{F})$.
After Step 14
$R(\mathbf{V}_i) = \pi_{\mathbf{V}_i}(\bowtie D[\mathcal{V}_i]),\ \ \mathbf{V}_i = \mathbf{V}_{i-1} \cup (\mathbf{W}_i \cap (\cup\mathcal{F}))$.

We can show the following equalities, though we omit the proof.

Throughout Step 17
$R(\mathbf{V}) = \pi_{\mathbf{V}}(\bowtie D[\mathcal{V}]),\ \ \mathbf{V} \subseteq \cup\mathcal{V}$.
After Step 17
$R(\mathbf{V}) = \bowtie D[\mathcal{V}]$,

□

Proof of Theorem 2: In this proof we employ the notations used in Algorithm 1 and 2. We also refer all facts presented in the proof of Theorem 1.

Let $\mathcal{V}$ be the root of T_2 or an arbitrary node that is the right child of its parent. Let $\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_n = \mathcal{V}$ be the sequence of nodes in T_2 defined in Algorithm 2. Since T_2 is created from T_1 by Algorithm 1, observe that the sequence $\mathcal{V}_1, \ldots, \mathcal{V}_n = \mathcal{V}$ is divided into sub-sequences each of which, say $\mathcal{V}_p, \ldots, \mathcal{V}_q$ $(1 \leq p \leq q \leq n)$, is generated by Algorithm 1 when we visit an internal node $\mathcal{U}$ in T_1. Precisely speaking, $\mathcal{U}$ has a component $\mathcal{C}$ that is the union of a set Γ consisting of some components in $\mathcal{L}$ and some components in $\mathcal{R}$, where $\mathcal{L}$ and $\mathcal{R}$

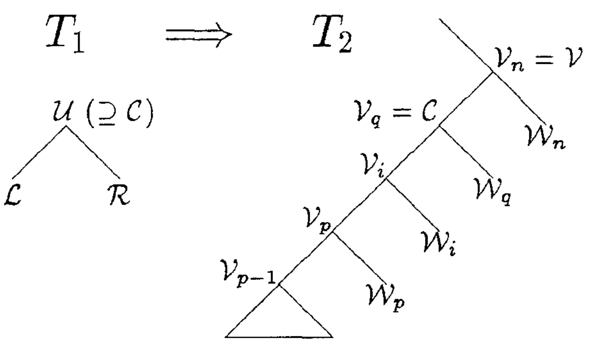

Figure 5: Sequence of nodes $\mathcal{V}_p, \ldots, \mathcal{V}_q$ in T_2 is generated from a component $\mathcal{C}$ of a node $\mathcal{U}$ in T_1 by Algorithm 1

are the left and right children of $\mathcal{U}$ respectively, and the subsequence $\mathcal{V}_p, \ldots, \mathcal{V}_q$ is generated from Γ by Steps 1~5 in Algorithm 1 (see Figure 5). Note that $\mathcal{V}_q$ is equal to $\mathcal{C}$.

We will first prove the following claim.

Claim A: Throughout the outer for-loop (Steps 2~16) is executed for $p \leq i \leq q$, $|R(\mathbf{V})|$ and $|R(\mathbf{F})|$ are bounded by $|\bowtie D[\mathcal{L}]|$ or $|\bowtie D[\mathcal{R}]|$.

Note that the sequence $\mathcal{V}_p, \ldots, \mathcal{V}_q$ is generated from $\Gamma = \{\mathcal{V}_{p-1}, \mathcal{W}_p, \ldots, \mathcal{W}_q\}$ by Algorithm 1. Each element in Γ is either a component of $\mathcal{L}$ or a component of $\mathcal{R}$. In what follows we assume that $\mathcal{V}_{p-1}$ is a component selected from $\mathcal{L}$. The other case that $\mathcal{V}_{p-1}$ is a component of $\mathcal{R}$ can be proved similarly. Note that for $p \leq i \leq q$

$$\mathcal{V}_i = \mathcal{V}_{p-1} \cup \mathcal{W}_p \cup \ldots \cup \mathcal{W}_i.$$

Let $\mathcal{L}_i$ $(\mathcal{R}_i)$ be the union of all elements in $\{\mathcal{V}_{p-1}, \mathcal{W}_p, \ldots, \mathcal{W}_i\}$ that are components of $\mathcal{L}$ $(\mathcal{R})$; that is,

$$\mathcal{L}_i = \mathcal{V}_{p-1} \cup (\cup\{\mathcal{W}_j \mid p \leq j \leq i, \mathcal{W}_j \text{ is a component of } \mathcal{L}\}).$$
$$\mathcal{R}_i = \cup\{\mathcal{W}_j \mid p \leq j \leq i, \mathcal{W}_j \text{ is a component of } \mathcal{R}\}.$$

Since $\mathcal{V}_i$ is equal to $\mathcal{L}_i \cup \mathcal{R}_i$, we have the following equality;

$$\bowtie D[\mathcal{V}_i] = (\bowtie D[\mathcal{L}_i]) \bowtie (\bowtie D[\mathcal{R}_i]).$$

Then we will show the following inequalities;

If $\mathbf{X} \subseteq \cup\mathcal{L}_i$,
$|\pi_{\mathbf{X}} \bowtie D[\mathcal{V}_i]| \leq | \bowtie D[\mathcal{L}_i]| \leq | \bowtie D[\mathcal{L}]|$.
If $\mathbf{X} \subseteq \cup\mathcal{R}_i$,
$|\pi_{\mathbf{X}} \bowtie D[\mathcal{V}_i]| \leq | \bowtie D[\mathcal{R}_i]| \leq | \bowtie D[\mathcal{R}]|$.

We will show the former one. The latter can be proved similarly. $| \pi_{\mathbf{X}} \bowtie D[\mathcal{V}_i] | \leq |\bowtie D[\mathcal{L}_i] |$, because we can easily prove the following general inequality;

$$|\pi_{\mathbf{X}}(R(\mathbf{Y}) \bowtie R(\mathbf{Z}))| \leq |R(\mathbf{Y})| \text{ if } \mathbf{X} \subseteq \mathbf{Y}.$$

Now we will prove $|\bowtie D[\mathcal{L}_i] | \leq |\bowtie D[\mathcal{L}] |$. Let $\{\mathcal{X}_1, \ldots, \mathcal{X}_m\}$ be the set of all components in $\mathcal{L}$. Since $\mathcal{X}_l$ $(1 \leq l \leq m)$ share no common attributes with each other we have

$$| \bowtie D[\mathcal{L}]| = | \bowtie D[\mathcal{X}_1]| \times \ldots \times | \bowtie D[\mathcal{X}_m]|.$$

We have assumed that $\bowtie D \neq \phi$, which implies $| \bowtie D[\mathcal{X}_l]| \geq 1$. Since $\mathcal{L}_i$ is a set of some components of $\mathcal{L}$, say $\{\mathcal{X}_{i_1}, \ldots, \mathcal{X}_{i_k}\}$, $| \bowtie D[\mathcal{L}_i]|$ is equal to $| \bowtie D[\mathcal{X}_{i_1}]| \times \ldots \times | \bowtie D[\mathcal{X}_{i_k}]|$, and therefore we have $| \bowtie D[\mathcal{L}_i]| \leq |\bowtie D[\mathcal{L}] |$.

As we have seen in the proof of Theorem 1, in the i-th execution of the outer for-loop the values of $R(\mathbf{V})$ and $R(\mathbf{F})$ change as follows;

Throughout Step 5
$R(\mathbf{V}) = \pi_{\mathbf{V}}(\bowtie D[\mathcal{V}_{i-1}]), \mathbf{V} \subseteq \mathbf{V}_{i-1} \cup (\cup\mathcal{F})$.
After Step 6
$R(\mathbf{V}_i) = \pi_{\mathbf{V}_i}(\bowtie D[\mathcal{V}_i]),\ \mathbf{V}_i = \mathbf{V}_{i-1} \cup (\cup\mathcal{F})$.
Throughout Steps 10~12
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_{i-1}]),\ \mathbf{F} \subseteq \cup\mathcal{F}$.
After Step 13
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_i]), \quad \mathbf{F} \subseteq \cup\mathcal{F}$.
After Step 14
$R(\mathbf{V}_i) = \pi_{\mathbf{V}_i}(\bowtie D[\mathcal{V}_i]),\ \mathbf{V}_i = \mathbf{V}_{i-1} \cup (\mathbf{W}_i \cap (\cup\mathcal{F}))$.

Recall that $R(\mathbf{V}_i)$ denote the value of $R(\mathbf{V})$ after the outer for-loop is executed i-times. Now we will show the following properties on $\cup\mathcal{F}$;

$\cup\mathcal{F} \subseteq \cup\mathcal{R}_{i-1}$ if $\mathcal{W}_i$ is a component of $\mathcal{L}$, and
$\cup\mathcal{F} \subseteq \cup\mathcal{L}_{i-1}$ if $\mathcal{W}_i$ is a component of $\mathcal{R}$.

Observe that for any $\mathbf{W}_j$ in $\mathcal{F}$ we have $\mathbf{W}_j \cap \mathbf{W}_i \neq \phi$. Since $\mathbf{W}_j = \cup\mathcal{W}_j$ and $\mathbf{W}_i = \mathcal{W}_i$, $\mathcal{W}_j$ and $\mathcal{W}_i$ have some common attributes. If $\mathcal{W}_i$ is a component of $\mathcal{L}$, $\mathcal{W}_j$ is a component of $\mathcal{R}$ and it is not a subset of $\mathcal{V}_{p-1}$, because any two distinct components of $\mathcal{L}$ do not share common attributes. Thus $\mathbf{W}_j = \cup\mathcal{W}_j \subseteq \cup\mathcal{R}_{i-1}$, and therefore $j \leq i-1$, $\cup\mathcal{F} \subseteq \cup\mathcal{R}_{i-1}$. Now we consider the case when $\mathcal{W}_i$ is a component of $\mathcal{R}$. If $j < p$, $\mathcal{W}_j$ is a subset of $\mathcal{V}_{p-1}$. Since $\mathcal{V}_{p-1}$ is a subset of $\mathcal{L}_{i-1}$, $\mathbf{W}_j = \cup\mathcal{W}_j \subseteq \cup\mathcal{V}_{p-1} \subseteq \cup\mathcal{L}_{i-1}$. If $j \geq p$, $\mathcal{W}_j$ must be a component of $\mathcal{L}$, because $\mathcal{W}_j$ and $\mathcal{W}_i$ have some common attributes. Thus $\cup\mathcal{F} \subseteq \cup\mathcal{L}_{i-1}$.

Next we show by an induction on i $(p \leq i \leq q)$ that

$$\mathbf{V}_i \subseteq \cup\mathcal{L}_i.$$

When $i = p$, we have $\mathbf{V}_{p-1} \subseteq \cup\mathcal{V}_{p-1}$ and $\mathcal{V}_{p-1} \subseteq \mathcal{L}_i$. When $i > p$, first suppose that $\mathbf{V}_{i-1} \cap \mathbf{W}_i \neq \phi$, which implies that $(\cup\mathcal{L}_i) \cap (\cup\mathcal{W}_i) \neq \phi$, because $\mathbf{V}_{i-1} \subseteq \cup\mathcal{L}_{i-1}$ by the inductive hypothesis. Since $\mathcal{W}_i$ and $\mathcal{L}_i$ have some common attributes, $\mathcal{W}_i$ must be a component of $\mathcal{R}$. Thus from the properties on $\cup\mathcal{F}$ we have $\cup\mathcal{F} \subseteq \cup\mathcal{L}_{i-1}$. Since $\mathbf{V}_{i-1} \cap \mathbf{W}_i \neq \phi$, we go through Steps 5 and 6, and after Step 6 we have $\mathbf{V}_i = \mathbf{V}_{i-1} \cup (\cup\mathcal{F})$. Therefore $\mathbf{V}_i \subseteq \cup\mathcal{L}_{i-1}$. Now suppose that $\mathbf{V}_{i-1} \cap \mathbf{W}_i = \phi$ at Step 4. Then we go through Steps 10~14, and after Step 14 we have $\mathbf{V}_i = \mathbf{V}_{i-1} \cup (\mathbf{W}_i \cap \cup\mathcal{F})$. By the inductive hypothesis $\mathbf{V}_{i-1} \subseteq \cup\mathcal{L}_{i-1}$. If $\mathcal{W}_i$ is a component of $\mathcal{L}$, recall that $\mathbf{W}_i = \cup\mathcal{W}_i$, and therefore $\mathbf{V}_i \subseteq \cup\mathcal{L}_i$. If $\mathcal{W}_j$ is a component of $\mathcal{R}$, by the properties on $\cup\mathcal{F}$ we have $\cup\mathcal{F} \subseteq \cup\mathcal{L}_{i-1}$, and therefore $\mathbf{V}_i \subseteq \cup\mathcal{L}_i$.

Consequently we have the following facts;

Throughout Step 5
$R(\mathbf{V}) = \pi_{\mathbf{V}}(\bowtie D[\mathcal{V}_{i-1}]),\ \mathbf{V} \subseteq \cup\mathcal{L}_{i-1}$.
After Step 6
$R(\mathbf{V}_i) = \pi_{\mathbf{V}_i}(\bowtie D[\mathcal{V}_i]),\ \mathbf{V}_i \subseteq \cup\mathcal{L}_i$.
Throughout Steps 10~12
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_{i-1}]),\ \mathbf{F} \subseteq \cup\mathcal{L}_{i-1}$ (or $\cup\,\mathcal{R}_{i-1}$).
After Step 13
$R(\mathbf{F}) = \pi_{\mathbf{F}}(\bowtie D[\mathcal{V}_i]), \quad \mathbf{F} \subseteq \cup\mathcal{L}_i$ (or $\cup\,\mathcal{R}_i$).
After Step 14
$R(\mathbf{V}_i) = \pi_{\mathbf{V}_i}(\bowtie D[\mathcal{V}_i]),\ \mathbf{V}_i \subseteq \cup\mathcal{L}_i$.

Since we have already proved that

$|\pi_{\mathbf{X}} \bowtie D[\mathcal{V}_i]| \leq | \bowtie D[\mathcal{L}]|$ if $\mathbf{X} \subseteq \cup\mathcal{L}_i$ and
$|\pi_{\mathbf{X}} \bowtie D[\mathcal{V}_i]| \leq | \bowtie D[\mathcal{R}]|$ if $\mathbf{X} \subseteq \cup\mathcal{R}_i$,

Claim A holds.

Next we will show the following claim.

Claim B: If $\mathcal{V}_q = \mathcal{V}$, throughout Step 17 $|R(\mathbf{V})|$ is bounded by $| \bowtie D[\mathcal{U}]|$.

As we have seen in the proof of Theorem 1, throughout Step 17 in Algorithm 2 we have $R(\mathbf{V}) = \pi_{\mathbf{V}}(\bowtie D[\mathcal{V}])$, where $\mathbf{V} \subseteq \cup\mathcal{V}$. Thus

$$|R(\mathbf{V})| = |\pi_{\mathbf{V}}(\bowtie D[\mathcal{V}])| \leq | \bowtie D[\mathcal{V}]|$$
$$= | \bowtie D[\mathcal{C}]| \leq | \bowtie D[\mathcal{U}]|,$$

because $\mathcal{V} = \mathcal{V}_q = \mathcal{C}$, $\mathcal{C}$ is a component of $\mathcal{U}$ and $\bowtie D \neq \phi$

Finally we will show the following claim.

Claim C: The number of all statements in P is less than $r(a+5)$, where a is the number of attributes in $\mathcal{D}$ and r is the number of relation schemes in $\mathcal{D}$.

We will show that in Algorithm 2 when we visit a node $\mathcal{V}$, the number of statements appended to P is less than or equal to $a + 5n$. Recall that n is the number of the internal nodes on the branch from $\mathcal{V}_0$ to $\mathcal{V}(= \mathcal{V}_n)$. It is clear that the number of statements appended by Steps 6, 10, 12, 13, 14 or 17 is less than or equal to $5n$. We will show that the number of statements appended by Steps 5 or 11 is less than or equal to a. Observe that in each execution of the outer for-loop Steps 5 or 11 append one statement to P for each $\mathbf{W}$ in $\mathcal{F}$. Furthermore by Property 2 presented in Theorem 1 for any $\mathbf{W}$ in $\mathcal{F}$ all attributes in $\mathbf{W} - \mathbf{V}_{i-1}$ are *unique* in $\mathcal{F}$; that is, any attribute in $\mathbf{W} - \mathbf{V}_{i-1}$ does not appear in any other $\mathbf{W}'$ in $\mathcal{F}$. Note that at least one attribute in $\mathbf{W} - \mathbf{V}_{i-1}$ is put into $\mathbf{V}_i$ by Steps 5 or 11, and therefore in the i-th execution of the outer for-loop the number of statements generated by Steps 5 or 11 is less than or equal to the number of attributes in $\mathbf{V}_i - \mathbf{V}_{i-1}$. Thus the number of all statements appended by Steps 5 or 11 while $\mathcal{V}$ is visited is less than or equal to the number of attributes in $\mathbf{V}_n$, and therefore it is also less or equal to a. Consequently the number of all statements generated while $\mathcal{V}$ is visited is less than or equal to $a + 5n$. Observe that the number of all internal nodes in T_2 is equal to $r-1$. Thus the number of all statements in P is less than $a|S|+5r$, where recall that S is the set of the root and all internal nodes in T_2 each of which is the right child of its parent. Therefore the number is also less than $r(a+5)$.

From Claims A, B and C we conclude that $\text{cost}(P(D)) < r(a+5) \times \text{cost}(T_1(D))$. □

4 Discussions

To compute the join of some set of relations we have employed programs that use joins, semijoins, and projections. We have shown that a quasi-optimal program can be derived from a CPF join expression. This is a novel approach to computing multiple joins, and there are some interesting open problems.

Avoiding Cartesian products is one heuristic commonly used to reduce the search space of join expressions. We can further restrict the search space by using linear join expressions. Then the following question naturally arises;

Among linear and CPF join expressions, does there exist a join expression from which we can derive a quasi-optimal program ?

Our method may not be generalized in order to solve the above question, because Algorithm 1 does not necessarily produce a linear join expression tree. Even though we reduce the search space of join expressions by avoiding Cartesian products and using linear join expressions, the size of the space may be exponential in the size of the input database scheme. Thus it is an important open question to find a subspace of join expressions such that its size is polynomial in the size of the database scheme and in it there exists a join expression from which a quasi-optimal program can be derived.

Acknowledgements

I would like to thank Jeff Ullman for giving me many insightful examples and suggestions that motivated me to pursue this work. His comments on earlier drafts of this paper were invaluable.

References

[1] Beeri, C. R. Fagin, D. Maier and M. Yannakakis [1983]. On the desirability of acyclic database schemas: J.ACM, 30(3), pp. 479-513.

[2] Bernstein, P.A. and N. Goodman [1981]. The Power of Natural semijoins: SIAM J. Computing, 10(4), pp. 751-771

[3] Goodman, N. and O. Shmueli [1984]. The tree projection theorem and relational query processing: Journal of Comp. and Sys. Science, 28(1), pp. 60-79.

[4] Sagiv, Y. and O. Shmueli [1986]. The equivalence of solving queries and producing tree projections: Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, pp. 160-172.

[5] Selinger, P.G., M.M. Astrahan, D.D. Chamberlin, P.A. Lorie and T.G. Price [1979]. Access path selection in a relational database system: Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 23-34

[6] Smith, D.E. and M.R. Genesereth [1985]. Ordering conjunctive queries: Artificial Intelligence, 26, pp. 171-215.

[7] Swami, A. and A. Gupta [1988]. Optimization of large join queries: Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 8-17.

[8] Swami, A.[1989]. Optimization of large queries: combining heuristics and combinatorial techniques: Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 367-376.

[9] Tay, Y.C. [1990]. On the optimality of strategies for multiple joins: Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, pp. 124-131. Full version to appear in J.ACM.

[10] Ullman, J.D. [1988] Principles of database and knowledge-base systems: Volume 1, Computer Science Press, New York

[11] Ullman, J.D. [1989] Principles of database and knowledge-base systems: Volume 2, Computer Science Press, New York

[12] Wong, E. and K. Youseffi [1976]. Decomposition – a strategy for query processing, ACM Trans. on Database Systems, 1(3), pp. 223-241

[13] Yannakakis, M. [1981] Algorithms for acyclic database schemes: Proc. Intl. Conf. on Very Large Data Bases. pp. 82-94