Optimization of Join-Type Queries in Nested Relational Databases

Yaxin Li* Hiroyuki Kitagawa† Nobuo Ohbo†

*Doctoral Degree Program in Engineering

†Institute of Information Sciences and Electronics

April 1994

ISE-TR-94-112

Summary

Nested relational models were proposed as natural extensions of the relational model to support new emerging database applications. Prototype implementations of nested relational database systems (NRDBSs) have been done by some research groups. However, there remain many research issues on nested relations. One important issue is query processing, in particular query optimization. In NRDBSs, efficient execution of queries involving hierarchical data structures inherent in nested relations is required. In this paper, we focus on two join-type operations on nested relations: nested join and embed, and propose an algorithm to derive a cost optimal execution sequence of nested joins and embeds for a given query graph. The complexity of the algorithm is proved to be $O(N^2)$, when N nested relations are included in the query graph.

1 Introduction

The relational database technology has had a significant impact on data processing applications. However, it is now commonly recognized that the relational model, with its flat representation of data, is not sufficiently powerful to support new application domains such as engineering design and office automation[10],[17],[26]. The difficulty relates to the recognized semantic mismatch between the entities that are commonly encountered in these application domains and the representations provided by the underlying database management system.

A number of approaches have been taken to remedy this drawback. The nested relational model, which abandons the first normal form assumption in the original relational model, has been studied as an approach to resolve this problem[1]–[7],[9],[12]–[14],[18]–[22],[27]. A variety of algebras have been proposed for nested relations[1]–[3],[7],[9],[12],[19]–[21]. Prototype implementations of nested relational database systems (NRDBSs) were reported by some research groups[2],[4],[5],[19].

However, there remain many research issues on nested relations. One important issue is query processing, in particular query optimization. A query optimizer in the NRDBS translates non-procedural queries into a procedural plan for execution as in the relational database system (RDBS)[23]. It generates a number of candidate plans for the execution, estimates the execution cost of each, and chooses the plan having the lowest estimated cost. Increasing this set of feasible plans improves the chances that it will find a better plan, while increasing query optimization cost. In the study of query optimization in RDBSs, a special attention has been paid on execution of join operations[11],[15],[25]. Since the join is implemented in most systems as a 2-way operator, the optimizer must generate plans that achieve an N-way join as a sequence of 2-way joins. In joining more than a few relations, giving the cost optimal sequence is important, because evaluating the joins in a wrong order could require much processing time and produce an enormous number of intermediate tuples, even if the final result is small.

The same discussion applies to nested relations, and the join query optimization is an important research issue in implementing NRDBSs. Nested relations allow attributes to be relation-valued, which enables direct and concise representation of hierarchical structures, or more precisely trees. Tree structures are inherent in many complex data objects which appear in advanced database applications. This feature of nested relations indeed contributes to elimination of some

join operations which would be required in the decomposed flat data representation in the relational model[22]. However, real world objects have complex structures and relationships. Their structures usually form DAGs and networks rather than simple trees. Therefore, we still have to decompose complex data object structures into trees to get the database schema in the nested relational model. In such cases, joins are required in the query and navigation to restore original data structures and relationships. Joins are also indispensable in processing many ad-hoc queries, which are posted based on a variety of users' viewpoints. Importance of joins and their efficient execution in the NRDBS is also pointed out by Korth and others[6],[14].

In the research on nested relations, a number of variants of join have been proposed. In this paper, we consider two join-type operations: *nested join* and *embed*. The nested join is a straightforward extension of the join in the original relational algebra, and represents basic and standard functionality of joins in many nested relational algebras[3],[6],[12],[21]. The embed was introduced in our nested relational algebra[12],[13], and creates a new nested structure combining two nested relations. Logically, embed can be regarded as a combination of nested join and nest operations. Although creation of new nested structures is essentially important in manipulation of nested relations, optimization of queries including nested joins and nests in general is a tough research issue. However, their specific combination, namely the embed, can be discussed with a slight extension of the framework of study for the nested join. For this reason, we consider embeds as well as nested joins to make our discussion more general.

In this paper, therefore, generating an execution plan is ordering the sequence of nested joins and embeds. We represent a query in a *query graph* and use a *processing tree* to represent a query execution plan. We give an algorithm producing a *linear processing tree (LPT)* for a given query graph. The algorithm derives a cost optimal LPT for the given query under some assumptions. The time complexity of the algorithm is proved to be $O(N^2)$, when N nested relations are included in the query graph. Some researchers have been investigating query processing schemes in advanced database systems[6],[8],[16],[24]. However, no query optimization algorithm has been reported on execution of join-type operations on nested relations to the best of our knowledge.

The remainder of this paper is organized as follows: Section 2 introduces basic concepts and terminology in this paper. It also explains nested join and embed operations discussed in the

paper. Section 3 clarifies our assumptions and presents a cost model for nested join and embed operations. Section 4 presents the optimization algorithm and its sample application. Section 5 shows that our algorithm generates a cost optimal LPT and discusses its time complexity. Section 6 concludes the paper.

2 Basic Concepts and Terminology

2.1 Nested Relations

A *nested relation* is a relation which allows attributes to be *relation-valued*, abandoning the first normal form assumption in the original relational model. A relation-valued attribute brings a nested structure into the flat table structure in the relational model, and this nesting can continue recursively a finite number of times. Fig. 1 shows a nested relation R_1. The schema of R_1 is denoted by $S_0(A, B, S_1(C))$. Attributes S_0 and S_1 are relation-valued, while attributes A, B, and C are atomic.

More formally, let V be the universe of attribute names. Then, the schema NS of a nested relation R is given as a set of *rules* of the form $S_i = (A_{i1}, \cdots, A_{in_i})$, where S_i, $A_{i1}, \cdots, A_{in_i} \in V$ and $A_{ij} \neq A_{ik}$ for $j \neq k$. Attributes whose names appear on the left hand side of some rule are called *relation-valued*, and others are called *atomic*. If a relation-valued attribute S_j appears on the right hand side of the rule $S_i = (A_{i1}, \cdots, A_{in_i})$, S_i is called a *parent* of S_j and S_j is called a *child* of S_i. For a set NS of rules to be qualified as a schema of a nested relation, two conditions must be satisfied. First, for each relation-valued attribute S_i, there can be only one rule in NS where S_i appears on the left hand side. Secondly, relation-valued attributes in NS must form a rooted tree based on the parent-child relationship. Attributes except the root are called *internal*. The terms "*descendant*" and "*ancestor*" are defined in an obvious way. If we follow this more formal definition, the schema R_1 is specified as a set consisting of the two rules: $S_0 = (A, B, S_1)$, $S_1 = (C)$. For notational convenience, we will use the concise linear representation form $S_0(A, B, S_1(C))$.

Each attribute A has a set of potential *instances*, namely attribute values, named the *domain* $dom(A)$. A set of primitive values is associated with each atomic attribute. Given a relation-valued attribute S_i accompanied by the rule $S_i = (A_{i1}, \cdots, A_{in_i})$, the domain of S_i is defined as

follows:

$$dom(S_i) = 2^{dom(A_{i1})\times\ldots\times dom(A_{in_i})}\dagger,$$

where elements in $dom(A_{i1}) \times \ldots \times dom(A_{in_i})$ are called *S_i-tuples* or generally *tuples*. An instance of a nested relation, or simply a nested relation, is formally an instance of its root relation-valued attribute. When a relation-valued attribute S_i is internal, instances of S_i are called *S_i-subrelations* or generally *subrelations*, and tuples in an S_i-subrelation are called *S_i-subtuples* or generally *subtuples*. The number of S_i-(sub)tuples in R is denoted by $\gamma(R, S_i)$. When S_i is the root of R, $\gamma(R, S_i)$ is simply denoted by $\gamma(R)$. In Fig. 1, $\gamma(R_1, S_0) = \gamma(R_1) = 2$ and $\gamma(R_1, S_1) = 5$.

In the remaining part of this paper, we use the term "*relation*" as a synonym of "*nested relation,*" if there is no possibility of confusion.

S 0		
A	B	S 1 C
a1	b1	c1 c2
a2	b2	c3 c4 c5

Figure 1: Nested Relation

2.2 Join-Type Operations

2.2.1 Nested Join[(3),(6),(11),(20)]

The *nested join* is a straightforward extension of the original relational join. In the remaining part of this paper, we simply refer to nested joins as joins, and joins in the original relational algebra are called relational joins. Let us consider the join $R_1 \bowtie[S_k, JC]\ R_2$ of two relations R_1 and R_2. Here, S_k denotes a relation-valued attribute of R_1, called the *target attribute*, and JC denotes the *join condition*. We assume that the join condition may only reference child

†For a set X, 2^X stands for the power set of X.

attributes of S_k in R_1 and child attributes of the root in R_2. We also assume that they are all atomic. The join $R_1 \bowtie[S_k, JC]\ R_2$ combines each S_k-(sub)relation with R_2 in a way similar to the relational join operation based on the join condition. Fig. 2 (a) illustrates the join $R_1 \bowtie[S_1, C = C]\ R_2$. Note that $\gamma(R_1 \bowtie[S_k, JC]\ R_2, S_m) = \gamma(R_1, S_m)$ holds for a relation-valued attribute S_m in $R_1 \bowtie[S_k, JC]\ R_2$, if S_m is not S_k nor its descendant.

In analogy to the original relational algebra, *natural join* is defined to remove the obvious redundancy when the join condition involves only equality conditions. Fig. 2 (b) shows the natural join $R_1 \bowtie[S_1]\ R_2$. When S_k is the root of R_1, the natural join $R_1 \bowtie[S_k]\ R_2$ is simply denoted by $R_1 \bowtie R_2$.

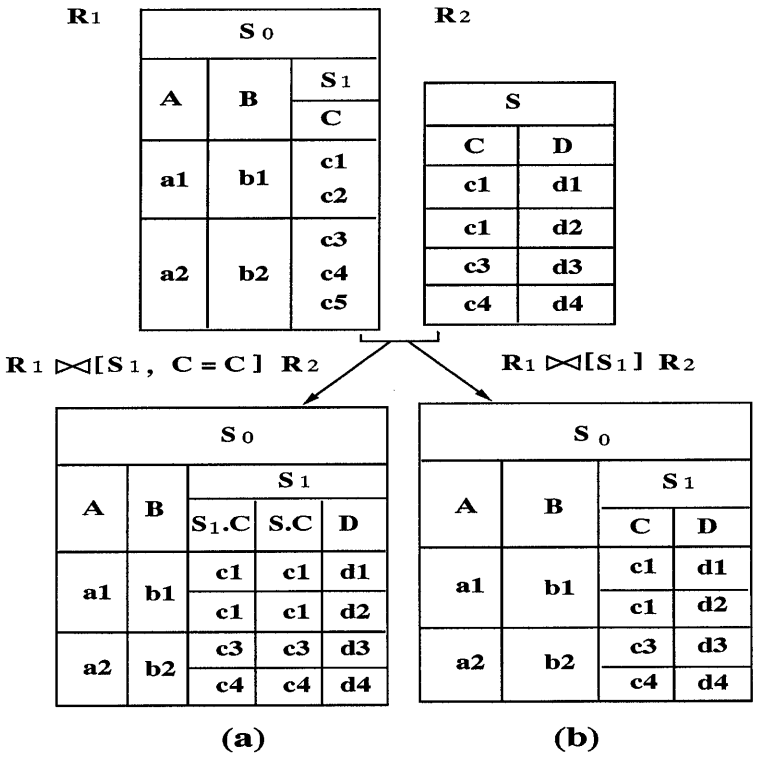

Figure 2: Join and Natural Join

As is the case with the relational join, there are a number of strategies to execute the join[6]. The *nested-loop join* as specified in Fig. 3 is the simplest one. In Fig. 3, we assume that S_0 is the root of R_1 and S_i is the parent relation-valued attribute of S_{i+1} for $i = 0, \cdots, k-1$. S_1 attribute value of a tuple t_1 is denoted by $t_1.S_1$. This algorithm becomes the nested-loop join

begin

for (each tuple t_1 in R_1)

begin

Build a partial result tuple from t_1 except the S_1-subrelation;

for (each S_1-subtuple t_{11} in $t_1.S_1$)

begin

Build a partial result subtuple from t_{11} except the S_2-subrelation;

$\vdots$

for (each S_k-subtuple t_{1k} in $t_1.S_1.\cdots.S_k$)

for (each tuple t_2 in R_2)

if (t_{1k} and t_2 satisfies JC)

Build a partial result subtuple concatenating t_{1k} and t_2;

$\vdots$

end

end

end

Figure 3: Nested-loop Join

devised for the relational join, when S_k is the root of R_1. In the following part of this paper, we consider only natural joins for simplicity of discussion. However, the discussion applies to joins in general with a slight modification.

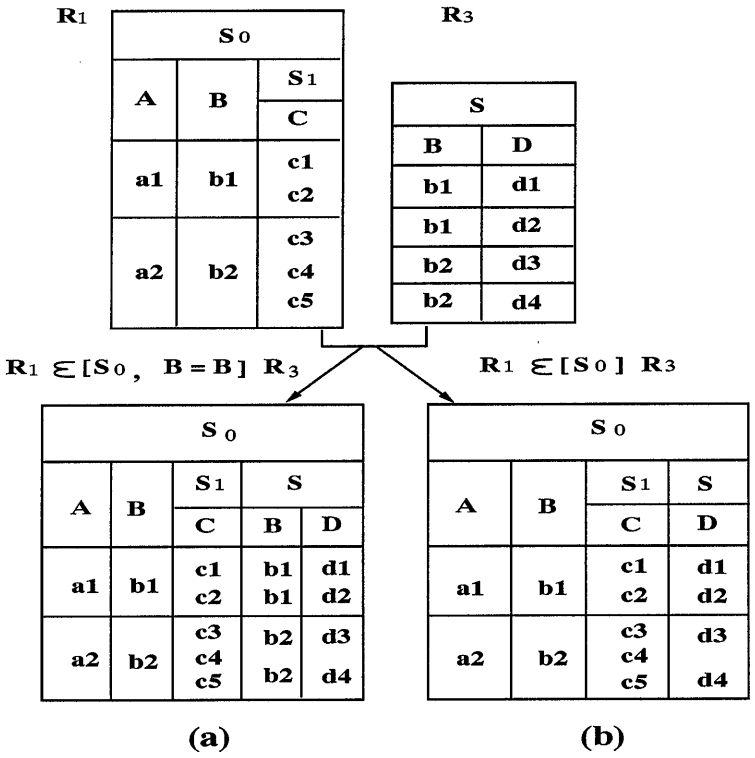

Figure 4: Embed and Natural Embed

2.2.2 Embed[12],[13]

The *embed* differs from the join in that it creates a new subrelation by embedding the second relation. Let us consider the embed $R_1\ \varepsilon[S_k, EC]\ R_3$ of two relations R_1 and R_3. Here, S_k denotes a relation-valued attribute of R_1, called the *target attribute*, and EC denotes the *embed condition*. As in the join condition, the embed condition may only reference child attributes of S_k in R_1 and child attributes of the root in R_3. We also assume that they are all atomic. The embed $R_1\ \varepsilon[S_k, EC]\ R_3$ creates a new relation-valued attribute, say S, in R_1 as a child of S_k, and, for each S_k-(sub)tuple in an S_k-(sub)relation, appends the S-subrelation consisting of tuples from R_2 that satisfy the embed condition. Here, S is called the *embedded attribute*. Fig.

4 (a) illustrates the embed $R_1\ \varepsilon[S_0, B = B]\ R_3$. Note that $\gamma(R_1\ \varepsilon[S_k, EC]\ R_3, S_m) = \gamma(R_1, S_m)$ holds for a relation-valued attribute S_m in $R_1\ \varepsilon[S_k, EC]\ R_3$, if S_m is not S nor its descendent.

In analogy to the natural join, *natural embed* is defined to remove the obvious redundancy when the embed condition involves only equality conditions. Fig. 4 (b) shows the natural embed $R_1\ \varepsilon[S_0]\ R_3$. When S_k is the root of R_1, the natural embed $R_1\ \varepsilon[S_k]\ R_3$ is simply denoted by $R_1\ \varepsilon\ R_3$. The *nested-loop embed* as specified in Fig. 5 is the simplest algorithm to execute the embed. In the following part of this paper, we consider only natural embeds for simplicity of discussion.

```
begin
    for (each tuple t1 in R1)
    begin
        Build a partial result tuple from t1 except the S1-subrelation;
        for (each S1-subtuple t11 in t1.S1)
        begin
            Build a partial result from t11 except the S2-subrelation;
                ⋮
            for (each Sk-subtuple t1k in t1.S1.···.Sk)
            begin
                Build a partial result subtuple from t1k except the S-subrelation;
                for (each tuple t2 in R2)
                    if (t1k and t2 satisfies EC)
                        put t2 as an S-subtuple;
            end
                ⋮
        end
    end
end
```

Figure 5: Nested-loop Embed

2.3 Query Graph

In this paper, we consider queries consisting of joins and embeds. A *query graph* is used to represent a query. In the query graph, relations involved in the query are represented by vertexes and joins and embeds are represented by edges. The join $R_1 \bowtie[S_k] R_2$ is represented by a *join edge* (directed solid edge) from R_1 to R_2 with the label S_k. In case S_k is the root of R_1, the label can be omitted. In that case, we can also omit the arrow, since the join is essentially symmetric if S_k is the root as the relational join operation. The embed $R_1\ \varepsilon[S_k]\ R_2$ is represented by an *embed edge* (directed dotted edge) from R_1 to R_2 with the label S_k. In case S_k is the root of R_1, the label can be omitted. Fig. 6 shows a sample query graph. A maximal connected subgraph that does not include embed edges is called a *join cluster*. The query graph in Fig. 6 has four join clusters C_1 - C_4.

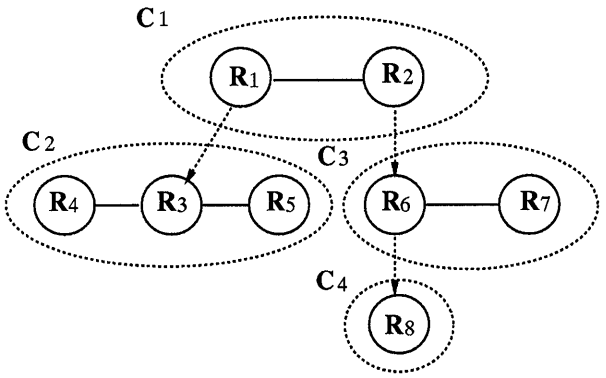

Figure 6: Query Graph

2.4 Processing Tree

A query execution plan is represented by a *processing tree (PT)*. A PT is a binary tree where leaf nodes represent base relations involved in the query and non-leaf nodes represent join and embed operations. In PTs, circles, rectangles, and triangles represent base relations, joins, and embeds, respectively. Operations in a PT is executed from the bottom to the top. A PT is called a *linear processing tree (LPT)*, if all operations appear in a linear sequence, in other words, they are totally ordered. LPTs considered in this paper are sometimes called *left-deep LPTs*, since base relations always become inner relations in the nested-loop join and embed

algorithms. We use the term "LPT" as a synonym of "left-deep LPT." Fig. 7 shows an example of an LPT. LPTs can be represented in a linear form. The LPT of Fig. 7 is represented as $R_1 \varepsilon R_3 \bowtie R_5 \bowtie R_4 \bowtie R_2 \varepsilon R_6 \bowtie R_7 \varepsilon R_8$, or simply $R_1R_3R_5R_4R_2R_6R_7R_8$.

A class of PTs defines an *execution space* over which the optimization is performed. The LPT execution space is the set of query executions whose processing trees are LPTs. The LPT execution space is the search space assumed by many query optimizers.

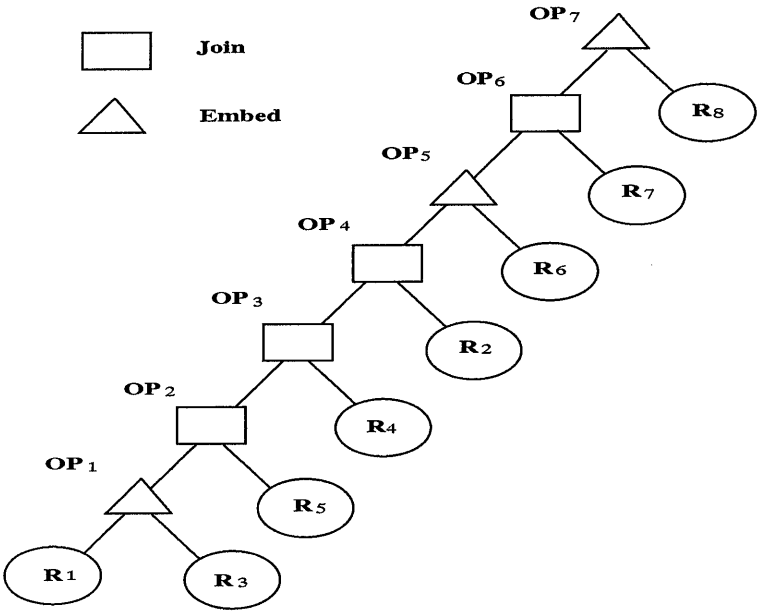

Figure 7: LPT

3 Cost Model

3.1 Selectivity

For the join $R_i \bowtie [S_k]\ R_j$, *join selectivity* denoted by SJ_{ij} represents the ratio of the number of tuple pairs from R_i and R_j that meet the join condition. That is,

$$SJ_{ij} = \frac{\gamma(R_i \bowtie [S_k]\ R_j, S_k)}{\gamma(R_i, S_k) * \gamma(R_j)}.$$

Similarly, for $R_i\ \varepsilon[S_k]\ R_j$, *embed selectivity* denoted by SE_{ij} represents the ratio of the number of tuple pairs from R_i and R_j that meet the embed condition. That is,

$$SE_{ij} = \frac{\gamma(R_i\ \varepsilon[S_k]\ R_j, S)}{\gamma(R_i, S_k) * \gamma(R_j)},$$

where S is the embedded attribute in $R_i\ \varepsilon[S_k]\ R_j$.

3.2 Assumptions

In this paper, we make the following assumptions to develop a cost model and to construct a query optimization scheme.

1. Query graphs satisfy the following conditions:

   (1) For each $R_i \bowtie[S_k]\ R_j$ or $R_i\ \varepsilon[S_k]\ R_j$ represented by an edge in a query graph, S_k is the root of R_i,

   (2) The internal structure of each join cluster forms a tree, and

   (3) At most one embed edge may exist between any pair of join clusters, and the structure connecting join clusters forms a rooted tree.

2. The processing tree is restricted to be an LPT, and the left bottom relation of the LPT is restricted to be one in the root join cluster in a given query graph.

3. Database is memory resident, and the execution costs of joins and embeds are evaluated in terms of in-memory processing costs.

4. Joins and embeds in a given query do not interact with each other as far as their selectivities are concerned. For example, the same join selectivity SJ_{ij} is not only applicable to $R_i \bowtie [S_k]\ R_j$ but also to the joins $R_i \bowtie[S_k]\ E(R_j)$ and $E(R_i) \bowtie[S_k]\ R_j$, where $E(R_j)$ and $E(R_i)$ stand for valid subexpressions joining and/or embedding other relations with R_j and R_i, respectively. A similar remark applies to the embed selectivity.

5. Any base or intermediate relation R satisfies the condition that, for an arbitrary internal relation-valued attribute S_k, every S_k-subrelation in R has the same number of S_k-subtuples.

The query graph of Fig. 6 and the LPT of Fig. 7 satisfy the assumptions 1 and 2, respectively.

3.3 Cost Equation

The cost of a query execution plan represented by an LPT is evaluated as the sum of the costs of all joins and embeds in the LPT. In the following, we give formulas to derive processing costs.

3.3.1 Join and Embed Costs

In the execution of join and embed, we have to compare a (sub)tuple from one relation with one from the other relation many times to check the join and embed conditions. In the memory resident model, the number of (sub)tuple comparisons is one of the most important cost measures. In our model, costs of join $R_i \bowtie[S_k]\ R_j$ and embed $R_i\ \varepsilon[S_k]\ R_j$ are expressed as follows:

$$\begin{aligned} Cost(R_i \bowtie[S_k]\ R_j) &= \gamma(R_i, S_k) * CJ(R_j) \\ Cost(R_i\ \varepsilon[S_k]\ R_j) &= \gamma(R_i, S_k) * CE(R_j), \end{aligned}$$

where $CJ(R_j)$ and $CE(R_j)$ are called *unit costs* and represent the join and embed costs, respectively, per S_k-(sub)tuple of R_i. They depend on join and embed execution algorithms. When we use the nested-loop join and nested-loop embed algorithms, both of them are γ values, since the numbers of (sub)tuple comparisons required in the join and embed are given as follows:

$$\begin{aligned} Cost(R_i \bowtie[S_k]\ R_j) &= \gamma(R_i, S_k) * \gamma(R_j) \\ Cost(R_i\ \varepsilon[S_k]\ R_j) &= \gamma(R_i, S_k) * \gamma(R_j). \end{aligned}$$

3.3.2 Cardinalities of Intermediate Relations

As mentioned above, γ values are required to evaluate the join and embed costs. Although they are given from the beginning for base relations, we have to calculate them for intermediate relations derived in the query processing. From the basic property of join and the assumption 5 in Subsection 3.2, $\gamma(R_i \bowtie[S_k]\ R_j, S_m)$, is calculated as follows:

Case 1: S_m is S_k or a descendant of S_k

$$\gamma(R_i \bowtie[S_k]\ R_j, S_m) = SJ_{ij} * \gamma(R_i, S_m) * \gamma(R_j),$$

Case 2: Otherwise

$$\gamma(R_i \bowtie[S_k]\ R_j, S_m) = \gamma(R_i, S_m).$$

Similarly, $\gamma(R_i\ \varepsilon[S_k]\ R_j, S_m)$ is calculated as follows:

Case 1: S_m is S or a descendant of S

$$\gamma(R_i\ \varepsilon[S_k]\ R_j, S_m) = SE_{ij} * \gamma(R_i, S_m) * \gamma(R_j),$$

where S is the embedded attribute in $R_i\ \varepsilon[S_k]\ R_j$,

Case 2: Otherwise

$$\gamma(R_i\ \varepsilon[S_k]\ R_j, S_m) = \gamma(R_i, S_m).$$

3.3.3 Cost of LPT

Let P be an LPT which satisfies the assumption 2 in Subsection 3.2. As mentioned before, the cost of an LPT is computed as the sum of join and embed costs. Fig. 7 shows such an LPT $R_1R_3R_5R_4R_2R_6R_7R_8$. From the expressions in Subsection 3.3.1 giving join and embed costs, we get the following expression giving the total cost of P.

$$Cost(P) \quad = \quad \sum_{i=1}^{k-1} \left(\gamma(R_{12\cdots i}, S_i) * CJE(R_{i+1})\right),$$

where $R_{12\cdots i}$ $(i > 1)$ is the $(i-1)$-th intermediate relation, S_i is the target relation-valued attribute of the i-th (join or embed) operation, and $CJE(R_{i+1})$ is the unit cost of the i-th (join or embed) operation. In P, $k = 8$. $\gamma(R_{12\cdots i+1}, S_i)$ is calculated by the expressions in Subsection 3.3.2.

4 Query Optimization

In this section, we show an algorithm which derives a cost optimal LPT for a given query graph under the assumptions and the cost model in Section 3. In our algorithm, we utilize the KBZ method, which was proposed to optimize join queries in the relational database, as a subprocedure. We first give an overview of the KBZ method, and then present our algorithm with an example.

4.1 KBZ Method

The KBZ method was originally proposed by Krishnamurthy, Boral, and Zaniolo[15]. Given a join tree for a relational database, the KBZ method derives a cost optimal LPT of joins. In the KBZ method, the database is assumed to be memory resident, and the cost of join $R_i \bowtie R_j$ is calculated by the expression $\gamma(R_i)*g(R_j)$, where $g(R_j)$ depends on the join execution algorithm. Furthermore, joins are assumed to be independent of each other as far as their selectivities are concerned. These assumptions just coincide with our discussion here.

The KBZ method is composed of two levels of procedures. The first procedure KBZ_1 generates a cost optimal LPT of joins for a given rooted join tree under the restriction that the root relation always comes at the left bottom of the LPT. The second procedure KBZ_2 inputs a join tree, and finds a cost optimal LPT by invoking KBZ_1 for each selection of the root.

The procedure KBZ_1 works on the rooted join tree in a bottom-up manner. All relations in subtrees are sorted to form a linear sequence based on the value of the *rank* $(\gamma(R_i)*JS_i-1)/g(R_i)$, where JS_i is the join selectivity of R_i and its parent, and $g(R_i)$ is the above mentioned factor used to determine the join cost. This process is recursively continued from the bottom to the top. When this process stops, we get an LPT, which gives a cost optimal join sequence for the given rooted join tree.

The procedure KBZ_2 invokes KBZ_1 so as to find a cost optimal LPT for each selection of the root in the input join tree. If there exist N relations in a join tree, it calls KBZ_1 N times. After that, KBZ_2 selects the cost optimal LPT. It has been shown that the time complexity of KBZ_1 is $O(N \log N)$ and that the whole KBZ method can be accomplished in $O(N^2)$ time[15].

4.2 Basic Strategy

As mentioned in Subsection 3.2, the left bottom relation of an LPT is restricted to one in the root join cluster. Therefore, in any LPT under consideration, one of the relations in the root join cluster is selected as the left bottom relation, and then adjacent relations are repeatedly joined or embedded with the intermediate result along the join and embed edges. This process stops when all the base relations in the query graph are joined or embedded, and the final result relation is obtained.

Here, we define *Depth-First-Execution LPTs (DFE-LPTs)* as LPTs which satisfy the follow-

ing additional restriction:

> Let e is an arbitrary embed edge in the query graph. Assume e goes from join cluster C_i to C_j. Then, in a DFE-LPT, once the embed e is performed, all the joins and embeds involved in the join cluster C_j and its descendants are performed before any other join or embed.

The LPT of Fig. 7 is a DFE-LPT for the query graph of Fig. 6. For example, once $R_1 \varepsilon R_3$ is performed, $R_3 \bowtie R_4$ and $R_3 \bowtie R_5$ are performed before any other join or embed. The above condition is satisfied for any embed included in the query graph.

Our optimization algorithm considers only DFE-LPTs to find a cost optimal LPT. In Section 5, we prove that there always exist a cost optimal DFE-LPT. Therefore, it is sufficient to consider DFE-LPTs as candidates of the cost optimal LPT.

Let us consider embed $R_i \varepsilon R_j$, where R_i and R_j are included in join clusters C_i and C_j, respectively. Let $REL(C_j)$ and $OP(C_j)$ be sets of relations and (join and embed) operations, respectively, included in the join cluster C_j and its descendants, and let $Q(C_j)$ be a query (sub)graph consisting of $REL(C_j)$ and $OP(C_j)$. Note that $OP(C_j)$ does not include the embed $R_i \varepsilon R_j$ itself. Then, in a DFE-LPT P, all the operations in $OP(C_j)$ come just above the embed $R_i \varepsilon R_j$ as shown in Fig. 8. In Fig. 8, the triangle represents the embed $R_i \varepsilon R_j$. (Note that this embed is actually executed as $R \varepsilon[S] R_j$ for the intermediate relation R in the context of P.) Therefore, in the linear notation, P can be specified as $P = TR_jR_{j1}\cdots R_{jm}U$, where T and U are sequences of relations outside $REL(C_j)$, and $R_{j1}\cdots R_{jm}$ is a sequence of relations in $REL(C_j) - \{R_j\}$ giving the execution sequence of operations in $OP(C_j)$.

From the definition, $P_j = R_jR_{j1}\cdots R_{jm}$ is also a valid DEF-LPT for the query graph $Q(C_j)$. Let its cost be $Cost(P_j)$. Then, the cost contribution of operations specified by $R_{j1}\cdots R_{jm}$ in P is given by $SE_{ij} * \gamma(R,S) * Cost(P_j)$, where R and S are given above. The reason is as follows. Let R' be the result relation of LPT $T' = TR_j$, and let S_j be the embedded attribute in embedding R_j. Then, from our assumptions, the cost contribution of $R_{j1}\cdots R_{jm}$ is obviously proportional to $\gamma(R', S_j)$ and the processing cost per S_j-(sub)tuple is given by $Cost(P_j)/\gamma(R_j)$. Therefore,

$$\gamma(R', S_j) * Cost(P_j)/\gamma(R_j) \quad = \quad SE_{ij} * \gamma(R,S) * \gamma(R_j) * Cost(P_j)/\gamma(R_j)$$

$$= SE_{ij} * \gamma(R,S) * Cost(P_j).$$

Since the cost of the embed $R\ \varepsilon[S]\ R_j$ itself is given by $\gamma(R,S) * \gamma(R_j)$, the subtotal cost contribution of operations specified by $R_jR_{j1}\cdots R_{jm}$ is given by

$$\gamma(R,S) * (\gamma(R_j) + SE_{ij} * Cost(P_j)).$$

Then, from the above discussion, the subsequence $R_jR_{j1}\cdots R_{jm}$ is equivalent to joining a dummy relation $R(C_j)$ with the following unit cost and the join selectivity as far as the execution cost is concerned:

$$\begin{aligned} CJ(R(C_j)) &= \gamma(R_j) + SE_{ij} * Cost(P_j), \\ SJ &= 1/\gamma(R(C_j)). \end{aligned}$$

Furthermore, from the basic property of embed, we can say that ordering of operations in $OP(C_j)$ represented by $R_jR_{j1}\cdots R_{jm}$ contributes the total processing cost only through the above derived cost factor $Cost(P_j)$ and does not affect processing costs of operations outside $OP(C_j)$. In other words, the subsequence $R_jR_{j1}\cdots R_{jm}$ that makes P cost optimal can be determined locally in the context of $OP(C_j)$ and $REL(C_j)$.

Based on the above consideration, we propose an algorithm to derive a cost optimal LPT for a given query graph. The meat of our algorithm is a procedure $SUBOPT$. The procedure $SUBOPT$ is applied recursively to each join cluster C_i to find a cost optimal execution sequence of operations in $OP(C_i)$. Assume that $SUBOPT$ is applied to join cluster C_i. If this join cluster has any child join clusters, we apply $SUBOPT$ recursively to each child C_j to get the cost optimal subsequence $R_jR_{j1}\cdots R_{jm}$ of operations in $OP(C_j)$. Then, we replace the sequence $R_jR_{j1}\cdots R_{jm}$ as a single join with $R(C_j)$ as mentioned above. At this point, the problem we have to solve is to obtain a cost optimal sequence of joins which are either originally involved in C_i or introduced in the above procedure. This problem can be solved by direct application of the KBZ method explained in Subsection 4.1.

4.3 Optimization Algorithm

Our optimization algorithm OPT is based on the above-mentioned consideration. OPT is shown in Fig. 9. OPT calls procedures $SUBOPT$ (Fig. 10) and KBZ (Fig. 11). The optimization

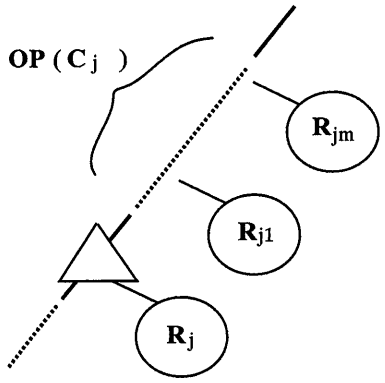

Figure 8: DFE-LPT

algorithm is basically recursive, and a cost optimal subsequence is derived in a bottom-up manner for the given query graph. OPT applies the procedure $SUBOPT$ to each child join cluster C_j of the root join cluster C_R. Each invocation of $SUBOPT(C_j, R_j)$ returns a cost optimal LPT P_j for the subgraph $Q(C_j)$ under the restriction that the left bottom relation of P_j is R_j. Once the cost optimal subsequence P_j is obtained, we can regard it as a join with $R(C_j)$ as mentioned in Subsection 4.2. Then, the remaining problem of finding a cost optimal join sequence is solved by the KBZ method. The procedure $KBZ(Q', R_k)$ returns a cost optimal LPT under the restriction that the given relation R_k comes at the left bottom of the LPT. Since every relation in the root may come at the left bottom in our problem, KBZ is called from OPT for each relation in the root join cluster. The final step is to find a cost optimal join sequence P and to substitute P_j for $R(C_j)$ to get the complete LPT for Q.

Subprocedure $SUBOPT$ is very similar to OPT. The difference comes from the restriction that the left bottom relation of target cost optimal LPTs for $Q(C)$ be R given as an argument at the invocation of $SUBOPT$. $SUBOPT$ also calls KBZ internally.

4.4 Example

We show application of OPT to the query graph Q of Fig. 6. When OPT is applied to Q, it invokes $SUBOPT(C_2, R_3)$ and $SUBOPT(C_3, R_6)$. Since C_2 has no child join clusters, $SUBOPT(C_2, R_3)$ finds a cost optimal join sequence for the subquery Q_1 of Fig. 12 (a) under the restriction that left bottom relation is R_3. This problem can be solved by invocation of

Algorithm *OPT*

Input: Query Graph Q

Output: Cost Optimal LPT P for Q

begin

$C_R \leftarrow$ the root join cluster of Q;

$Q' \leftarrow$ a query graph consisting of all the relations and joins in C_R;

for (each child join cluster C_j of C_R) /* Assume C_R and C_j are connected by an embed edge from R_i to R_j */

begin

$P_j \leftarrow SUBOPT(C_j, R_j)$;

Add the relation $R(C_j)$ and a join edge from R_i to $R(C_j)$ to Q' assuming the following cost parameters:

$$CJ(R(C_j)) = \gamma(R_j) + SE_{ij} * Cost(P_j)$$

$$SJ = 1/\gamma(R(C_j));$$

end

for (each relation R_k in C_R)

$P_{Rk} \leftarrow KBZ(Q', R_k)$;

$P \leftarrow$ the minimum cost P_{Rk};

for (each $R(C_j)$)

Replace $R(C_j)$ with P_j in P;

Return P;

end

Figure 9: OPT

Procedure *SUBOPT*

Input: Join Cluster C in Q

Relation R in C

Output: LPT P for $Q(C)$

begin

$Q' \leftarrow$ a query graph consisting of all the relations and joins in C;

for (each child join cluster C_j of C) /* Assume C and C_j are connected with an embed edge from R_i to R_j */

begin

$P_j \leftarrow SUBOPT(C_j, R_j)$;

Add the relation $R(C_j)$ and a join edge from R_i to $R(C_j)$ to Q' assuming the following cost parameters:

$$CJ(R(C_j)) = \gamma(R_j) + SE_{ij} * Cost(P_j)$$

$$SJ = 1/\gamma(R(C_j));$$

end

$P \leftarrow KBZ(Q', R)$;

for (each $R(C_j)$)

Replace $R(C_j)$ with P_j in P;

Return P;

end

Figure 10: SUBOPT

Procedure KBZ

Input: Query Graph Q (Consisting of a Join Cluster)

Relation R in Q

Output: LPT P for Q

begin

for (each relation $R_i(\neq R)$ in Q)

Calculate the rank $(\gamma(R_i) * JS_i - 1)/CJ(R_i)$;

$P \leftarrow$ an LPT constructed by the KBZ_1 procedure representing a cost optimal join sequence for Q under the restriction that the left bottom relation is R;

Return P;

end

Figure 11: KBZ

$KBZ(Q_1, R_3)$. Let us assume that $SUBOPT(C_2, R_3)$ returns $R_3R_4R_5$. Since C_3 has a child join cluster C_4, $SUBOPT(C_3, R_6)$ recursively invokes $SUBOPT(C_4, R_8)$. $SUBOPT(C_4, R_8)$ obviously returns R_8. Then, $SUBOPT(C_3, R_6)$ finds an optimal join sequence for the subquery Q_2 of Fig. 12 (b) in a similar manner. Assume that $SUBOPT(C_3, R_6)$ returns $R_6R_7R(C_4)$. Then, a cost optimal subsequence for subquery $Q(C_3)$ is derived as $R_6R_7R_8$. Finally, OPT finds a cost optimal join sequence for subquery Q_3 of Fig. 12 (c). Since either R_1 or R_2 may come at the left bottom of the final LPT, OPT calls $KBZ(Q_3, R_1)$ and $KBZ(Q_3, R_2)$. Assume that the LPT $R_1R(C_2)R_2R(C_3)$ returned by $KBZ(Q_3, R_1)$ be cheaper than that returned by $KBZ(Q_3, R_2)$. Then, we get the LPT $R_1R_3R_4R_5R_2R_6R_7R_8$ as a cost optimal solution for the given query Q.

5 Discussion

In this section, we show that the optimization algorithm OPT in Subsection 4.3 really gives a cost optimal LPT under the assumptions in Subsection 3.2. We also show that the time complexity of the algorithm is $O(N^2)$ when N relations are involved in a given query graph.

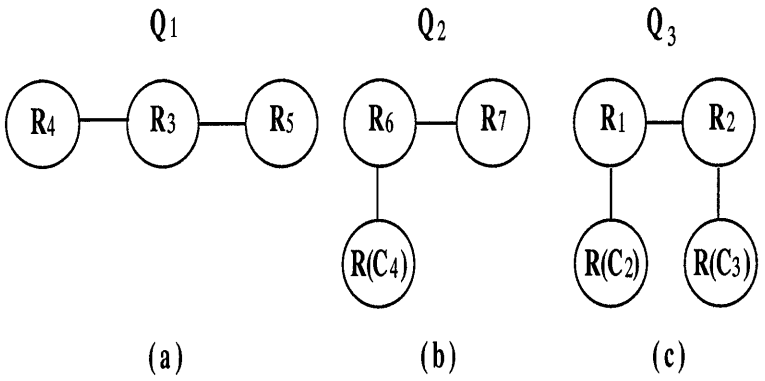

Figure 12: Subqueries

5.1 Cost Optimality

Before we prove the cost optimality of the algorithm OPT, we prove four lemmas. The first three lemmas show basic properties of join and embed regarding the processing cost.

Lemma 1 *Given a query graph in Fig. 13, let* P_{123} *and* P_{132} *be LPTs such that* $P_{123} = R_1R_2R_3$ *and* $P_{132} = R_1R_3R_2$. *Then,* $Cost(P_{123}) = Cost(P_{132})$.

(Proof) Since $\gamma(R_1\ \varepsilon[S_{12}]\ R_2, S_{13}) = \gamma(R_1, S_{13})$ and $\gamma(R_1\ \varepsilon[S_{13}]\ R_2, S_{12}) = \gamma(R_1, S_{12})$, the lemma obviously holds. ■

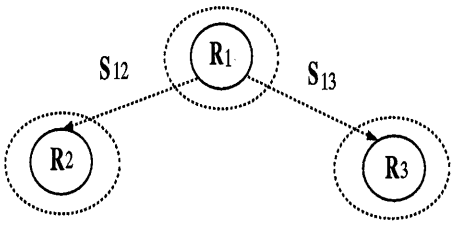

Figure 13: Case of Lemma 1

Lemma 2 *Given query graphs in Fig. 14 (a)(b), let* $P_{1234} = R_1R_2R_3R_4$, $P_{1324} = R_1R_3R_2R_4$, *and* $P_{1342} = R_1R_3R_4R_2$. *Then,* $Cost(P_{1234}) = Cost(P_{1324}) = Cost(P_{1342})$.

(Proof) Proved similarly to Lemma 1 based on basic properties of join and embed. ■

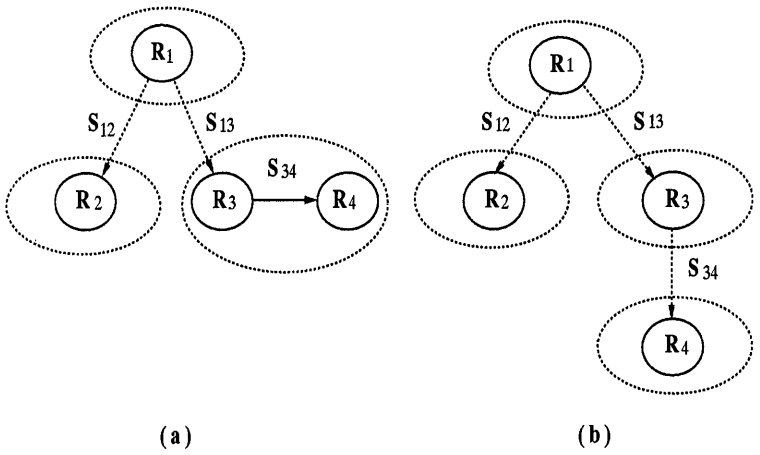

Figure 14: Cases of Lemma 2

Lemma 3 *Given query graphs in Fig. 15 (a)(b), let* $P_{1234} = R_1R_2R_3R_4$, $P_{1324} = R_1R_3R_2R_4$, *and* $P_{1342} = R_1R_3R_4R_2$. *Then, their costs always satisfy the following condition:*

$$Cost(P_{1234}) \leq Cost(P_{1324}) \leq Cost(P_{1342})$$

or

$$Cost(P_{1234}) > Cost(P_{1324}) > Cost(P_{1342}).$$

In other words, P_{1324} *cannot become a unique cost optimal LPT among the three.*

(Proof) See Appendix A.

The following Lemma 4 is essential in proving the cost optimality of our algorithm.

Lemma 4 *There always exists a DFE-LPT which is cost optimal for a given query graph.*

(Proof) See Appendix B.

Theorem: *The algorithm OPT gives a cost optimal LPT for a given query graph.*

(Proof) From the discussion in Subsection 4.2, it is proved that the algorithm *OPT* derives a cost minimum DFE-LPT among all the valid DFE-LPTs. From Lemma 4, we can conclude that the DFE-LPT is a cost optimal LPT for a given query graph. ■

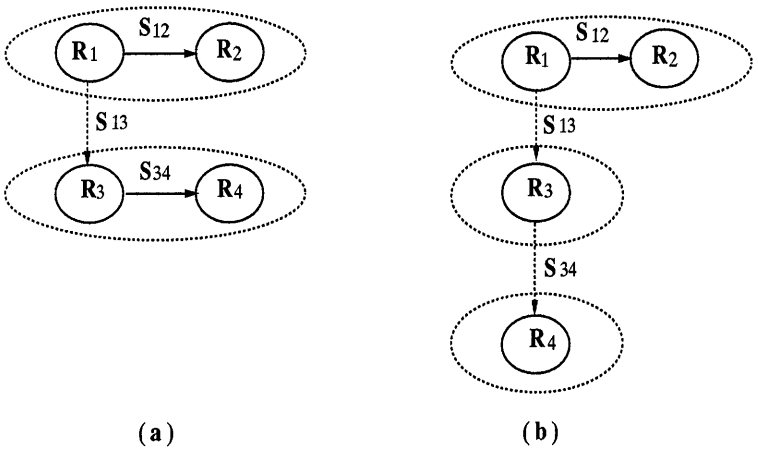

Figure 15: Cases of Lemma 3

5.2 Time Complexity

Let us assume that the query graph includes N relations and that the root join cluster includes N_R relations and has N_c child join clusters. In the algorithm OPT, $SUBOPT(C_j, R_j)$ is called for each child join cluster C_j. In the invocation of $SUBOPT$, $SUBOPT$ is recursively called for each child join cluster of C_j and then KBZ is executed for C_j. As mentioned in Subsection 4.1, the time complexity of KBZ is known to be $O(N \log N)$ when N relations are involved. Therefore, the time complexity of the invocation of $SUBOPT(C_j, R_j)$ from OPT is bounded by $O(N_j \log N_j)$, where N_j represents the number of relations included in the $REL(C_j)$. Therefore, the total time required to invoke $SUBOPT$ from OPT is given by $O((N - N_R)log(N - N_R))$. In OPT, KBZ is performed N_R times. This process essentially requires the time to perform the whole KBZ method for a query involving $N_R + N_c$ relations, namely $O((N_R + N_c)^2)$. Therefore, the overall complexity of the algorithm OPT is $O(N^2)$.

6 Conclusion

We have proposed an optimization algorithm for join-type queries in nested relational databases. We have focussed two operations: join and embed. Our algorithm gives a cost optimal LPT for a query graph including joins and embeds under the assumptions in Subsection 3.2. We have also shown that the time complexity of the algorithm is $O(N^2)$ when N relations are involved

in the query graph. The $O(N^2)$ time complexity makes our algorithm very promising for the query optimizer in NRDBMSs.

Although we assumed the nested-loop join and embed algorithms in our discussion, the basic framework can be applied to other join and embed algorithms as long as their processing costs can be formulated in the following forms:

$$\begin{aligned} Cost(R_i \bowtie[S_k]\ R_j) &= \gamma(R_i, S_k) * CJ(R_j) \\ Cost(R_i\ \varepsilon[S_k]\ R_j) &= \gamma(R_i, S_k) * CE(R_j). \end{aligned}$$

For example, the hash-based algorithm can be discussed by setting $CJ(R_j)$ and $CE(R_j)$ to the average collision chain length, if we regard the navigation in the chain as the major cost factor. Similarly, we can incorporate the index-based algorithm, if we take $\log_m(\gamma(R_j))$ representing the height of the index tree for $CJ(R_j)$ and $CE(R_j)$. However, these cost expressions are all memory-based in that page structures are ignored. Extension of our cost model to the disk resident model is a remaining research issue. Other future research issues include evaluation of our method in more practical situations where some of our assumptions are not satisfied in a strict sense, extension to queries including other types of operations, and design of a query optimizer incorporating our method. Research results on these issues will be reported in forthcoming papers.

Appendix A: Proof of Lemma 3

Let us consider the case of Fig. 15 (a). According to the cost model in Subsection 3.3, we get the following formulas:

$$\begin{aligned}
Cost(P_{1234}) &= \gamma(R_1, S_{12}) * CJ(R_2) + \gamma(R_{12}, S_{13}) * CE(R_3) \\
&\quad + \gamma(R_{123}, S_{34}) * CJ(R_4), \\
Cost(P_{1324}) &= \gamma(R_1, S_{13}) * CE(R_3) + \gamma(R_{13}, S_{12}) * CJ(R_2) \\
&\quad + \gamma(R_{132}, S_{34}) * CJ(R_4), \\
Cost(P_{1342}) &= \gamma(R_1, S_{13}) * CE(R_3) + \gamma(R_{13}, S_{34}) * CJ(R_4) \\
&\quad + \gamma(R_{134}, S_{12}) * CJ(R_2).
\end{aligned}$$

(**Case 1:** S_{13} is identical with S_{12} or its descendant)

From the discussion in Subsection 3.3.2, we get the following equations:

$$\begin{aligned}
\gamma(R_1, S_{12}) &= \gamma(R_{13}, S_{12}), \\
\gamma(R_{123}, S_{34}) &= \gamma(R_{132}, S_{34}), \\
\gamma(R_{12}, S_{13}) &= SJ_{12} * \gamma(R_1, S_{13}) * \gamma(R_2), \\
\gamma(R_{13}, S_{12}) &= \gamma(R_{134}, S_{12}), \\
\gamma(R_{132}, S_{34}) &= SJ_{12} * \gamma(R_{13}, S_{34}) * \gamma(R_2).
\end{aligned}$$

By substitution, we can get

$$\begin{aligned}
Cost(P_{1234}) - Cost(P_{1324}) &= \gamma(R_1, S_{13}) * CE(R_3) * (SJ_{12} * \gamma(R_2) - 1), \\
Cost(P_{1324}) - Cost(P_{1342}) &= \gamma(R_{13}, S_{34}) * CJ(R_4) * (SJ_{12} * \gamma(R_2) - 1).
\end{aligned}$$

Therefore, if $SJ_{12} * \gamma(R_2) \leq 1$,

$$Cost(P_{1234}) \leq Cost(P_{1324}) \leq Cost(P_{1342}),$$

otherwise

$$Cost(P_{1234}) > Cost(P_{1324}) > Cost(P_{1342}).$$

(**Case 2:** Otherwise)

From similar discussion, we get

$$Cost(P_{1234}) = Cost(P_{1324}) = Cost(P_{1342}).$$

Therefore, we have proved Lemma 3 for the case of Fig. 15 (a). Lemma 3 is proved for the case of Fig. 15 (b) in a similar way. ■

Appendix B: Proof of Lemma 4

Let us assume that P is an LPT but not a DEF-LPT. Then, P can be specified in the linear form $P = TR_1UVR_2W$, where

(1) R_1 is a relation in some join cluster C_1, and the embed edge from R_0 in the join cluster C_0 to R_1 exists in the query graph Q,

(2) U is (possibly null) sequence of relations in $REL(C_1)$,

(3) V is (non-null) sequence of relations in $REL(C_0) - REL(C_1)$,

(4) R_2 is a relation in $REL(C_1)$ and belongs to the join cluster C_2,

(5) T is a (non-null) sequence of relations outside $REL(C_1)$ and T locally satisfies the condition of DFE-LPT, and

(6) W is a (possibly null) sequence of relations.

Fig. 16 illustratively shows the situation. There are two cases that R_2 is joined with or embedded into the intermediate result. Let us consider the former case, and let the intermediate relation resulted from T be R_A. Note that the subsequence R_1U specifies an execution sequence of joins and embeds, and let its result be R_B. Then, the execution of TR_1U is equivalent both in its result and cost to embedding the relation R_B into R_A with the following cost parameters:

$$\begin{aligned} CE_B(R_B) &= CE_1(R_1) + SE_{01} * Cost(R_1U) \\ SE_{AB} &= SE_{01}. \end{aligned}$$

Let its result be R_{AB}. As mentioned above, V is a sequence of relations in $REL(C_0) - REL(C_1)$. In analogy to the above discussion for the sequence U, we can construct a sequence V' equivalent to V consisting of an optional leading join and zero or more embeds directly applied to R_{AB}.

(**Case 1**: V' starts with a join.)

Let us assume that V' consists of a join $R_{AB} \bowtie[S_{VJ}]\ R_{VJ}$ and embeds $R_{AB}\ \varepsilon[S_{VE_1}]\ R_{VE_1}$, $\cdots$, $R_{AB}\ \varepsilon[S_{VE_m}]\ R_{VE_m}$, namely $V' = R_{VJ}R_{VE_1}\cdots R_{VE_m}$. Then, $R_AR_BV'R_2$ is an LPT for a query graph in Fig. 17. (In Fig. 17, edge labels are omitted for simplicity). From the basic properties of join and embed,

$$\begin{aligned} Cost(R_AR_BV'R_2) &= Cost(R_AR_BR_{VJ}R_{VE_1}\cdots R_{VE_m}R_2) \\ &= Cost(R_AR_BR_{VJ}R_2R_{VE_1}\cdots R_{VE_m}). \end{aligned}$$

By Lemma 3,

$$Cost(R_A R_B V' R_2) \geq Cost(R_A R_B R_2 R_{VJ} R_{VE_1} \cdots R_{VE_m})$$
$$= Cost(R_A R_B R_2 V')$$

or

$$Cost(R_A R_B V' R_2) \geq Cost(R_A R_{VJ} R_B R_2 R_{VE_1} \cdots R_{VE_m}).$$

By Lemmas 1 and 2,

$$Cost(R_A R_{VJ} R_B R_2 R_{VE_1} \cdots R_{VE_m}) = Cost(R_A R_{VJ} R_{VE_1} \cdots R_{VE_m} R_B R_2)$$
$$= Cost(R_A V' R_B R_2).$$

Therefore,

$$Cost(R_A R_B V' R_2) \geq Cost(R_A R_B R_2 V')$$

or

$$Cost(R_A R_B V' R_2) \geq Cost(R_A V' R_B R_2).$$

This implies

$$Cost(R_A R_B V' R_2 W) \geq Cost(R_A R_B R_2 V' W)$$

or

$$Cost(R_A R_B V' R_2 W) \geq Cost(R_A V' R_B R_2 W).$$

(**Case 2**: V' does not include a join.)

Let us assume that V' consists only of embeds $R_{AB}\ \varepsilon[S_{VE_1}]\ R_{VE_1}, \cdots, R_{AB}\ \varepsilon[S_{VE_m}]\ R_{VE_m}$. Then, by Lemmas 1 and 2,

$$Cost(R_A R_B V' R_2) = Cost(R_A R_B R_2 V') = Cost(R_A V' R_B R_2).$$

Therefore,

$$Cost(R_A R_B V' R_2 W) = Cost(R_A R_B R_2 V' W) = Cost(R_A V' R_B R_2 W).$$

Thus, both in Cases 1 and 2, we get

$$Cost(T R_1 U V R_2 W) \geq Cost(T R_1 U R_2 V W)$$

or

$$Cost(T R_1 U V R_2 W) \geq Cost(T V R_1 U R_2 W).$$

The above expression also holds even if R_2 is embedded into the intermediate relation. It means that replacing TR_1UVR_2W with TR_1UR_2VW or TVR_1UR_2W eliminates the assumed violation of the DFE-LPT condition without inducing a new violation nor increasing the total processing cost. This implies that we can find a cost optimal LPT in the set of DFE-LPTs. ■

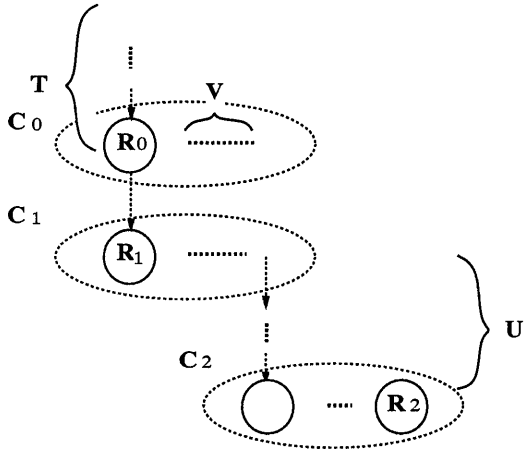

Figure 16: Case of Lemma 4

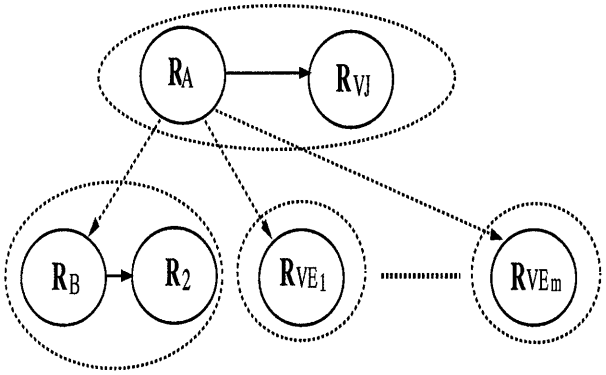

Figure 17: Query Graph for $R_AR_BV'R_2$

Acknowledgements

The authors are grateful to Prof. Yuzuru Fujiwara and Prof. Isao Suzuki, Institute of Information Sciences and Electronics, University of Tsukuba, for their encouragement to this research. This work is partially supported by grants from University of Tsukuba Project Research.

References

[1] Abiteboul, S. and Bidoit, N., "Non First Normal Form Relations to Represent Hierarchically Organized Data," *Proc. 3rd ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pp. 191-200, March 1984

[2] Abiteboul, S., Fischer, P. C., and Schek, H. J. (eds.), *Nested Relations and Complex Objects in Databases*, Lecture Notes in Computer Science 361, Springer-Verlag, 1989

[3] Colby, L.S.,"A Recursive Algebra and Query Optimization for Nested Relations," *Proc. ACM SIGMOD Conf.*, Portland, pp. 273-283, June 1989

[4] Dadam, P., et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies," *Proc. ACM SIGMOD Conf.*, Washington, D. C., pp. 356-367, May 1986

[5] Deshpande, A. and Van Gucht, D., "An Implementation for Nested Relational Databases," *Proc. 14th VLDB Conf.*, Los Angeles, pp. 76-87, August 1988

[6] Deshpande, V. and Larson, P. A., "The Design and Implementation of a Parallel Join Algorithm for Nested Relation on Shared-Memory Multiprocessors," *Proc. 8th International Conf. on Data Engineering*, pp. 68-77, 1992

[7] Fischer, P. C. and Thomas, S. J., "Operators for Non-First-Normal-Form Relation," *Proc. IEEE COMPSAC 83*, Chicago, pp. 464-475, November 1983

[8] Freytag, J. C., Maier, D. and Vossen, G. (Eds.), *Query Processing for Advanced Database Systems*, Morgan Kaufmann Publishers, 1994

[9] Gyssens, M. and Van Gucht, D., "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra," *Proc. ACM SIGMOD Conf.*, Chicago, pp. 225-232, June 1988

[10] Haskin, R. L. and Lorie, R. A., "On Extending the Functions of a Relational Database System," *Proc. ACM SIGMOD Conf.*, pp. 207-212, June 1982

[11] Ioannidis, Y. E. and Kang, Y. C.,"Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimizations," *Proc. ACM SIGMOD Conf.*, Denver, pp. 168-177, June 1991

[12] Kitagawa, H. and Kunii, T. L., *The Unnormalized Relational Data Model - For Office Form Processor Design -*, Springer-Verlag, 1989

[13] Kitagawa, H., Kunii, T. L. and Ohbo, N.,"Classification of Nested Tables under Deeply Nested Algebra," *Proc. 24th Hawaii International Conf. on System Sciences*, Hawaii, pp. 165-173, January 1991

[14] Korth, H. F., "Optimization of Object-Retrieval Queries," *Proc. 2nd International Workshop on Object-Oriented Database Systems* (Lecture Notes in Computer Science 334), Springer-Verlag, pp. 352-357, September 1988

[15] Krishnamurthy, R., Boral, H. and Zaniolo, C., "Optimization of Nonrecursive Queries," *Proc. 12th VLDB Conf.*, Kyoto, Japan, pp. 128-137, August 1986

[16] Lanzelotte, R. S. G., et al., "Optimization of Non-recursive Queries in OODBs," *Proc. 2nd DOOD*, Munich, Germany, pp. 1-21, December 1991

[17] Lorie, R. and Plouffe, W., "Complex Objects and Their Use in Design Transactions," *Proc. ACM SIGMOD Conf.*, pp. 115-121, May 1983

[18] Makinouchi, A., "A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model," *Proc. 3rd VLDB Conf.*, Tokyo, Japan, pp. 447-453, October 1977

[19] Ozsoyoglu, Z. M. (ed.), *Special Issue on Nested Relations, IEEE Data Engineering*, Vol. 11, No. 3, September 1988

[20] Roth, M. A., Korth, H. F. and Silberschatz, A., "Extended Algebra and Calculus for Nested Relational Databases," *ACM Transactions on Database Systems*, Vol. 13, No. 4, pp. 389-417, December 1988

[21] Schek, H. J. and Scholl, M. H., "The Relational Model with Relation-Valued Attributes," *Information Systems*, Vol. 11, No. 2, pp. 137-147, 1986

[22] Scholl, M. H., Paul, H.B. and Schek, H. J., "Supporting Flat Relations by a Nested Relational Kernel," *Proc. 13th VLDB, Conf.*, Brighton, pp. 137-147, 1987

[23] Selinger, P. G., et al., "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD Conf.*, pp. 23-34, 1979

[24] Straube, D. D. and Ozsu, M. T., "Queries and Query Processing in Object-Oriented Database Systems, " *ACM Transactions on Information Systems*, Vol. 8, No. 4, pp. 387-430, October 1990

[25] Swami, A., "Optimization of Large Join Queries," *Proc. ACM SIGMOD Conf.*, Chicago, pp. 8-17, June 1988

[26] The Committee for Advanced DBMS Function, "The Third-Generation Database System Manifests," *ACM SIGMOD Record*, Vol. 19, No. 3, pp. 31-44, 1990

[27] Valduriez, P., Khoshafian, S. and Copeland, G., "Implementation Techniques of Complex Objects," *Proc. 12th VLDB, Conf.*, Kyoto, Japan, pp. 101-110, August 1986