Hot-Spot Based Composition Algorithm

Shu-Shang Wei and Yao-Nan Lien*
Dik L. Lee and Ten-Hwang Lai
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
(614) 292-4029,wei-s@cis.ohio-state.edu

Abstract

A composition requires three operations: join, projection and duplicate elimination. When eliminating duplicates on a large file, an external sort is needed. Both join and external sort operations are expensive in database systems because they incur a nonlinear I/O overhead. Conventional composition algorithms (either sort-merge-join based or hash-join based) achieve saving on join operation but spend a tremendous amount of time on external sort. On the other hand, the single-sided composition algorithm [2] is inefficient in performing the join but it eliminates duplicates on-the-fly. None of the above algorithms achieve savings on both operations. In this paper, a hot-spot composition *algorithm is proposed in an attempt to achieve savings on both operations. Several experiments have been conducted and it is shown that our hot-spot composition algorithm outperforms other algorithms under almost every condition. Composition operation is implemented as a primitive operation in our algorithm.*

1 Introduction

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. (We only consider binary relations in this paper. For relation that has more than two attributes, all non-joining attributes can be considered as one attribute. The respective sizes of relations R and S are assumed to be $|R|$ and $|S|$.) Composition is a very common operation in computing transitive closure and other types of linear recursions in deductive databases [7,8]. Much work have been done on the processing of transitive closure and linear recursion [1,9,11]. Therefore, composition is an important primitive operation to support many emerging new database applications and new database models, such as deductive databases and knowledge bases.

Three operations: (1) join, (2) projection, and (3) duplicate elimination are required to perform a composition operation. The join and projection operations are usually executed together, referred to as *join-project* operation in this paper, and then a duplicate elimination operation is followed to remove all redundant results created by the join-project operation. An external sort is needed before eliminating the redundant results if the size of the results after the join-project operation is large [3]. Both join and external sort operations are expensive in database systems because they incur a nonlinear I/O overhead.

Conventional composition algorithms (either sort-merge-join based or hash-join based) achieve saving on join operation but spend a tremendous amount of time on external sort. On the other hand, the single-sided composition algorithm [2] is inefficient in performing the join but it eliminates duplicates on-the-fly. None of the above algorithms achieve savings on both operations.

In this paper, a *hot-spot composition* algorithm is proposed in an attempt to achieve savings on both join and external sort operations. The proposed algorithm reduces the effort of performing the join operation by using a new *hot-spot* technique, whose detail will be discussed later in the paper. It is shown in the paper that on the average the join overhead using the hot-spot technique is only half of that in the single-sided algorithm. Moreover, the costly external sort operation is totally avoided because no duplicates are created in this algorithm.

Two other algorithms, namely, the *multiple-bucket* and the *double-sided* composition algorithms, are also

*Yao-Nan Lien is currently with AT&T Bell Laboratories, Naperville, Illinois. This work was done when the author was at Ohio State University.

0-8186-2545-7/92 $3.00 © 1992 IEEE

proposed in this paper. The multiple-bucket algorithm is proposed for the purpose of illustrating the *best*-case performance of the hash-join based composition algorithm; while the double-sided algorithm is for illustrating the *worst*-case performance of the hot-spot composition algorithm.

This paper is organized as follows. Section 2 discusses conventional composition algorithms and single-sided composition algorithm [2]. Section 3 proposes our hot-spot composition algorithm. Then, Section 4 analyzes the above algorithms and Section 5 shows their comparisons and experimental results. Finally, the conclusion remarks are given in Section 6.

2 Previous work

The conventional composition algorithms perform join and projection as one combined operation, and remove the duplicates in a separate step. In order to save time in doing the join, both relations are clustered on the joining attribute so that only pairs of clusters from the two relations within the same range need to be joined. This can be done by using either a sort-merge join or a hash join. Therefore, join can be performed very efficiently.

The problem with conventional composition algorithms is on 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 the duplicates. The external sort takes about $2|T| * log_2(|T|/|M|)$ I/O operations [5], where $|T|$ is the size of the result after the join-project operation and $|M|$ is the number of tuples that can fit in memory. (Actually, if a b-way merge-sort is performed [4], the external sort only costs $2|T|*log_b(|T|/|M|)$ I/O operations; however, it is still very expensive.)

Agrawal et al. proposed a single-sided composition algorithm to eliminate duplicates at the same time as performing the join-project operation [2]. R is first divided into a sufficient number of clusters based on A. The clusters are carefully chosen such that the potential size of each bucket (i.e., the join-projected result of each cluster of R with relation S) is small enough to fit into memory. Therefore, duplicates can be eliminated on-the-fly when performing the join. The detailed algorithm is depicted in Appendix 1.

The join operation of this algorithm is not very efficient because S has to be read in 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 $|S| * |R|/|M|$.

One variant of this algorithm [2] is to cluster S on B. Then, instead of fetching the whole S for every cluster of R, say R_i, the B values of all tuples in R_i are first examined to determine which clusters of S need to be fetched into memory. However, because the values of B for tuples in each R_i are arbitrary, it is very likely that most clusters of S still need to be read into memory. Furthermore, each time a join-projected result is created, the entire bucket needs to be searched once in order to get rid of the duplicates.

3 Hot-spot composition algorithm

Our hot-spot composition algorithm is described below. R is first clustered according to A and the number of occurrences of each B value is recorded at the same time. S is then clustered according to 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 has the same B value as some tuple of R. This can filter out many irrelevant tuples in S. In the extreme case, all tuples of S are discarded; hence, the following composing phase is even not needed. Note that the single-sided composition algorithm is not aware of this feature; therefore, it can not take this advantage to reduce the I/O cost on its composing phase.

Based on the number of occurrences of each B value, the tuple in each S cluster whose B value is most frequently referenced by R is maintained in memory. This tuple is referred to as the ***hot spot*** of the corresponding S cluster. Note that hot spots can be obtained and maintained at very little cost.

Each cluster of R is then retrieved and composed with the hot spots first. Tuples of each cluster in R are compared with each hot spot one by one. If the two compared tuples have the same B value, a composed result is immediately available and the remaining tuples in the R cluster need not be compared with the same hot spot any more. Only when all tuples in the R cluster failed to match with a hot spot, the corresponding S cluster of the hot spot is read into memory. The R cluster is then composed with the S cluster tuple by tuple. As long as two matching tuples are found, the composed result is immediately available and no further comparison between the two clusters is needed. Hence, the costly duplicate elimination operation is totally avoided because no duplicates are created. Composition is implemented as a primitive operation in this algorithm. The detailed algorithm is given in Appendix 2.

Note that our hot-spot composition algorithm has many unique features. (1) The composed results can be produced in any order (i.e. either in ascending or descending order) based on any or even both non-joining attributes without any extra overhead on sorting. The conventional composition algorithm needs a costly external sort for ordering the results; while the single-sided composition algorithm still needs an internal sort to achieve the ordering. (2) No duplicates are created; it means that any composed result is immediately available at the time when it is produced. Hence, a result can be either immediately given to the user or immediately pipelined to other processors for further processing without any waiting. (3) Its performance is sensitive to the distribution of the joining attribute values. On the clustering phase, many irrelevant tuples can be filtered out and will reduce some I/O cost on the composing phase. On the composing phase, each hot spot in memory serves as a surrogate of its corresponding cluster on disk and is chosen such that it can match with as many clusters of R as possible, which also depends on the distributions of the joining attribute values on both relations. In some extreme cases, most clusters of S need not be read into memory. Note that the performance when the joining attribute values are non-uniformly distributed is better than that when they are uniformly distributed. This is because, under uniform distribution, the B value of each tuple in S has the same number of occurrences in R; hence, the average chance of removing irrelevant tuples in the clustering phase is reduced and that of reading a cluster of S into memory in the composing phase is increased. (4) A variant that guarantees better performance can be easily achieved. A straightforward variant is to keep more than one hot spot in memory for each cluster of S. Then, the chance of reading an S cluster is reduced and the I/O cost is also reduced.

4 Analysis of different algorithms

In this section, we analyze the above composition algorithms. The detail experimental results are given in next section. The algorithms analyzed in this section are two-phase hash-join based composition[1], basic model of single-sided composition, and basic model of hot-spot composition algorithm.

Assume that the source relations R and S are initially resident on 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 in 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 the two-phase hash-join based composition is:

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

In the clustering phase, each relation has to be read in once for clustering and each cluster needs to be written back to disk after clustered. Therefore, the cost for clustering is $2|R|+2|S|$. Join is done cluster by cluster and only relevant pair of clusters are retrieved at each time. 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 [4], which requires $2|T|log_b(|T|/|M|)+|T|+|T|/f$ I/O cost.

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

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

R is clustered first, which requires $2|R|$ effort; then it is read into memory cluster by cluster, which takes $|R|$ tuples. 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 $|S| * |R|/|M|$. Duplicates are eliminated on-the-fly; therefore, the cost for writing out the result tuples is only $|T|/f$.

The total I/O cost for the hot-spot composition is:

cost of clustering R and S	$2\|R\| + 2\|S\|$
+ cost of reading R	$\|R\|$
+ cost of reading unmatched clusters	$\|A\| * m * \|S\|/\|C\|$
+ cost of writing the result relation	$\|T\|/f$
$= 3\|R\| + \|S\|(2 + m * \|A\|/\|C\|) + \|T\|/f$.	

The first two items are very straightforward. The third item is the product of the number of clusters in

[1] i.e. a two-phase hash-join [6] followed by the duplicate elimination.

R and the average unmatched clusters of S per cluster of R. Therefore, it is equivalent to $|A| * m * |S|/|C|$; where m is the average number of unmatched clusters for each cluster in R, $0 \leq m \leq |C|$. $|A|$ and $|C|$ are the numbers of clusters in R and S, respectively; hence, $|S|/|C|$ is the average size of a cluster in S.

From the above three equations, it can be seen that whether the hot-spot algorithm can outperform the other two algorithms mainly depends on its cost of reading unmatched clusters. 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.

5 Comparisons and experimental results

In this section, the proposed hot-spot composition algorithm is evaluated against the hash-join based and single-sided composition algorithms. Two other algorithms, namely, *multiple-bucket* composition and *double-sided* composition, are also discussed in order to illustrate the *best-* and *worst*-case performances of the hash-join based and hot-spot composition algorithms, respectively.

5.1 The multiple-bucket and double-sided composition algorithms

The *multiple-bucket composition* algorithm is a variant of the hash-join based composition algorithm. Both R and S are first clustered on B with the same set of subdomains. Hash join is then performed together with the projection. Each join-projected result is further hashed on A to decide which bucket it should go into. The hash function is carefully chosen such that the potential size of each bucket is small enough to fit in memory. After the join-project operation, 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 grouped in the same bucket which is small enough to fit into memory. In fact, the bucket a join-projected result should go into can be predetermined if each cluster of R is further clustered on A before performing the composition. However, we do not discuss this variant in this paper. Since it takes advantage of hash join but eliminates the external sort by using many small buckets to store different groups of duplicates when performing the join, its performance is the same as the best-case performance of hash-join based composition algorithms.

The *double-sided composition* is a variant of hot-spot composition algorithm. Relations R and S are clustered using the same method as in the hot-spot algorithm, except that no hot spot is retained in memory. Then, a nested loop is constructed to compose each pair of clusters of the two relations. Both clusters are sorted on their respective non-joining attributes before performing the composition. If a tuple in one cluster can compose with a tuple in another cluster then the remaining tuples in the clusters having the same value on the respective non-joining attributes need not be compared with each other. Hence, no duplicates are created in this algorithm. Since every cluster of relation S needs to be read in memory, the performance of this algorithm reveals the worst-case performance of hot-spot algorithm.

5.2 Performance model

We assume that R and S both have 2000 tuples. The effects of the following three parameters on the I/O traffic are the major parameters studied in the experiment: (1) size ratio (s), which equals to $|T|/|R|$, (2) duplicate ratio (f), which is defined as $|T|/|T'|$, and (3) memory ratio, which is equivalent to $|M|/|R|$. The miss ratio ($missr$), defined as $m/|C|$ (m and $|C|$ are defined in the previous section), is assumed to be 0.5 to observe the average performance of the hot-spot composition algorithm.

We assume that the values in attributes A, B, and C are of integer type and are uniformly distributed on the same domain. Based on the given values of the three parameters s, f, and $missr$, R and S are synthesized. Our measurement is based on the I/O traffic instead of the real CPU response time. Thus, the experiments conducted in this paper are immune to the size of each operand relation. The assumption of uniform distribution on each attribute decreases the performance of the hot-spot algorithm. As discussed before, the performance under a non-uniform distribution should be better than that of a uniform distribution.

In the ideal case, each pair of composing clusters in the above algorithms are assumed to be small enough to be memory resident at the same time. However, under certain conditions, such as data skewing or the memory being too small, the above assumption is not true. Whenever such situation occurs, a nested loop approach is performed.

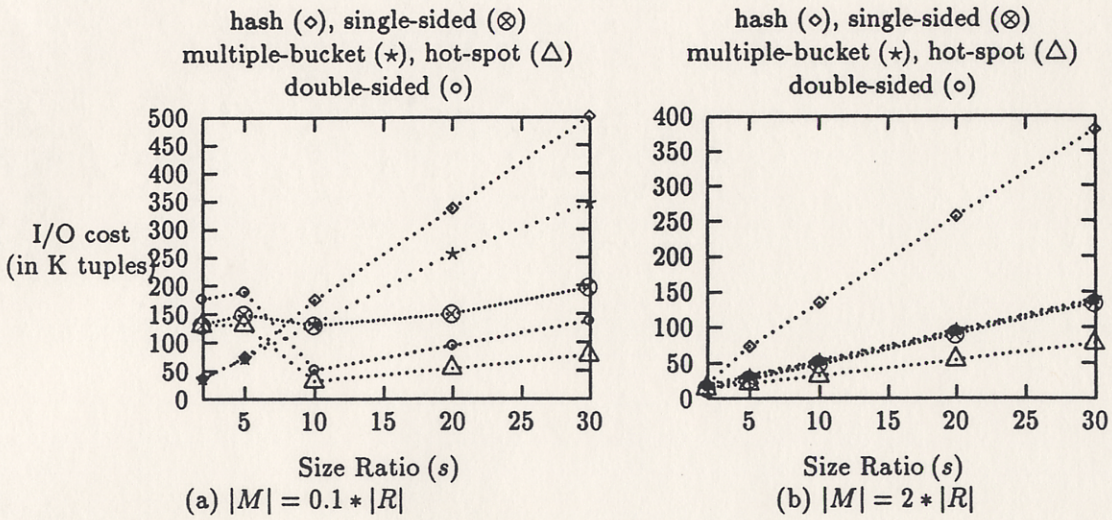

Figure 1: The effects of size ratio when $f = 10$.

Theoretically, only the hash-join based composition algorithm needs an extra external sort in order to eliminate duplicates. However, for the same reason described above, external sort is still needed for multiple-bucket and single-sided composition algorithms to eliminate duplicates and for double-sided composition algorithm to sort each pair of composing clusters. For the hot-spot composition algorithm, external sort is absolutely not needed because it neither needs to sort each pair of composing clusters nor creates any duplicated tuples. The external sort is done by a b-way merge-sort, where the value of b depends on the given memory size and the limit of the number of files that can be opened simultaneously (which is 64 in our system).

5.3 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 at 10, but varying the size ratio. The memory ratio was also changed to observe its effect on the performance. The results when the memory ratio is 0.1 are depicted in Fig. 1-a.

The multiple-bucket and hash-join based composition algorithms outperform the others when the size ratio is small, but they become the worse ones when the size ratio is very large. The explanation for this behavior is that the cost of the composing phase dominates that of the duplicate elimination when the size ratio is small, and the situation is reversed when the size ratio is large. This phenomenon makes the double-sided, single-sided, and hot-spot composition algorithm perform poorly when the size ratio is small because they spend more time on the composing phase but a small amount of time on duplicate elimination (which is unnecessary in the ideal case for the single-sided algorithm and absolutely unnecessary for the double-sided and hot-spot algorithms). On the other hand, the multiple-bucket and hash-join based algorithms have poor performance when the size ratio is large because they incur more I/O traffic for duplicate elimination. The reason why the curves of single-sided, double-sided, and hot-spot compositions go down when size ratio is 10 is because the number of clusters of R is very small at that point, which has significant effect on the cost of the composing phase of these algorithms. When the memory is large, the above phenomenon is not significant because most of the composing cluster pairs can be memory resident, as can be observed in Fig. 1-b. Since the hot-spot algorithm doesn't produce any duplicate and only accesses limited part of S (which depends on the miss ratio), it has the best performance among the five algorithms almost everywhere in the above two figures. Even the double-sided algorithm, whose performance is considered the same as the worst-case performance of the hot-spot one, outperforms the others almost under all conditions for the same reason.

The second experiment is to observe the effects of different duplicate ratios, when the size ratio is fixed at 10. The results are depicted in Fig. 2. In both 2-a and 2-b, all curves, except that of the single-sided composition in Fig. 2-a (where memory ratio is 0.1), decrease only slightly with the duplicate ratio. This is because

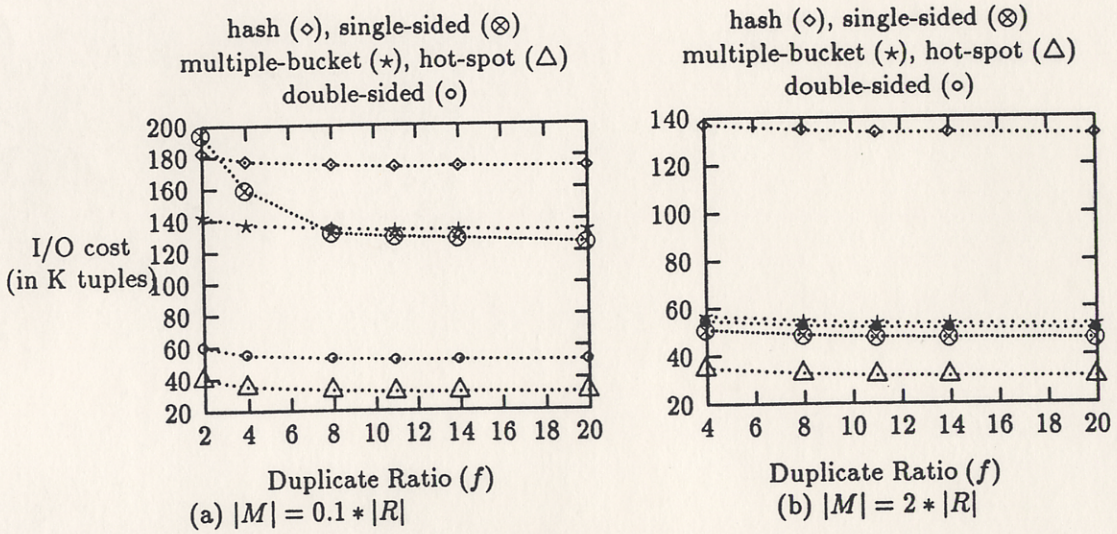

Figure 2: The effects of duplicate ratio when $s = 10$.

$|T|$ is the same for different duplicate ratios and $|T'|$ is monotonically decreasing when the duplicate ratio is increasing. However, the curve of single-sided composition drops significantly when the duplicate ratio is between 2 and 8. This is because not many duplicates were removed in the composing phase before that range since the size of the memory is not large enough to eliminate all duplicates on-the-fly. Therefore, an extra external sort is still needed to eliminate the remaining duplicates. The problem disappears when the memory size is large, which can be seen from Fig. 2-b. The hot-spot algorithm is still the best. Moreover, the performance of double-sided algorithm is very close to the single-sided and the multiple-bucket algorithms, which means that even in the worst case, the hot-spot algorithm is still superior.

Finally, the effects of memory size on the I/O traffic are investigated in the last experiment. The size ratio and duplicate ratio were held at 10 and 10, respectively. The results are displayed in Fig. 3.

When memory size is very small, duplicates can hardly be eliminated on-the-fly and the cost of nested loop is very significant. Therefore, the single-sided algorithm becomes the worst when the memory size is small because it requires not only an extra external sort to sort the join-projected results but also a costly nested loop to perform the composition. Thus, its performance deteriorates very significantly as can be seen from Fig. 3-a. The double-sided algorithm also suffers when the memory is small. This is because an extra external sort is also needed for each pair of composing clusters. However, the effect is not as much as that in the single-sided algorithm because the latter needs to sort each join-projected result file which is much larger than each source cluster. When the memory size becomes larger, all algorithms, except the hash-join based algorithm, take the advantage very promptly. The curve of hash-join based algorithm only moves down significant when the memory size is large enough to avoid the costly external sort; this part of graph is not displayed in our graph for simplicity purpose.

The curves of single-sided and multiple-bucket algorithms go down substantially when the memory size is varying from 10 to 100 percent of the source relation. This also has to do with duplicate elimination. In the above range, most duplicates for both algorithms can be eliminated without an extra external sort; therefore, their costs drop down very significantly. On the other hand, the double-sided and hot-spot algorithms have less effect on the memory size. This can be seen from the fact that the two algorithms do not produce any duplicates. Note that the hot-spot algorithm is again the best when memory size is large and it has good performance even when the memory size is small.

From the above experiments, the hot-spot algorithm seems to be the best under almost every condition. However, the hot-spot algorithm does not permit any conflict in each cluster, which means that each cluster of R and S can only have the same A and C attribute values, respectively; otherwise, the miss ratio will increase significantly. In order to fulfill this requirement, each cluster may contain only few tuples and lots of clusters are created. This affects response time because if each cluster is stored as a single file then much CPU time is required to manage those files.

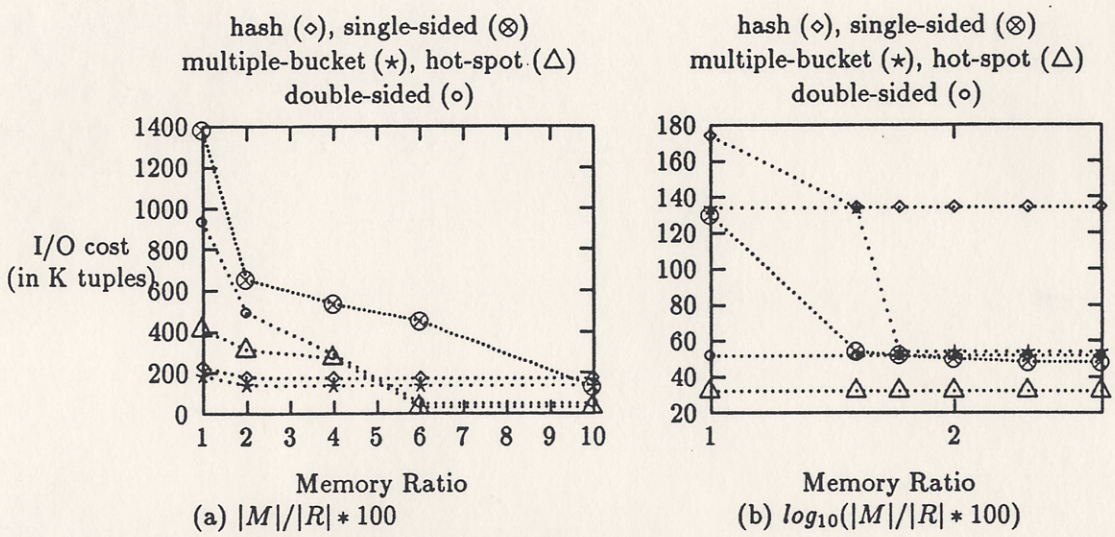

Figure 3: The effects of memory ratio when $f = 10$ and $s = 10$.

However, a more sophisticated structure can be used to organize the clusters such that the CPU time can be reduced. For example, many clusters can be stored in a single data file and an index file is constructed on top of the data files so that only few files are needed.

6 Conclusion

Composition operation is an important primitive operation in deductive database. However, it requires either a costly natural join operation or a costly external sort operation. Neither the conventional composition algorithms nor the single-sided composition algorithm can achieve savings on both operations. In this paper, we proposed a *hot-spot* composition algorithm in an attempt to achieve savings on both operations.

Several experiments were conducted to compare the performances of our hot-spot composition algorithm, hash-join based composition algorithm, and single-sided composition algorithm. We observe the average performance of the hot-spot algorithm; its worse-case performance can be determined by that of the double-sided algorithm. The best-case performance of hash-join based algorithm is observed by that of multiple-bucket composition algorithm.

From the experiments, it can be seen that the hot-spot algorithm has a very good performance under almost every condition; even in its worse case (which is depicted by the double-sided algorithm), its performance is still close to the single-sided composition algorithm. This is because (1) hot-spot composition algorithm requires no cost for duplicate elimination because no duplicates are created and (2) only part of an operand relation need to be read into memory because most composed results can be found via hot spots. Therefore, our conclusion is that the hot-spot composition algorithm is recommended for most applications. Moreover, the hot-spot composition algorithm seems to be more suitable for parallel and distributed systems because the filtering effect on its clustering phase is similar to the semi-join technique used in distributed systems and the pipelining ability on its composing phase is also a common technique adopted in parallel systems. However, more experiments need to be conducted to judge the above argument.

Appendix 1. The single-sided composition algorithm.

```
FOREACH cluster R_i of R DO
    Read in R_i
    Foreach page S_i of S DO
        Read in S_i
        foreach tuple (b_j, c_k) in S_i DO
            foreach tuple (a_ix, b_j) in R_i DO
                "ADD" tuple (a_ix, c_k) to the result T_i
            end
        end
    End
END
```

For 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_i of R DO
  Read in P_i
  Foreach tuple (a_l, b_m) in P_i Do
    Write (a_l, b_m) into a cluster file R_l
    Rcount(b_m) = Rcount(b_m) + 1.
  End
END
Initialize every HS to blank
FOREACH page P_i of S DO
  Read in P_i
  Foreach tuple (b_m, c_n) in P_i Do
    If Rcount(b_m) ≠ 0 then
      Write (b_m, c_n) into a cluster file S_n
      If HS(n) is blank
      then { Write (b_m, c_n) to HS(n) }
      else if Rcount(b_m) ≥ Rcount(b_m')
        then replace (b_m', c_n) with (b_m, c_n)
        /* (b_m', c_n) is the old tuple in HS(n) */
      end
    end
  End
END

/*************************/
/* Second phase: Composing */
/*************************/

FOREACH cluster R_i in R DO
  Read in R_i
  Foreach hot spot HS(k) DO
    Match(k) := false; Assume that HS(k) = (b_j', c_k)
    foreach tuple (a_i, b_j) in R_i and not Match(k) DO
      If b_j = b_j' then
        {Match(k) := true; write (a_i, c_k) to result file.}
    end
  End
  /* The unmatched clusters are handled below */
  Foreach k such that Match(k) = false DO
    Read in cluster S_k
    foreach tuple (b_j', c_k) in S_k and not Match(k) DO
      foreach tuple (a_i, b_j) in R_i and not Match(k) DO
        If b_j = b_j' then
          {Match(k) := true; write (a_i, c_k) to result file}
      end
    end
  End
END
```

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, L. A., CA, Feb. 3-5, 1987.

2 R. Agrawal, S. Dar and H. V. Jagadish, Composition of Database Relations, Proc. of 5th Int'l Conf. on Data Engineering, L. A., CA, 1989.

3 D. Bitton and D. J. DeWitt, Duplicate Record Elimination in Large Data Files, *ACM Trans. of Database Systems* 8, 2:255–265 (June 1983).

4 H. K. Choi and M. Kim, Hybrid Join: An Improved Sort-Based Join Algorithm, *Information Processing Letters* 32, 2:51–56 (July 1989).

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* 15, 10 (Oct. 1985).

6 M. Kitsuregawa, H. Tanaka and T. Moto-oka, Application of Hash to Database Machine and its Architecture, *New Generation Computing* 1, 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 (DEXA '90), Vienna, Austria, 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, Edmonton, Canada, 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, Washington D. C., May 1986.

10 G. M. Sacco and M. Schkolnick, Buffer Management in Relational Database Systems, *ACM Trans. on Database Systems* 11, 4:473–498 (Dec. 1986).

11 P. Valduriez and H. Boral, Evaluation of Recursive Queries Using Join Indices, Proc. of the 1st Int'l Conf. Expert Database Systems, Charleston, South Carolina, April 1986.