SOME EXPERIMENTAL RESULTS ON DISTRIBUTED JOIN ALGORITHMS IN A LOCAL NETWORK

by

Michael J. Carey and Hongjun Lu

Computer Sciences Technical Report #587

March 1985

Some Experimental Results on Distributed Join Algorithms in a Local Network

Michael J. Carey
Hongjun Lu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by the Wisconsin Alumni Research Foundation, National Science Foundation Grant Number DCR-8402818, and an IBM Faculty Development Award

Some Experimental Results on Distributed Join Algorithms in a Local Network

Michael J. Carey
Hongjun Lu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

A number of algorithms have been proposed for processing distributed join queries in recent years. In contrast to the large amount of work that has been done on algorithms, relatively little work has been done on the performance of distributed join methods. This paper presents some experimental results on the performance of distributed join algorithms in a local network. Eight different join methods have been implemented in an experimental distributed system, the Crystal multicomputer, and tested for join queries with a variety of relation sizes, join selectivities, and join column value distributions. The results obtained indicate that pipelined join methods outperform sequential methods over a wide range of join queries. It was also found that the communications costs in a local network environment are not a dominant factor with respect to performance, and that shipping a whole relation from one site to another is not an unreasonable decison as long as it is done in the framework of pipelined algorithms. Two pipelined variants of a centralized nested loop join algorithm (with an index) were found to perform well for many of the queries tested.

1. INTRODUCTION

In relational database systems, queries are typically posed in a high-level, nonprocedural query language based on the relational calculus such as QUEL or SQL [Ullm82]. It is the task of the query optimizer to decompose the query into primitive relational operations such as selection, projection and join. The join operator has attracted a great deal of research interest since the cost of available join methods tend to vary widely with the characteristics of the data and the available access paths [Blas76, Seli79, DeWi82a, Brab84, DeWi85]. The join operation and join orders have also been the primary focus of many query optimization algorithms for centralized database systems [Wong76, Seli79]. The distribution of data in distributed database systems further increases the complexity and tradeoffs associated with the join operation. A great deal of research work has gone into the problem of developing distributed query processing algorithms, particularly into finding good methods for performing distributed joins. A new operation, the semijoin, was introduced as

This research was partially supported by the Wisconsin Alumni Research Foundation, National Science Foundation Grant Number DCR-8402818, and an IBM Faculty Development Award.

a way of minimizing the communications cost for performing distributed joins, and a number of researchers have addressed the problem of finding optimal semijoin sequences for various classes of queries [Bern81, Bern81a]. A related method, known as "fetch the inner tuple as needed", is among the algorithms used for processing distributed joins in System R^* [Seli80, Lohm84].

Along with the development of new join algorithms has come research on the performance of the various algorithms. Four methods for processing a general select-project-join query in a centralized database system were investigated analytically and their performance was compared in [Blas76]. They found that, in most cases, either the nested loop algorithm or the sort-merge algorithm offered the best (or close to the best) performance. Other researchers have also analyzed alternative centralized join algorithms [Nieb76, Yao78, Yao79, Brab84]. There have also been a few studies of distributed query processing performance issues. Epstein and Stonebraker tested 14 versions the of Distributed INGRES query processing algorithms [Epst80]. Their results indicated that exhaustive search performs consistently better than limited search, that dynamic optimization is beneficial, and that using a "worst case" estimate for intermediate relation sizes is overly pessimistic. Some simulation results on the performance of different join strategies for a distributed database system based on a star computer network were presented in [Kers82], and the effects of join selectivity and communications speed on the optimal algorithm choice were investigated in this context. Some results on query processing in a locally distributed system were reported in [Page83], where query processing costs for the INGRES database system were analyzed in the context of the LOCUS distributed operating system.

Compared to the number of distributed query processing *algorithm* papers in the literature, relatively few papers have addressed the *performance* of distributed join algorithms. This is one of the major motivations which lead us to perform this study. We have implemented eight different distributed two-way join methods in an experimental locally distributed computer system at the University of Wisconsin. The join methods examined in our study include methods based on both traditional joins and semijoins using several different access methods and data transfer strategies. The work reported here differs from most of the work mentioned above in the sense that it is an *empirical* study — we present measurements of actual response times, disk activity, and message transfers for the join algorithms that we consider.

The organization of the remainder of the paper is as follows. The eight different join algorithms that we studied are described in section 2. Section 3 provides an overview of our experimental environment. Our experiments are described and the results are presented in Section 4. Section 5 summarizes what we have learned from this study and its influence on what we plan to do in the future.

2. DISTRIBUTED JOIN METHODS

Given an equi-join query R_a[A=B]R_b in a distributed database system, where R_a and R_b reside at (different) sites S_a and S_b, respectively, there are a number of distributed join methods available for processing it. Considering all of the possible combinations of access paths, local processing algorithms, execution paradigms, etc., would lead to a prohibitively large search space for an empirical investigation. Thus, to reduce the number of experiments necessary we categorize distributed join methods along three dimensions and consider the following options in each:

(1) General approach — "traditional" join methods versus semijoin.

(2) Execution paradigm — sequential versus pipelined execution for the pair of sites involved.

(3) Local join processing — sort-merge versus nested loop (with an index).

In the remainder this section we elaborate on each of these dimensions, and we then present descriptions of our implementations of eight join algorithms that are produced by combining options from these dimensions.

2.1. Join Versus Semijoin

The semijoin operator was introduced as a primitive for processing distributed queries with less data transfer than traditional join methods [Bern79a, Bern79b, Bern81, Bern81a]. Using the semijoin method, only the join column values of one relation and the matching tuples of the second relation need to be transferred between the two sites. If intersite data transfer is expensive, the join field width is relatively small compared to the width of an entire tuple, and there are not many matching tuples, the use of semijoins can result in a significant savings. In local area networks, however, the data transfer rate between two sites is much higher — on the same order of magnitude as that between memory and a local disk. It is questionable whether or not semijoins will be beneficial in such an environment, as using them requires multiple scans of one of the source relations, and therefore more disk accesses.

2.2. Sequential Versus Pipelined Processing

When an operation at a site requests remote data, as in a distributed join, a choice exists — the two sites can work in either a sequential fashion or in a pipelined fashion. If the sites work sequentially, the site receiving data will not begin its processing until all of the required data has arrived. In the pipelined case, processing will begin at the receiving site as soon as the first tuple or packet of data has arrived. One advantage of the pipelined approach is its parallelism — the two sites work in parallel, so the elapsed time for the query will be reduced in proportion to the amount of overlapped processing. Second, and perhaps more important, is the fact that the receiving site doesn't actually store the incoming data in a temporary relation, thus saving the time required to store and then re-retrieve the data received from the remote site.

2.3. Sort-Merge Versus Nested Loop Join

Since any distributed join involves local processing, the join algorithm and associated access methods are still important factors in a distributed database system. For centralized joins, it was found in [Blas76] that, except for very small relations, the nested loop join or sort-merge join methods were always optimal or near optimal. We thus chose these two local join methods to implement for our distributed join experiments. For the nested loop join method, we assume the availability of a B+ tree index on the join column of the inner relation, as would likely be the case in System R^* [Seli80, Lohm84].[1] If the inner relation is shipped to the outer relation's site, a secondary B+ tree index (i.e., non-clustered) is constructed for the inner relation at the outer site because the WiSS system [Chou83] does not support clustered indexes on non-unique (i.e., non-key) attributes. In addition, it is not always reasonable to assume that the join column will have a clustered index available.

[1] The cost of a nested loop join without an index for relations of reasonable size is prohibitive [Bitt83], so we do not consider this possibility.

2.4. Join Algorithm Details

As described earlier, we have implemented eight join algorithms for our study. The algorithms, classified according to the three dimensions presented above, are SJSM, SJNL, PJSM, PJNL, SSSM, SSNL, PPSM, and PPNL. For the first letter, "S" stands for sequential and "P" stands for pipelined. The second letter, "J" or "S", is used to represent "traditional join" versus semijoin. The last two letters indicate either sort-merge ("SM") or nested loop ("NL") join. In the remainder of this section we describe each of these methods in turn. In our tests, site S_a initiates the join query and is both the join site and the result site.

2.4.1. SJSM and SJNL

The sequential join methods SJSM and SJNL are the simplest of the distributed join methods. The remote relation R_b is shipped to join site S_a as a whole. The two relations are then joined at site S_a using either the sort-merge method (SJSM) or the nested loop method (SJNL). Figure 2.1 illustrates the details of the two methods. For SJSM, the two relations are each sorted at their local sites to increase parallelism. For SJNL, a B+ tree index is built on the join column of the received relation $R_b{}'$ at site S_a as discussed above.

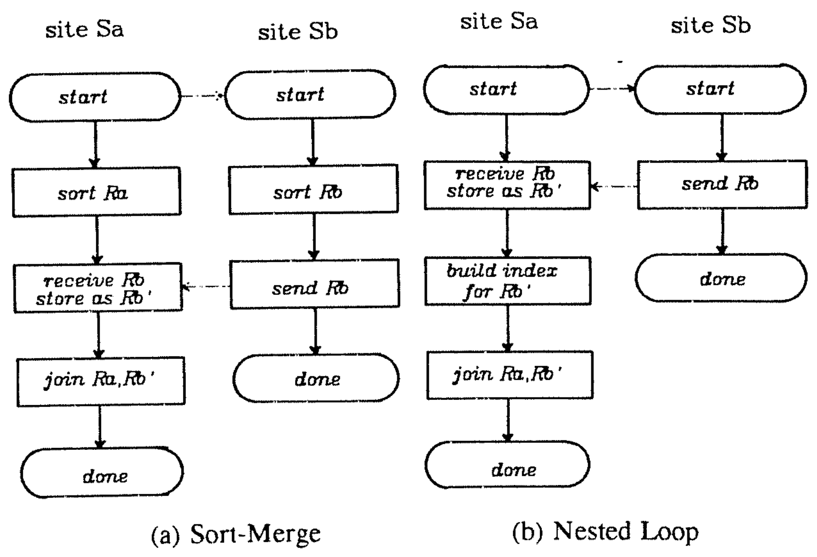


Figure 2.1: Sequential Join Methods.

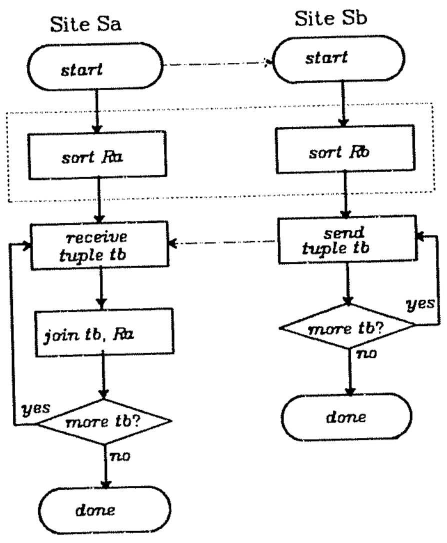


Figure 2.2: Pipelined Join Methods.

2.4.2. PJSM and PJNL

Like SJSM and SJNL, the PJSM and PJNL algorithms transfer the whole relation R_b from site S_b to site S_a. The difference is that R_b is not stored as a temporary relation at site S_a. Instead, tuples of R_b are joined with R_a tuples on the fly as they arrive, as shown in Figure 2.2 (with the sorting steps only being present in the sort merge case). Using the PJSM method, both relations are sorted first. Then, upon receiving tuples from relation R_b, scan cursors on R_a are incremented to find matching tuples. Matches are merged with the incoming tuples of R_b and written to the result relation. The scan cursor is then reset to the its last starting point in R_a and the process is repeated for the next group (packet) of R_b tuples. In the PJNL case, since no temporary relation R_b' is stored at site S_a, the local relation R_a always serves as the inner relation and the remote relation R_b serves as the outer relation. Both PJSM and PJNL can be viewed as distributed executions of centralized join algorithms.

2.4.3. SSSM and SSNL

SSSM and SSNL are two implementations of the semijoin method. One variation in our implementation is that the join column $R_a.A$, which is sent from site S_a to site S_b, is not stored on disk at site S_b — the

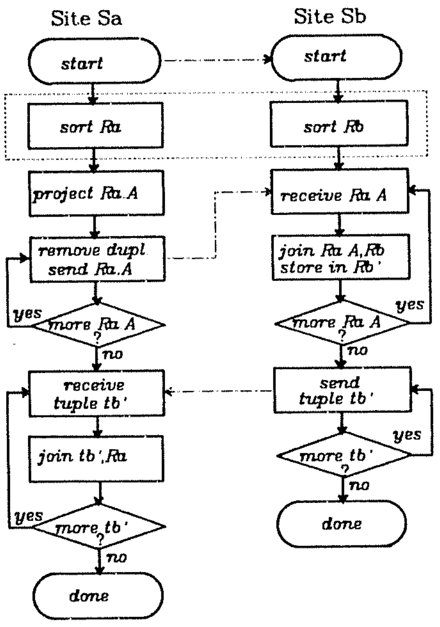


Figure 2.3: Sequential Semijoin Methods.

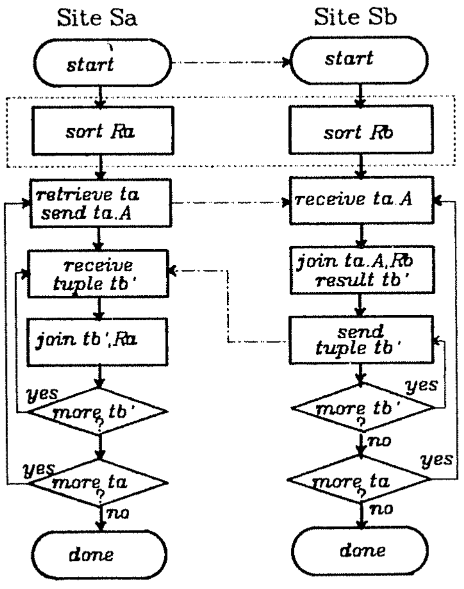


Figure 2.4: Pipelined Semijoin Methods.

incoming values are processed on the fly as they arrive. Similarly, the relation R_b', which is transferred back to site S_a, is processed on the fly as it arrives at S_a instead of being stored there as a temporary relation. Although there is therefore some limited pipelining involved in the SSSM and SSNL execution strategies, we still categorize them as being sequential as compared to the truly pipelined PSSM and PSNL algorithms to be described next. Figure 2.3 illustrates the details of the SSSM and SSNL methods.

2.4.4. PSSM and PSNL

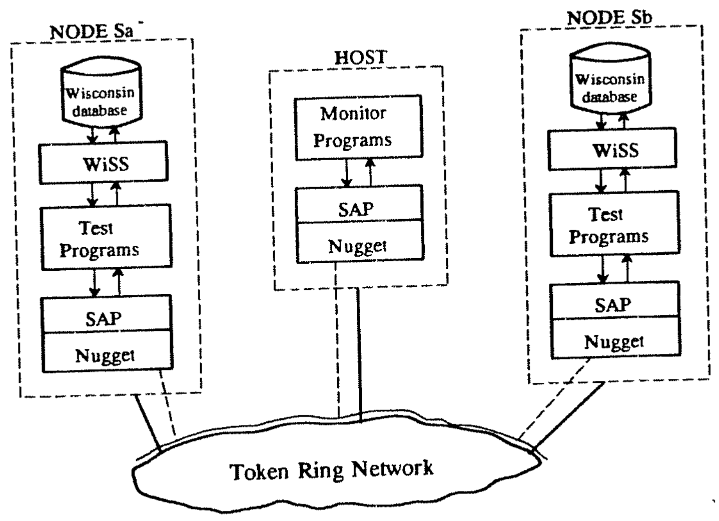
The pipelined semijoin methods, referred to as "fetch inner tuple as needed" in System R^* [Seli80], were the most complicated join methods to implement. As shown in Figure 2.4, relation R_a is scanned in a tuple-by-tuple manner (conceptually), and join column values $R_a.A$ are sent to site S_b. Upon receiving the $R_a.A$ values, site S_b selects the matching tuples from R_b and sends them to S_a; a null message is sent if there are no matching tuples. These tuples are then merged with the corresponding tuples of R_a, which are waiting for them (still in main memory). Our implementation actually processes R_a tuples in one-page batches, so one buffer page is allocated for keeping the tuples from R_a.

2.5. Discussion

The eight distributed join methods described in this section represent a range of possible methods. The sequential join methods, pipelined join methods, and pipelined semijoin methods are all among the methods used System R^* [Seli80, Lohm84], although our implementation may differ in minor ways. Of these methods, sequential join methods are attractive for their simplicity and the pipelined methods are attractive because they allow more concurrency and avoid the cost of storing and retrieving tuples from a temporary relation. The pipeline methods, of course, require some synchronization of the two processing sites (in the form of flow control, so the receiving site can indeed avoid having to store incoming tuples). One limitation of PJNL (the pipelined nested loops join method) is that R_a must be the inner relation, regardless of how its size compares to that of R_b, as the inner relation has to be available for multiple scans. The semijoin methods are attractive because they reduce communications costs. The main difference between the pipelined and sequential semijoin methods is related to duplicates — since the pipelined version simply scans R_a instead of projecting on $R_a.a$, it will send duplicate join column values if they are present in R_a; however, the sequential semijoin method requires multiple scans of R_a, increasing the local processing cost. Clearly, there are tradeoffs among all of these algorithms — these are the tradeoffs to be empirically investigated in Section 4.

3. THE EXPERIMENTAL TESTBED

Figure 3.1 depicts the testbed system used for our performance study. A collection of test programs were written to implement (hard-wired) distributed join queries using the different methods described in section 2. These programs access a synthetic database, the Wisconsin database [Bitt83], via WiSS (the Wisconsin Storage System) [Chou83]. The programs run on a pair of node machines from the Crystal multicomputer, an experimental distributed computer system [DeWi84]. Monitor programs run on a VAX/Unix host machine to initiate test program execution and to collect performance statistics after the test programs terminate. For communications between node machines, or between node machines and the host, we used a Crystal communications package called the Simple Application Package (SAP). In this section we briefly describe each of these components of the system.



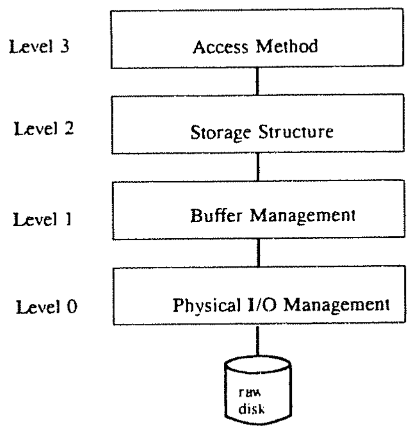


Figure 3.1: The Testbed for Distributed Join Methods. Figure 3.2: The Architecture of WiSS.

3.1. The Crystal Multicomputer

The Crystal multicomputer [DeWi84] currently consists of 20 DEC VAX 11/750's interconnected via a 10-Mb/sec Proteon token ring network. The ring network is also connected to several of the Computer Science Department's research VAXes, each of which can serve as a host machine. Crystal multicomputer users can claim a number of node machines as a partition. The partitions of different users in the system are logically isolated from each other — each partition is basically a virtual distributed computer system. For our distributed join algorithm experiments, we used a partition of two node machines with 2 megabytes memory and 160 megabyte Fujitsu disks to create a distributed database system testbed.

There are several levels of software available on Crystal. We used two of the lowest levels, the Crystal Nugget and the Simple Applications Package (SAP), to avoid unnecessary overhead. The Nugget is a simple communications kernel that resides permanently on each node machine, providing low-level message-passing primitives and ensuring that no messages are sent to nodes outside the user's partition (i.e., enforcing the logical isolation of partitions). SAP is a set of subroutines that sit on top of the Nugget, providing buffered communications using two queues, one for incoming messages and other for outgoing messages. With SAP, the sender busy-waits until a send buffer is available, fills the buffer with data, indicates its destination, and then invokes a non-blocking send routine. To receive a message, the receiver busy-waits on a flag that indicates that the input buffer is full. The flag is set when a message is received from the Nugget and reset when it is consumed by the user's program. Thus, SAP provides a somewhat higher-level message facility for our applications.

3.2. The Wisconsin Storage System

The Wisconsin Storage System, or WiSS, is an access-method level data storage system that can run either on top of Unix or directly on top of a "raw" disk [Chou83]. For our experiments, it was installed on the Crystal node machines and accessed their disks directly. WiSS manages devices, deals with physical storage allocation, manages buffer pages, and provides a primitive concurrency control mechanisms. From the viewpoint of a WiSS user, it serves to provide high performance sequential and indexed access to data on disk. Figure 3.2 illustrates the architecture of the WiSS system. WiSS implements an extent-based file system on top of the raw disk. The physical I/O management level is responsible for reading pages from and writing pages to the disk. The buffer management level maintains a buffer pool of pages; buffer page pools are associated with each user. When the buffer pool is full, the buffer pool manager makes a replacement decision using an LRU replacement strategy and some hints from the system about which pages are important. These hints give preference to pages such as B-tree root pages and system directory pages. Level 2 of WiSS is responsible for implementing the record-level storage abstraction. Sequential files, long data items, and B+ tree indexes (primary or secondary) are provided as structured files at this level. Finally, the highest level of WiSS implements the access methods of sequential scan, index scan, and long data item scan. This level also provides the routines for creating and destroying files, indexes and long data items. Some explicit control over scans, such as the capability to reset a scan cursor (search pointer), is also provided. Our test programs interface with WiSS mainly at level 3.

3.3. The Wisconsin Database

The Wisconsin Database was designed for use in systematically benchmarking relational database systems [Bitt83]. There are four basic relations in the database, referred to as "thoustup", "twothoustup", "fivethoustup", and "tenthoustup". These relations contain 1000, 2000, 5000, and 10,000 tuples, respectively. Tuples in all of these relations are 182 bytes long, each consisting of thirteen 2-byte integer attributes and three 52-byte string attributes. All of the integer attributes have uniformly distributed values, but the

unique1	unique2	two	four	ten	thousand	fivethous	odd	even	...
1347	6709	0	0	0	937	929	1	100	...
9354	1591	1	1	1	985	1762	3	98	...
1595	2651	0	2	2	829	1967	5	96	...
3806	4474	1	3	3	936	2923	7	94	...
8727	1930	0	0	4	820	3849	9	92	...
9282	1293	1	1	5	187	1042	11	90	...
8124	885	0	2	6	198	4737	13	88	...
3228	9584	1	3	7	734	4082	15	86	...
3654	2307	0	0	8	184	1621	17	84	...
3761	2508	1	1	9	445	3231	19	82	...

Figure 3.3: A Fragment of the "tenthoustup" Relation.

range of their distributions varies. Figure 3.3 shows a portion of the "tenthoustup" relation to illustrate the purpose of the different integer attributes. The "unique1" and "unique2" attributes are both candidate keys, taking on values from 0 to 9999. The other integer attributes all take on random values chosen in a way indicated by their name — for instance, the "ten" attribute takes on values from 0 to 9 (i.e., 10 distinct values). The integer attributes in the "thoustup", "twothoustup", and "fivethoustup" relations are similar in nature. The string attributes were not used in our study, so we will not describe them here.

4. EXPERIMENTS AND RESULTS

4.1. Some Considerations

The first problem that arose in designing our tests was the issue of choosing an appropriate set of test queries. In their classic study of join methods for centralized database systems, Blasgen and Eswaren used a query that selected a subset of tuples from two relations, joined these together, and finally projected out a subset of the resulting fields as a general query for their analyses [Blas76]. Our initial inclination was to do the same for our study of distributed joins. However, since we are more interested in the effects of data distribution on the various join options, we decided to use the simple two-site join of Figure 4.1 for our test queries. The sizes of relations R_a and R_b, the size of the result relation R, and the value distributions of the join attributes are varied in our experiments to observe their respective effects on performance. Using this simple join query allowed us to limit our experimental search space, and we will see later that it has not really limited what we have learned from the study — adding the two pre-join selections and a post-join projection would only increase the fraction of the execution time due to local processing, strengthing the conclusions that we will present at the end of the paper.

```
range a is Ra at Sa
range b is Rb at Sb
retrieve into r(a.all,b.all) at Sa
where (a.A = b.B)
```

Figure 4.1: General Form of the Test Query.

Our choice of source relations followed the methodology presented in [Bitt83]. There are several considerations here. First, relation sizes should be large enough to be realistic. The basic relations used in our tests have 1,000 tuples and 10,000 tuples (the "thoustup" and "tenthoustup" relations of the Wisconsin database), occupying about 46 and 456 pages, respectively. Second, random attribute value distributions are desirable in order to provide an unbiased treatment of each of the join methods. This was particularly important in the sort-merge join case. Third, in order to assure that the results of one query test were not biased by previous ones, we had to ensure that no test query was likely to find useful pages sitting in the buffer from its predecessors. We used a technique described in [Bitt83], where two copies of source relations are maintained (at each site in our case), and alternate queries use alternate copies of the source relations.

Another important decision for our study was the choice of an appropriate set of performance metrics and a reasonable measurement approach. For our experiments, the elapsed time of a query was the main metric measured. This time is defined as the time interval beginning when site S_a initiates the query and ending when the result is completely stored at site S_a. The Crystal Nugget provides a timing procedure that is accurate to the nearest 10 milliseconds; this procedure was used for our elapsed time measurements. For each query, we also measured the number of disk accesses performed and the number of messages sent. Our disk access measurements were taken using a special version of WiSS that is instrumented to trace disk operations. For each disk access, the start and completion times of the access are recorded. An analysis of the trace records from our experiments indicates that the average disk access in our test environment takes about 25.5 milliseconds (for a 4K-byte page). To measure network traffic, we counted messages in our own communications interface routines. To measure the actual message send and receive times, we ran separate tests to send and receive a large number of single-packet "null" messages between two node machines using the same communications interface routines used for our test queries. Our results indicate that the average message transfer time is about 16.6 milliseconds (for a 2K-byte packet). Finally, while we would also like to have measured the CPU time used by our test queries, this was not easily done at the level at which our experiments run (i.e., stand-alone on Crystal nodes).

4.2. The Experiments and Results

We designed test queries to investigate the effects of a number of different factors on the performance of the alternative distributed join algorithms. The factors investigated include the sizes of the source relations and the join selectivity (i.e., the result relation size and the distributions of the join column values). We describe our experiments and the results that we obtained in this section. First, however, we describe the results of one of our distributed join executions in great detail in order to illustrate the costs and benefits of the various approaches and to provide the reader with useful background knowledge for later discussions.

4.2.1. Query Resource Demands: A Detailed Example

The example that we will examine in this section involves a query where both R_a and R_b are "thoustup" relations and the result relation has 100 tuples. Figure 4.2 lists the notation that is used in the discussion.

Figure 4.3 shows the elapsed time for the example query processed using the different join methods. The elapsed time for site S_a is the actual elapsed time, and the elapsed time for site S_b shows the portion of time during which S_b was involved in the query. The general trend is that the pipelined join methods — PJSM, PSSM, PJNL, and PSNL — executed the join more quickly than the sequential methods did. Of the pipelined methods, the nested loop join method outperformed the sort-merge method for this example. (That is, PJNL did better than PJSM, and PSNL did better than PSSM). This can be explained by taking a look at the resource demands of the various join methods.

Figure 4.4 shows the number of messages that were required to transfer data between the two sites S_a and S_b (measured at site S_a), illustrating the communications cost of each of the join methods. For each of the join methods (SJNL, SJSJ, PJSM and PJNL), all of relation R_b is shipped to site S_a, and site S_a sends no messages to site S_b. Thus, for these methods, the number of messages received by site S_a will be approximately $\frac{||R_b||}{M}$. For the sequential semijoin methods (SSNL and SSSM), site S_a sends its join column values to site S_b, and site S_b sends back its matching tuples. The number of messages sent and received by site S_a in this case will be $\frac{||R_a.A||}{M}$ and $\frac{J_S(Ra,Rb) \times ||R_b||}{M}$, respectively. In our example, since $||R_a.\text{A}||$ is only 2000 bytes (just exceeding the size of a single message packet with control information), and $Ja(Ra,Rb)$ is

Notation	Meaning
$\|\| R \|\|$	size of relation R in bytes
M	size of message packet in bytes
$R_a <A=B]R_b$	semijoin of R_a, R_b
$J_s(Ra,Rb)$	semijoin selectivity of $R=(Ra<A=B]R_b$, which is $\frac{\|R\|}{\|Rb\|}$
$J_j(Ra,Rb)$	join selectivity of $R=(Ra[A=B]Rb)$, which is $\frac{\|R\|}{\|Rb\| \times \|Ra\|}$

Figure 4.2: Notation.

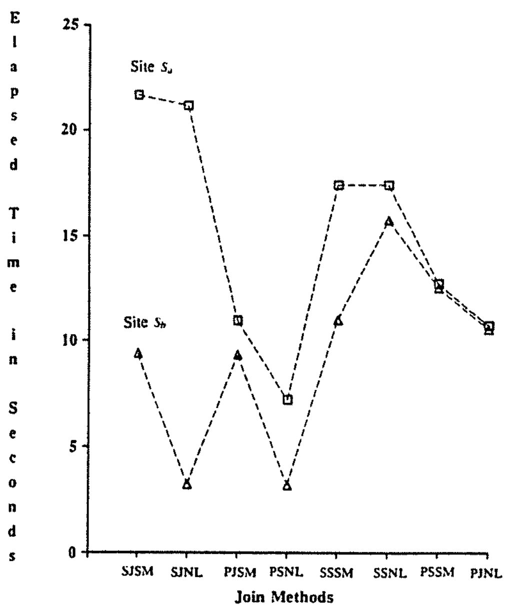


Figure 4.3: Elapsed Time.

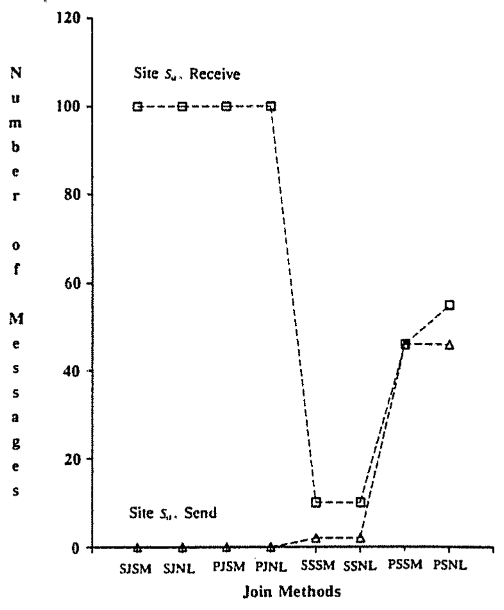


Figure 4.4: Number of Messages.

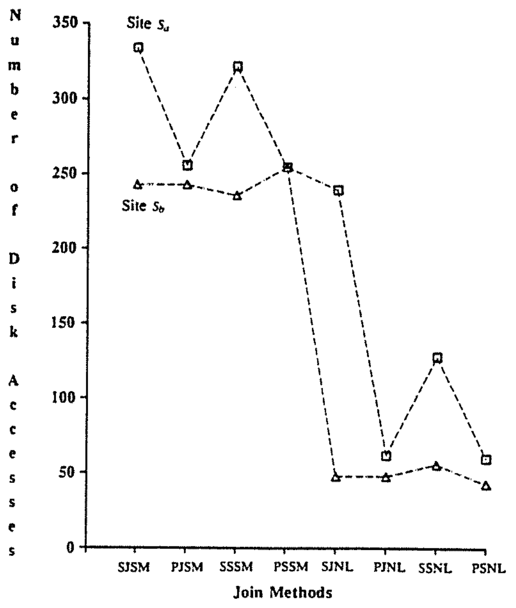


Figure 4.5: Number of Disk Accesses.

0.1, the communications cost of the sequential semijoin methods is much lower than that of the other join methods. However, it is important to realize that the fractional communications cost (i.e., communications cost as a portion of the total elapsed time) is not high in any of the join methods. For the non-semijoin methods, 100 messages were required in all, yielding a total message time of about 1.65 seconds.

The message cost analysis for the pipelined semijoin methods (PSNL and PSSM) is a bit more complicated for our implementation. First of all, the number of messages for PSNL and PSSM is affected both by the buffer space size P (in bytes) that is allocated at site S_a for holding R_a tuples, and also by the message packet size M, in the following way. To process the whole relation R_a, the number of "batches", where a batch involves filling the buffer space P with R_a tuples and processing these tuples, is given by $\frac{||R_a||}{P}$. The number of messages needed for sending the join column values from S_a to S_b is the same as for the sequential semijoin case (ignoring duplicate join column values for now). The minimum of these two values will therefore be the actual number of messages needed. As shown in Figure 4.4, the pipelined semijoin methods required more messages than the sequential semijoin methods. This is because our implementation allocates only one buffer page for scanning R_a, and only 22 tuples fit in a page: thus, we were limited by the buffer space factor above. (This may indicate that, since one message packet can hold many join column values, it may be better in practice to select the number of R_a buffer pages used according to the number of join column values that fit in one message packet.) Another complication involved in the pipelined semijoin message analysis is that the number of messages received by site S_a is influenced by the distribution of the join column values. That is, each message sent to S_b causes at least one return message that contains either matching tuples or tells S_a to read another page; the degree to which a given return message fills its message packet will depend on the number of matching R_b tuples found. For some messages sent by S_a, many values will be returned, but other messages may simply say "no matches" or may contain just a few tuples. This is in contrast to the sequential semijoin case, which will totally fill all but the last of the messages returned from S_b to S_a. For all of the reasons cited above, then, the number of messages for the PSSM and PSNL algorithms usually exceeds the number for SSSM and SSNL, as is the case in Figure 4.4. (Again, however, we remind the reader that the message cost is far from being the dominant cost factor here.)

The number of disk accesses for a join method depends on the number of different pages accessed during the operation (of course), but it also depends strongly on the available buffer space, on the page replacement policy used by the buffer manager, and on the physical allocation of pages in each relation. For example, for the nested loop join method, the number of pages touched will be:

$$num\_pages(outer) + num\_tuples(outer) \times (depth(inner\ index) + J_j)$$

where J_j is the join selectivity for the operation. In the worst case, the number of disk accesses will be equal to the number of pages accessed. This may be overly pessimistic, however, as some of the pages accessed may already be in the buffers. Figure 4.5 shows the measured number of disk accesses for the example query, and provides insight into the disk usages of the various join methods. At site S_b, with the exception of the sequential sort-merge semijoin method (SSSM), all of the sort-merge methods require the same number of disk accesses — this number is the sum of the accesses required for sorting relation R_b and those needed to scan R_b once to send its tuples to S_a. For the nested loop join methods, in both the sequential and pipelined join cases (SJNL and PJNL), just one scan of R_b is required at S_b (to send it to S_a). For the nested loop versions of the semijoin methods (SSNL and PSNL), the number of disk accesses at site S_b depends on the semijoin selectivity in a manner similar to that described in the equation above. Both of the sequential semijoin methods (SSNL and SSSM) require somewhat more disk accesses at S_b because they have to store the intermediate semijoin result $R_a.A[A=B]R_b$ and then retrieve it again to send it back to S_a. Similar trends are observed at site S_a. As observed in the measurements at S_b, the sort-merge methods require more disk accesses than the nested loops methods due to sorting. Among the nested loops methods, the sequential ones have higher disk costs than the pipelined methods due to the storage and retrieval of the received relation; this is especially true for the sequential join case (SJNL), which builds an index on the received relation at S_a.

4.2.2. The Effect of Relation Sizes

Two groups of queries, QG1 and QG2, were tested to investigate the behavior of the different join methods as the relation sizes were varied. QG1 consists of joins between two relations of the same size; the result relation in QG1 is the same size as a source relation (making the join selectivity simply the inverse of the source relation size). QG2 consists of joins between two relations of various differing sizes; the join selectivity is kept constant in query group QG2 (at a value of 10^{-4}). Since the two sites in these queries are asymmetrical, QG2 is further divided into two subgroups of queries, QG2.a and QG2.b. In QG2.a, the site having the larger relation was chosen as the join site; in QG2.b, the smaller relation resided at the join site.

(As before, the result site is taken to be the join site for these tests.) These query groups are listed in Figures 4.6.

Figure 4.7 shows the elapsed time measured for each of the QG1 queries. For the join of the two "tenktup" relations, all of the nested loop methods lost to the sort-merge methods even though the sort-merge methods must sort these large relations. This is because the amount of work saved through sorting significantly outweighs the work required to perform the sorts. This is illustrated by Figure 4.10, which shows the measured elapsed times and disk accesses for sorting the "tenthoustup", "thoustup" and "hundredtup" relations, and by the following analysis. Figure 4.10 shows that it takes 64.89 seconds to sort the "tenthoustup" relation, and that this involves 1911 disk accesses. This constitutes the per-join "overhead" portion of the sort-merge methods for this case. After sorting, the merge phase accesses each page of each relation just once. In contrast, for the nested loop join using a nonclustered index, the number of disk accesses is much larger; this is due to the number of data pages (randomly) accessed. Figure 4.7 shows this clearly. Of the four nested loop methods, three of them required more than 10,000 disk accesses, which is what was chosen as an upper limit for the number of disk accesses traced due to space considerations. The one exception was PSNL (pipelined semijoin-based nested loops), which keeps tuples in memory at site S_a, scanning R_a only once. However, this method involved a large number of disk accesses at site S_b, where R_b is searched using the index to find the matching tuples for the 10,000 join attribute values sent by R_a. The elapsed time was mainly determined by the processing rate at site S_b in this case, which explains its elapsed time as compared to the sort-merge methods.

GQ1			QG2					
			QG2.a			QG2.b		
$\|R_a\|$	$\|R_b\|$	$\|R\|$	$\|R_a\|$	$\|R_b\|$	$\|R\|$	$\|R_a\|$	$\|R_b\|$	$\|R\|$
10,000	10,000	10,000	10,000	1,000	1,000	1,000	10,000	1,000
1,000	1,000	1,000	10,000	100	100	100	10,000	100
500	500	500	10,000	10	10	10	10,000	10
100	100	100	10,000	1	1	1	10,000	1

Figure 4.6: Sizes of Relations Used in Query Group QG1 and QG2

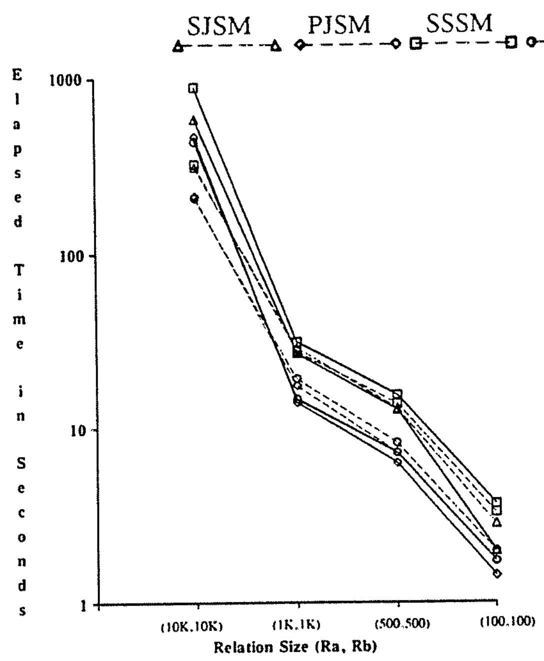


Figure 4.7: Elapsed Time (QG1).

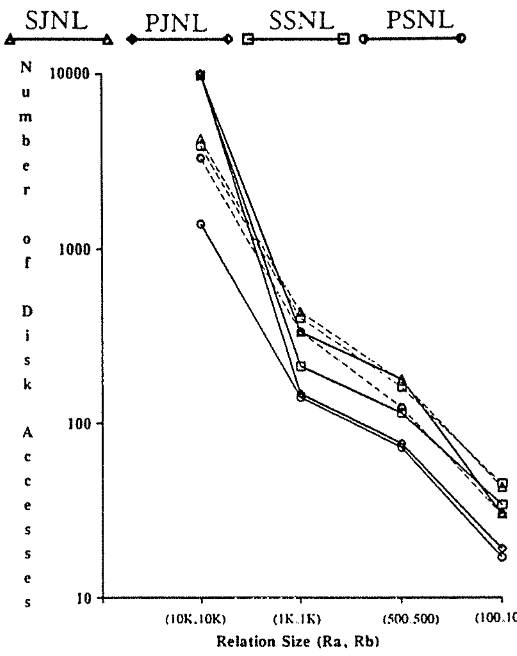
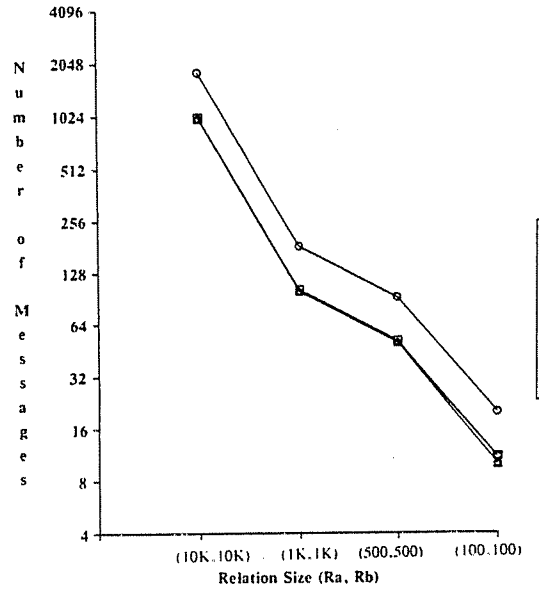


Figure 4.8: Number of Disk Accesses (QG1).



Relation	Elapsed Time	Number of Disk Accesses		
		Total	Reads	Writes
hundredtup	0.54	13	6	7
thoustup	6.46	196	97	99
tenthoustup	64.89	1911	955	956

Figure 4.9: Number of Messages (QG1). Figure 4.10: Costs for Sorting a Relation.

Figure 4.7 shows that, as the relation sizes are decreased, the cost of sorting the relations begins to outweigh the cost of performing an inner relation disk access per outer relation tuple. With smaller relation sizes, Figure 4.8 shows that the total numbers of disk accesses for the the pipelined nested loop methods (PJNL and PSNL) are lower than those for the sort-merge methods. Thus, the pipelined nested loop methods are the best performers except at the largest relation size tested for QG1.

Figure 4.9 shows the total number of messages involved in executing each of the eight join methods tested. Only three curves are evident; as in Figure 4.4 in the previous section, all of the join (i.e., non-semijoin) methods have equal message costs, the two sequential semijoin methods have equal costs. The highest cost here is for the pipelined semijoin methods, the next highest cost is for the sequential semijoin methods, and the lowest among the message costs are the non-semijoin methods. This is because, in this case, the join is a "one-to-one join" — each tuple of R_a joins with one and only one tuple of R_b. Thus, the use of semijoins here does not reduce the amount of data ultimately transferred to the join site; rather, it increases the overall message cost by the amount of data sent to the remote site from the join site initially. In all cases, given our packet transfer time of 16.6 milliseconds, the overall message time is never more than about 5-10% of the overall elapsed time. (Also, the effect of the message time will be even less significant for the pipelined algorithms, as there is processing going on while messages are in transit.)

Figures 4.11 and 4.12 give the measured elapsed times for the queries in query groups QG2.a and QG2.b. The results clearly illustrate the differences between the various join methods tested. The diversity of

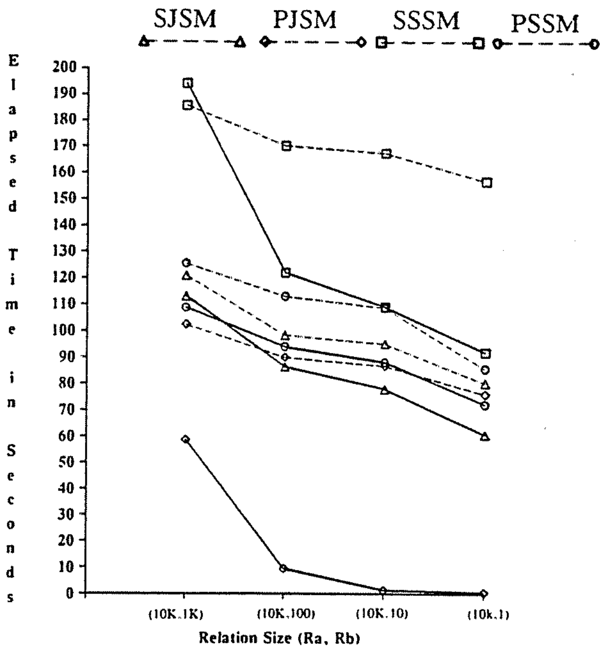


Figure 4.11: Elapsed Time (QG2.a).

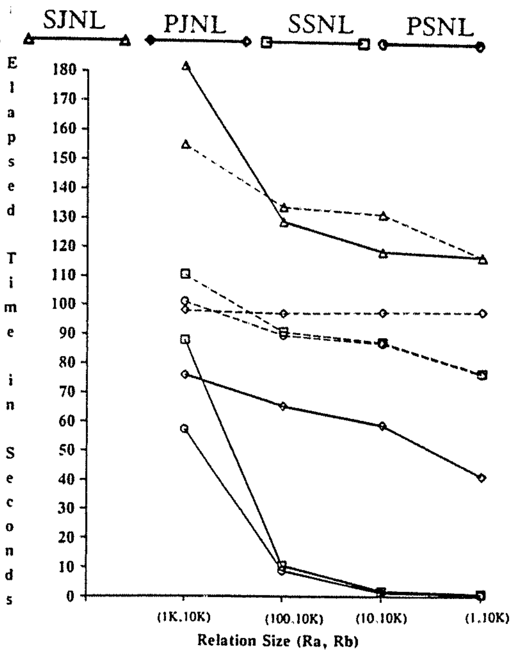


Figure 4.12: Elapsed Time (QG2.b).

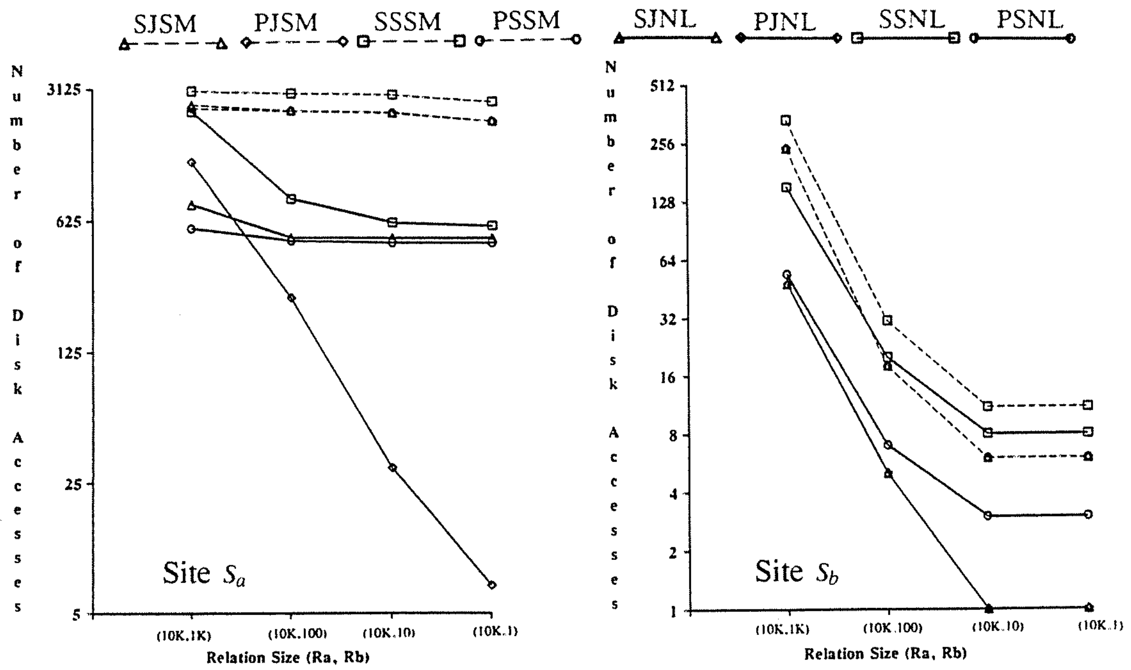


Figure 4.13: Number of Disk Accesses (QG2.a).

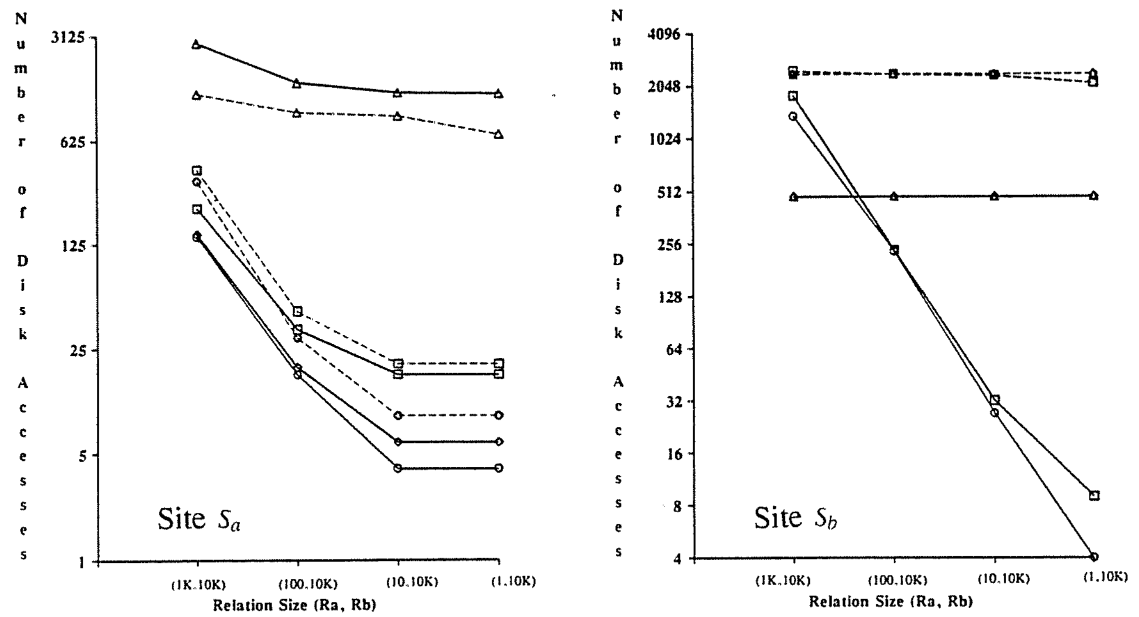


Figure 4.14: Number of Disk Accesses (QG2.b).

the results can be explained based on the discussion of the detailed example and analysis given in Section 4.2.1. It is evident from the figures that the nested loop join methods are more sensitive to relation size differences than the sort-merge methods, particularly at the larger relation sizes. This is because the sort-merge methods have a fixed component of their costs due to sorting the "tenthoustup" relation (see Figure

4.10 for this cost). An extreme case is illustrated in Figure 4.12 for the the pipelined join case (PJSM). With the smaller relation site as the join site, its cost remains nearly constant over the whole size range investigated. The main components of the cost of PJSM are sorting R_b at site S_b and scanning R_b to send it to site S_a. (These two factors alone account for about 90% of the elapsed time.) The sort merge methods can never execute faster than the time it takes to sort and scan the larger of its relations. The nested loop join methods are different, however. When the size of one of the source relations decreases, the number of disk accesses decreases dramatically for at least one of the nested loops methods in both query groups, as shown in Figures 4.13 and 4.14. This is due to the absence of sorting overhead and the effectiveness of the index for smaller outer relation sizes. The winner for query group QG2.a is the pipelined join version of nested loops (PJNL). The winner for query group QG2.b is the pipelined semijoin version of nested loops (PSNL); the sequential semijoin method ((SSNL) is the next best choice, with nearly identical performance for the smaller relation sizes. While the message counts are not given here, they represent an insignificant portion of the overall query processing cost (as in the previous cases examined).

One note here: It seems to us that the queries in QG2 are representative of a class of queries that is likely to arise in real database systems — that is, queries with a small number of tuples in one relation (the result of a selection) being joined with tuples from a much larger relation. An important observation from the tests covered by query groups QG1 and QG2 is that. when one relation is small, the pipelined nested loops join methods perform much better than their sequential counterparts or any of the sort-merge methods. When both relations are large, however, as when both were "tenthoustup" relations in our tests, the optimal methods will be the pipelined sort-merge methods.

4.2.3. The Effects of Join Selectivity

Join selectivity, which is the ratio of the size of the result relation to the product of the sizes of the source relations (see Figure 4.2), is an influential factor with respect to join algorithm performance. To see just how various join selectivities affect performance, we ran tests on the eight distributed join methods using the relation sizes shown in Figure 4.15. Each of the source relations used in this experiment had 1000 tuples, but these were not just "thoustup" relations from the Wisconsin database. Rather, the join selectivity was varied to obtain result sizes of 1000 tuples, 100 tuples, 10 tuples, and 1 tuple. Getting a result size of N tuples was accomplished by selecting R_a from the "tenthoustup" relation with "unique1" values in the range [4500..5499], selecting N of R_b's tuples from the "tenthoustup" relation randomly in the same range, and selecting the remainder of R_b's tuples randomly outside this range.

Figure 4.16 shows the measured elapsed times for the different join methods for the various join selectivities tested. Higher join selectivities (i.e. smaller result relations) mean that fewer tuples will match during the join, which leads to several cost savings. First, the result relation is smaller, so fewer disk accesses are needed to write out the result. Second, fewer data pages are accessed for the indexed nested loop join methods. Third, fewer tuples will be retrieved from the remote site for the semijoin methods, so the communications cost is reduced in their case. The pipelined nested loops join and semijoin methods were the winners in this experiment, with the semijoin method doing somewhat worse than the join-based method. Their sort-merge counterparts were the next best in terms of elapsed time here.

4.2.4. The Effects of Duplicate Attribute Values

Another factor which can potentially influence the performance of a join method is the degree of join column value duplication. In the merge phase of a sort-merge join, duplicate join attribute values will cause multiple scans of pages of the relation. Perhaps more significant is the effect of duplicates on sequential versus pipelined semijoin performance. In semijoin methods, site S_a sends the join column values of R_a to site S_b, and site S_b uses the values to fetch and return any matching tuples in relation R_b. In the sequential variant of the semijoin method, duplicate $R_a.A$ values are removed, which has two effects. First, less data is sent — duplicate join column values are avoided in messages from S_a to S_b, and (as a result) each matching

$\lvert R_a \rvert$	$\lvert R_b \rvert$	$\lvert R \rvert$
10,000	1,000	1,000
10,000	100	100
10,000	10	10
10,000	1	1

Figure 4.15 Query Group QG3.

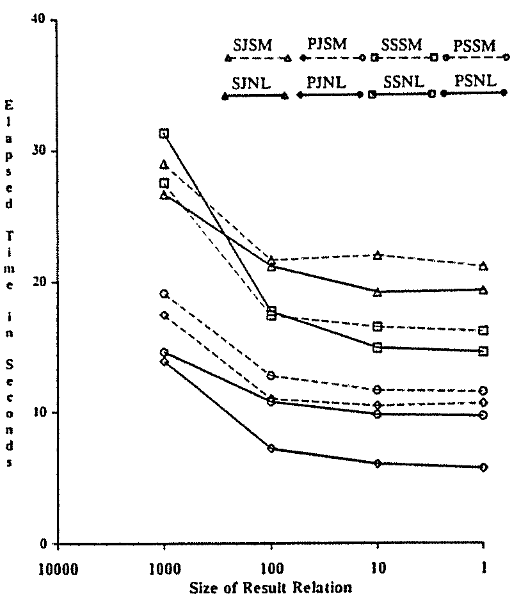


Figure 4.16: Effects of Join Selectivity.

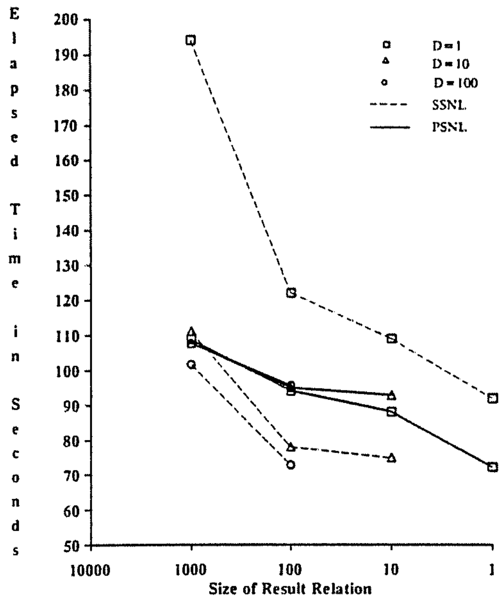


Figure 4.17: Effects of Duplicate Values.

R_b tuple is sent to S_a just once. Second, and related, is the avoidance of multiple disk accesses in R_b for a given $R_a.A$ value. These two savings reduce both the communications cost and the local processing cost. We tested a group of queries which join two relations on an attribute with duplicate values in this experiment.

We use the *duplication factor* D of attribute A of relation R_a to quantitatively describe the degree of duplication. This factor is defined as the ratio of the number of tuples in relation R to the number of distinct values of attribute A. For example, in the "tenthoustup" relation shown back in Figure 3.3, the duplication factor of the "hundred" attribute is 100, and that of the "thousand" attribute is 10. Figure 4.17 shows the effects of duplicates on the sequential and pipeline semijoin methods (SSNL and PSNL). In all tests, both source relations have 1000 tuples. Join columns in one of the source relations are chosen with D=1, 10, 100, respectively. It can be seen from the figure that duplicates have almost no effects on PSNL. For SSNL, however, an increase in the duplication factor moves it from being much much worse than PSNL to being much better. Obviously, the benefit due to the elimination of duplicates outweighs the usual disadvantages of the sequential methods when the duplication factor is high.

4.2.5. Summary of Test Results

There are a few observations that we would like to make here. First, if we set the issue of join site selection aside, Figures 4.11 and 4.12 show that the three nested loop methods PJNL, SSNL and PSNL provide the best performance when joining a large relation with a small one. For "medium" similar size relation joins, such as those shown in Figure 4.16, PJNL and PSNL also perform the best, and the sequential SSNL algorithm becomes worse than the two pipelined sort-merge methods (PJSM and PSSM). This leads to the conclusion that pipelined join and semijoin methods seem to be the most promising of the distributed join methods tested. To join very large relations of similar size, the pipelined sort-merge methods seem to be the best choices, as shown in Figure 4.7. These conclusions hold for the entire range of queries that we investigated.

Another general conclusion of this study is that the communications cost did not play a significant role in determining algorithm performance in our environment. The contribution of communications costs to the overall measured elapsed times was typically around 10-15%. As an illustrative example, to transfer a "tenthoustup" relation from one site to another requires about 1000 messages, but it only takes about 16-17 seconds for this transmission. In queries where such a transfer might be useful, however, the processing cost may be as high as several hundred seconds. The relatively poor performance of the sequential semijoin method is also evidence that communication cost savings alone does not help much. As a more concrete example, Figure 4.18 compares local join costs with distributed join costs in our environment. In all cases, $|R_a| = 10K$, and the local and distributed join methods used were those that gave the least elapsed time. It is clear from the example that the message cost is not the major determining factor for performance.

Relation Size	Elapsed Time (seconds)		
	local join	distributed join	degeneration (%)
|R| = |Rb| = 1K	55.8739	58.6128	4.90
|R| = |Rb| = 100	8.0877	9.3826	16.00
|R| = |Rb| = 10	0.9503	1.1164	17.48
|R| = |Rb| = 1	0.3500	0.3895	11.29

Figure 4.18: A Comparison of Local Join and Distributed Join (|Ra| = 10K).

The last point to be made is that choosing the right combination of a join processing site and a join method is important. Figures 4.11 and 4.12 indicate that, if the two relations to be joined have different sizes. the pipelined nested loop join method (PJNL) needs the site with larger relation to be the join site (i.e., it needs the outer relation to be the smaller of the two source relations). In contrast, the semijoin nested loop methods (PSNL and SSNL) perform much better when the smaller relation site is chosen as the join site (i.e., they also need the outer relation to be the smaller of the two). The intuition behind these results is fairly simple. in retrospect: PJNL and PSNL are basically the same algorithm if communications cost is zero. both being distributed executions of a simple nested loop join; they have the same local processing costs. The outer relation should be the smaller of the two in the centralized case as well [Blas76, Seli79]. Note also that in a low communications cost environment such as this, we can switch our join site choice with little or no significant impact on performance. For example, suppose that R_a is a small relation, R_b is a large relation. that S_a is to be the result site, and therefore that the pipelined semijoin (PSNL) method is the best choice. In this case, S_a ends up being the join site. If we would prefer to have S_b be the join site for some reason. such as load balancing considerations [Care85], we can accomplish this by doing a pipelined join (PJNL) at site S_b and shipping the results back to site S_a.

5. CONCLUSIONS

In this paper, we have studied the performance of a number of different join methods for a distributed database system. Eight different methods were implemented on top of the Wisconsin Storage System and run on an experimental distributed computer system, the Crystal multicomputer. at the University of Wisconsin. Join queries with various sizes, join selectivities, and attribute value distributions were tested. Our results have shown that. in a local network, communication cost is not the dominant factor. Shipping an entire relation from one site to another site is a reasonable way to process a distributed join query — as long as it is done correctly. Correctly in this case means that a pipelined join algorithm. where the outer relation is shipped to the join site (the inner site) in parallel with the local join processing itself, is employed. Although traditional (sequential) semijoin methods can reduce the communications cost. and they perform well in cases where the join column duplication factor is high (many matching inner relation tuples per outer relation tuple), pipelined semijoin methods were found to be preferable in most of the test cases examined. These results hold over a wide range of query characteristics. For the case where two very large relations are to be joined, pipeline sort-merge methods are recommended. We also found that the combination of the join method and the join site are important, that it is very important to ensure that the outer relation for the join is the smaller of the two source relations (as in centralized database systems).

Our results are related to several other pieces of work on distributed query processing techniques. First, the pipelined semijoin methods that we implemented are the ones used in System R^*, known there as the "fetch the inner tuple as needed" methods [Seli80]. They opted to use the pipelined version of semijoin over the more traditional sequential version because they believed that it would tend to win in most situations due to lower local processing costs. Our results indicate that this is indeed the case in a local network. Our results also concur with the claims of Page, which indicate that, in a distributed database system based on a local network, it is far more important that joins be done in the correct order and with the correct inner and outer relations than that they be done at the site which minimizes communications [Page83]. The key difference between Page's results and ours are that his conclusions were based on a cost analysis of the INGRES database system and the LOCUS distributed operating system, whereas ours were obtained from measuring the performance of a number of actual distributed join queries. Finally, earlier analytical studies have indicated that pipelined query evaluation techniques provide the best performance in centralized database systems [Yao79, Smit75]. Our results in favor of pipelined join methods can be viewed as showing experimentally that pipelining is still the method of choice in a locally distributed database system.

There are several directions that can be taken from here in terms of future research on query processing methods for locally distributed database systems. One direction that we are actively pursuing is the incorporation of load balancing techniques into a distributed database system [Care85]. We intend to use the results obtained here to guide the design of query processing algorithms that incorporate such techniques. We also plan to use our detailed measurements to drive a simulation model that we have developed for research in this area. Another direction that one might pursue is a study of the performance of processing strategies for n-way joins (where $n > 2$). While we feel that our results paint a fairly complete picture of the relative merits of the alternative join methods for our environment, we do not claim to have taken an exhaustive look at every possibility.

ACKNOWLEDGEMENTS

The authors wish to acknowledge helpful discussions with David DeWitt. We must also thank Hong-Tai Chou, who provided us with the instrumented version of WiSS used in these experiments. We also want to thank Bart Miller, who helped us learn about Unix/IPC for an earlier implementation of the distributed join methods. Finally, we wish to acknowledge the NSF-sponsored Crystal multicomputer project at the University of Wisconsin, which provided the experimental environment for this work and a helpful group of staff members to aid us in learning how to work in the Crystal environment.

REFERENCES

[Bern79a] Bernstein, P. A. and Goodman, N. *The theory of semijoins*. Tech. Rep. CCA-79-27, Computer Corp. of America.

[Bern79b] Bernstein, P. A. and Goodman, N. ''Full reducers for relational queries using multi-attribute semi-joins,'' in *Proc. 1979 NBS Symp. on Comp. Network.*, Dec. 1979.

[Bern81] Bernstein, P. A., and Chiu, D. W. ''Using semijoins to solve relational queries,'' *J. ACM. 28*, 1 Jan. 1981, 25-40.

[Bern81a] Bernstein, P. A., and Goodman, N. ''The power of natural semijoins,'' *SIAM J. Comput. 10*, 4, Nov 1981, 751-771.

[Bitt83] Bitton, D., DeWitt, D. J., and Turbyfill, C. ''Benchmarking database systems; a systematic approach''. In *The 9th International Conference on Very Large Database* , October, 1983.

[Blas76] Blasgen, M. W., and Eswaren, K. P. ''On the evaluation of queries in a relation data base system,'' IBM Research Rep. RJ1745, April, 1976.

[Brab84] Brabergsengen, K. ''Hashing methods and relational algebra operations''. In *the 10th International Conference on Very Large Data Base*, September 1984.

[Care85] Carey, M. J., Livny, M., and Lu, H. ''Dynamic task allocation in a distributed database system'', to appear in *The 5th International Conference on Distributed Computing Systems*, May 1985.

[Chou83] Chou, H.T., DeWitt, D.J., Katz, R. H., and Klug, A. C. *Design and implementation of the Wisconsin storage system*, Technical Report, Computer Sciences Department, University of Wisconsin-Madison, November 1983.

[DeWi84a] DeWitt, D. J., Katz, R. H., Shapiro, L. D., Stonebraker, M., and Wood, D. ''Implementation techniques for main memory database systems''. *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, June 1984.

[DeWi84] DeWitt, D. J., Finkel, R., and Solomon, M. *The Crystal multicomputer: design and implementation experience*, Technical Report, Computer Sciences Department, University of Wisconsin-Madison, September, 1984.

[DeWi85] DeWitt, D.J. and Gerber, R, *Partitioned hashing algorithms*, Technical Report #583, Computer Sciences Department, University of Wisconsin-Madison, February, 1985

[Epst80] Epstein, R., and Stonebraker, M "Analysis of distributed database processing Strategies" In *the 6th International Conference on Very Large Data Base*, October 1980, 82-101.

[Kers82] Kerschberg, L., Ting, P. D., and Yao, S. B. "Query optimization in star computer networks," *ACM Trans. Database Syst. 7*. 4, Dec. 1982, 678-711.

[Lohm84] Lohman, G. M., Mohan, C., Haas, L. M., Lindsay, B. G., Selinger, P. G., Wilms, P. F., and Daniels, D., "Query Processing in R^* ", IBM Research Report, RJ 4272, April 1984.

[Nieb76] Niebuhr, K. E. and Smith, S. E. "N-ary joins for processing Query by Example", *IBM Tech. Disclosure Bull. 19*, 6, 2377-2381.

[Page83] Page, T. W. Jr., "Distributed query processing in local network databases", Master Thesis, University of California, Los Angeles, 1983.

[Seli79] Selinger, P., et al. "Access path selection in a relational database management system", *Proceedings of the ACM-SIGMOD International Conference on Management of Data*. June 1979.

[Seli80] Selinger, P., and Adiba, M., "Access path selection in distributed database management systems", *Proceedings of the First International Conference Conference on Distributed Data Bases*. Aberdeen, 1980.

[Smit75] Smith, J. M. and Chang, P. Y. T., "Optimizing the performance of a relational algebra database interface", *Commun. ACM 18*, 10, October 1975, 568-579.

[Ullm82] Ullman, J. D. *Principles of Database Systems*, Computer Sciences Press, Rockville, Maryland. 1982.

[Wong76] Wong, E., and Youssefi, K, "Decomposition -- A strategy for query processing," *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 223-241

[Yao78] Yao, S. B., and DeJong, D. "Evaluation of database access paths". In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1978, 66-77.

[Yao79] Yao, S. B. "Optimization of query evaluation algorithms," *ACM Trans. Database Syst. 4*, 2, June 1979, 133-155