The Twenty-First Annual International
Computer Software & Applications Conference
(COMPSAC'97)
August 13-15, 1997 Washington, D.C.

Composition Can Be Faster Than Join §

by
Yao-Nan Lien and Chih-Lin Hu
Department of Computer Science, National Chengchi University
Taipei, Taiwan, R.O.C., lien@cherry.cs.nccu.edu.tw

Abstract

In relational databases, a composition requires three operations: join, projection and duplicate elimination. An expensive external sort is required to eliminate duplicates in a large file. Most conventional composition algorithms take join and duplicate elimination as separate operations to reduce overhead 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 the direct composition can even outperform its component operation, join, under various conditions. Moreover, when the density of the joining attribute is high enough, the hot-spot direct composition may even run faster as the operand relations become bigger. These results can encourage real DBMSs to remove duplicates in some relational operations, and thus, to preserve the closure property of the relational data model.

INDEX TERMS: 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 various new database models [1,9].

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 which follow the definition to process a composition in three separate steps can reduce overhead on either join or duplicate elimination, but not both [2]. Direct composition algorithms [8,10] take a different approach to process a composition as a single primitive. It had been shown in [8,10] that direct composition algorithms can easily outperform conventional algorithms. In this paper, we will show that they can even outperform one of its component operation, join.

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. (Π' denotes a projection without duplicate elimination.)

1.2 Join vesus Composition

It is a common experience that, by direct execution, a join operation can be more efficient than its component operation, cross product. Similarly, we will show that direct composition algorithms may outperform its component operation, join. This has a very significant implication to the real world DBMSs.

In order to save the significant overhead of duplicate elimination, most real world DBMSs do not remove duplicate tuples in answering queries unless explicitly specified by users. A user has to specify the "DISTINCT" modifier in a SQL query to request duplicate tuples removed. However, sometimes it is essential to preserve the closure property of the relational data model, in which every tuple in a relation must be unique even after a relational operation. This SQL flaw has been extensively criticized by the research community.

Let's examine the following popular* SQL 'join' statement over relations R(A,B) and S(B,C):

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

As we can see that the above operation is actually a

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

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

compromised composition, a composition without duplicate elimination. This type of queries is always executed as a join operation followed by a projection, without duplicates being eliminated. The invention of the direct composition can eliminate this compromise. With comparable efficiency, duplicated tuples will not be kept in the composition result. Of course, the the above query must be rewritten by adding a DISTINCT modifier as follows:

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

This paper will compare the performance of direct composition and hash join under various conditions. Section 2 will briefly describe various direct composition algorithms. Analytical and experimental comparisons of join and direct composition will be given in Section 3 and 4 respectively. Finally, future research will be discussed in Section 5.

2. Previous Work

2.1 Conventional Composition Algorithms

Most conventional composition algorithms take either join or duplicate elimination as separate operations to reduce overhead by one of them. 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 [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 when it performs the join-project operation [2]. A typical join-saving based algorithm such as the sort-merge-join or the hash-join, performs join and projection together firstly, and then remove the duplicates by a separate step [4,7]. An example is shown in Fig. 2.

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

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

2.2 Direct Composition

Direct composition algorithms [8,10] take another approach to achieve a better performance. They do not follow the definition to process those three component operations in sequence. Instead, they execute composition directly as a single primitive so that no duplicate is generated in the intermediate steps. This is similar to what we did in join operation by executing it directly, instead of following the formal definition to execute its component operation sequentially.

We use the example shown in Fig. 1 to illustrate some properties of the composition operation. Relation R is firstly clustered 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

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

It is clear 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 problem, depending on whether $R_i.B$ and $S_j.B$ have any common value:

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

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

By utilizing this property, direct composition algorithms achieve a better performance than conventional implementations as shown in [8]. 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 entirely read into

memory 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 the matched values 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 [8].

2.2.2 The Hot-Spot Algorithm

The hot-spot composition algorithm takes one step further to cache the "hot-spot tuples" (which will be explained later) right in the memory, such that many S clusters can be discarded if their corresponding in-memory hot-spot tuples are already matched to some B value in R.

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. 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. The hot-spot tuple of each cluster, say S_j, is kept in memory and is used for matching the B values of the R cluster in a cluster composition, say $R_i \circ S_j$. If a B value of R_i match the B value of the hot-spot tuple of S_j in the memory, the cluster composition can be stopped immediately without reading S_j 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.) Furthermore, 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;
Composing Ri with the hot-spot tuple of Sj;
If there is a match,  write (i,j) and stop;
If there is no match, composes Ri with Sj.
```

Fig. 4 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

Figure 4. 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 resulting tuples. They are executed without reading their corresponding S clusters since they all have a hot-spot hit.

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 systems. 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 be compared with every tuple of S. One good way to improve its efficiency is to cluster both R and S based on the joining attribute, B, so that only the tuples in the same cluster need to be compared. The complexity is reduced from $|R| \times |S|$ to $\sum |R_i| \times |S_i|$. 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. Assuming that the source relations R and S are initially resident in disk, and 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 which 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

The total I/O cost for the hash-join operation is then:

$$\begin{aligned} &\text{cost of clustering } R \text{ and } S && 2|R| + 2|S| \\ &+ \text{ cost of joining clusters} && |R| + |S| + \frac{\rho |R| |S|}{|B|^2} \\ &+ \text{ cost of writing } T && |T| \\ &= 3|R| + 3|S| + |T| + \rho\left(\frac{|R||S|}{|B|^2}\right) \end{aligned}$$

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 cluster and S clusters depends on the size of each cluster and the size of memory. It is a function of $|R||S|/|B|^2$. The parameter ρ 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 composition is:

$$\begin{aligned} &\text{cost of clustering } R \text{ and } S && 2|R| + 2|S| \\ &+ \text{cost of reading } R \text{ clusters} && |R| \\ &+ \text{cost of reading unmatched } S \text{ clusters} && \lambda(1-\mu)\frac{|R||S|}{|M|} \\ &+ \text{cost of writing out result relation} && |T|/f \\ &= 3|R| + (2 + \lambda(1-\mu)(|R|/|M|))\cdot|S| + |T|/f \end{aligned}$$

where f is the duplication factor (thus, $|T|/f$ is the size of composition result), μ ($0 \geq \mu \geq 1$) is the *hit ratio*, the average ratio of matched clusters to the total S clusters, per R cluster; and λ ($0 > \lambda \geq 1$) is the average fraction of a S cluster that is read. The total I/O cost for cluster composition is then $\lambda(1-\mu)\frac{|R||S|}{|M|}$.

The best case happens when the hot-spot hit ratio, μ, is 1. A small λ value can also lead to a good performance. One good way to reduce λ is to sort S clusters according to 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 the above analysis that the performance of the 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 μ, λ, ρ, 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]. The SET Query Benchmark is specially designed for testing the database performance in decision support environments, rather than OLTP environments. (Since TPC-D Benchmark has been announced, we will consider running the experiment on the TPC-D Benchmark in the future.)

4.1 Experiments

In this experiment, we focus on the sensitivities of the following parameters on the overall performance: (1) size of relations; (2) cardinality of the domain of joining attribute; (3) cardinality of the domain of non-joining attribute; (4) data distribution; (5) order of relations; and (6) number of cached hot-tuples.

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 three different integer domains: (1-25), (1-100), and (1-1000). In each experiment, R and S are first defined by selecting A, B, and C attributes from the defined domain. Each defined relation is materialized by randomly drawing values from the domain of its attributes and is materialized into several different sizes from 1000 up to 20000 tuples. (Because of duplicated tuples must be removed in the generated relations, the actual size of a generated relation may be slightly 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. Finally, all possible definitions (combinations of attributes) of R and S are simulated to observe the sensitivity of the desired parameters under various conditions. Each experiment is done twice, one on uniform data distribution and the other on exponential distribution.

4.2 Experimental Results

4.2.1 Sensitivity to the Size of Relations

As we have already known, the complexity of hash-join is nonlinearly proportional to the size of joining relations. While the performance of the hot-spot direct composition is worse than the hash join when the relations are small. However, when the relation size increases, its execution time increases slowly and, impressively, decreases when the size is sufficiently large. This is because the denser the joining attribute is, the higher the hop-spot hit ratio will be. As a consequence, the hot-spot composition performs better than the hash-join when the size of relations are larger than a threshold. The results are shown in Figure 5.

4.2.2 Sensitivity to the Joining Attribute

Both hash-join and the hot-spot composition are sensitive to the cardinality of the domain of joining attributes. The 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 pairs of larger clusters due to the linearization of quadratic complexity.)

On the other hand, the 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. This is depicted in Figure 6(a) and 6(b).

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 the hash-join, a smaller non-joining

attribute size will reduce the joining overhead in the bucket joining process. For the hot-spot composition, it needs to perform a composition for every possible pair of R and S clusters. A smaller number of clusters leads to fewer cluster compositions. Different from a join operation where the time complexity is quadratically proportional to the size of clusters, the execution time of a cluster composition does not directly proportional to the size of clusters since its execution can be stopped immediately after finding a match of B value on both clusters.

4.2.4 Sensitivity to the data distribution

As we expected, the direct composition prefers a skewed data distribution on the joining attributes if the joining attributes of both relations are skewed in the same way. We compare the performance of the hot-spot algorithm under uniform and exponential distributions. (Thus, both relations are skewly distributed in the same way.) We found that there is a significant difference in performance. Skewed distribution obviously increases the possibility of hot-spot hit. This is depicted in Figure 7.

4.2.5 Sensitivity to the Order of Relations

Similar to a joining algorithm, the hop-spot composition prefers the smaller relations to be the first relation. This is because R clusters are always read into memory while S clusters may not be the case. Further, a smaller R cluster can avoid (or reduce) any nested loop for composing two clusters. The results are depicted in Figure 8.

4.2.6 Sensitivity to the cache size

More than one hot-spot tuples from each S cluster can reduce the execution time. Aforementionedly in Section 2, to minimize the I/O overhead, we can strive to compare a matched B value as early as possible. Furthermore, we found that hot-spot tuples play an important role in minimizing I/O overhead. To further enhance the effect of hot-spot hit, we cached more hot-spot tuples in memory from each S cluster to increase the probability of hot-spot hit and hoping to further reduce the I/O overhead for reading S clusters. Surprisingly, the improvement is insignificant as shown in Figure 9. This may be because the hit-ratio has been already very high when caching the first hot-spot tuple in our experiments. Further study is needed to understand its real cause.

5. Concluding Remarks

Composition is an important primitive operation in deductive databases. Direct composition algorithms have been proved to be more efficient than the conventional approaches. In this paper we show that the direct composition may also outperform its component operation, join. Within a very broad range, the performance of direct composition is comparable with join operation.

We have observed in the experiment that the hot-spot composition performs better on the the following cases:

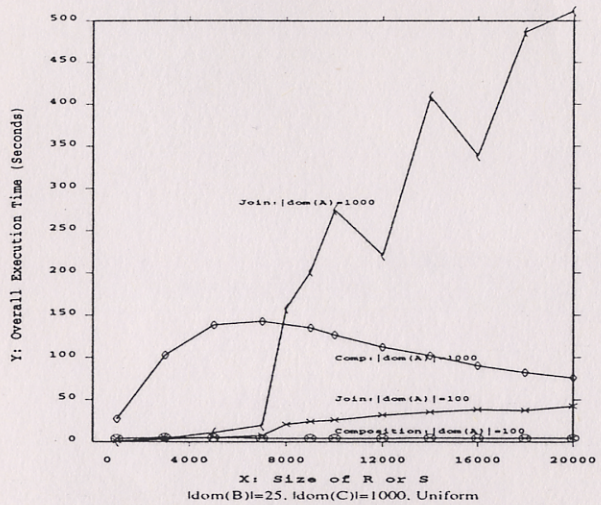
1. when the domain cardinality of attribute A is small and that of attribute C is large;
2. when the size of relations is large;
3. when the density of joining attribute is high; and
4. when the joining attributes of both relations are skewed in the same way.

The main reason that the hot-spot algorithm can outperform join operation is that it reduces I/O overhead for reading S clusters by caching hot-spot tuples in the memory.

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 usually implemented as a join operation to avoid the overhead for duplicate elimination. The results presented in this paper will encourage practitioners stop using join operation as a compromised composition. Not only the performance may be better, but also the closure property of relational data model can be preserved.

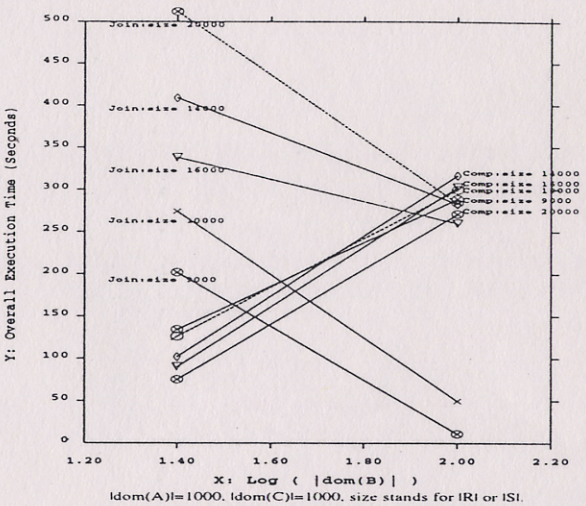
Reference

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 5th 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.
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", *Journal of Information Science and Engineering*, Dec. 1996, vol. 12, no. 4, pp. 547-565.
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. Wei, Shu-Shang, Lien, Yao-Nan, Lee, Dik and Lai, T. H., "Hot-Spot Based Composition Algorithm", *Eight International Conference on Data Engineering*,, Feb. 3, 1992, pp. 48-55.



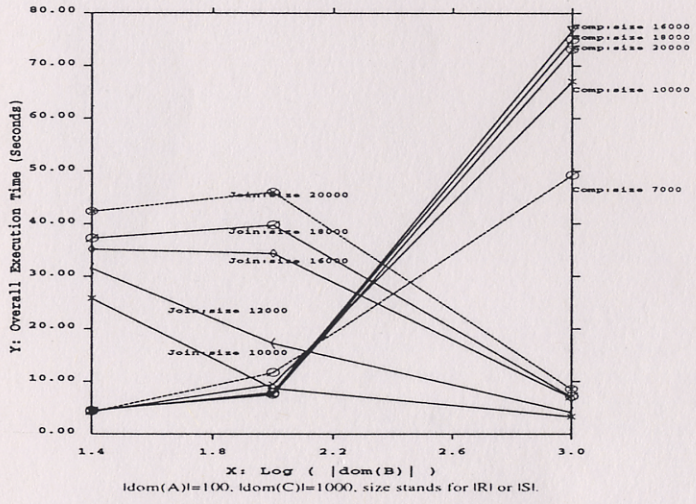
|dom(B)|=25, |dom(C)|=1000, Uniform

Figure 5. The sensitivity to the size of original relations.



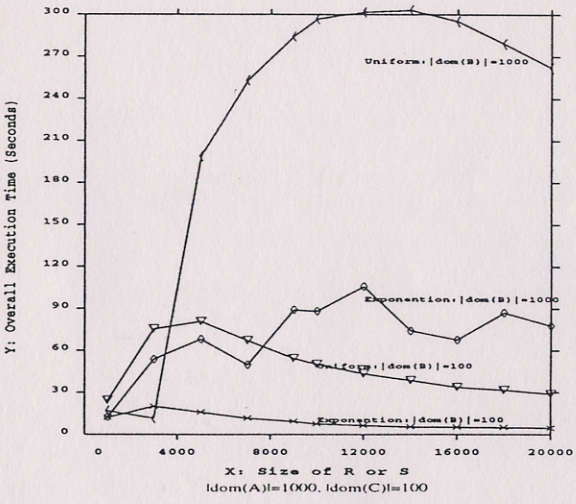
|dom(A)|=1000, |dom(C)|=1000, size stands for |R| or |S|.

Figure 6(a). Hash-Join: The sensitivity to the joining attribute.



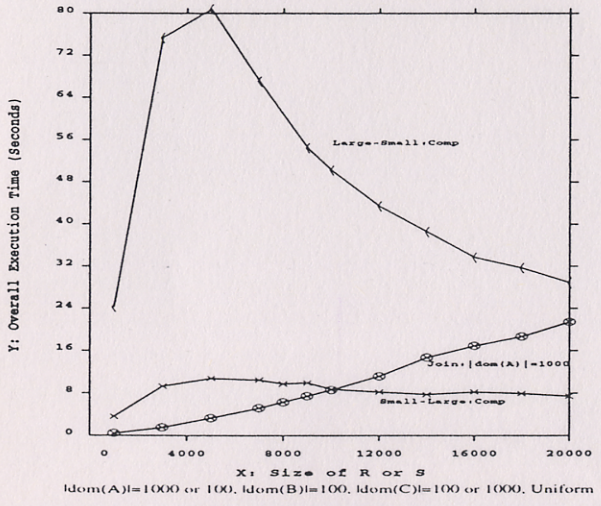
|dom(A)|=100, |dom(C)|=1000, size stands for |R| or |S|.

Figure 6(b). Hash-Join: The sensitivity to the joining attribute.



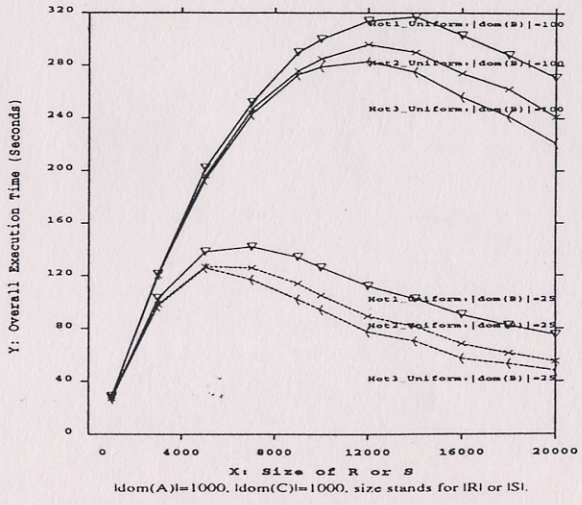
|dom(A)|=1000, |dom(C)|=100

Figure 7. Composition: Exponential vs. Uniform distributions.



|dom(A)|=1000 or 100, |dom(B)|=100, |dom(C)|=100 or 1000, Uniform

Figure 8. Effect of Smaller_Relation_First (composition).



|dom(A)|=1000, |dom(C)|=1000, size stands for |R| or |S|.

Figure 9. Composition: More than One Hot-tuple