JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 12, 547-565 (1996)

Direct Composition Algorithms

YAO-NAN LIEN
Department of Computer Science
National Chengchi University
Taipei, Taiwan 116, R.O.C.

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 in database systems 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. In this paper, we analyze the characteristics of the composition operation and find that executing the composition as a single primitive may require much less I/O overhead. Several algorithms taking this approach are proposed to achieve savings on both operations. Several experiments have been conducted showing that this new approach outperforms other approaches under almost every condition.

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 Π 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 [7, 8]. Much work has been done on processing of transitive closure and linear recursion [1, 9, 11]. 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.

Three operations, join, projection, and duplicate elimination, are required to perform a composition operation. The join and profection operations are usually executed together, referred to as a *join-project* operation in this paper, 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.

Received June 28, 1995; revised March 13, 1996.
Communicated by Wei-Pang Yang.

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

Fig. 1. An example of composition operation. (Π' denotes a projection without duplicate elimination.)

1.2 Previous Work

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 the join operation cost while sort-saving based algorithms strive to reduce the duplicate elimination cost.

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. To save time in the join operation, both relations are clustered based on the joining attribute such that only pairs of clusters within the same range need to be joined. (The clustering can be done by either sorting or hashing.) Therefore, their join operations can be performed very efficiently. However, they suffer from an expensive duplicate elimination because duplicates are scattered over all clusters after the join-project operation. Since the result is usually too large to fit into memory, it is necessary to perform an external sort to eliminate duplicates. The number of I/O operations required by an external sort is about $2|T| \cdot \log_2(|T|/|M|)$, where $|T|$ is the size of the intermediate result generated by the join-project operation, and $|M|$ is the number of tuples that can fit into memory [5]. (Actually, if a b-way merge-sort is performed, the external sort only costs $2|T| \cdot \log_b(|T|/|M|)$ I/O operations [4]; however, it is still very expensive.) An example is shown in Fig. 2.

Note that although an index on the joining attribute may be available in real world databases such that the join operation can be performed very efficiently, it may not be always available in many cases. For example, in a deductive database, the relations to be composed may actually be the intermediate results of some logic computation and, thus, may have no index available for join.

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]. Relation R is first divided into a sufficient number of clusters based on attribute A. The clusters are carefully chosen such that the potential size of each bucket (i.e., the result of a cluster of R join-projected with S) is small enough to fit into memory. Since all resulting tuples of the same A value reside in the same bucket, duplicates can be eliminated

R		
	A	B
R_1	1	1
R_2	2	2
	1	2
R_3	2	3
R_4	3	5

S		
	B	C
S_1	1	4
	1	3
S_2	2	4
	2	3
S_3	4	3

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

$R \circ S$	
A	C
1	3
1	4
2	3
2	4

Fig. 2. An example showing an hash join based composition algorithm.

within the bucket on-the-fly when performing the join. (In general, duplicate elimination for a bucket can be done within the memory and, thus, is relatively inexpensive as compared to the I/O cost.) A detailed description is given in Appendix 1. The join operation of this algorithm is not very efficient because S has to be read into memory repeatedly. For each cluster of R in memory, the entire S is read once. Since, there are about $|R|/|M|$ clusters in R, the total cost of reading S is about $|S| \cdot |R|/|M|$.

One improvement to this algorithm is to cluster S on B as well [2]. Then, for each R cluster, say R_i, there is no need to fetch the entire S into memory; instead, S clusters can be brought into the memory selectively be matching the B values in R_i and in S clusters. However, because B values in each R_i are arbitrary, most clusters of S may still need to be brought into memory for join.

1.3 Direct Composition

After careful analysis of the previous algorithms which take join and duplicate elimination as separate operations, we can easily find that it is difficult for this type of algorithm to achieve savings on both operations. An efficient join execution requires that both R and S be clustered on B while a duplicate elimination requires them to be clustered on (A, C). After close examination of the composition, we find that it is not necessary to follow the definition to process those three component operations in sequence. A direct implementation may result in a more efficient composition by not generating duplicates 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.

In this paper, we analyze the characteristics of the composition operation and propose several algorithms that execute composition as a single primitive to achieve greater reduction in the I/O cost. Several experiments have been conducted which show that our approach outperforms other algorithms under almost every condition.

The rest of the paper is organized as follows. Section 2 presents our direct composition algorithms; Section 3 analyzes these algorithms; and Section 4 shows comparisons and experimental results. Finally, concluding remarks are

given in Section 5.

2. DIRECT COMPOSITION ALGORITHMS

2.1 Characteristics of Composition

We use the example shown in Section 1 to illustrate some properties of the composition operation. Relation R is first clusterd on attribute A and S on C as follows:

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

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 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:

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

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 utilizing this property, direct implementation of the composition operation would be more efficient than other conventional implementations. In the rest of this section, a basic algorithm will be presented first; then, several more advanced

algorithms will be shown.

2.2 Basic direct Composition Algorithm

In this algorithm, relations R and S are first clustered based on A and C, respectively; then, a nested loop is used to perform all the cluster compositions. If a match on the joining attribute is found in a composition, the operation stops immediately. Therefore, no duplicate is created. A detailed description follows:

```
(1) clustering R on A
(2) clustering S on C
(3) foreach R_i {
(4)     foreach S_j {
(5)         if (Π_B(R_i) ∩ Π_B(S_j) ≠ ∅)
(6)         then write (i, j)
            }
        }.
```

As mentioned earlier, clustering can be easily done using conventional techniques such as hashing [6]. The only restriction is that each R (or S) cluster must have a single A (or C) value. If, for any reason, multiple values have to be hashed into the same cluster, the algorithm can be easily modified by executing a rehash when two clusters are to be composed [10].

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. An equally bad situation is where R_i and S_j have no common value so that both R_i and S_j have to be read entirely.

There are many ways to minimize the I/O overhead in this step. A good algorithm should focus on comparing a matched 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. Sorting can be done either in natural order (ASCII order) or artificial order (customized order). An example of artificial order is to sort according to the occurring frequency of B values in relation R. The following sections first present a sort-match direct composition algorithm, which is an extension of the basic direct composition algorithm, to sort the clusters of both relations before matching. (A similar algorithm that uses hashing, instead of sorting, for a lower I/O overhead can be easily derived from this 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.

2.3 Sort-Match Direct Composition Algorithm

This is an extension of the basic direct composition algorithm where all steps are identical except the 5th step, in which a composition between two clusters is performed. The clusters that are to be composed are first sorted on their respective joining attributes and then are compared tuple by tuple as shown in the following algorithm:

```
(1)  clustering R on A
(2)  clustering S on C
(3)  foreach S_j { sort S_j in ascending order of B }
(4)      foreach R_i {
(5)      sort R_i in ascending order of B
(6)              foreach S_j {
(7)                      set iptr pointing to the first tuple of R_i
(8)                      set jptr pointing to the first tuple of S_j
(9)                      while (both iptr and jptr are not null) {
(10)                         if (iptr → B == jptr → B)
(11)                         then {
(12)                             write (i, j)
(13)                             break to (6)
                             }
(14)                         elseif (iptr → B < jptr → B)
(15)                         then iptr ++
(16)                         else jptr ++
                         }
                 }
     }
```

As we mentioned earlier, sorting can be performed in any order. This algorithm assumes ascending order and can be easily modified to accommodate other sorting order. This algorithm is very similar to the sort-merge joining algorithm, where both clusters are sorted first and then compared and merged one by one. The only difference is that this algorithm stops the comparison immediately when a match is found while the sort-merge join needs to complete the entire comparison procedure.

2.4 The Hot-Spot Algorithm

The hot-spot composition algorithm takes one step further to cache hot-spot tuples, which will be explained later, in memory to eliminate I/O operations where the values to be matched can be found right in memory [12].

Relation R is first clustered on A (i.e. using hash), and the number of occurrences of each B value is recorded at the same time. Relations 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. Hence, every tuple in a cluster of S shares the same B value with some tuple in R. Many irrelevant tuples in S can be filtered

out. In the extreme case, the entire S can be discarded such that the following composing phase is not even needed. Other algorithms do not make use of this advantage to reduce the I/O cost because they are not aware of this feature.

For each cluster of S, the tuple whose B value is most frequently referred 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 and the fact that execution can be stopped immediately if a match on B value is found when composing R_i with S_j, 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 during 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. 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 R_i into memory;
compose R_i with the hot-spot tuple of S_j;
if there is a match on B value, write (i, j) and stop;
if there is no match, compose R_i with S_j.

A detailed description is given in Appendix 2. Note that we assume again that each R cluster can be fit into memory entirely, so there is no need to execute a nested loop join, which is a component of cluster composition. If this is not the case, then the algorithm should be modified to include a sort-merge or hush-join process in joining R_i with S_j. Fig. 3 is an example depicting this algorithm.

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

Fig. 3. A 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 $R_2 \circ S_2$, produce results. 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

eliminated 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 duplicated tuple.

Besides I/O reduction, this algorithm has some other unique features:

1. Since the order of the composed results is determined by the order in which $R_i \circ S_j$ are executed, the results can be produced in any order (i.e. in ascending or descending order) by executing $R_i \circ S_j$ in the desired order with no extra cost. The single-sided composition algorithm needs an internal sort, and other algorithms need an external sort to order their results.
2. Its performance is sensitive to the distribution of values in the joining attribute B. In the clustering phase, many irrelevant tuples can be filtered out, reducing somewhat the I/O cost in the composing phase. In the composing phase, each hot spot tuple in memory serves as a surrogate of its corresponding cluster in disk and is chosen in such a way that it can match as many clusters of R as possible. As we have shown, the corresponding $R_i \circ S_j$ can be done without reading the entire R_i if the hot spot tuple of S_j matches any B value in R_i. Therefore, the distribution of the values in the joining attributes of both relations has a significant impact on the performance of this algorithm. In general, this algorithm prefers skewed distributions to the uniform distribution if R.B and S.B are skewed toward the same direction.
3. As we mentioned earlier, the performance can be easily improved by accommodating more hot-spot tuples into memory (more than one hot spot tuple in each S cluster). In this way, the chance of reading an S cluster and, thus, the I/O cost can be further reduced.

3. PERFORMANCE ANALYSIS

In this section, we analyze the above composition algorithms. The details of the experimental results will be presented in next section. The algorithms analyzed in this section are the two-phase hash-join based composition, the basic model of single-sided composition, the sort-match direct composition, and the basic model of hot-spot composition algorithms.

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 composition results before and after duplicate elimination. The duplication factor f is defined as $|T|/|T'|$. Let $|M|$ be the number of tuples that can fit into memory. 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.

The total I/O cost for two-phase hash-join based composition is:

cost of clustering R and S	$2\|R\| + 2\|S\|$
+ cost of joining R and S clusters	$\|R\| + \|S\|$
+ cost of writing T	$\|T\|$
+ cost of external sort	$2\|T\| \cdot \log_b(\|T\|/\|M\|)$
+ cost of duplicate elimination	$\|T\|$
+ cost of writing out result	$\|T\|/f$
$= 3\|R\| + 3\|S\| + \|T\|(2 + 1/f + 2\log_b(\|T\|/\|M\|))$.	

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 of 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. Hence, each relation is read in only once in the joining phase. The duplicates in T are assumed to be removed by using the b-way external sort-merge algorithm, which requires an I/O cost of $2|T|\log_b(|T|/|M|) + |T| + |T|/f$ [4].

The total I/O cost for the basic model of single-sided composition is:

cost of clustering R	$2\|R\|$
+ cost of reading R clusters (comp. phase)	$\|R\|$
+ cost of reading S clusters (comp. phase)	$\|S\| \cdot \|R\|/\|M\|$
+ cost of writing out result	$\|T\|/f$
$= 3\|R\| + \|S\| \cdot \|R\|/\|M\| + \|T\|/f$.	

R is clustered first, which requires $2|R|$ effort, and is then read into memory cluster by cluster, which takes $|R|$ tuples. For each cluster of R, the entire S is read once, so the total cost of reading S is about $(|R|/|M|) \cdot |S|$. Duplicates are eliminated on-the-fly; hence, only non-duplicate resulting tuples are written to disk, which costs only $|T|/f$.

The total I/O cost for the sort-match and hot-spot compositions is:

cost of clustering R and S	$2\|R\| + 2\|S\|$
+ cost of sorting S clusters (sort-match only)	$2\|S\|$
+ cost of reading R clusters (comp. phase)	$\|R\|$
+ cost of reading unmatched S clusters	$\lambda(1-\mu)(\|R\|/\|M\|) \cdot \|S\|$
+ cost of writing out result relation	$\|T\|/f$
$= 3\|R\| + (2 + 2\mathcal{P} + \lambda(1-\mu)(\|R\|/\|M\|)) \cdot \|S\| + \|T\|/f$.	

The first and the third items are straightforward. The second item is only for the sort-match direct composition algorithm, which is the cost of sorting all S clusters. The fourth item is the total I/O traffic for bringing the tuples of S clusters that need to be brought into memory, where μ is the *hit ratio*, the average ratio of matched clusters to the total S clusters, per R cluster; and λ is the average fraction of a S cluster that is read. The total I/O cost for cluster composition is, then, $\lambda(1-\mu)(|R|/|M|) \cdot |S|$. The value of $\mathcal{P}$ is 1 for sort-match direct composition and is 0 for hot-spot composition.

From the above three equations, it can be easily seen that whether the hot-spot algorithm can outperform the other two algorithms mainly depends on its

cost of reading unmatched S clusters. Composing hot-spot tuples first can significantly reduce the number of unnecessary I/O operations. In the next section, we will show that this cost is smaller than that of the external sort in the hash-join based algorithm and that of the composing phase in the single-sided algorithm under almost all conditions. That is, the hot-spot composition algorithm outperforms the other two algorithms under almost all conditions.

4. EXPERIMENTAL PERFORMANCE COMPARISON

In this section, the proposed hot-spot composition algorithm is evaluated against the hash-join based and single-sided composition algorithms. One other algorithm, *multiple-bucket* composition, is also discussed in order to illustrate the *best-case* performances of the hash-join based algorithms. The sort-match direct composition algorithm is also evaluated.

4.1 The Multiple-Bucket Composition Algorithm

The *multiple-bucket composition* algorithm is a variant of the hash-join based composition algorithms. Both R and S are first clustered on B with the same set of sub-domains (i.e. using the same hash function). Hash join is then performed together with the projection. Each join-projected result is further hashed on A to decide into which bucket it should be placed. The hash function is carefully chosen so that the potential size of each bucket is small enough to fit into memory. After the join-project operation, the tuples in each bucket need to be retrieved again in order to eliminate the duplicates. However, external sort is avoided because all duplicates of the same value are placed in the same bucket which is small enough to fit into memory. In fact, the bucket into which a join-projected result should be placed can be predetermined if each cluster of R is further clustered on A before composition is performed. Since the implementation is straightforward, the details are not discussed in this paper. This algorithm has the efficiency of hash join but without any need to eliminate the expensive external sort by using many small buckets to store different groups of duplicates when join is carried out. Obviously, its performance is an upper bound of that of hash-join based composition algorithms.

4.2 Experiments

In this experiment, we focus on the sensitivity to the following three parameters in the experiment:

1. scaling factor (s), the ratio of the composing result with duplicates not removed to the relation R in their cardinalities, $|T|/|R|$;
2. duplication ratio (f), the ratio of the composing result with and without duplicate, $|T|/|T'|$; and
3. memory ratio, the ratio of memory size to the size of relation R, $|M|/|R|$.

Both R and S are assumed to have 2000 tuples. The hit ratio (μ) is set to 0.5 (μ is defined in Section 3). R and S are synthesized for each given set of s, f, and μ. Values in attributes A, B, and C are assumed to be integers and to be uniformly distributed. Only I/O traffic, instead of the real CPU response time, is measured. We choose to use uniform distribution to avoid any potential bias toward the hot-spot algorithm, which prefers skewed distributions to the uniform distribution.

For simplicity, each pair of composing clusters in the above algorithms is assumed to be small enough to reside in memory at the same time. However, whenever this assumption is not valid for certain conditions such as data skewing or insufficient memory space, a nested loop is needed to carry out the composition.

The external sort used in this experiment is a b-way merge-sort, where the value of b depends on the given memory size and the limit on the number of files that can be opened simultaneously (which is 64 in our system).

4.3 Experimental Results

Three experiments were conducted on a Sun SPARC SLC machine, under C and UNIX. The first experiment was conducted by holding the duplicate ratio to 10 but varying the scaling factor. The memory ratio was also changed in order to observe the effect of this change on performance. The results when the memory ratio was 0.1 are depicted in Fig. 4-a.

The multiple-bucket and hash-join based composition algorithms outper-

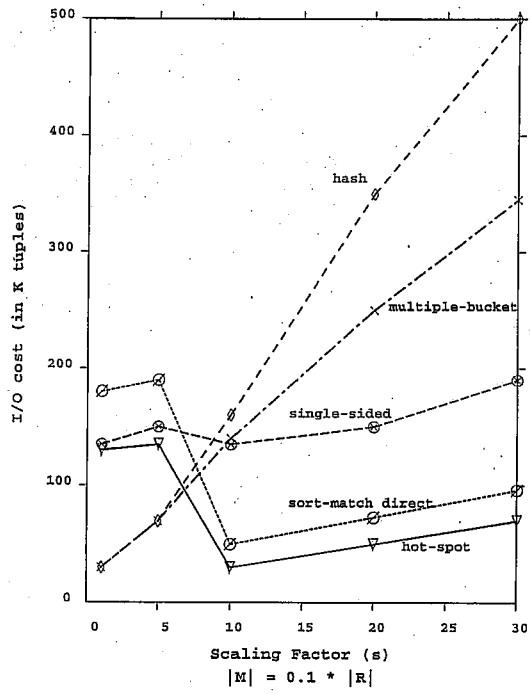

Fig. 4-a. The effects of scaling factor when f = 10.

formed others when the scaling factor was small, but they had the worst performance when the scaling factor was very large. This is because, when the composing result was small, the cost of duplicate elimination was nominal, so the overall cost was dominated by the join operation; thus, the join-saving based composition algorithms outperformed the others. On the other hand, they performed poorly when the scaling factor was large because they incurred more I/O traffic for duplicate elimination. Since the hot-spot algorithm doesn't produce any duplicates and only accesses a fraction of S (which depends on the hit ratio), it had the best performance among all five algorithms almost everywhere as shown in Figs. 4-a and 4-b. Even the sort-match direct algorithm, whose performance is a lower bound of the hot-spot algorithms presented in this paper, outperformed the others under almost all conditions for the same reason.

The second experiment was designed to observe the impact of duplication

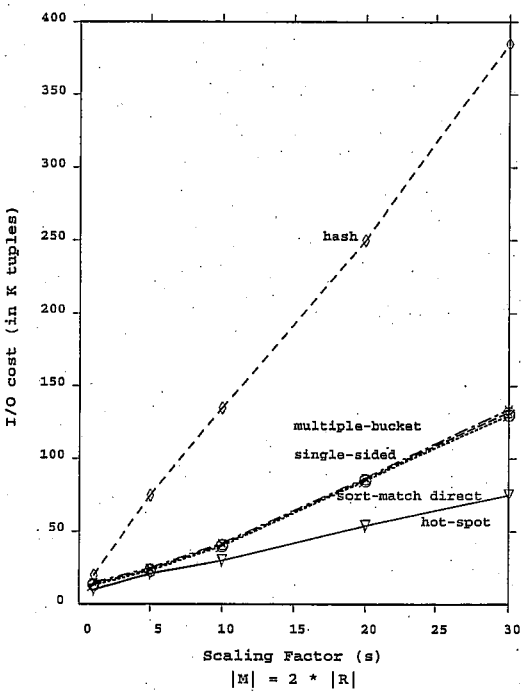

Fig. 4-b. The effects of scaling factor when f = 10.

ratios when the scaling factor was fixed at 10. The results are depicted in Fig. 5. Figs. 5-a and 5-b show that all the algorithms except the single-sided one were only slightly sensitive to the duplication ratio. The single-sided composition was very sensitive to the duplication ratio when it was smaller than 8. This is because not many duplicates could be removed in the composing phase, so the memory space was too small to support complete on-the-fly duplicate elimination. Thus, an extra external sort was still needed to eliminate the remaining duplicates. When the memory size was large enough to support on-the-fly duplicate elimina-

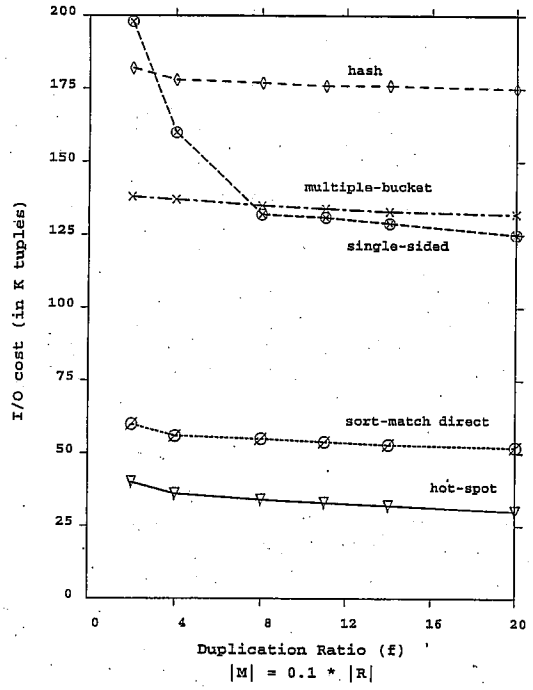

Fig. 5-a. The effects of the duplication ratio when s = 10.

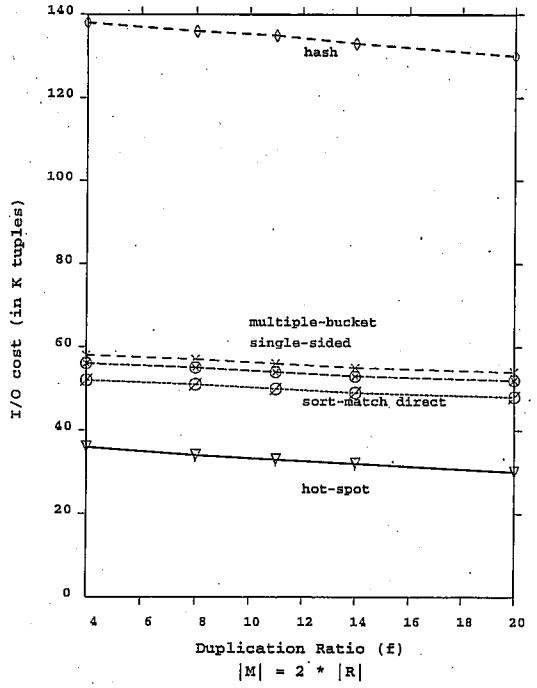

Fig. 5-b. The effects of the duplication ratio when s = 10.

tion at lower duplication ratios, the problem disappeared as shown in Fig. 5-b. Nevertheless, direct composition algorithms were still better than conventional algorithms even in the worse case, the sort-match direct algorithm.

Finally, sensitivity to one of the most important resources, memory, was investigated in the last experiment. The scaling factor and the duplication ratio were both held at 10. The results are shown in Fig. 6.

When memory size was very small, duplicates could hardly be eliminated on-the-fly, so the cost of a nested loop was very significant. Hence, the single-sided algorithm had the worst performance because it required not only an extra external sort in order to sort the join-projected results, but also a costly nested loop in order to perform composition. Thus, its performance deteriorated very significantly as can be seen in Fig. 6-a. The sort-match direct algorithm was also affected by small memory size. This is because an extra external sort was also needed for each pair of composing clusters. However, it was still better than the single-sided algorithm because the latter needed to sort a much larger join-projected result. When the memory size became larger, all the algorithms, except the hash-join based algorithm, took advantage of this very promptly (see Fig. 6-b). (The curve of the hash-join based algorithm only moved down significantly when the memory size was large enough to avoid a costly external sort; this part of the graph is not shown in these figures.)

The curves of the single-sided and multiple-bucket algorithms go down substantially when the memory size varies from 10 to 100 percent of the source relations. This also has to do with duplicate elimination. In the above range, most of the duplicates for both algorithms could be eliminated without an extra external sort; therefore, their costs dropped very significantly. On the other hand, the sort-match direct and hot-spot algorithms were almost insensitive to memory size when the memory ratio was larger than 0.06. This can be seen from the fact that the two algorithms did not produce any duplicates. Note that the hot-spot algorithm was again the best when the memory size was large, and it still had good performance even when the memory size was small.

From the results of the above experiments, the hot-spot algorithm seems to be the best under almost every condition. Nevertheless, it requires a single A value and C value in every R and S cluster. In order to fulfill this requirement, each cluster may contain only a few tuples and lots of clusters are created. This may affect the response time because, if each cluster is stored as a single file, then much CPU time is required to manage these files. This problem can actually be solved by allowing multiple A and S values in a cluster and by then using a more sophisticated data structure to organize them within each cluster.

5. CONCLUSION

The composition operation is an important primitive operation in deductive databases. It consists of a costly natural join operation and a costly external sort operation. Most conventional composition algorithms take these as separate operations and strive to achieve savings on either operation, but not on both. Just as an efficient join operation can be executed directly as a single primitive, we

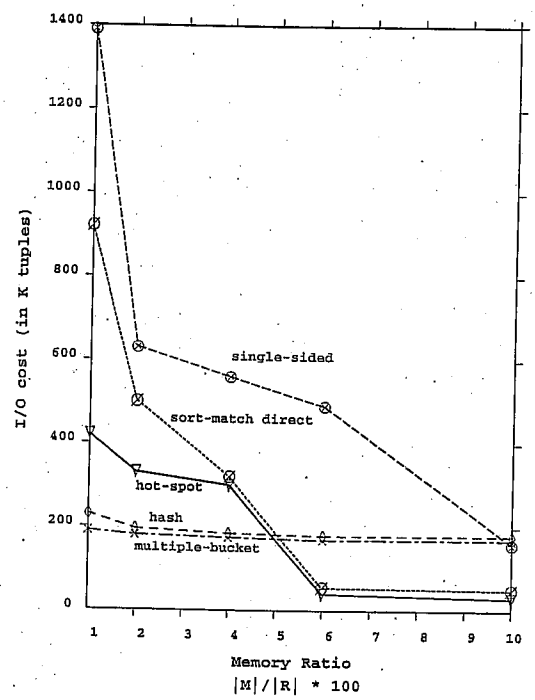

Fig. 6-a. The effects of the memory ratio when f = 10 and s = 10.

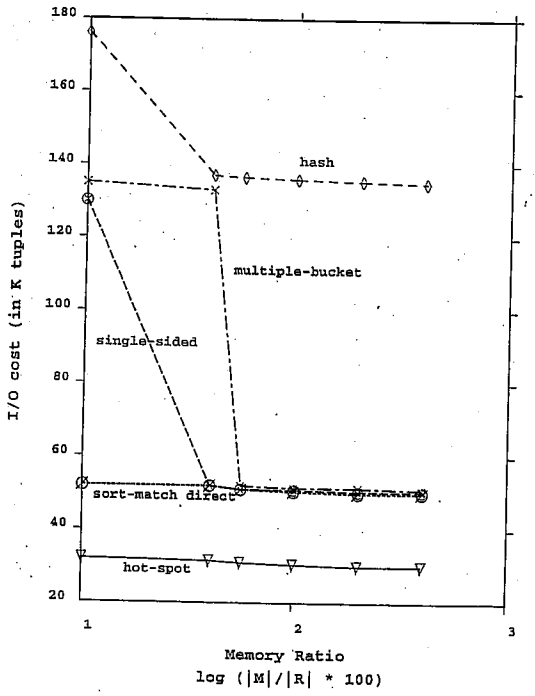

Fig. 6-b. The effects of the memory ratio when f = 10 and s = 10.

have shown that executing composition directly as a single primitive may require much less I/O overhead. Several algorithms taking this approach have been proposed to achieve greater savings in I/O overhead.

These algorithms achieve greater savings in I/O overhead by not generating any duplicate in the intermediate steps. Not only is the cost of duplicate elimination is eliminated, but the cost wasted in generating duplicates is also eliminated. The hot-spot algorithm goes one step further to reduce more I/O overhead by storing the tuples of one relation that are most likely to be matched to another into main memory so that many compositions can be performed right in memory.

It can be seen from our experiments that the hot-spot algorithm outperforms other algorithms under almost every condition. Even in its worst case (which is depicted by the sort-match direct algorithm), its performance is still close to that of the single-sided composition algorithm.

Finally, it is worth mentioning that, although newer algorithms use memory more elegantly to reduce the number of I/O operations, there is still plenty of room for improvement since the memory size available to a DBMS is getting larger and larger. It would be a reasonable approach to trade more memory for further reduction in the number of I/O operations.

APPENDIX 1. THE SINGLE-SIDED COMPOSITION ALGORITHM

```
foreach cluster R_i do {
    read R_i
    foreach page S_j do {
        read S_j
        foreach tuple (b_k, c_j) in S_j do {
            foreach tuple (a_{i,x}, b_k) in R_i do {
                write tuple (a_{i,x}, c_j)
            }
        }
    }
}
```

For a detailed explanation of the above algorithm, readers are referred to [2].

APPENDIX 2. THE HOT-SPOT COMPOSITION ALGORITHM

```
/*-----------------------------------------------------
  First phase: Hashing and Clustering
-----------------------------------------------------*/

foreach page P_ℓ of R do {
    read P_ℓ
    foreach tuple (a_i, b_k) in P_ℓ do {
```

```
            write (a_i, b_k) into a cluster file R_i
            rcount[b_k] = rcount[b_k] + 1
        }
    }

    foreach page P_ℓ of S do {
        read P_ℓ
        foreach tuple (b_k, c_j) in P_ℓ do {
            if (rcount[b_k] ≠ 0) then {
                write (b_k, c_j) into a cluster file S_j
                if (hotspot[j] is empty)
                then hotspot[j] = (b_k, c_j)
                elseif (rcount[b_k] > rcount[HotB[j]]) then {
                    hotspot[j] = (b_k, c_j)
                    HotB[j] = b_k
                    /* HotB[j] is the B value of the current hot spot tuple of S_j */
                }
            }
        }
    }
    /*-----------------------------------------------------
      Second phase: Composing
    -----------------------------------------------------*/
    foreach cluster R_i do {
        read R_i /* assume R_i can fit into memory */
        foreach hotspot[j] do {
            /* the content of hotspot[j] is (HotB[j], c_j) */
            match[j] = false
            foreach tuple (a_i, b_k) in R_i do {
                if (b_k = HotB[j]) then {
                    match [j] = true
                    write (a_i, c_j)
                    break /* stop the inner loop */
                }
            }
        }

        foreach hotspot[j] do {
            if (match[j] = false) then {
                foreach tuple (b_k', c_j) in S_j do {
                    read (b_k', c_j) from S_j
                    foreach tuple (a_i, b_k) in R_i do {
                        if (b_k = b_k') then {
                            write (a_i, c_j)
                            break /* stop the inner loop */
                        }
                    }
```

```
            }
        }
    }
}
```

ACKNOWLEDGMENTS

We appreciate the constructive comments received from the reviewers who have helped to improve the presentation of this article.

REFERENCES

1. R. Agrawal, "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. R. Agrawal, S. Dar and H.V. Jagadish, "Composition of database relations," *Proc. of the 5rd Int'l Conf. on Data Engineering*, Feb. 1989.
3. D. Bitton and D.J. DeWitt, "Duplicate record elimination in large data files," *ACM Trans. of Database Systems*, Vol. 8, No. 2, June 1983, pp. 255-265.
4. H.K. Choi and M. Kim, "Hybrid join: an improved sort-based join algorithm," *Information Processing Letters*, Vol. 32, No. 2, July 1989, pp. 51-56.
5. H.T. Chou, D.J. DeWitt, R.H. Hatz and A. Klug, "The design and implementation of the Wisconsin Storage System," *Software Practice and Experience*, Vol. 15, No. 10, oct. 1985.
6. M. Kitsuregawa, H. Tanaka and T. Moto-oka, "Application of hash to database machine and its architecture," *New Generation Computing*, Vol. 1, No. 1, 1983.
7. W. Y. Lu and D. L. Lee, "The design of a recursive query processor," *Proc. Int'l Conf. On Data Base and Expert Systems Applications*, Aug. 1990.
8. W.Y. Lu, D.L. Lee, I.M. Hsu and S.S. Wei, "Minimizing search redundancy in processing bounded recursions," *Proc. CIPS Edmonton '90 Information Technology Conference*, Oct. 1990.
9. A. Rosenthal, S. Heiler, U. Dayal and F. Manola, "Traversal recursion: A practical approach to supporting recursive applications," *Proc. of Int'l Conf. on Management of Data*, May 1986.
10. G.M. Sacco and M. Schkolnick, "Buffer management in relational database systems," *ACM Trans. of Database Systems*, Vol. 11, No. 4, Dec. 1986, pp. 473-498.
11. P. Valduriez and H. Boral, "Evaluation of recursive queries using join indices," *Proc. of the 1st Int'l Conf. Expert Database Systems*, April 1986.
12. S.S. Wei, Y.N. Lien, D.L. Lee and T.H. Lai, "Hot-spot based composition algorithm," *Eighth International Conference on Data Engineering*, Feb. 3, 1992, pp. 48-55.

Yao-Nan Lien（連耀南）is an associate professor of computer science at National Chengchi University. His research interests include mobile computing, database systems, distributed computing, and software engineering. He received his B.S. degree from National Cheng Kung University in 1979, and his M.S. and Ph.D. degrees from Purdue University in 1981 and 1986, all in electrical engineering. Previously, he was an assistant professor of computer and information science at Ohio State University from 1986 to 1989, a member of the technical staff at AT&T Bell Laboratories from 1989 to 1993, and the Deputy Director of the Computer Software Technology Division in the Industrial Technology Research Institute, Computer and Communication Research Laboratories. Contact him at the CS Department, National Chengchi University, Mucha, Taipei, Taiwan, R.O.C.; lien@cherry.cs.nccu.edu.tw; http://www.cs.nccu.edu.tw/~lien.