Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

# A Comparison of Composition and Join Operations

*by*

*Yao-Nan Lien, Gillin Hu* and Chian-Yuan Chiou*
*Department of Computer Science*
*National Chengchi University*
*Taipei, Taiwan, R.O.C.*
*lien@cherry.cs.nccu.edu.tw, (02) 9393091-2275, Fax:(02)2341494*

## Abstract

*In relational databases, a composition requires three operations: join, projection and duplicate elimination. An external sort is required to eliminate duplicates in a large file. Both join and external sort are expensive because they incur a nonlinear I/O overhead. Most conventional composition algorithms take join and duplicate elimination as separate operations to achieve savings on either operation but not both. Direct composition algorithms outperform all these algorithms by executing the composition as a single primitive. In this paper, we show that direct composition can even outperforrn its component operation, the join operation, when there is a high degree of duplication after join and projection operations. This result can encourage the correct implementation of join operation to preserve the integrity of relational data model in real DBMSs where duplicate tuples are usually not removed.*

**Keywords:** Database, composition, join, duplicate elimination.

## 1. Introduction

### 1.1 Composition Operation

Given two binary relations *R(A,B)* and *S(B,C)*, a composition $R \circ S$ is defined as $\Pi_{AC}(R \bowtie S)$, where $\Pi$ is a projection and $\bowtie$ is a natural join. The sizes of relations $R$ and $S$ are denoted as $|R|$ and $|S|$, respectively. Composition is a very common operation in computing transitive closure and other types of linear recursions in deductive databases [9,10]. Much work has been done on processing of transitive closure and linear recursion [1,11,12]. Therefore, composition is an important primitive operation for support of many emerging new database applications and new database models such as deductive databases, knowledge bases, and object-oriented databases.

* This work is supported by NSC Grant (86-2213-E-004-002).

| $R$ | |
|---|---|
| A | B |
| 1 | 2 |
| 2 | 3 |
| 1 | 1 |
| 2 | 2 |
| 3 | 5 |

| $S$ | |
|---|---|
| B | C |
| 1 | 3 |
| 2 | 4 |
| 2 | 3 |
| 1 | 4 |
| 4 | 3 |

| $R \bowtie S$ | | |
|---|---|---|
| A | B | C |
| 1 | 2 | 4 |
| 1 | 2 | 3 |
| 1 | 1 | 3 |
| 1 | 1 | 4 |
| 2 | 2 | 4 |
| 2 | 2 | 3 |

| $\Pi'_{AC}(R \bowtie S)$ | |
|---|---|
| A | C |
| 1 | 4 |
| 1 | 3 |
| 1 | 3 |
| 1 | 4 |
| 2 | 4 |
| 2 | 3 |

| $(R \circ S)$ | |
|---|---|
| A | C |
| 1 | 3 |
| 1 | 4 |
| 2 | 3 |
| 2 | 4 |

Figure 1. An example of composition operation. ($\Pi'$ denotes a projection without duplicate elimination.)

Three operations: join, projection, and duplicate elimination are required to perform a composition operation. The join and projection operations are usually executed together, and then a duplicate elimination operation is followed to remove all redundant tuples. In general, an external sort is required to perform a duplicate elimination [3]. Both join and external sort operations are expensive in database systems because they incur a nonlinear I/O overhead. Fig. 1 shows an example of composition executed in the described steps.

Most conventional composition algorithms that follow the definition to process a composition in three separate steps can achieve savings on either join or duplicate elimination but not both. It it found in [8] that a direct execution that process a composition as a single primitive not only outperforms all conventional algorithms, it may even outperform one of its component operation, the join.

### 1.2 Join vs. Composition

It is a common experience that, by executing directly, a join operation can be more efficient than its component operation, the cross product. Similarly, we can see that a

direct composition algorithm may outperform its component operation, the join operation. This has a very significant implication to the real world DBMSs.

In order to reduce siginificant overhead of duplicate elimination, most real world DBMSs do not remove duplicate tuples in answering queries unless explicitly specified by users. For example, the user has to specify the "DISTINCT" modifier in a SQL query to request duplicate tuples be removed. However, it is an essential property of relational data model that every tuple in a relation must be unique. In other words, no duplicate tuple is allowed. This SQL flaw has been extensively criticized by the research community because it ruins the integrity of relational data model.

Not only join operation is implemented handicaply, composition operation is usually implemented in a compromised fashion too. Let's examine the following popular SQL statement:

```
SELECT  A, C
FROM    R, S
WHERE   R.B = S.B
```

As we can see that above operation is actually a *compromised composition*, a composition without duplicate elimination. Because composition has been considered a composite operation and the join operation has been studied extensively, this type of query is always executed as a join operation followed by a projection, while duplicated tuples are not removed. This compromise seriously ruins the integrity of relational data model. The invention of direct composition can eliminate this compromise. With comparable efficiency, duplicated tuples will not be kept in the composition result. Of course, the query must be rewritten into the following format if the SQL definition is not changed:

```
SELECT  DISTINCT (A, C)
FROM    R, S
WHERE   R.B = S.B
```

This paper will compare the performance of composition and join operations under various conditions. Section 2 will briefly describe various direct composition algorithms. Analytical and experimental comparison of join and direct composition will be given in Section 3 and 4 respectively.

# 2. Previous Work

## 2.1 Conventional Composition Algorithms

Most conventional composition algorithms take join and duplicate elimination as separate operations to achieve savings on either operation but not both. They can be divided into two categories: *join-saving*, and *sort-saving*. Join-saving based algorithms strive to reduce join operation cost while sort-saving based algorithms strive to reduce duplicate elimination cost [1,2,3,8].

A typical sort-saving based algorithm is the single-sided composition algorithm proposed by Agrawal et al., which eliminates duplicates at the same time it performs the join-project operation [2]. Typical join-saving based algorithms, such as sort-merge-join or hash-join algorithms, perform join and projection together first, and then remove the duplicates in a separate step [4,7]. An example is shown in Fig. 2.

| $R$ | A | B |
|---|---|---|
| $R_1$ | 1 | 1 |
| $R_2$ | 2<br>1 | 2<br>2 |
| $R_3$ | 2 | 3 |
| $R4$ | 3 | 5 |

| $S$ | B | C |
|---|---|---|
| $S_1$ | 1<br>1 | 4<br>3 |
| $S_2$ | 2<br>2 | 4<br>3 |
| $S3$ | 4 | 3 |

| $\Pi'_{AC}(R \bowtie S)$ | A | C |
|---|---|---|
| $\Pi'_{AC}(R_1 \bowtie S_1)$ | 1<br>1 | 4<br>3 |
| $\Pi'_{AC}(R_2 \bowtie S_2)$ | 2<br>2<br>1<br>1 | 4<br>3<br>4<br>3 |

| $(R \circ S)$ | |
|---|---|
| A | C |
| 1 | 3 |
| 1 | 4 |
| 2 | 3 |
| 2 | 4 |

Figure 2. An example showing a hash-join based composition algorithm.

## 2.2 Direct Composition

Direct composition algorithms [8,13] take another approach to achieve a better performance. They do not follow the definition of composition to process those three component operations in sequence. Instead, they execute composition operation directly as a single primitive such that no duplicate is generated in the intermediate steps. This is similar to what we did on join operations by executing join directly instead of following the formal definition to execute cross product, selection, and projection sequentially.

We use the example shown in Section 1.1 to illustrate some properties of the composition operation. Relation $R$ is first clustered on attribute $A$ and $S$ on $C$ as shown in Figure 3:

It is easy to see that the entire composition operation can be carried out by performing six smaller composition operations: $R_1 \circ S_1$, $R_1 \circ S_2$, $R_2 \circ S_1$, $R_2 \circ S_2$, $R_3 \circ S_1$, and $R_3 \circ S_2$. Thus, $R \circ S = \bigcup_{i,j} (R_i \circ S_j)$. The composition of $R_i$ with $S_j$ is referred to as *cluster composition*. It is interesting to see that $R_i \circ S_j$ produces either a single

| | R | |
|---|---|---|
| | A | B |
| $R_1$ | 1 | 2 |
| | 1 | 1 |
| $R_2$ | 2 | 2 |
| | 2 | 3 |
| $R_3$ | 3 | 5 |

| | S | |
|---|---|---|
| | B | C |
| $S_1$ | 1 | 3 |
| | 2 | 3 |
| | 4 | 3 |
| $S_2$ | 1 | 4 |
| | 2 | 4 |

Figure 3. The result of clustering $R$ on $A$ and $S$ on $C$.

tuple or null; thus, the computation effort is reduced to a binary choice, depending on whether $R_i.B$ and $S_j.B$ have any common value:

$$R_i \circ S_j = \begin{cases} (i,j) & \Pi_B(R_i) \cap \Pi_B(S_j) \neq \varnothing, \\ \varnothing & \text{otherwise.} \end{cases}$$

For instance:

$R_1 \circ S_1 = \{(1,3)\}$,
$R_1 \circ S_2 = \{(1,4)\}$,
$R_2 \circ S_1 = \{(2,3)\}$,
$R_2 \circ S_2 = \{(2,4)\}$,
$R_3 \circ S_1 = \varnothing$,
$R_3 \circ S_2 = \varnothing$, and
$R \circ S = \{(1,3), (1,4), (2,3), (2,4)\}$.

By making use of this property, direct composition algorithms achieve a better performance than the conventional implementations. In [8], several algorithms were presented. They are briefly described in the rest of this section.

### 2.2.1 Basic Direct Composition Algorithm

The Basic Direct Composition algorithm is as follows:

```
(1) clustering R on A
(2) clustering S on C
(3) foreach Ri {
(4)     foreach Sj {
(5)         if ( ΠB(Ri) ∩ ΠB(Sj) ≠ ∅ )
(6)         then  write (i,j)
            }
        }
    }
```

The overall performance of this algorithm depends on the efficiency of the fifth step, the cluster composition for $R_i$ and $S_j$. The main task in this step is to determine whether there is a common $B$ value in $R_i$ and $S_j$ or not. Whenever a match is found, the current cluster composition can be stopped immediately; thus, no duplicate will be generated. Its worst case happens whenever every cluster has to be read into memory entirely where the matched values are found in their last tuples or $R_i$ and $S_j$ have no common value.

To minimize the I/O overhead in this step, a good algorithm should strive to compare a matched $B$ value as early as possible. The most straightforward way is to sort (or hash) both clusters that are to be composed and to compare their contents tuple by tuple. An algorithm taking this approach is presented in [8].

### 2.2.2 The Hot-Spot Algorithm

The hot-spot composition algorithm takes one step further to cache the "hot-tuples" (which will be explained later) right in the memory, such that many $S$ clusters may even not be needed if their corresponding in-memory hot-tuples are already matched to some $B$ value in $R$.

Relation $R$ is first clustered on $A$ (e.g. using hash), and the number of occurrences of each $B$ value is recorded at the same time. Relation $S$ is then clustered on $C$. Any tuple of $S$ whose $B$ value has no occurrence in $R$ is discarded and is not written to any cluster of $S$.

For each cluster of $S$, the tuple whose $B$ value is most frequently referenced by $R$ is called the *hot-spot* tuple of that cluster. To utilize the property that $R_i \circ S_j$ is a binary choice problem such that the execution can be stopped immediately if a match on $B$ value is found, all hot-spot tuples are kept in memory and compared first. Thus, there is a very good chance of finding a match right in the memory in a cluster composition without reading the demanded $S$ clusters into memory. We say that a cluster composition $R_i \circ S_j$ has a *hot-spot hit* if $R_i$ matches the hot spot tuple of $S_j$, and a *hot-spot miss* otherwise. Note that we assume that memory is large enough to accommodate all hot-spot tuples since they can be obtained and maintained at very little cost. Further, more than one hot spot tuple from each $S$ cluster can be kept in memory if memory space is permitted. The composition of $R_i$ with $S_j$ is briefly described as follows:

```
read Ri into memory;
compose Ri with the hot-spot tuple of Sj;
if there is a match on B value,
      write (i,j) and stop;
if there is no match,
      compose Ri with Sj.
```

Fig. 4 is an example depicting this algorithm.

---

* All but one join query in Date's famous database textbook are this type of query [5].

| | R | |
|---|---|---|
| | A | B |
| $R_1$ | 1 | 2 |
| | 1 | 1 |
| $R_2$ | 2 | 2 |
| | 2 | 3 |
| $R_3$ | 3 | 5 |

| Freq. of $B$ in $R$ | |
|---|---|
| $B$ | freq. |
| 1 | 1 |
| 2 | 2 |
| 3 | 1 |
| 4 | 0 |
| 5 | 1 |

| | S | |
|---|---|---|
| | B | C |
| $S_1$ | 1 | 3 |
| | 2 | 3 |
| $S_2$ | 1 | 4 |
| | 2 | 4 |

| Hot Spot Tuples | | |
|---|---|---|
| | $B$ | $C$ |
| $S_1$ | 2 | 3 |
| $S_2$ | 2 | 4 |

| $(R \circ S)$ | |
|---|---|
| A | C |
| 1 | 3 |
| 1 | 4 |
| 2 | 3 |
| 2 | 4 |

Figure 4. An example showing the hot-spot Algorithm.

As we can see from the above example, the tuple (4,3) is removed from $S$ during the clustering phase since its $B$ value, 4, doesn't exist in $\Pi_B R$. Only four out of six cluster compositions, $R_1 \circ S_1$, $R_1 \circ S_2$, $R_2 \circ S_1$, and $R_2 \circ S_2$, are executed. All of them are executed without reading their corresponding $S$ clusters since they all have a hot-spot hit.

By implementing the composition operation as a primitive, much I/O can be saved by composing $R$ with in-memory hot-spot tuples first. Further, if a cluster of $S$ needs to be brought into memory, the tuple reading can be stopped immediately when a match on a $B$ value is found; not only are the remaining I/O operations, but the duplicate elimination is completely eliminated since no duplicate is produced in the first place. That is, this algorithm eliminates duplicate elimination entirely by not producing any duplicate.

# 3. Analytical Comparison

In this section, we compare join and direct composition algorithms analytically. The details of the experimental results will be presented in next section. We only consider the I/O cost, which is the dominant cost in most database applications. The I/O cost measured is the number of tuples read and written.

## 3.1 Join Operation

There are many joining algorithms such as nested-loop, sort-merge, and hash joins [4,8]. Basically, join is a quadratic complexity operation since every tuple $R$ has to compare with every tuple of $S$. One good way to improve its efficiency is to cluster (i.e. hash) both $R$ and $S$ based on the joining attribute, $B$, such that only the tuples in the same cluster (i.e. bucket) need to be compared. The complexity is reduced from $|R| \times |S|$ to $\sum |R_i| \times |S_j|$. Sort-merge and hash joins are typical examples. Since hash join is one of the best joining algorithms, we choose to compare with hash join.

## 3.2 Assumptions

Because join and composition are two different operations, it is not easy to have a strictly fair comparison. In this paper, most assumptions are made in favor of join when a compromise is necessary. Assume that the source relations $R$ and $S$ are initially resident in disk, and that the final results need to be written back to disk. Let $|T|$ and $|T'|$ be the respective sizes of the join and composition results, the duplication factor $f$ is then defined as $|T|/|T'|$. Let $|M|$ be the number of tuples that can fit into memory. We further assume that each cluster generated in the hash operation of a hash join can fit into the memory. Therefore, the joining operation between two clusters can be performed right in the memory. No I/O overhead is incurred. This assumption may not be accurate. However, the performance analysis obtained based on this assumption represents an upper bound of the hash join algorithm.

## 3.3 Analytical Performance Estimation

Assuming the average number of pages each cluster (hash bucket) has is $\rho$, then the total I/O cost for hash-join operation, then, is:

cost of clustering $R$ and $S$,
+ cost of joining $R$ and $S$ clusters,
+ cost of writing $T$,

$$= 2|R| + 2|S| + |R| + |S| + \lambda_1 \rho^2 (|R|/|B| \times |S|/|B|) + |T|$$

$$= 3|R| + 3|S| + |T| + \lambda_1 \rho^2 (|R|/|B| \times |S|/|B|),$$

where $\lambda_1$ is a filtering parameter ($0 \le \lambda_1 \le 1$), indicating the ratio of the cluster pairs that are needed to be joined to the total cluster pairs. It is one if all cluster pairs must be joined. In the clustering phase, each relation has to be read once for clustering and each cluster needs to be written back to disk after clustering. Therefore, the cost for clustering is $2|R| + 2|S|$. Relevant clusters ($\Pi_B R_i \cap \Pi_B S_j \neq \emptyset$) are joined together cluster by cluster. The cost of joining $R$ clusters and $S$ clusters depends on the size of each cluster and the size of memory. It is a function of $|R|/|B| \times |S|/|B|$. $\rho$ is an empirical coefficient between zero and $|B|^2$. In the best case, each relation is read into memory only once in the cluster joining phase.

The total I/O cost for the hot-spot compositions is:
cost of clustering $R$ and $S$,
+ cost of reading $R$ clusters,
+ cost of reading unmatched $S$ clusters,
+ cost of writing out result relation,
$= 2|R| + 2|S|$
$+ |R|$
$+ \lambda_2(1-\mu)(|R|/|M|)\cdot|S|$
$+ |T|/f$
$= 3|R| + (2 + \lambda_2(1-\mu)(|R|/|M|))\cdot|S| + |T|/f$,

where $f$ is the duplication factor (thus, $|T|/f$ is the size of composition result), $\mu$ ($0 \le \mu \le 1$) is the *hit ratio*, the average ratio of matched clusters to the total $S$ clusters, per $R$ cluster; and $\lambda_2$ ($0 < \lambda_2 \le 1$) is the average fraction of a $S$ cluster that is read. The total I/O cost for cluster composition, then, is: $\lambda_2(1-\mu)(|R|/|M|)\cdot|S|$.

The best case happens when the hot-spot hit ratio, $\mu$, is one. In other words, the hot-spot tuples of all $S$ clusters match to every $R$ cluster. A small $\lambda_2$ value can also lead to good performance. One good way to reduce $\lambda_2$ is to sort $S$ clusters according to the the descending order of occurring frequency of $B$ value in $R$. Nevertheless, further research is needed to balance the extra overhead for sorting and the saved overhead in I/O reduction.

As we can see from above analysis that the performance of direct composition is comparable with hash join. The best case of hot-spot direct composition outperforms the best case of hash join by a small margin of $|S|$. This is because all $S$ clusters will not be read into memory for hot-spot hit. Nevertheless, since many parameters such as $\mu$, $\lambda_1$, $\lambda_2$, $\rho$, $f$, are all empirical data, it is necessary to execute these algorithms in real databases to obtain a more realistic comparison.

# 4. Experimental Performance Comparison

In this section, the hot-spot composition algorithm is evaluated against the hash-join algorithm based on the simulated database defined by the Set Query Benchmark [6]. Different from the TPC benchmark, the Set Query Benchmark is specially designed for testing the performance of databases in decision support environments, rather than OLTP environments. We envision that the composition operation is the most popular in decision support environments.

## 4.1 Experiments

In this experiment, we focus on the sensitivity of the following three parameters on the overall performance.

1. size of relations;
2. cardinality of the domain of attribute $B$;
3. cardinality of the domain of attribute $A$, and $C$;
4. duplication ratio ($f$), the tuple ratio of joining result and composition result;

The experiment is running on a lightly loaded HP755 system running Unix operating system. The overall execution time is measured.

### 4.1.1 Data Generation

The two operand relations, $R$ and $S$, are subsets of the SET database. The values of attributes $A$, $B$, and $C$ are drawn from the following four different integer domains, 1-1000, 1-100, 1-25, and 1-10 as shown in table 1.

Table 1. Domains of simulated relations.

| Domain Name | Range |
|---|---|
| K1000 | 1-1000 |
| K100 | 1-100 |
| K25 | 1-25 |
| K10 | 1-10 |

In each experiment, $R$ and $S$ are first defined by selecting $A$, $B$, and $C$ attributes from the defined domain (K1000, K100, K25, K10). Each defined relation is materialized by randomly drawing values from the domain of its attributes and is materialized into seven different sizes: 1000 tuples to 7000 tuples stepped by 1000 tuples. (Because of duplicated tuples must be removed in the generated relations, the actual size of a generated relation may be fewer than the designed size.) To obtain a statistical stable performance measurement, each pair of relations is generated 10 times with different random seeds and is fed into hash join and composition operations. We choose to use uniform distribution to obtain a performance lower bound of the hot-spot algorithm which prefers skewed distributions to the uniform distribution. Finally, all possible definitions of $R$ and $S$ are simulated to observe the sensitivity of the desired parameters in various conditions.

## 4.2 Experimental Results

### 4.2.1 Sensitivity to the Size of Relations

As we already know, the complexity of hash-join is nonlinearly proportional to the size of joining relations. Hot-spot composition performs better than hash-join when the size of relations are large.

Further, it is interesting to see that, like joining algorithm, the hop-spot composition prefers the smaller relations to be the first relation and the larger one to be the second. This is because $R$ clusters are always read into memory while $S$ clusters may not be the case. Further, a smaller $R$ cluster can prevented nested loop

for composing two clusters from happening. The result is shown in Figure 5.

### 4.2.2 Sensitivity to the Joining Attribute

Both hash-join and hot-spot composition are sensitive to the cardinality of the domain of joining attributes. Hash-join prefers a larger cardinality because it reduces the size of each cluster. (In general, joining many pairs of smaller clusters is more efficient than joining fewer pair of larger clusters due to the linearization of quadratic complexity.)

On the other hand, hot-spot composition prefers a smaller cardinality because it leads to a denser *B* attribute and has a larger chance to find a match on *B* value between two composing clusters. It is less sensitive to the cardinality of joining domain as compared to that of non-joining domain as shown in next section. This is depicted in Figure 6(a) and 6(b).

Further, it is interesting to see that the execution time of hot-spot algorithm may not always monotonically increase with the size of original relations. This is beacuse, in some certain range, the larger the relation, the higer the spot-spot hit ratio. In the future, we will further investigate this phenomenon.

### 4.2.3 Sensitivity to the Non-Joining Attribute

Both hash-join and hot-spot composition are very sensitive to this parameter. Both of them prefer a smaller cardinality. For hash-join, a smaller non-joining attribute size will reduce the joining overhead in the bucket joining process. For hot-spot composition, it needs to perform a composition for every possible pair of *R* and *S* clusters. A smaller number of clusters lead to fewer number of cluster compositions. Different from join operation where the time complexity is quadratically proportional to the size of clusters, the execution time of cluster composition does not directly proportional to the size of clusters since execution can be stopped immediately after finding a match of *B* value on both clusters. This result is shown in Figure 7.

### 4.2.4 Sensitivity to the Duplication Factor

As we expected, the composition performs better when the duplication factor is high. This is consistent with the results found in [8]. There is no general threshold on which the composition is always better than join. Further analysis and simulation experiments are needed to have more useful conclusion.

### 4.2.5 Comparison of Join and Composition

In average, the composition is three times slower than hash-join over all 302 simulated cases. About one third out of the simulated cases (102/302), the performance of hot-spot composition is within 150% of the hash-join algorithm. About one quarter (71/302) cases, the composition are better. The following cases are observed in the experiment:

1. Hot-spot composition performs better when the domain cardinality of attribute *A* is small and that of attribute *C* is large.
2. Hot-spot composition performs better when the size of relations are large.
3. Hot-spot composition performs better when the duplication factor is high.

## 5. Concluding Remarks

The composition operation is an important primitive operation in deductive databases. Direct composition algorithms that process composition as a single primitive have been proved to be more efficient than the conventional approaches that process composition as three separated steps: join, projection, and duplicate elimination. In this paper, it has been shown that direct composition not only produce no duplicate, it may even outperform its component operation, the join. Within a very broad range, the performance of direct composition is comparable with join operation.

Because join and composition are different operations, it is not appropriate to replace join with composition. Instead, many join operations in the real world are actually compromised compositions. They are implemented as a join to reduce I/O overhead for duplicate elimination. The results presented in this paper will encourage practitioner stop using join operation as a compromised composition. Not only the performance may be better, the integrity of relational data model can be preserved.

**Reference**

1. Agrawal, R., "ALPHA: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proc. of the 3rd Int'l Conf. on Data Engineering*, Feb. 1987.
2. Agrawal, R., Dar, S. and Jagadish, H. V., "Composition of Database Relations", *Proc. of the 5rd Int'l Conf. on Data Engineering*, Feb. 1989.
3. Bitton, D. and DeWitt, D. J., "Duplicate Record Elimination in Large Data Files", *ACM Trans. of Database Systems*, vol. 8, no. 2, June 1983, pp. 255-265.
4. Choi, H. K. and Kim, M., "Hybrid Join: An Improved Sort-Based Join Algorithm", *Information Processing Letters*, vol. 32, no. 2, July 1989, pp. 51-56.

5. Date, C. J., "An Introduction to Database Systems", vol. 1, 1986.

6. Gray, J., "The Benchmark Handbook", 1991, pp. 209-245.

7. Kitsuregawa, M., Tanaka, H. and Moto-oka, T., "Application of Hash to Database Machine and its Architecture", *New Generation Computing*, vol. 1, no. 1, 1983.

8. Lien, Y. N. Lien, "Direct Composition Algorithms", *Accepted to appear in Journal of Information Science and Engineering*, JISE-84-31.

9. Lu, W. Y. and Lee, D. L., "The Design of a Recursive Query Processor", *Proc. Int'l Conf. On Data Base and Expert Systems Applications*, Aug. 1990.

10. Lu, W. Y., Lee, D. L., Hsu, I. M. and Wei, S. S., "Minimizing Search Redundancy in Processing Bounded Recursions", *Proc. CIPS Edmonton '90 Information Technology Conference*, Oct. 1990..

11. Rosenthal, A., Heiler, S., Dayal, U. and Manola, F., "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", *Proc. of Int'l Conf. on Management of Data*, May 1986.

12. Valduriez, P. and Boral, H., "Evaluation of Recursive Queries Using Join Indices", *Proc. of the 1st Int'l Conf. Expert Database Systems*, April 1986.

13. Wei, Shu-Shang, Lien, Yao-Nan, Lee, Dik and Lai, T. H., "Hot-Spot Based Composition Algorithm", *Eighth International Conference on Data Engineering,*, Feb. 3, 1992, pp. 48-55.