Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

3-WAY HASH JOIN QUERY PROCESSING IN DISTRIBUTED RELATIONAL DATABASE SYSTEMS

Scott E. Spetka
Gerald J. Popek

January 1989
CSD-890008

3-way Hash Join Query Processing in Distributed Relational Database Systems *

Scott E. Spetka and Gerald J. Popek

University of California, Los Angeles

ABSTRACT

Initial distribution of relations as well as storage structures and organization have an important impact on performance and the appropriate choice of processing techniques for database operations. Consideration of data distribution for partitioned relations used in hash join processing lead us to experiment with a new algorithm for processing 3-way join queries in a distributed system.

Database cacheing is also important for performance of distributed database management systems. An important goal is to provide an algorithm that can complement existing algorithms to provide sufficient generality to operate in a network transparent environment where the location of available resources may be changing, and to use those resources effectively. We present a new algorithm for processing 3-way join queries that can take advantage of cacheing by providing improved performance when data is not ideally distributed for some other algorithms.

1 Introduction

Organizations are becoming richer in computer hardware as it becomes cheaper and more powerful. At the same time they are accumulating unprecedented amounts of data. Some organizations buy computers specifically designed for the database functions. Others accumulate database processing potential through addition of general purpose workstations to service specific functions for their employees. Distributed database technology provides the opportunity to effectively utilize distributed resources for improved performance through parallel execution. A variety of algorithms for query processing provide a rich target for query optimization. The more approaches to processing a query that are considered by an optimization algorithm, the greater is the potential to find an algorithm that performs significantly better for a particular query.

In the future, we anticipate that knowledge based front ends to conventionally structured databases will become increasingly important. One approach to interfacing a logic programming system to a relational database system is to translate queries, for example, from an implementation based on such a language as Prolog to a relational language. The queries which can be expected from such an environment may be significantly more complex than the simple queries that are the mainstay of database technology today.

We focus here on nested queries, those that involve more than two variables. They are considered nested since subqueries involving at most two variables may be executed with their result replacing the subquery in the main query which is then processed recursively. The combination of increasing database size and increasing query complexity leads to increasing potential for improved performance from discovery of new distributed nested query

[*] This research was sponsored by DARPA Contract No. F29601-87-C-0072

processing algorithms. For example, large databases are being implemented to make increasingly large collections of libraries available to users. The MELVYL system contains the library catalog for all campuses of the University of California. Borrowers may borrow books from any library. The ORION system at UCLA contains the UCLA collection. Both systems are readily accessible to any engineering student from his computer account. A possible relational schema for the implementation of such a system in a relational database [Ullman 82] is given in figure 1.

Books(Title, Author, Publisher, Book_Number)
Publishers(Name, Address, City)
Borrowers(Name, Address, City, Card_Number)
Loans(Card_Number, Book_Number, Date_out, Library_City)

Figure 1 - Relational Schema for Library Database

Printing a statement for overdue books requires a join of Books, Borrowers, and Loans. Notice that the Books, Borrowers and Loans relations may be independently administered and in fact might well be stored on independent machines. The performance of a join operation is extremely sensitive to data location. This example 3-way join of distributed data illustrates the potential for improved performance. Given the size of this query, even a small percentage increase in performance would result in a significant reduction in response time. The distributed algorithm that we introduce here can result in a significant reduction in response time over currently available algorithms. It may be a useful addition to the distributed query processing algorithms currently available.

Implementing a distributed database to support these increasingly complex applications motivates new distributed operating system facilities and new distributed query processing algorithms. Little emphasis has been placed on multi-variable nested queries for DBMS or DDBMS. It is a difficult area to study since algorithm performance can be closely related to data distribution. Instead of considering how to move data into a configuration, we propose to use a better algorithm that can process the data where it is more cheaply than the cost to move the data to process it efficiently using existing algorithms. The recent development of hash join algorithms [DeWitt 84] [Brat 83] has lead to a number of interesting results [Nakayama 88]. [DeWitt 85] studied distributed hash join algorithms in the framework of existing distributed query processing technology [Epstein 80] [Stonebraker] [Lohman 84]. In this paper, a new approach to distributed hash join query processing for nested queries is presented. We focus on the three way join to illustrate the utility of our approach to distributed query processing.

Conflicting goals exist for query processing. In this study, improved response time is our goal, by which we mean the time required to produce the entire result. There are some cases where it is more important to produce a single tuple of a result quickly. Processing can proceed on that result while other results are being produced. This is the standard model for logic programming languages. We evaluate two important factors in distributed query processing algorithm design. Performance measurements are presented for an experimental implementation that quantifies the cost of tuple response optimization in single site systems. The other part of this study uses an analytic model to compare our distributed hash join query processing algorithm with other approaches to show that our algorithm provides improved response time.

2 Overview

In the introduction, we argued that 3-way join processing is a good example where a new approach to query processing can improve distributed database processing. In section three, we go on to describe three algorithms for processing nested queries. The first is the standard Ingres tuple substitution algorithm. Second, we present our algorithm. Finally, we present a well known algorithm that has been studied extensively for distributed query processing. These algorithms can be easily adapted for distributed hash join query processing.

In section four, we present an introduction to hash join and distributed hash join query processing algorithms. We also describe how the nesting technique for extension of simple two variable join queries to more complex

multivariable queries is affected by application of hash join techniques. We argue that hash join algorithms restrict the degree of parallelism that can be achieved through the standard pipelining approach to distributed query processing. Our algorithm achieves increased parallelism and improved performance by executing as many subqueries in parallel as possible but not depending on pipelining.

Distributed query optimization is often considered to have two parts. Storage allocation and relation distribution have been studied extensively. Replication is used in many schemes to assure that data is available for efficient processing when conflicting query processing requirements exist for distinct queries. Storage is allocated in anticipation of future needs. The other part of the problem is to dynamically determine the best query processing strategy given the distribution of data when a query is made. [DeWitt 85] showed that if sufficient memory exists his multiprocessor hash join algorithm is superior to existing algorithms even if he must redistribute data dynamically. In section five, we describe a scenario that we feel will occur frequently where data is not cached ideally for DeWitt's algorithm. For data distributed as such, our algorithm performs better.

In section six, we describe two hash join algorithms. The first is based on single subquery nesting. The second is our algorithm in which several subqueries are executed simultaneously. We use a simple analytic model to study the performance of the two algorithms. Concentrating on I/O requirements shows that the basic principle of our algorithm is worthwhile. Parallel execution allows I/O to be done simultaneously, reducing response time. Response time is measured in terms of the number of I/Os that are not performed in parallel. We compare the performance for join selectivity less than one. We also study the effect of increased relation size on performance. Results show that our algorithm provides a significant performance improvement for values of join selectivity less than one.

Some query processing algorithms are inherently "depth first". Such execution characterizes the University Ingres [1] algorithm as well as typical Prolog based knowledge oriented systems. In Ingres, each tuple from a relation is recursively substituted into a query. For each substitution, the decomposition of the query tree is repeated. Prolog searches for matches on the next clause each time a match is found for a variable in the current clause. Backtracking causes the next clause with the same name to be tested for a match. It is important to quantify the cost for "tuple recursion" because it could potentially be used for a distributed query processing algorithm. In section seven, we describe an experiment that we performed to quantify the cost of tuple recursion. We also consider when this approach is better than running subqueries to completion by processing a join completely before substituting the result into the main query. Section eight presents conclusions and describes our plans for future research.

3 Query Processing Algorithms for 3-way Join Processing

In this section, we present three related algorithms. The first is the standard University Ingres recursive tuple substitution algorithm. The second is our proposal for a new algorithm that is similar to the Ingres algorithm but that eliminates recursive invocation of nested subqueries. The final algorithm presented here is the traditional approach to 3-way join processing. It was proposed by [Epstein 80] for extension of the Ingres query processing algorithm to exploit parallelism in distributed systems. We show below that single site Ingres can also benefit significantly by its use. We give the main features and an example for each algorithm.

The example schema from section 1 (see figure 1) will be used in this section. An example of a fully connected 3-way query is used to illustrate the various query processing requirements for each algorithm. We use this example 3-way join query:

---

[1] Any reference to Ingres in this paper is to the University Ingres system. We use version 8.8 Ingres, released 5/1/86 as a testbed for this work. Our processors are Vax 750s.

```
Retrieve (Borrowers.Name, Books.Title)
Where Borrowers.Card_Number = Loans.Card_Number
And  Loans.Book_Number = Books.Book_Number
And  Books.Author = Borrowers.Name
```

In simple English:

Find all Authors who borrowed books written by themselves.

Now consider the projection of an example database instance on relevant domains:

Borrower		**Loans**		**Books**	
Name	*Card_Number*	*Card_Number*	*Book_Number*	*Book_Number*	*Author*
Jones	J312	J312	H115	H115	Jones
Smith	S222	S222	Q019	Q019	Brown
Brown	B845	B845	E772	E772	Smith

3.1 The Ingres Query Processing Algorithm

The original Ingres algorithm [Wong 76] features recursive tuple substitution. The original query is only modified by detaching subqueries and replacing variables by temporary results from subqueries. Subquery target lists only include inner relation domains that are needed for other equality clauses in the main query. The outer relation, corresponding to the substituted variable, is not needed again until the next tuple is recursively substituted.

A variable is selected for substitution. Then a tuple from the substituted variable replaces that variable in the main query. One variable subqueries are then detached from the main query for each variable that is related to the substituted variable. The results from these subqueries replace the variables for each variable that was originally related to the substituted variable. The process is repeated recursively until all variables are processed. It then proceeds recursively for each tuple of each variable up until all tuples of all variables have been processed.

If we consider substitution for Borrowers, and detach all subqueries for that variable, we get the decomposition:

Main Query

```
(1) Retrieve (Borrowers.Name)
        Where (Temp1.Book_Number = Temp2.Book_Number)
```

Subqueries

```
(2) Retrieve (Loans.Book_Number)
        into (Temp1)
        Where (Loans.Card_Number = Borrowers.Card_Number)

(3) Retrieve (Books.Book_Number)
        into (Temp2)
        Where (Borrowers.Name = Books.Author)
```

For the standard Ingres tuple substitution, we execute 2 and 3 for each substituted tuple from Borrowers. For the first tuple (Jones, J312) the result of subquery 2 is (H115) and the result of subquery 3 is (H115). Recursively executing the main query gives the result (Jones). For the second tuple (Smith, S222) the result of subquery 2 is (Q019) and the result of subquery 3 is (Brown) so substitution of (Smith, S222) from Borrowers fails to produce a

meaningful result. Similarly, substitution of Borrowers tuple (Brown, B845) fails to produce a result tuple.

3.2 Our Query Processing Algorithm

Our algorithm is similar to the original Ingres algorithm except that the algorithm is not invoked recursively for each tuple that is substituted. Instead, subqueries are executed for each tuple of a substituted variable before further processing on the main query proceeds, using the output of the subqueries as replacement for the inner variables of those subqueries. When all variables have been substituted, processing is complete. This method avoids repeating the decomposition process for each tuple of each relation. Multiple detached subqueries can be executed in parallel on a distributed system.

When the query was processed recursively, each domain from a substituted tuple in the target list of a subquery was associated with the same tuple of the outer relation for that subquery. We must assure that this relationship holds for further processing of temporary relation output from subqueries when subqueries run to completion. To do so, we must include domains from the outer relation in those temporary relations. We also must modify the original query to transitively equate all domains from the outer relation that are used in subquery equality clauses. We must also, as in the original Ingres algorithm, include domains from the inner relation that are needed in other equality clauses in the main query in the subquery target lists.

Adding the joining variable from Borrowers to Temp1 and Temp2 and adding the clauses equating those variables to the main query (query 1) gives the decomposition:

Main Query

(1) Retrieve (Temp1.Name)
Where (Temp1.Name = Temp2.Name)
And (Temp1.Book_Number = Temp2.Book_Number)
And (Temp1.Card_Number = Temp2.Card_Number)

Subqueries

(2) Retrieve (Loans.Book_Number, Borrowers.Name, Borrowers.Card_Number)
into (Temp1)
Where (Loans.Card_Number = Borrowers.Card_Number)

(3) Retrieve (Books.Book_Number, Borrowers.Name, Borrowers.Card_Number)
into (Temp2)
Where (Borrowers.Name = Books.Author)

The results of subqueries 2 and 3 are:

Subquery 1 (Temp1)

Name	*Book_Number*	*Card_Number*
Jones	H115	J312
Smith	Q019	S222
Brown	E772	B845

Subquery 2 (Temp2)

Name	*Book_Number*	*Card_Number*
Jones	H115	J312
Smith	E772	S222
Brown	Q019	B845

So, the result of the main subquery is (Jones), the correct result.

Considering the example decomposition of section 3.1 we see that addition of clauses to the main query is necessary

for correct operation. Executing our non-recursive algorithm without additional equality clauses and subquery output domains, we compile the results from subqueries 2 and 3 and then execute subquery 1. The result of subquery 2 is (H115) (Q019) (E772) and the result of subquery 3 is (H115) (E772) (Q019). Executing subquery 1 should give suppliers associated with the resulting part numbers (H115), (Q019) and (E772). This incorrect result illustrates the need for the modifications to the original algorithm to remove tuple recursion that we introduced here. The functionality that we add by modification of the main query and subqueries for our algorithm is handled by the recursive algorithm by "setting" target list variables in the main query, then executing the remainder of the query recursively. If the remainder of the query executes successfully, the target list value is output.

3.3 The Epstein Query Processing Algorithm

Epstein's "splitting" algorithm [Epstein 80] is similar to our algorithm except that it is restricted to single subquery detachment. The requirement to add clauses to the main query to transitively equate outer relation domains from subquery temporary output relations is not needed since only one such relation is produced for each processing step. Domains from the outer relation are still needed in the temporary relation for the subquery for input to other subqueries that refer to the outer relation. Inner relation domains found elsewhere in the query must also be output and their variables set to refer to the temporary output relation as in each case above.

Selecting Borrowers as the substituted variable in the main query and detaching the clause equating the Borrowers and Loans relations leads to the decomposition shown below.

Main Query

```
(1)     Retrieve (Temp1.Name)
            Where (Temp1.Book_Number = Loans.Book_Number)
            And   (Loans.Card_Number = Temp1.Card_Number)
```

Subquery

```
(2)     Retrieve (Borrowers.Name, Borrowers.Card_Number, Books.Book_Number)
            into (Temp1)
            Where (Books.Author = Borrowers.Name)
```

The results of subquery execution is:

Subquery 2 (Temp1)

Name	*Book_Number*	*Card_Number*
Jones	H115	J312
Smith	E772	S222
Brown	Q019	B845

So, the result of the main subquery (subquery 2) is (Jones), the correct result.

4 Distributed Hash Join Query Processing Algorithms

The hash join algorithm that has been discussed extensively in several papers [DeWitt 84], [Kits 83] performs better than most currently available algorithms for join processing. There are two phases to hash join query processing. In the first phase, the two relations to be joined are partitioned into "buckets" according to hash key value ranges for the join domain in each relation. Care is taken to assure that hash key value ranges are chosen so that buckets are approximately the same size for the inner relation and that inner relation buckets fit in main memory. In the join phase, each inner relation bucket is read into memory. The corresponding outer relation buckets are read and a hash join performed to determine the qualifying inner relation tuples.

[Dewitt 85] extends the hash join algorithm to allow parallel processing for multiprocessor systems. In the partitioning phase, buckets are distributed to remote sites instead of writing them to the local disk. The join phase can be executed in parallel when distribution is complete. In some cases, when enough processors are available and sufficient memory is available, the partition and join phases may overlap. A distributed version of DeWitt's hybrid hashing technique allows such processing to overlap, at least for the first partition to be processed at each join site. In his study of multiprocessor hash join, DeWitt assumed that data was horizontally partitioned. He also assumed that it was possible to fit the inner relation into distributed memory. He studied the performance of his algorithm for two way join processing.

We consider 3-way hash join processing and study the impact of partitioned data cache distribution on performance. Little has been done to study 3-way join processing. For example, [Richardson 88] does not consider them in his study of parallel pipelined join algorithms. The assumption is that classic nested join techniques are sufficient. Standard pipelined nested join query processing techniques are well suited to single site processing but place unnecessary restrictions on distributed hash join algorithms. In nested join, a join query such as JOIN(A, B, C) would be processed by first processing a two variable join such as TEMP = JOIN(A, B), then using that result to form another two variable query, JOIN(TEMP, C). This solution is efficient for both standard single site DBMS [Selinger 79] and distributed DBMS (DDBMS) [Epstein 80].

The problem with the pipelined approach is that they significantly restrict the amount of parallelism that can be exploited by distributed hash join query processing algorithms and hence limit the performance gains that they can achieve. The principle problem is that the pipelining approach requires that simultaneous execution of subqueries. But hash join algorithms require that each step in processing be completed before the next begins. The first inner relation partition must remain in memory until all tuples from the outer relation have been processed for the corresponding key range. But, when the outer relation is the result of execution of another subquery, tuples with join domain values for that key range may be produced at any time up until completion of that subquery.

The algorithms presented in section 3 above can be easily adapted for distributed hash join processing. Our algorithm allows parallel execution of multiple subqueries to increase parallelism and obtain improved performance. Our approach has been largely overlooked because, for techniques other than hash join, it restricts pipelining and causes greater delays than the first algorithm. The algorithm presented in section 3.3 is limited to execution of a single subquery at each step. The performance of these algorithms is compared in section 6 below. In section seven, we examine issues related to use of tuple recursive algorithms for distributed hash join query processing.

5 Cacheing Issues

Cacheing can have an important impact on accessibility of data and performance. Relations may not always be ideally distributed for a particular algorithm. Cacheing can help to reduce the cost of the expensive initial relation partitioning and distribution for parallel hash join processing. Attempting to use data where it is cached instead of moving it can also have an important impact on algorithm selection for join processing. Moving data into an optimal processing configuration then executing a join can in many cases be more expensive than the total cost of performing the join of the cached relations without moving them. We show the potential performance improvement

for cacheing in distributed hash join algorithms. Then we present a scenario in which relations may not be optimally cached, but for which our algorithm is particularly useful. In section 6, we analyze the performance of our distributed hash join algorithm and Epstein's distributed hash join algorithm for this scenario.

We use an analytic model for distributed hash join query processing, based on the model used in [DeWitt 84], to study the potential performance improvement for optimal cacheing of inner relations in a two-way join. In our model, main memory is evenly distributed among processing sites. Distributed memory is large enough to cache the partitioned inner relation. We choose the partition size to be equal to the size of memory, instead of choosing the smallest partition that could be chosen for the number of processors available. Choosing the smallest possible partition would have decreased the cost for the join phase but increased the cost for partitioning. While the model takes both I/O cost and cpu cost into account, I/O cost is most important distributed query processing. The inner relation for our test query had 1,284 pages and 35,952 tuples. The outer relation had 2,291 pages and 64,148 tuples. The same test data was used by [DeWitt 84] to study the performance of join operations in single site systems.

Figure 2 compares the cost for the distributed hybrid hash algorithm (DHH), including inner relation distribution, to the cost of the distributed hybrid hash algorithm with cacheing (DHH/C). For DHH/C, the inner relation is cached so that initial distribution of the inner relation is not required for join processing. For the hybrid hash algorithm, outer relation buckets may be processed as they arrive at remote sites. Since each site only processes a single partition (hash key value range) there is no need to write outer relation buckets to disk to await reading their corresponding inner relation tuples to a memory resident hash table. Memory size indicates the amount of memory available at each site. The figure shows that a significant performance gain can be achieved by initiating the first phase of hash join processing without paying the expense of relation partitioning and distribution for the inner relation.

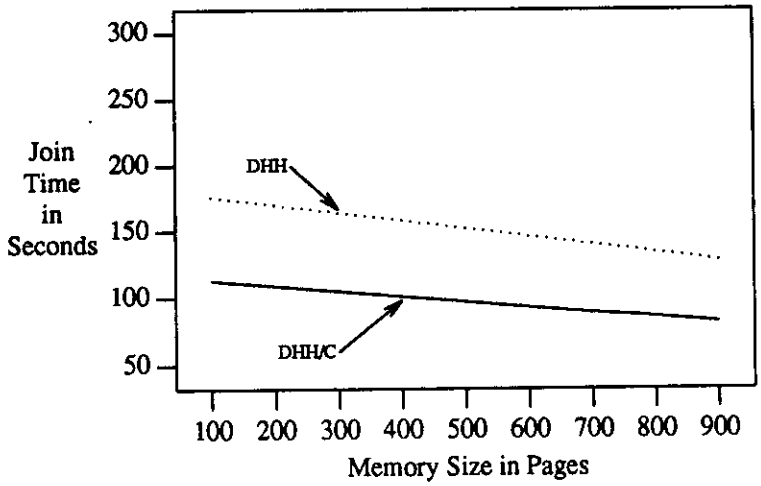

Figure 2 - Comparison of Hybrid Hash Algorithms

Data distribution and distribution of available memory and processing resources have a significant impact on the performance of distributed query processing. Data distribution in network transparent systems can be ignored by a user application. However, the impact of performance and autonomy concerns often dictate that care be taken in distribution. Distributed query processing takes place in organizations where each department stores data on multiple machines maintained locally, such as work stations. Other organizations distribute data without regard for departmental boundaries and local autonomy concerns.

In the *logically partitioned network model*, data is grouped according to departmental requirements. Join operations on relations that are in two independent departments may utilize resources in both departments. A common

occurrence would be for a central administration to require data that combined relations from two departments with a relation from the central administration. In this model, we would like to use an algorithm for join processing that can, as much as possible, take advantage of the natural partitioning of data. When logically partitioned networks are geographically distributed, the same set of tradeoffs apply as when data is cached into subnets of a local area network. The cost of re-distribution of relations is high so processing of data at its current location results in better performance. The partitioning and distribution of relations into disjoint sub-networks is depicted in figure 3. In the figure, A, B, and C represent three relations. One third of each relation is distributed to each of the three sites in its sub-network. We do not consider data replication.

In the *database machine model* each machine is a member of a pool of equivalent processors. Every node can be used as easily as another. Network transparency allows data to be moved for maintenance or performance. In this model, most relational join operations will choose to distribute relations as much as possible to reduce the size of each piece of a join that may be processed in parallel. However, the cost of distibution for a relation may exceed its benefits if relations are already partitioned among the processing sites. The choice of sites for cacheing a partitioned relation depends on immediately available resources. This can often result in partitioned relations that are to be joined being disjointly partitioned among sites in the network as shown in figure 3 below. Our algorithm is especially useful for this configuration as we show in the next section.

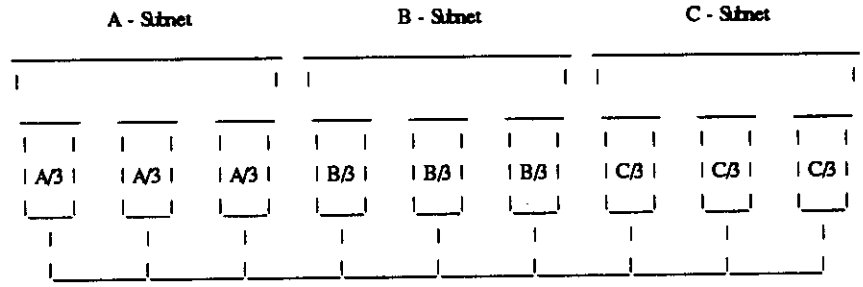

Figure 3 - Relations A, B and C each Partitioned at 3 Sites

6 Performance of 3-way Hash Join Processing

Three way distributed hash join query processing requires us to rethink the basic notion of nested pipelined query processing. Nested single variable subqueries can enjoy less pipelining and parallelism for techniques that require data partitioning. This is because results from a join operation cannot be input into another join operation until they are completed. The problem is that the second join cannot read the second partition into its hash table until all tuples have been matched for the first partition. But we can not, in general, be sure that all such matches have been found until the entire first join has been processed.

We study the performance of the two basic approaches for application of hash join techniques to distributed three way query processing that are not tuple recursive. *Nested hash join (NHJ)* results from application of hash join techniques to non-recursive single subquery detachment (section 3.3). *Parallel hash join (PHJ)* results from application of hash join techniques to non-recursive multiple subquery detachment (section 3.2). We will describe exactly how each of these algorithms operates and evaluate their performance. We assume cacheing according to figure 3 as discussed in section five. The basic queries and processing orders that we compare for NHJ and PHJ are illustrated by figure 4. We propose performance models for NHJ and PHJ below and then compare their performance. The parameter p represents join selectivity in our model. In our analysis, we also assume that the initial relations, A, B, and C are each the same size. A, B and C represent the number of pages in each source relation. N is the number of sites where each of the source relations is initially partitioned.

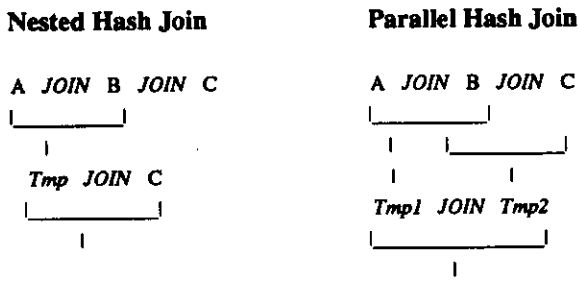

Figure 4 - Join Order for Test Queries

6.1 Nested Hash Join Performance

The NHJ algorithm joins relation A with relation B at the B storage sites. The result of that join is partitioned for use in joining with relation C. We assume that the I/O of the partitioned temporary relation at C storage sites is done in parallel with Join(A, B) processing. The cost of temporary relation distribution is just the cost of I/O for the temporary relation at B storage sites, (pB)/N. This is in addition to the cost of Join(A, B) at the B storage sites. The Join with C, except for the initial partition, cannot proceed in parallel with Join(A, B). The first partition of C at each C storage site can be used to join with corresponding partition buckets from the output outer relation tuples resulting from Join(A, B). This potential overlap for processing of the first partition is ignored in our simple analysis. The model for performance of cached NHJ is presented here.

Step 1: Join A and B at B storage sites. - (A/N) + (B/N)

Step 2: Partition Join(A, B) at C storage sites. - (pB)/N

Step 3: Join temporary relation with C. - (C/N) + (pB)/N

The total I/O cost for the NHJ algorithm (recall that A = B = C) is:

$$NHJ = (A/N) + (B/N) + (pB)/N + (C/N) + (pB)/N$$

$$NHJ = 3*(A/N) + 2*(pB)/N$$

6.2 Parallel Hash Join Performance

The PHJ algorithm joins relations A and B in parallel with relations A and C at the B and C storage sites. Temporary result output from these joins is partitioned among the resources at the 2N sites used for the initial joins. Each site will have (pB)/N pages after distribution of both temporary relations to the 2N processing sites used for Join(A, B) and Join(A, C). This is the same number of result output pages that each will have after the initial join step. So each site must output (pB)/N - (pB)/(2N) pages of the data to the various sites on the network. Each site must input approximately the same amount of data from the network. Storing the partitioned data on the local disk at each site costs (pB)/N. The distribution step is done simultaneously, but not in parallel with the join step for parallel execution of Join(A, B) and Join(A, C). The cost for distribution is then:

$$2 * [(pB)/N - (pB)/(2N)] + (pB)/N = 2 * (pB)/N$$

That is each site must do about 2*[(pB)/N] I/Os for the distribution. The third step in processing involves a distributed hash join of the two temporary relations that result from the first step. (pB)/2N pages from each temporary relation are joined at each of the 2N processing sites. The model for performance of cached PHJ is

presented here.

Step 1: Join A and B at B storage sites. - (A/N) + (B/N)
(and in parallel Join A and C)

Step 2: Partition Join(A, B) and Join(A, C) at 2N sites. - 2*(pB)/N

Step 3: Join temporary relations. - (pB)/N

The total I/O cost for the PHJ algorithm (recall that A = B = C) is:

$$PHJ = (A/N) + (B/N) + 2*(pB)/N + (pB)/N$$

$$PHJ = 2*(A/N) + 3*(pB)/N$$

6.3 Performance Comparison

We compare the performance of NHJ and PHJ using the simple models described above. We assume that each of the three relations is distributed evenly among three sites. The total number of sites that we use is then nine. The three source relations are assumed to be 100 pages each.

Figure 5 shows how the performance of the NHJ and PHJ algorithms is affected by changes in p, the join selectivity. We observe that, for small values of p, the number of I/Os required for PHJ is significantly lower than for NHJ as a percentage of total I/O. For low join selectivity, the savings should be approximately the cost of input for the source relation that results from performing Join(A,B) and Join(A,C) in parallel, avoiding the cost of input for one of the three source relations. We see that this is the case exactly when p is equal to zero. The difference in cost for NHJ and PHJ is then A/N. As the amount of I/O associated with redistribution of the temporary relation increases for increased join selectivity, the savings decrease to the point that the cost for NHJ is equal to that for PHJ. This point occurs when join selectivity is one as we show below.

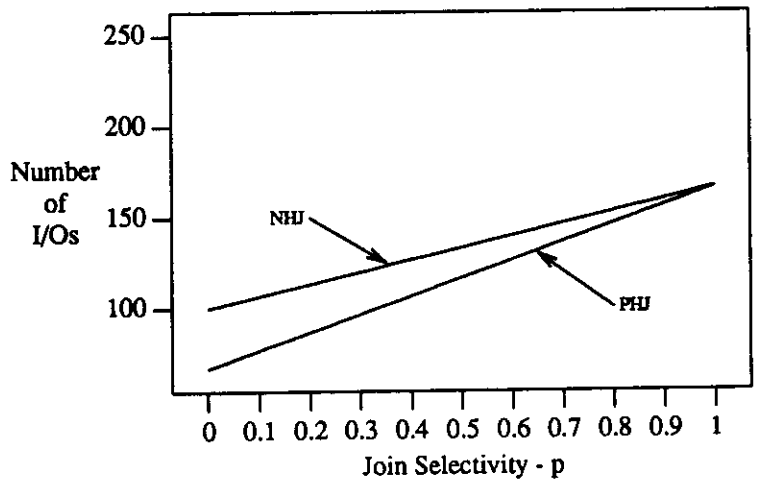

Figure 5 - Comparison of NHJ and PHJ Algorithms for variation in join selectivity - p

Figure 6 shows how the performance of the NHJ and PHJ algorithms from sections 6.1 and 6.2 above is affected by

changes in the size of source relations. We assume that all three relations are the same size. The graph uses a join selectivity of .2. We observe that savings in I/O for the PHJ algorithm are directly proportional to relation size for fixed selectivity. The graph shows that we achieve a greater improvement in performance for larger relations.

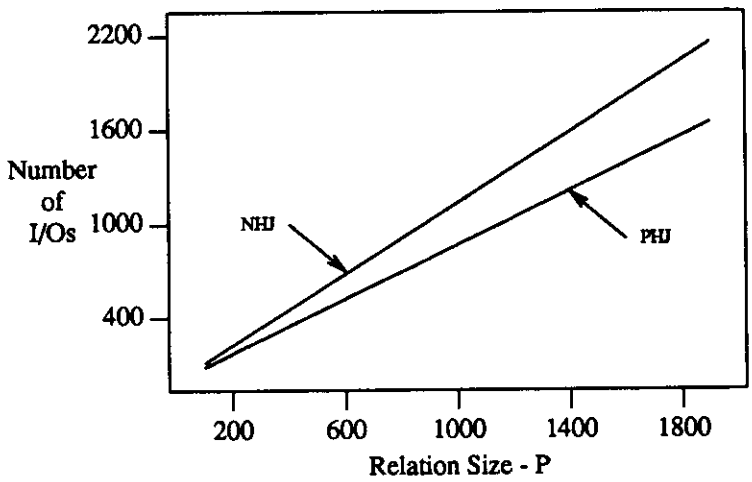

Figure 6 - Comparison of NHJ and PHJ Algorithms For varying Relation Size

7 Tuple Recursion in Query Processing

The recursive tuple substitution algorithm (section 3.1) may be useful for distributed processing of complex queries for relational database support of knowledge based systems. It has the advantage that it may return a query results more quickly than our other algorithms. Applying hash partitioning to tuple recursion techniques offers the same benefits as for other techniques. Matching inner relation tuples are found in memory hash tables for all outer relation tuples considered. There are two principle reasons that little research has been done to study extension of this algorithm for distributed query processing. First, the cost of network communication for each tuple substituted seemed to be high compared to batch processing of tuples for a subquery. Second, repeated decomposition of the query for each tuple substituted seems to be inherently inefficient.

Network communication can be reduced significantly through bitmap filtering techniques [Brat 84]. These techniques can be used to improve performance for many query processing techniques. A bitmap indicates the set of hash values that match tuples in a relation for a particular domain. The bitmap can be used to avoid unnecessary I/O in the partitioning phase of a hash join algorithm. The join domain hash value for each tuple of a relation that is being partitioned is checked to determine if it is in the bitmap for the other relation involved in the join. For cached hash partitioned relations that were not bitmap filtered, a bitmap can still be used to avoid network transmission of tuples that have no match.

The first step in a study to determine when tuple recursive distributed hash join may be beneficial is to quantify the overhead for tuple recursion for that algorithm. We can then begin to experiment in a distributed system to quantify other parameters needed for effective optimization. To quantify the cost of repeated decomposition of the query for each tuple substituted we compared the standard Ingres algorithm with an implementation of Epstein's algorithm in a single site system. [Epstein 80] showed that he could remove the recursion from the University Ingres query processing algorithm to facilitate distributed pipelined join processing. His approach detached a single subquery and executed it with the output used for processing the remaining query at a remote site. The remaining query was similarly decomposed to transform the query into a pipeline of single subqueries operating in parallel at several

sites. He demonstrated this to be an effective technique for distributed query processing.

We implemented the Epstein 3-way join processing algorithm in order to study its effectiveness in a single site environment. Eliminating recursion was shown by Epstein to be an effective approach to distribution but he did not study whether it would be an effective technique for single site database implementation or quantify the effect that it might have on single site processing. In fact, the inefficient University Ingres recursive algorithm was left intact for single site processing. The 1986 University Ingres release uses the classic recursive tuple substitution algorithm. We show quantitatively the value of removing the recursion from the University Ingres algorithm. The use of Epstein's algorithm for a single site achieved significant performance gains.

In our experiment, we used the library database introduced earlier (see figure 1) as our test database. We used four example queries and two database instances to provide quantitative performance comparison of the two algorithms. The queries are shown in Appendix I. Tuples in each relation had 128 characters. Significant domains in each relation were modeled using seven characters. One domain in each relation was sequentially numbered. Other domains were randomly generated. For database 1, relation cardinalities were 999 999, 1999, 2999, and 3999 tuples. For Database 2, each relation had 99 tuples. The experimental queries executed varied in join selectivity as well as source relation size and number of join clauses. The results for our experiment are shown in figure 7. The tuple substitution algorithm is designated TS and the other algorithm that is not tuple recursive is designated NR. Response times are given in seconds.

Experiment #	Query #	Database #	Result Tuples	Response Time TS	Response Time NR
1	1	1	1083	11488.3	5960.2
2	2	1	1	2934.7	2209.2
3	3	1	8	2683.4	2212.1
4	4	1	0	1068.5	289.6
5	1	2	99	265.1	82.9
6	2	2	0	26.6	25.1
7	3	2	0	26.9	25.3
8	4	2	0	17.1	15.5

Figure 7 - Experimental Results

Our experiment showed that Epstein's non-recursive single subquery detachment is superior to the classic recursive tuple algorithm with multiple subquery detachment. The non-recursive algorithm performed better for all cases. Figure 8 shows the results of our implementation compared to the classic algorithm, plotted against a logarithmic scale. The recursive algorithm spends a significant amount of time repeatedly generating query trees and deleting them, for each tuple substituted from each relation. Removal of recursion significantly reduces this overhead by allowing subqueries to run to completion and then using their output for input to the next processing step.

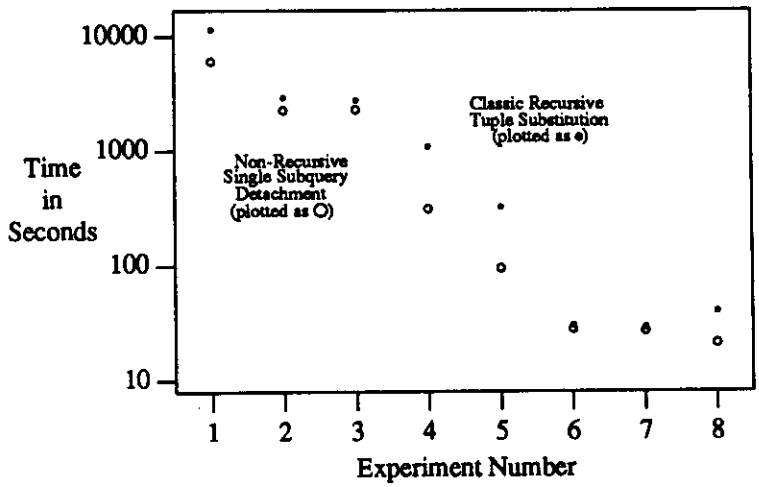

Figure 8 - Comparison of Recursive and Non-Recursive Algorithms

Distributing tuple substitution can be a reasonable approach for query processing when it is important to return the "next" result as soon as possible. Tuple substitution would likely be most useful when join selectivity is greater than one. In that case, we would be more likely to find the first result in less time than it would take to generate all results using an algorithm such as ours. For a query with low selectivity, our algorithm would perform well and a tuple recursive algorithm would be less likely to return a result quickly.

8 Conclusions and Future Research

We introduced a new algorithm for 3-way join query processing. Our algorithm is similar to the recursive University Ingres algorithm, but without recursion. It differs from Epstein's algorithm [Epstein 80] in that we detach multiple subqueries whereas Epstein is limited to single subquery detachment. We then described algorithms for distributed hash join processing of nested queries and potential limitations for hash join processing. Issues in data cacheing motivated the consideration of an important scenario for distribution of partitioned relations. An analytic model helped to quantify the performance advantage that our algorithm has for the distribution that we considered.

Our work was motivated by two principle observations. First, we recognized that for hash partitioned join processing pipelining approaches were less significant than for other techniques. Rather, increased parallelism would have to be found in each step of join processing. This is because a join step must complete before the first partition of the inner relation for the next join step can be replaced in memory. Second, consideration of cacheing issues lead us to propose a new query processing algorithm that can improve performance when partitioned relations are not ideally distributed for processing by existing algorithms. A performance study, based on an analytic model, helped to quantify the potential for performance improvement that can be achieved by our algorithm.

Bitmap filters were introduced above in section seven. In addition to their potential use in a distributed tuple recursive query processing, they could be applied to our proposed nested join processing algorithm. Cacheing bitmap filters is also potentially advantageous. However cacheing partitioned relations that have been filtered is not very useful since they could only be used for join operations with the filtering relation.

We discussed the potential for a tuple recursive distributed query processing algorithm. Experimentation showed that non-recursive single subquery detachment is better than recursive multiple subquery detachment, even for single site queries. Measured performance for our experiment helped to quantify the overhead for a simple tuple

recursive algorithm This study showed the potential for improved performance from relation cacheing, including cacheing of hash partitioned relations.

Consideration of new technology, including hash join query processing techniques and bitmap filtering lead us to introduce a new algorithm for distributed nested query processing. Increasing database size and complexity, including requirements for support of knowledge based applications motivated a re-evaluation of the potential of tuple recursive query processing techniques for distributed query processing. Further study of these techniques, including the affect on performance of variations in relation distribution, will require extensive experimentation in a distributed system testbed that offers facilities for database experimentation. In particular, distributed resource maps for memory and hash partitioned relations can be supported by a distributed system. DDBMS control of distributed cache resources including memory will require that new facilities be provided by a distributed operating system. Our study supports the hypothesis that for appropriate algorithms, such DDBMS control can be used productively. Experimentation will verify the analytic results that we presented.

Appendix I - Test Queries

Query 1:

```
range of a is Borrowers
range of b is Borrowers
range of c is Borrowers
range of d is Borrowers
retrieve ( a.Name, d.Name )
where ( a.Address = b.Address  and
        b.City = c.City  and
        c.Card_Number = d.Card_Number )
```

Query 2:

```
range of borr is Borrowers
range of loan is Loans
range of book is Books
range of publ is Publishers
retrieve ( borr.Name, publ.City )
where ( borr.Card_number = loan.Card_Number  and
        loan. Book_Number = book.Book_Number  and
        book.Publisher = publ.Name )
```

Query 3:

```
range of borr is Borrowers
range of loan is Loans
range of book is Books
retrieve ( borr.Name, book.Title )
where ( borr.Card_Number = loan.Card_Number  and
        loan.Book_Number = book.Book_Number )
```

Query 4:

```
range of borr is Borrowers
range of loan is Loans
range of book is Books
range of publ is Publishers
retrieve ( borr.Name,publ.City )
where(  borr.Name = book.Author  and
        borr.Address = publ.Address  and
        borr.Card_Number = loan.Card_Number  and
        book.Publisher = publ.Name  and
        book.Book_Number = loan.Book_Number  and
        loan.Library_City = publ.City )
```

Appendix I - Test Queries

Query 1:

```
range of a is Borrowers
range of b is Borrowers
range of c is Borrowers
range of d is Borrowers
retrieve ( a.Name, d.Name )
where ( a.Address = b.Address  and
        b.City = c.City  and
        c.Card_Number = d.Card_Number )
```

Query 2:

```
range of borr is Borrowers
range of loan is Loans
range of book is Books
range of publ is Publishers
retrieve ( borr.Name, publ.City )
where ( borr.Card_number = loan.Card_Number  and
        loan. Book_Number = book.Book_Number  and
        book.Publisher = publ.Name )
```

Query 3:

```
range of borr is Borrowers
range of loan is Loans
range of book is Books
retrieve ( borr.Name, book.Title )
where ( borr.Card_Number = loan.Card_Number  and
        loan.Book_Number = book.Book_Number )
```

Query 4:

```
range of borr is Borrowers
range of loan is Loans
range of book is Books
range of publ is Publishers
retrieve ( borr.Name,publ.City )
where(  borr.Name = book.Author  and
        borr.Address = publ.Address  and
        borr.Card_Number = loan.Card_Number  and
        book.Publisher = publ.Name  and
        book.Book_Number = loan.Book_Number  and
        loan.Library_City = publ.City )
```

References

[Richardson 88] Richardson, "Design and Evaluation of Parallel Pipelined Join Algorithms", ACM SIGMOD, 1988.

[Brat 84] Bratbergsengen, Kjell, "Hashing Methods and Relational Algebra Operations", Proceedings of the 1984 Very Large Database Conference, August, 1984.

[DeWitt 84] DeWitt, David J., Katz, Randy H., Olken, Frank, Shapiro, Leonard D., Stonebraker, Michael R., Wood, David, "Implementation Techniques for Main Memory Database Systems", ACM SIGMOD Conference Proceedings, 1984.

[DeWitt 85] DeWitt, David J., "Multiprocessor Hash-Based Join Algorithms", Proceedings of the 1985 Very Large Database Conference, 1985

[Epstein 80] Epstein, R., "Query Processing Techniques for Distributed, Relational Data Base Systems", Memorandum No. UCB/ERL M80/9, March 15, 1980. Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720.

[Kits 83] Kitsuregawa, M. et al, "Application of Hash to Database Machine and its Architecture", New Generation Computing, No. 1, 1983, 62-74.

[Nakayama 88] Nakayama, M., Kitsuregawa, M., Takagi, M., "Hash-Partitioned Join Method Using Dynamic Destaging Strategy", Proceedings of the 1988 Very Large Database Conference, Los Angeles, 1988.

[Selinger 79] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., and Price, T.G., "Access Path Selection in a Relational Database Management System", ACM SIGMOD 1979.

[Ullman 82] Ullman, J.D., "Principles of Database Systems", Second Edition, Computer Science Press, 1982.

[Wong 76] Wong, E. and Youssefi, K., "Decomposition - A Strategy for Query Processing", ACM SIGMOD International Conference on Management of Data, Washington D.C., June 1976.

[Wong 83] Wong, E. and Katz, R.H., "Distributing a Database for Parallelism", ACM SIGMOD Conference Proceedings, 1983.