NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Query Processing For Distributed Main Memory Database Systems

by

Xiao Wang

B.Sc, Zhongshan University, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1987

ISBN 0-315-48840-9

Approval

Name: Xiao Wang

Degree: Master of Science

Title of Thesis: Query Processing For Distributed Main Memory Database Systems

Chairman: Dr. Binay Bhattacharya

Dr. Wo-Shun Luk
Senior Supervisor

Dr. Tiko Kameda

Dr. Jiawei Han
External Examiner

December 16, 1987

Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

QUERY PROCESSING FOR DISTRIBUTED MAIN MEMORY DATABASE SYSTEMS

Author: _____
(signature)

XIAO WANG
(name)

DECEMBER 8, 1987
(date)

Abstract

Issues on centralized main memory database systems are becoming popular in the recent literature. However, little has been done on the issues regarding distributed main memory database systems (DMMDBs). This thesis studies generally relational join strategies for DMMDBs. The empirical data obtained are used as the bases for performance evaluation.

We have constructed an analytical system model and a cost model for a local area network environment to evaluate distributed join operations, and designed and implemented a number of algorithms over an Ethernet network of diskless Sun-3 workstations running the V-system. Our study shows that hashing can be an effective method for a main memory environment. However, no single algorithm is the overall best one. A certain strategy only fits best into a specific situation.

The major contribution of this research is the development of the load sharing strategies for distributed join operations. Our experiments show that costly local processing tasks of the join operations such as sorting and merging can be spread over other otherwise idle machines to reduce the total processing cost at the expense of increased communication overhead. The communication/local processing tradeoffs should be fully exploited to increase system performance. It also indicates that in a similar distributed environment, parallel processing strategies may be employed for performance enhancement.

To my wife, Wei Bai.

Acknowledgement

I am most grateful to my supervisor Dr. Wo-Shun Luk for his thougtful guidance, invaluable support and encouragement throughout this research. Many thanks to Dr Tiko Kameda for his teaching and suggestions which improve the work considerably. I am also grateful to Dr. Jiawei Han for his careful reading of the thesis and his valuable comments. I would like to thank Dr. Binay Bhattacharya for serving as the Chairman of my thesis examination committee.

My discussions with fellow students Franky Ling and Garnik Haftevani were very helpful in setting up the experimental environment. Many thanks to Dr. Joe Peters, Dr. Pavol Hell, Dr. Veronica Dahl, Dr. Lou Hafer and Dr. Binay Bhattacharya for permitting me to use their Sun-3 workstations which are necessary for the experiments.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

1.1. Distributed Systems and LANs

The terms *parallel* and *distributed computing systems* appear frequently in the literature. The major difference between the two lies in the degree of coupling and the level of interactions among the component processors. By a parallel computing system, we mean a system in which all processors are tightly coupled and sharing is through shared main memory. A distributed system, on the other hand, is one in which all processors are loosely coupled, each with its own local memory, and sharing among processors is through a certain communication medium.

Distributed systems, in particular, distributed database systems, are becoming more and more important in recent years. A distributed database is a collection of data which belong logically to the same system but are geographically spread over the sites of a computer network [CEP 84]. There are two important aspects: distribution and logical correlation. Distribution means that the data are physically distributed. Queries usually involve data from different geographical sites and data transmission through a communication medium is necessary. It distinguishes a distributed system from a centralized one. Logical correlation means that the data are logically interrelated, which distinguishes a distributed system from a collection of several local systems. The user does not even have to know where a set of data is located.

Distributed computing systems can be categorized by the degree of decentralization. At one extreme is the remote computer network, also called long haul network, where interconnected systems may be widely separated, with distance between two systems greater than 10 km, and data transfer rate usually less than .1 Mbps. At the other extreme is the tightly coupled multiprocessor system, with distance between two processors (usually much) less than .1 km and data transfer rate usually greater than 10 Mbps [MEB 76]. Near the middle of these two extremes is the so-called

local area network system (LAN). In such a system, we have the advantages of a network system for resource sharing, with a much more efficient communication medium. We also have the advantages of a multi-processor system for parallel computations, without the cost of building tightly coupled systems. Local networks are becoming more and more popular because of the decrease in cost and the increase in the computer hardware and local network performance.

1.2. Main Memory Systems

With the current trend of semiconductor technology, massive memory computing systems, having a main memory size of a gigabyte or more, will be economically feasible in the near future [LEH 86]. In such a system, a huge amount of primary memory is available and a large portion of, or even the whole application working storage will be able to fit into it. This will fundamentally change the conventional programming techniques. [BIT 86] and [PARK 86] have studied its impacts on the current methodologies. Database applications, which tend to be I/O bound, will immediately benefit from the large amount of main memory.

The implications are twofold. The huge primary storage can be used to hold a large portion of (or even the entire) database such that the disk I/O bottleneck could be eliminated. The memory space can also be traded for computation time to achieve desired speedup. In a relational database system, storing in memory several index structures of different types for a relation can facilitate multi-dimensional retrievals.

The availability of inexpensive, large primary memory is bringing new design issues to database management systems. It leads us to reexamine the components of traditional database managers. Any component whose operations are aimed at disk-based data should first be modified to reflect the new environment. There are basically two situations to the use of massive storage.

One is that the memory can be used to provide a large buffer pool where frequently accessed data can be kept in order to reduce the amount of I/O traffic. Performance is enhanced, but minimizing the amount of disk accessing is still the primary goal for the algorithm design. [DEW 84], [SHAP 86] and [ELH 84] are based on this assumption. In this situation, accuracy in estimation of space requirement is critical. When we perform query optimization, we have to estimate exactly the size of a temporary relation. Large errors may result in substantial performance degradation.

The other one is when the main memory is large enough to hold the entire database system. New schemes for physical data organization, query processing, concurrency control and recovery are needed. The emphasis will be the efficient use of CPU and memory resources rather than the economy of disk accessing and disk storage. Disk accesses are necessary only for activities for crash recovery purposes. [LEH 85], [LEH 86] and [SAL 86] are based on this assumption in their research. In our opinion, the actual memory size is not important, but there must be sufficient storage so that disk I/O is not necessary throughout the processing.

1.3. Thesis Contributions

This thesis studies general query processing strategies for distributed main memory database systems (DMMDBs). The database is spread over geographically separated sites of a local network, and each site has sufficiently large amount of primary memory to store the entire local portion of the database system.

Currently, much work has been done on issues regarding centralized main memory database systems. However, DMMDB issues, which are becoming more important as diskless workstations are more and more popular, are neglected.

We have developed two sets of algorithms for the join operations between two relations. One contains those derived from algorithms for conventional (disk-based) distributed database systems. The processing is confined to the two sites with the relations. Three algorithms, called simple sort-merge, nested loop hash join and tree merge, are designed and analyzed.

The load distribution concept is proposed. It is closer to a parallel processing procedure in that the otherwise 'indivisible' tasks are decomposed and shared among the processors over a local network. The motivation is based on the fact that the possibility that some machines are sitting idle while others are busy is remarkably high, and the fact that communication overhead in a local area network is fairly low with regard even to local processing in a main memory environment. The underlying idea is that parallel processing strategies can be employed in a distributed environment for performance enhancement.

Based on the load distribution strategy, the other set of algorithms contains the extended versions of the first set. They are the load sharing sort-merge, load sharing hash join and load sharing tree merge algorithms.

All the algorithms are implemented in our experimental test environment and their performance is analyzed. The performance trend is predicted as well. A system model and a cost model are developed as the bases of our theoretical and empirical studies.

Our results confirm that increasing communication parallelism can be an effective way of increasing system performance, which is specially true if load distribution strategy is employed.

1.4. Thesis Organization

The thesis is organized as follows. Chapter 2 gives a survey of the query processing strategies for centralized main memory database systems recently developed in the literature. In Chapter 3, three join processing algorithms based on the conventional strategies for disk-based systems are developed. Chapter 4 analyzes the empirical data for the previous algorithms and provides evidence in favor of load distribution strategies. Load sharing algorithms which employ the load distribution strategies are subsequently designed and analyzed theoretically in Chapter 5. Chapter 6 analyzes the experimental data for the load sharing algorithms and studies the impacts of communication parallelism, parallel processing and load distribution strategies. A conclusion of this study is given in Chapter 7.

Chapter 2

Query Processing for Centralized MMDBs

The basic assumption for a centralized MMDB is that the part of the database that is going to be referenced is completely memory resident. No disk I/O is involved during a transaction execution except for the activities for recovery purpose.

The essential considerations are therefore: how to make efficient use of primary memory and how to minimize the number of CPU cycles. This is especially true when we study query processing techniques. Two aspects need to be considered: data structures and processing algorithms. In a conventional database system, data structures are chosen and designed to minimize necessary disk traffic. Variations of B-trees are of this kind. In a main memory environment, however, other data structures may more efficiently use the storage space and have better retrieval performance. The processing algorithms should also be reconsidered and new methods should be developed to fit into the new environment.

This chapter will summarize what has been done on this issue.

2.1. Index Structures for MMDBs

There are basically two kinds of index structures: hash-based and order-preserving indices. Variants of hashing methods belong to the first category. Chained bucket hashing and linear hashing are examples of this kind. The hash function and the key value itself determine the location a data item is to be stored. The structures which preserve a certain logical order of the data belong to the the second category. Arrays and B-trees are examples of this type. In these data structures, a total order on data items is retained in the sense that, given a data item, there is an easy way to know where the "next" item is. "Adjacent" items are likely to be clustered.

Although it is not very efficient in a conventional environment due to its heavy disk I/O requirement, hashing is expected to have good performance in main memory systems since the working storage of an application is now entirely in main memory and references to it will not cause any disk I/O. [KNU 73] shows various techniques of calculating hash values. Searching a hash table is fast which is a good characteristic for equal-join operations. But it does poorly with range queries which ask for a set of items within certain range.

As representatives of order-preserving data structures, AVL-trees and B-trees are two well-known data structures, of which a good introduction can be found in [COM 79] and [AHU 74], respectively.

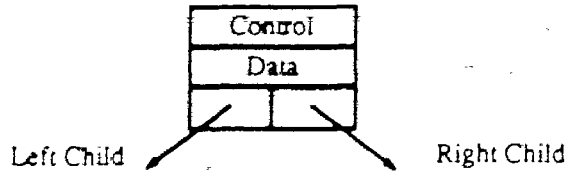

Figure 2-1: The Structure of an AVL-tree Node

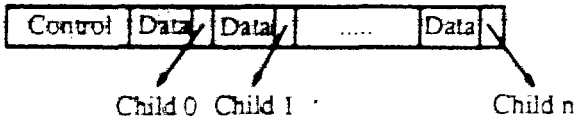

Figure 2-2: The Structure of a B-tree Node

The structures of an AVL-tree node and a B-tree node are shown in Figure 2-1 and 2-2, respectively. AVL-trees are binary trees in the sense that each node has at most two children. Search is fast because it requires less number of data comparisons. However, it has poor storage usage since each node contains only one data item and the data to pointer ratio is 1 to 2. Even in the main memory environment, we cannot assume that the amount of memory available is unlimited. Efficient use of primary storage is still one of the major issues. B-trees are well-known external data structures. Since each node contains multiple data items, the number of levels searched to retrieve an item is significantly reduced. It also has better storage usage. However, it has to determine where to go from among multiple data links of each node along a search path. [LEH 85] has studied the experimental aspects of B-trees and AVL-trees.

A new tree structure, called a T-tree which has advantages of both AVL-trees and B-trees, is introduced in [LEH 85]. The basic structure of a T-tree node is shown in Figure 2-3. T-trees are binary trees, with each node containing n data fields, one control field and two pointer fields to its left and right children. Since multiple data items are allowed in each node, and each node can have a maximum of two children, T-trees retain the intrinsic binary search nature of AVL-trees and the storage efficiency of B-trees. Binary search is used for intra-node searching.

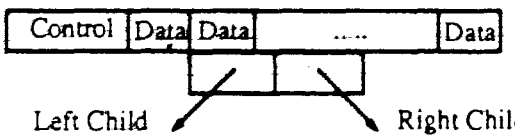

Figure 2-3: The Structure of a T-tree Node

Insertion and deletion operations for a T-tree are very similar to those for an AVL-tree. Intra-node searching or data movement is involved, and rebalancing has to be done when necessary. The rebalancing technique is the same as for AVL-trees except that data items may have to be transferred from one node to another. However, most updates are expected to involve only data movements within one node, rebalancing is much less often needed than in an AVL-tree. Algorithmic details can be found in [LEH 85].

[LEH 85] has done experiments on T-tree, B-tree, array, extendible hashing and linear hash structures. The results indicated that T-trees provide a good overall performance for mixes of searches, inserts and deletes.

2.2. Query Processing Strategies

When queries are presented to the database systems, it is the task of a query optimizer to decompose a query into primitive relational operations such as projections, selections and joins. Each operation involves one or two relations. We will concentrate only on join operations on two relations.

Conceptually, there are two general methods: nested looping and merging. For a pure nested loop method, one of the relations is operated as the outer and the other as inner relation. For every

tuple of the outer relation, the entire inner relation has to be scanned to search for matching tuples. This is a quadratic strategy which is too expensive to be employed. Various index structures can be used to speed up the operation. In the previous section, we have discussed retrieval structures good for main memory environment. Hash table and T-tree are two of them. Instead of scanning the entire inner relation, the index for the inner relation can be probed in search for matching tuples.

Merging is the other general strategy used for joining operations. Merging requires the ability to access tuples in a relation in a certain logical order, which can be achieved by either an order preserving index or a sorted array index. Both relations are scanned following the order provided. The strategy is linear in time complexity, because each relation has only to be traversed once. Tree structures are the commonly used order-preserving indices.

In the literature, various algorithms have been proposed, which are essentially derived from conventional join processing algorithms for disk-based systems. Index structures suitable for main memory operations are used. Processing strategies are also tuned to fit into the main memory environment. Five algorithms are presented and tested in [LEH 86]: simple nested loop, hash index nested loop, tree index nested loop, sort-merge and tree merge. The simple nested loop algorithm is shown too costly to be practical, and it is implemented only for comparison purpose. Hash index nested loop and tree index nested loop are two variants of the simple nested loop join algorithm with a hash index and a tree index, respectively, on the join column of one of the relations. The sort merge and the tree merge are two variants of the sort-merge join algorithm of [BLA 77]. For the sort merge algorithm, an array index for each relation is set up and subsequently sorted by quicksort. Then join is performed with this index. For the tree merge, a T-tree index is created on the join column for both relations and the merge join is performed subsequently. However, the tree merge is a practical method only if the indices already exist, since the tree index setup cost is very high.

The experimental results show that if a proper pair of tree indices on the join column(s) for both relations exists, the tree merge join method performs the best [LEH 86]. This finding is obviously predicted on the assumption that the relations are not already sorted according to the join column(s). However, in situations where one of the two relations is missing an index and the result

relation needs to be in sorted order, the sort merge is the choice. Otherwise, the hash join method would be the best one. One benefit from merging algorithms is that they produce ordered results, whereas the hashing methods give randomly collected ones.

Chapter 3
Distributed MMDBs

3.1. Why Distributed MMDBs?

A distributed MMDB can be considered as a database system distributed over a loosely coupled multi-machine system, with each individual machine having enough primary memory to store its complete local database. The data transfer rate of such a system should be reasonably fast.

But why distributed MMDBs on a local area network? In a long haul distributed system where communication expenses constitute a major portion of the total processing cost, it does not make too much sense to have a main memory system which does not aim at reducing data exchange cost. On the other hand, in a centralized MMDB, there is obviously a limit to which additional memory ceases to improve its performance, i.e. when the memory space is large enough that virtually no disk I/O is incurred due to local processing. Then, local processing, instead of disk I/O, is a major factor of the total expense.

By having the database distributed over a local network, the performance could be enhanced with the introduction of parallel computations. Some of the advantages include (1) reliability and availability are increased because of data duplication; (2) database expansions are easier through the addition of new sites to the network. The system response time may also be reduced.

One of the problems with main memory systems (either centralized or distributed) is that the volatility of primary storage makes crash recovery more difficult and costly. [DEW 84] and [SAL 86] have studied new recovery techniques for centralized MMDBs. But little has been done for distributed ones. We also need new concurrency control schemes since transactions tend to be short that locking small items may not be tolerable. However, this thesis does not address these problems. One of the main issues of distributed systems (main memory or disk-based systems) is

the overall performance which is partly determined by the communication overhead that accounts for a significant portion of entire computation expenses. This research deals with the issue of performance of distributed MMDBs.

3.2. The Models

We consider two kinds of models. System models are abstractions of major features of actual systems, which should be specific enough to reflect and summarize basic features of a class of actual systems and which should be general enough to ignore unimportant details of actual systems and make theoretical analysis possible. Cost models are basic formulas of how processing expenses should be calculated. They should include all dominant factors under a specific system model and eliminate negligible components.

In our system model for the distributed computing study, two homogeneous machines (or sites) are interconnected by a local area network. They run the same system software and communicate with each other through message passing. They are called the result site, denoted by S_{rst}, and the remote site, denoted by S_{rmt}, respectively.

There is one relation at each site. Join operation is to be performed between the two relations. One of the relations, usually at the remote site, has to be transferred to the other site in a way depending on the specific technique used. Each processor is assumed to have enough local primary memory to hold the entire two relations. For ease of discussion, we assume that the two relations are of the same size M integers. Since the two machines are autonomous, they may execute the same operation on their own data set independently. We call this kind of executions parallel operations. The cost of some simultaneous parallel operations is determined by the expenses of the most costly individual operation.

Our cost model consists of three components: preprocessing, communication, and local join. Preprocessing refers to the local handling prior to the local join of the relations. Local joining is the actual join of the two previously handled relations. Preprocessing usually includes operations such as index setup, array index sorting, relation partition, and so on, which may not be necessary but which will speed up the subsequent local join operation. A more costly preprocessing phase

could result subsequently in a cheaper local joining phase; and both of them belong to the local processing cost. But we make them separate components because we want to study the trade-off between the two cost components for different techniques. It is also because it is difficult to measure them in a unique unit. They are measured in terms of the number of data comparisons or data movements during the operation, which will be defined later when we pursue the cost analysis. The communication cost is usually proportional to the size of a relation to be transferred. It is measured in terms of the number of parallel packet transmissions or the size of data segment transmitted, depending on the technique used.

3.3. Conventional Algorithms

Queries coded in a high-level, procedural or non-procedural language are submitted to the database system which is distributed over a communications network. The database management system then translates the queries into a relational calculus form. It is the task of the query optimizer to decompose the relational calculus form into primitive operations such as selections, projections and joins which may involve operations on relations at different sites. For ease of discussion and analysis, we assume, without lose of generality, that a join operation involves only two relations residing at two different sites of the network.

Distributed join algorithms are distinguished by whether they employ the traditional join or semijoin operator, how the pair of sites involved cooperate during the join processing and which of the local processing methods is used.

To perform a join, one of the relations has to be shipped to the other site (called the result site). The semijoin operation can be used to limit the amount of data to be transferred. Only the join column values of one relation and the matching tuples of the second need to be transferred between the two sites. It played a crucial role in the query processing algorithm of SDD-1 [BERN 81] where the intersite data transfer is expensive. This is specially true for a long haul network system. However, in a local area network over which data transfer cost is much lower, it relies on the specific join processing strategy used to decide whether the semijoin operation is beneficial, as it requires multiple scans of a relation, resulting in increased local processing cost.

Many centralized join evaluation algorithms can be employed for local processing in each machine. Sort-merge and nested loop are the most common ones. We may use index structures such as B-trees or hash tables to speed up the operation.

The two sites involved can work in either a sequential or a pipelined fashion. For the sequential (sometimes called batching) approach, the receiving site will not begin working until all the required data have arrived. In the pipelined approach, processing will begin as soon as the first tuple has arrived. While the sequential strategy is easy to implement, the pipelined method allows the sites to work in parallel, and the receiving site does not need to store incoming data in a temporary relation. However, batching of tuples for transmission may be more economical than a series of transmissions of single tuples. The following table shows the performance of V-system in file transfer between two diskless Sun-3's.

0.5K bytes	1K bytes	2K bytes	4K bytes	8K bytes	16K bytes
4ms	6ms	9ms	11ms	16ms	25ms

Table 3-1: V-System IPC Timing Data

While much work has been done on distributed query processing in a disk-based environment [CAR 85], little has been done to investigate strategies for distributed main memory database systems. Here, we are going to present several algorithms for distributed MMDBs which are derived from the conventional algorithms.

As in disk-based systems, we have several choices of algorithms for evaluation: (1) join vs. semijoin; (2) sequential vs. pipeline; and (3) sort-merge vs. nested loop. We already have results on the performance difference between a sequential and a pipelined algorithm for conventional systems [CAR 85], which will, we expect, not change too much for a main memory environment. Therefore, we are only interested in sequential algorithms. Notice, however, that although [CAR 85] has shown the pipelined methods are better in their test environment, [LANT 85] gave the opposite results because pipelined methods may introduce high communication overhead. There is an added difficulty in pipelined processing over the V system since the interprocess communication protocal is a blocking one. It is a plausible research topic to investigate pipelined techniques vs. batching ones. Meanwhile, the batching approach is adapted in this thesis.

Furthermore, we always transfer key fields since in the main memory environment, the key extracting operation is cheap and we are not intending to compare join with semijoin operation. The semijoin vs. join is not an issue here. The result relations are always sent to the result site at the last stage if necessary.

Those algorithms we are going to investigate in this section are categorized by the local processing techniques used. Specifically, they are distinguished by whether they employ the nested loop or merging method for the join operation and whether they use a hash-based or order-preserving index structure to speed up the local calculation.

Based on the above discussion, we have the following three algorithms.

Algorithm 1. Simple Sort-merge Join

This algorithm is a modified version of the general sort-merge algorithm. An array index on the join column is created for each relation at two different sites. The indices are sorted in parallel and then the relation on the remote site (and its index) is transferred to the result site where subsequent merging is performed.

Algorithm 2. Nested Loop Join with Hashing Indices

A hashing index on the join column is created for the relation at the result site. The relation at the remote site is then transmitted to the result site and joining is performed using nested loop strategy.

Algorithm 3. Tree Merge Join

Both sites set up, in parallel, a T-tree index for their relations if they do not exist. Then, the trees are traversed, also in parallel, to get sorted array indices for the relations. The relation at the remote site (and its array index) is transferred to the result site where subsequent merging is performed.

Another possible processing strategy would be a nested loop join method with T-tree indices. Two relations are to be joined in a nested loop fashion. For each key in the outer relation, the T-tree index for the inner relation is probed for matching keys. The method performs better than the simple sort-merge algorithm in a centralized environment. However, it is not as good as it should be in a distributed situation. The simple sort-merge method now performs better in the

distributed environment because the sorting phase can be done in parallel. On the other hand, probing the tree index for each key value is not cheap. We have implemented the strategy and the result shows that the tree merge method (Algorithm 3) is five times faster than the nested loop join method for relations of size greater than 8K integers, if the tree setup is not included.

3.4. Cost Analysis

We are going to pursue cost analysis for the conventional algorithms given in the previous section based on the cost model. For each algorithm, the cost equation consists of three parts; preprocessing for local joining; data transfer; and local joining. Due to the diverse nature of different preprocessing techniques, different measurement units have to be used. Therefore, we can not compare different cost components on the basis of the cost analysis. The data transfer is measured by the total time used for data transmission. It is proportional to the amount of data transmitted. Since the physical communication medium is not sharable, simultaneous transmissions may cause collisions. However, as noted in [PAGP 85], actual data transfer time on the communication medium is only a small fraction of the total message passing time, 90% of which is spent on preprocessing and postprocessing by the transmitting and receiving processors. As a result, if two messages are initiated at approximately the same time, the two transmission processes may just overlap so that they appear to be going on in parallel. This phenomenon is called communication parallelism [LUK 87]. We refer to a data-exchange process as a parallel data transmission if communication parallelism can be fully exploited. In other words, the cost (in the time scale) of a parallel data transmission is proportional to the amount of data transmitted by the initiating machine, without being affected by other simultaneous messages. We measure the data transfer cost in terms of parallel data transmissions. For example, assume we have two transmission processes T_1 and T_2, and their communication costs, when they are pursued individually, are C_1 and C_2, respectively, with $C_1 > C_2$. Then, if T_1 and T_2 are initiated one after the other, the total communication cost will be the sum of C_1 and C_2. If T_1 and T_2 are initiated at the same time, the cost for the parallel transmissions will be measured by the more expensive one, which is C_1 in this case.

For Algorithm 1 (Simple Sort-merge Join), the arrays of keys for both relations are sorted in

parallel at different sites, using any of the popular internal sorting algorithms. The relation at the remote site is then transferred over to the result site. A merge join of the two is subsequently performed there. The number of comparisons is the usual unit to measure the sorting cost. Hence, the preprocessing cost of Algorithm 1 is $O(M \times \log(M))$. The data transfer cost is M, since the relation is of size M integers. The number of data comparisons is used to determine the merge join expense. Since both relations are of size M, it is $2cM$, where c is the average number of matching tuples. In our experiments, c is approximately 1.

For Algorithm 2 (Nested Loop Join with Hashing Indices), a hash table index is created for the relation at the result site. The relation at the remote site is then shipped over to the result site and the local joining is performed at the result site. During the hash table setup procedure, keys have to be added one by one. For each insertion, a hash value have to be calculated and the keys have to be inserted into the appropriate table entry. It takes a constant time to insert the key. We use the number of keys to be inserted to measure the index setup expense. Hence, the preprocessing cost of Algorithm 2 is M. The communication expense is obviously M. Moreover, we use the number of key comparisons to measure the local nested loop hash join operations. The local join cost will then be $c'M$, where c' is the average length of the chain list associated with each bucket in the hash table. In our experiments, c' is approximately 4.

For Algorithm 3 (Tree Merge Join), a tree index, if it does not exist, is set up for each relation in the machines. However, tree index is theh most likely one to exist for a relation. Then, the tree indices are traversed to produce sorted array indices. Later phases are the same as for the simple sort-merge algorithm. For the tree setup procedure, the relations have to be scanned and the keys are inserted into the index tree one after the other. For the i^{th} tuple to be inserted, a search for the inserting node has to be done and intra-node data movement may be necessary. We use the total search cost to measure the tree setup expense. The search cost can be measured by the number of tree nodes searched. For the i^{th} key to be added, the search costs $\log(\frac{i}{S})$, where S is the size of a T-tree node. Therefore, the total preprocessing expense is $\sum_{i=1}^{M} \log(\frac{i}{S})$, which is equal to $\log(\frac{M!}{S^M})$. From [KNU 68], we have the following formula:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Hence, the above asymptotic estimation will be estimated to be O ($M \log (\frac{M}{eS})$).

The traversing of the indices is linear since each node is only reached three times and scanned only once. We use the number of keys to measure the tree traversal operation. Hence, the cost of traversal is M. The data transfer cost here is also M, the same as previous algorithms. The last merging phase is the same as the one for the simple sort-merge. Table 3-2 summarizes the analytical results obtained in this section. Table 3-3 gives the list of units used.

Simple Sort-merge Algorithm:

Local Sort	Communication	Local Merge
O (Mlog (M))	M	2M

Nested Loop Join with Hashing Index:

Hash Index Setup	Communication	Local Join
M	M	4M

Tree Merge Join:

Tree Index Setup	Tree Traversal	Communication	Local Merge
O (Mlog ($\frac{M}{eS}$))	M	M	2M

Table 3-2: Summary of Cost Analysis

All the algorithms have the same communication overhead, which is necessary for the distributed join. Therefore, the performance difference of these algorithms lies on the various preprocessing and local join techniques used. However, since different units are used for different cost components, we can only compare relative performance between cost components of the same kind. We will evaluate the algorithms through experiments and study the best tradeoff between preprocessing strategies and local join methods.

Cost Components	Measurement Units
sorting	number of comparisons
merge join	number of comparisons
hash table setup	size of the relation
local hash join	number of comparisons
tree index setup	number of tree nodes searched
tree traversal	size of the relation
communication	size of the relation

Table 3-3: Measurement Units

Chapter 4
Experimental Validation I

4.1. Test Environment

The distributed experimental environment consists of five Sun-3 workstations connected by a 10Mbit Ethernet local network. Each workstation is equipped with a powerful MC68020 microprocessor and an MC68881 floating-point coprocessor. It is a homogeneous environment, with all workstations being diskless and possessing four mega-bytes of main memory (RAM). They also run the same operating system. The underlying Ethernet provides a fast communication medium.

Running on these workstations is the operating system, the V-system, developed at Stanford University. V-system is categorized as a distributed operating system by [TAN 85] in the sense that it provides the users with high transparency of the underlying system activities.

The greatest advantage of using the V-system is its efficient and cheap interprocess communication facilities. Among many features, the system only transmits short and fixed-size messages which is easy to implement and has less processing overhead. To transfer large amount of data, the sender has only to specify a segment of memory in its address space where the data are stored. The access is passed along with a fixed-size message to the receiver and the receiver can then copy the whole segment from the sender's address space over to its own address space directly. No intermediate buffering is necessary.

The V-system employs a blocking send operation, which means the process initiating the operation is blocked until a reply or acknowledgement is received. The design is chosen according to the nature of most applications: a process typically suspends execution to wait for a reply immediately after sending a message [CHER 84]. Providing one kernel primitive for both sending

the message and receiving the reply results in less overhead with a message transaction and makes the interface procedure easy to use.

Our experiments were all done during weekend nights when the network traffic was low and nobody else was using any of the machines. Two machines are used for the algorithms studied in the previous chapter.

4.2. Test Parameters

For each algorithm implemented, the two machines involved generate two relations of the same size in their primary memory respectively. The entire operation does not involve any disk I/O at all. Our relations consist of only key values which are integers in our experiments. The keys were randomly generated and were uniformly distributed over a certain range such that the number of tuples (i.e keys in our case) in the resulting relation was about the same as the size of the relations themselves. Duplicate keys were allowed.

Timing is measured by the GetTime system call provided by the V-system, which gives the instantaneous time since January 1, 1970 GMT. Because we had a dedicated system, the measurements are accurate. In fact, we ran our programs many times at different times and the results only showed about 1% difference. We designed the algorithms to consist of several steps, each representing a logical phase of processing. The elapsed time for each phase was recorded. The total cost, the response time in our research, of an algorithm is defined to be the sum of the time used during each step.

As was stated earlier, the V-system employs blocking communication interfaces, which means the data-exchange process is the period from the initiation of the operation till the arrival of an acknowledgement from the receiver. The implication is that some local processing such as copying data from one location to another is also included in the communication overhead.

4.3. Analysis of Experiments

4.3.1. Test Results

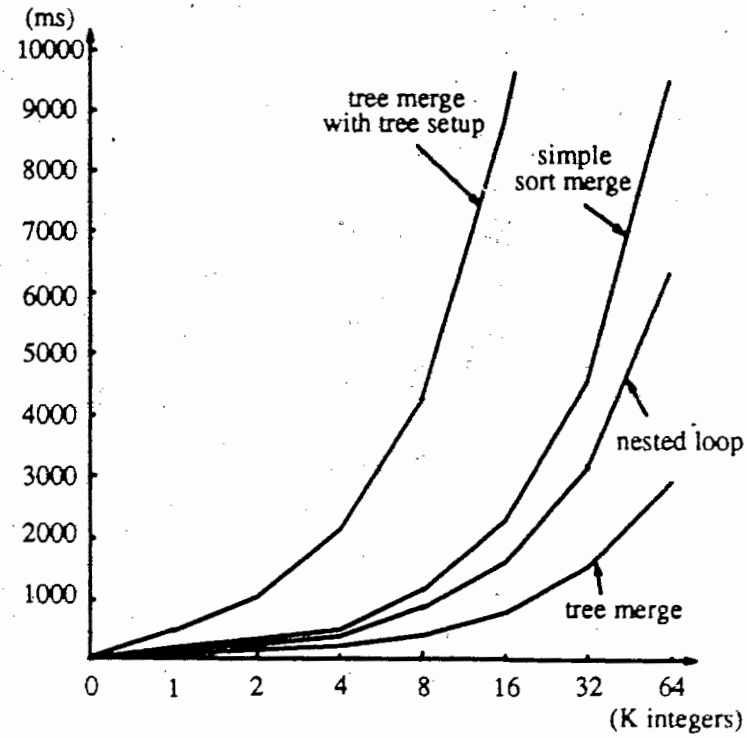

Figure 4-1: Performance Comparison

Appendix A.1 summarizes the experimental results obtained for Algorithms 1, 2 & 3. They are measured by the total elapsed time of a transaction in milliseconds. Figure 4-1 shows the performance of the algorithms, where the sizes are the number of keys for one of the two relations.

4.3.2. Preprocessing vs Local Join Operation

For Algorithm 1, the preprocessing execution is the local sorting operation. The two machines sort their relations in parallel and the timing is measured on the result machine. From the experimental data, we can see that the cost of local sorting grows fast (more than linearly) as the size of the relations increases. It is the most expensive component of the total cost. Local join operation refers to the merging of the two relations. Just as theoretical complexity analysis indicates, the merging cost is linear with respect to the size of the relations.

For Algorithm 2, preprocessing means the hash table setup for one of the relations. The data show the cost grows linearly with regard to the size. The same is true for the local nested loop join operation. But unlike Algorithm 1, where preprocessing cost is in most cases 3 times greater than the merging cost, the table setup is less expensive than the local join operation. There is clearly a tradeoff between preprocessing and local join components. Although this tradeoff varies from one processing strategy to another, a sophisticated preprocessing method produces a simple local join operation, at least in this case. Hence, a costly join operation may be eliminated by introducing a preprocessing phase. The tree merge algorithm is an exception because of the assumption that the tree indices already exist.

4.3.3. Local Processing vs Communication

We refer local processing to be the sum of preprocessing and (parallel) local join operations, which is largely determined by the sophistication of the processing strategies and the power of the participation processors. The data exchange process, on the other hand, depends largely on the sophistication of the system communication facilities and the nature of the underlying communication medium. Yet, it is also partly determined by the power of the processors since, as we know, some local processing is also included which will be affected by the hardware technologies.

As we can see from the results, the data exchange only constitutes a small portion of the total cost, whereas local processing constitutes the majority of the total cost. Table 4-1 shows the comparisons for Algorithms 1 & 2.

The results imply that we can employ more sophisticated strategies to improve the performance although the communication overhead may be increased. The communication medium is no longer the most critical resource and the strategies used for long haul network is no longer the best in a local area environment. The improvement could prove to be substantial.

Algorithm 1:					
Sizes	4K	8K	16K	32K	64K
Local Proc	494	1036	2127	4416	9116
	95.1%	95.2%	95.6%	95.8%	95.8%
Comm	25	52	98	192	402
	4.9%	4.8%	4.4%	4.2%	4.2%
Algorithm 2:					
Sizes	4K	8K	16K	32K	64K
Local Proc	381	755	1501	2998	6003
	92.5%	92.8%	93.8%	93.8%	93.8%
Comm	31	59	99	199	394
	7.5%	7.2%	6.2%	6.2%	6.2%

Table 4-1: Local Processing vs Communication

4.3.4. Merging and Nested Loop Strategies

In the previous chapter, we have discussed the theoretical analysis of the algorithms. However, we can only tell, from the theoretical complexity, that one is asymptotically better or worse than another. Since the different complexities are expressed in terms of different operations and one operation can be cheaper or more expensive than another, we can not predict whether one algorithm is *absolutely* better than the other.

Our empirical data show that tree-merge is the best provided both indices are available for the two relations. Tree traversal is surprisingly cheap because one data movement is much less costly than one data comparison or one arithmetic calculation. If the two relations are already in sorted order, the simple sort-merge proves to be the best. Otherwise, nested loop will be the best. Each algorithm fits best to a particular situation.

One advantage of using merging methods is that they produce sorted results, which meets the requirement of some applications. They are also suitable for non-equi-joins. The hashing nested loop join could be made fast if the hash functions are properly chosen and tuned.

4.3.5. Trend of Algorithm Performance

Figure 4-1 shows the performance of the algorithms. As we can see, the results are consistent in the sense that the costs increase in proportion to the growth of the relation size. All local processing cost components are doubled when the size of the relations is doubled. One exception is the local sorting cost component (Algorithm 1). It grows slightly faster than linearly, which agrees with the cost analysis.

The communication expense grows linearly, as we can predict from the complexity. Since the data exchange process involves only the two synchronized machines, the chance of getting contentions is very small. Therefore, the cost will grow in proportional to the change of the sizes.

For Algorithms 2 and 3, it is safe to predict that the total cost will continue to grow linearly for relation sizes greater than 64K, as long as the main memory is still large enough to hold the entire working storage. The total cost of Algorithm 1 will grow slightly faster than linearly.

4.3.6. Further Improvements

In all three algorithms, only two machines are involved. One of the relations, usually in the remote machine, has to be sent over to the other machine and local join is subsequently performed there. However, as we can see, the remote machine is virtually sitting idle after the transmission of its relation. Therefore, possible performance improvement can be achieved by allowing more parallel processing.

For the simple sort-merge algorithm, immediately after the local sorting phase, the two relations can be partitioned into two parts such that the first part of one relation only needs to be joined with the first part of the other relation. This can be easily done since the two relations are sorted at the moment. Then, the first part of the relation at the remote site can be transferred to the result site and the second part of the relation at the result site be transferred to the remote site. Merging can then be done in parallel at both sites. At the last phase, the result obtained at the remote site is transferred to the result site. Figure 4-2 shows the entire procedure. The strategy is based on the fact that communication overhead is relatively low so that local processing/data exchange tradeoff can be beneficial. Similar modification can be done to the tree merge algorithm.

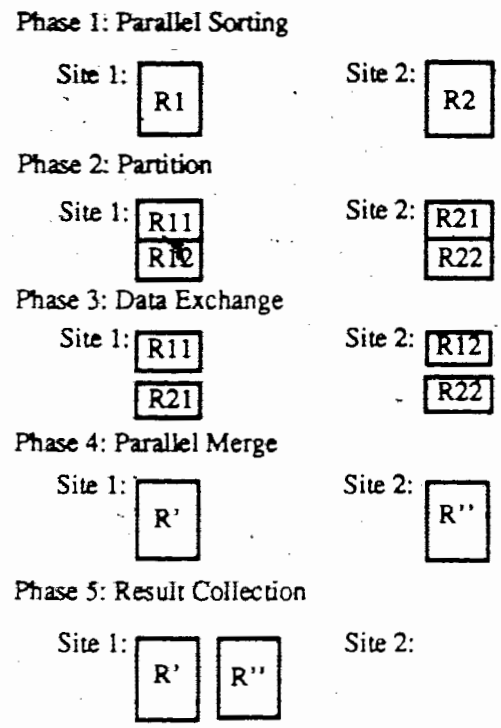

Figure 4-2: Example of the New Strategy

For the hashing nested loop method, the relations cannot be partitioned easily in the same way since they are not sorted. However, they can be simply partitioned into two equal parts. Then, the second half of the relation at the result machine is sent to the remote site; and a copy of the whole relation at the remote site is shipped to the result site. Next, a hash index is set up, in parallel, for the two parts of the original relation at the result site. Subsequent nested loop join is performed between the two parts and the relation from the remote site. The result obtained at the remote site is then transferred to the result site. With the strategy, the hash index setup cost is cut in half, at the cost of increasing data exchange overhead.

However, we are still not satisfied with the strategies developed. Notice that machines in a local network may not work 24 hours a day. The probability that additional machines are sitting idle or have light workload at a moment is high. We would like to develop more general strategies, which leads to the study of the following chapters.

Chapter 5

DMMDB Query Processing with Load Sharing

5.1. Motivations

In the conventional approaches, the amount of data transfer is minimized, or at least, extra traffic is not encouraged. A join operation involves only the two processors (or sites) at which the original two relations are stored. One relation is transferred from its site to the other in one transmission stage and join operation between the two is locally performed there.

With the development of distributed systems and communication technologies, more and more computers can be connected to a network with fast data transfer rate and good overall system performance. The fact is that the probability that at least one processor is sitting idle while tasks are waiting at other sites in a distributed system (a "wait while idle" state) is remarkably high over a wide range of network sizes and processor utilizations [LU 85]. Hence, with a high performance distributed system over a high-speed local area network, such as V-system ([CHER 84] and [BERG 86]), we may distribute over the network the workload incurred by join operations to achieve high parallelism as the result of the processing/communication tradeoff.

Communication cost can be traded for local computational expense. [LUK 87] has done theoretical analysis and practical experiments on distributed sorting algorithms. For the sorting algorithms, it has been shown that local processing cost is dominant while the total communication cost is only a small fraction of the total expense. Processing time should be minimized even with the increased expense of communication.

For join algorithms, the situation is similar in that it is also data intensive. Local processing cost tends to be a major factor. It is, therefore, very likely to be beneficial to distribute tasks over the network. Transferring part of system load from a congested area to a lightly loaded area will reduce the queuing time of tasks and speed up the local processing.

As we know, during the data transmission process, only less than 10% of the time is spent on real physical communication medium. The rest is spent on the pre-processing and post-processing in the transmitting and receiving sites [PAGP 85]. A main memory system can make the preprocessing and post-processing faster. However, as we will see, local processing cost is still dominant even in a main memory system. While the performance of a centralized MMDB ceases to be improved with even more memory if the memory size is already big enough, distributing workload tends to be an effective way of improving performance. We call the approach a load sharing one in the sense that more machines are involved to handle the original two-machine task. However, our approach is not the duplication of the traditional load sharing strategies. While the traditional load sharing strategies distribute tasks amongst processors over a network to achieve a system-wide performance improvement in throughput, our approach further decomposes a task into subtasks which run in parallel in other machines and hence improves performance of a particular task. Data associated with the subtasks may also be transmitted.

5.2. The Revised Models

Suppose there are N processors interconnected by a high performance network. Each of the processors is assumed to have reasonably large primary memory for query processing. A join operation is going to be performed between two relations residing at two different processors (or sites), which are called S_{rst} and S_{rmt} respectively. The result site S_{rst} is where the final result is to be collected.

The first stage of the new approaches is, unlike conventional ones, to preprocess the original relations for subsequent load distribution. The two relations are partitioned, according to a certain set of criteria, into a number of subrelations which are then transferred to other sites through the communication medium. The next stage is the parallel processing of the subrelations at each site to ease the later join operations, which usually includes such operations as index setup, array index sorting, etc. Subsequently, the subrelations are joined together. The result of each of these join operations is finally transferred to the result site.

Hence, basically, the whole procedure consists of four phases: (1) preprocessing of original

relations and distribution of the subrelations; (2) preprocessing of the subrelations at individual sites for subsequent join operations; (3) join operations between pairs of subrelations at each processor; (4) final result collection. We call the algorithms based on the approach load sharing algorithms.

Our cost model for a load sharing algorithm also consists of four parts, analogous to the four basic processing phases. The first component is the cost of preprocessing for load distribution which accounts for the expense of local processing prior to workload distribution. Since the load distribution is not intrinsic to the preprocessing and different measurement scale is used, it is counted as part of data exchange. The second component is the cost of preprocessing for local joining which accounts for the expense of local processing prior to pair-wise join operations. The third component is the total parallel data transfer cost which includes the costs for load distribution, data-exchange before and during pair-wise joins, if any, and final result collection. The last component is the cost for total parallel local join operations.

The third component depends on the amount of data to be transferred. The other three components are measured in terms of the number of data movements and comparisons. The sum of the four gives the response time of the entire join procedure.

5.3. Load Sharing Algorithms

We have argued in the previous section that the load distribution strategy is very likely to significantly improve performance for a main memory system, as well as for a disk-based system. We have developed three load sharing algorithms for distributed join processing. They are the load sharing version of the algorithms studied in Chapter 3.

Suppose the two original relations are resident at two sites called S_{rst} and S_{rmt}, respectively. We have also some even number of additional sites which are divided into two disjoint groups with S_{rst} and S_{rmt} being the group leaders respectively. Note that in the simple sort merge algorithm, sorting is the most costly operation. To apply our load distribution strategy, the original relations should be partitioned and distributed so that the sorting task can be shared. S_{rst} and S_{rmt} then partition their relations into a number of subrelations of approximately equal size, and distribute them within

their own group through the communication network. Each subrelation in one group is subsequently joined in parallel with every subrelation in the other group. The results are kept at sites led by S_{rst}. Finally, S_{rst} collects results from its group members. A formal description of the procedure is given in Algorithm 4. Detailed pseudo-code of the algorithm can be found in Appendix B. Since the algorithm is derived from the simple sort-merge method, we call it a load-sharing sort-merge algorithm.

Algorithm 4. Load Sharing Sort Merge

1. Suppose there are n sites on a local network with light loading, where n is an even number, and the sites are divided into two groups. The first consists of $\frac{n}{2}$ sites with S_{rst} being the leader. The second group also consists of $\frac{n}{2}$ sites, with S_{rmt} being the leader. The two relations are at S_{rst} and S_{rmt} respectively. S_{rst} is the result site.
2. Partition Phase
   Sites S_{rst} and S_{rmt} partition their relations into $\frac{n}{2}$ equal subrelations in parallel. Then, site S_{rst} sends $\frac{n}{2}-1$ of its $\frac{n}{2}$ subrelations to its group members, and S_{rmt} sends $\frac{n}{2}-1$ of its $\frac{n}{2}$ subrelations to its group members.
3. Parallel Sorting Phase,
   All sites sort their subrelations in parallel.
4. Pair-Wise Merge Phase
   Every member of one group perform pair-wire merging with a member from the other group, and the results are stored in the members of S_{rst}'s group.
   This process repeats $\frac{n}{2}$ times, until every member of one group has performed merging with every member of the other group.
5. Result Collection Phase
   S_{rst} collects results from its group members.

In the above algorithm, the workload is distributed and pair-wise mergings are performed in parallel. Therefore, the overall performance may be improved. However, as we have noticed, each subrelation in one group has to be joined with every subrelation of the other group since we do not employ, during the partitioning, any knowledge about which tuple should be joined with which other tuple(s). The data exchange cost may increase fast as the number of processors involved increases. One solution will be to divide the original relations into disjoint sets of tuples such that tuples in one set of a relation have to be joined with tuples of a corresponding set of the other relation. In this sense, the sets of one relation have a one-to-one matching relationship with the sets

of the other relation. We have developed a partitioning technique using hashing functions. A hash function is chosen and a relation is partitioned into several non-overlapping portions in such a way that two tuples in the same portion should have the same hash value. If two relations are partitioned this way by the same hash function, then to evaluate the join of the two, it suffices to join the corresponding portions respectively, since it is only possible for two tuples from the two corresponding portions to have the same key value.

Since each subrelation does not have to be joined with every other, we can send the two corresponding subrelations to the same site in the load distribution phase to eliminate the later data exchange overhead. From our previous experiments, the total cost of local sorting and subsequent merging is higher than the cost of setting up a hash index and pursuing subsequent joining. Therefore, nested loop join with hash index is employed for local processing, and we call the algorithm the load sharing nested loop algorithm.

The drawback of the partition method is that it requires one additional scan of the relations and the calculation of a hash value for each tuple. This may be costly compared with data transfer expense.

A formal and more precise description of the procedure is given in Algorithm 5.

Algorithm 5. Load Sharing Hash Join

1. Suppose there are n sites on a local network with light loading, where n is an even number, and the sites are divided into two groups. The first consists of $\frac{n}{2}$ sites with S_{rst} being the leader. The second group also consists of $\frac{n}{2}$ sites, with S_{rmt} being the leader. The two relations are at S_{rst} and S_{rmt} respectively. S_{rst} is the result site.
2. Partition Phase
   A hash function is chosen to partition the two original relations into n subsets each.
3. Load Distribution Phase
   a. S_{rst} and S_{rmt} each transfer $\frac{n}{2}-1$ subrelations to its group members.
   b. S_{rst} and S_{rmt} each transfer $\frac{n}{2}-1$ subrelations to the members of the other group.
   c. S_{rst} and S_{rmt} exchange one subrelation.

   At the end of the phase, each site has the two corresponding subrelations, one from S_{rst} and the other from S_{rmt}.

4. Index Setup Phase
   Each site creates a hash index for one of its subrelations.
5. Nested Loop Join Phase
   Each site performs the joining of the two subrelations.
6. Result Collection Phase
   S_{rst} receives results from all other sites.

Unlike usual load balancing algorithms which decompose a user query into a sequence of non-interrelated subqueries and dynamically distribute the subqueries over the network, our algorithms are concerned with the distribution of "indivisible" subqueries. It will certainly be beneficial if the relations are large.

We have developed two load sharing algorithms. Each of them aims at reducing an expensive cost component by making use of extra machines available. Naturally, the question arises as to whether they are the optimal algorithms. Consider the pair-wise merging phase of Algorithm 4, where simple sort-merge is employed as in Algorithm 1. There are two sites involved, one from S_{rst}'s group and one from S_{rmt}'s group for each pair-wise merging operation. After transferring its subrelation over to the other site, the one belonging to S_{rmt}'s group will sit idle until the next operation begins. Further reduction in local processing is still possible.

Consider the partition phase of Algorithm 5, where a hash function is chosen and the relations are scanned and partitioned into n parts while other machines are sitting idle. There is the possibility of further refinement by letting other sites share the partition task. Our aim is to achieve optimal solutions in terms of communication/local processing tradeoffs. We expect the processing cost will be reduced by half, except for the communication overhead, if the processor resource is doubled.

We have developed the following two modified algorithms. For Algorithm 4a, after the parallel sorting phase, each subrelation is further partitioned into n subcomponents. In the following component exchange phase, the i^{th} processor collects the i^{th} components of all subrelations. The pair-wise merge is then performed within each individual site. For Algorithm 5a, the relations are equally divided and distributed among the processors. The subrelations are then partitioned by a chosen hash function into n nonoverlapping parts which are subsequently transmitted to the corresponding sites. The partition task of Algorithm 5 is shared by other machines.

Algorithm 4a. Load Sharing Sort Merge (modified version)

1. Suppose there are n sites on a local network with light loading, where n is an even number, and the sites are divided into two groups. The first consists of $\frac{n}{2}$ sites, with S_{rst} being the leader. The second group also consists of $\frac{n}{2}$ sites, with S_{rmt} being the leader. The two relations are at S_{rst} and S_{rmt} respectively. S_{rst} is the result site.
2. Partition Phase
   S_{rst} and S_{rmt} partition their relations into $\frac{n}{2}$ equal subrelations in parallel. Then, they send $\frac{n}{2}-1$ of the $\frac{n}{2}$ relations to their group members, respectively.
3. Local Sorting Phase
   Each site sorts its subrelation using internal QuickSort.
4. Pivot Finding Phase
   S_{rst} finds the n-1 pivot values which divide its subrelation into n (almost equal) parts, and them send them to all other sites. The other sites, upon receiving the pivot values, further divide their subrelations into n parts by the n-1 values.
5. Component Exchange Phase
   All sites send, in turns, n-1 of their subrelations to the corresponding sites, in such a way that the corresponding subrelations will be sent to the same site.
6. Merging Phase
   All sites perform the merging operation on their subrelations in parallel.
   a. All subrelations from S_{rst}'s group members are to be merged together.
   b. All subrelations from S_{rmt}'s group members are to be merged together.
   c. The merge join is then performed between the two resulting relations.
7. Result Collection Phase
   S_{rst} receives results from all other sites.

Algorithm 5a. Load Sharing Hash Join (modified version)

1. Suppose there are n sites on a local network with light loading, where n is an even number, and the sites are divided into two groups. The first consists of $\frac{n}{2}$ sites, with S_{rst} being the leader. The second group also consists of $\frac{n}{2}$ sites, with S_{rmt} being the leader. The two relations are at S_{rst} and S_{rmt} respectively. S_{rst} is the result site.
2. Distribution Phase
   S_{rst} and S_{rmt} partition their relations into $\frac{n}{2}$ equal subrelations in parallel, and send them to their respective group members.
3. Partition Phase
   Choose a hash function.
   All sites partition, using the same hash function, their subrelations into n subsets.
4. Component Exchange Phase
   Each site in turn, one site at a time, sends its subsets to the corresponding sites.

All sites combine subsets from its own group members into one relation, and combine others into another relation.

5. Index Setup Phase
   Each site sets up, in parallel, a hash index for one of its two relations.
6. Local Joining Phase
   All sites perform the join operation on the two relations residing on them.
7. Result Collection Phase
   S_{rst} receives results from all other sites.

Appendix C gives two examples of how Algorithms 4a and 5a work.

The previous algorithms are developed from the conventional sort-merge and hash indexed nested loop strategies. Another load sharing algorithm can also be derived from the tree merge algorithm. Recall that the tree-merge algorithm gives the best performance if the T-tree indices exist for both relations. Tree traversal is cheap compared with other cost components. Since the array index obtained from the tree searching is sorted, it is easy to partition it into a number of almost equal parts. The key value at the beginning of each part is called a pivot value. Similar to the partition strategy for Algorithm 5, if we partition the two relations by the same pivot values, then only the tuples from two corresponding subrelations need to be joined together, and all the subrelations are also in sorted order. Distribution of the subrelations will allow other machines to share the merging cost. According to some of our experiments, tree traversal and relation partition costs are so low that it is not worth the effort to further distribute the task at the cost of increased communication overhead.

The following shows a formal description of the algorithm.

Algorithm 6. Load Sharing Tree Merge.

1. Suppose there are n sites on a local network with light loading, where n is an even number, and the sites are divided into two groups. The first consists of $\frac{n}{2}$ sites, with S_{rst} being the leader. The second group also consists of $\frac{n}{2}$ sites, with S_{rmt} being the leader. The two relations are at S_{rst} and S_{rmt} respectively. S_{rst} is the result site. Assume there are T-tree indices for both relations.
2. Tree Traversal Phase
   S_{rst} and S_{rmt} traverse the tree indices to get the sorted array indices.
3. Partition Phase
   a. S_{rst} partitions its relation into n almost equal subrelations.

b. S_{rst} sends the partitioning key values to S_{rmt}.
c. S_{rmt} partitions its relation into n subrelations according to the pivot values.

4. Data Transmission Phase

a. S_{rst} and S_{rmt} send $\frac{n}{2}-1$ subrelations to their own group members.

b. S_{rst} and S_{rmt} send another $\frac{n}{2}-1$ subrelations to members of the other group.

c. S_{rst} and S_{rmt} exchange a subrelation.

At the end of this phase, all sites contains two corresponding subrelations to be joined.

5. Merging Phase
All sites perform merging of their subrelations.

6. Result Collection Phase
S_{rst} receives results from all other sites.

Appendix B gives the pseudo code for the algorithms.

5.4. Cost Analysis

As before, costs are measured by the time complexity of each processing phase, based on our revised cost model. Recall that the cost equation for each algorithm consists of four parts: preprocessing for load distribution, preprocessing for pair-wise joining, communication cost, and local joining expenses. Parallel processing is the processings of many tasks that are initiated (almost) simultaneously by the processors over a network. Maximum parallelism is achieved when the time needed to process the simultaneous tasks is approximately the same as the time needed to process the longest one. The (parallel) cost of some simultaneous tasks is the cost to process the longest task. If several tasks are processed sequentially, the total cost will be the sum of each individual cost component. The four cost components above are all parallel cost measurements.

Assume there are N sites available in the network. The original two relations are both of size M, i.e., consists of M keys each. As for the algorithms discussed in Chapter 3, different quantity units are used for different preprocessing techniques employed. For the hash-based partition, since a hash value has to be calculated for each key, the number of keys, i.e., the size of the relations, serves as the measurement unit. Equal partition divides a relation into a number of equal parts. The number of parts is used to be the quantity unit. The pivot finding process is measured by the number of data comparisons, which, as we will see later, is negligible. The remaining components

are counted by the same units as in Chapter 3. Later in this section, we will not explicitly show the quantity units.

For Algorithm 4 (Load Sharing Sort Merge), the N processors are divided into two groups of $\frac{N}{2}$ processors, led by the result site, S_{rst}, and the remote site, S_{rmt}, respectively. The task of preprocessing for load distribution is to partition simultaneously the two relations into $\frac{N}{2}$ equal subrelations at sites S_{rst} and S_{rmt}. The cost for it is then $\frac{N}{2}$, which is negligible. Each subrelation will be of size $\frac{2M}{N}$. This algorithm is derived from the simple sort-merge method. The sorting task is distributed at the cost of increased data exchange expenses. The cost of the second component is now the cost of internal sorting of a subrelation, that is, $O(\frac{2M}{N}\log(\frac{2M}{N}))$.

In this algorithm, each site is involved in $\frac{N}{2}$ pair-wise joinings. Each time, one sorted relation has to be transferred to the other site, and local joining is then performed. The cost of transmission is proportional to the size of the subrelation, that is, $\frac{2M}{N}$. As before, the cost of merging is $\frac{4cM}{N}$. Recall that c is the average number of matching tuples for a given tuple, which is 1 in our experiments. Therefore, the total cost of local joining is $2M$, and the total cost of data exchange during pair-wise joinings is M.

The data exchange time for load distribution is $(\frac{N}{2}-1)\frac{2M}{N} = M-\frac{2M}{N}$, and the data collection time is $(\frac{N}{2}-1)\frac{2M}{N} = M-\frac{2M}{N}$ because the result site collects final data from the sites one by one. The total data exchange cost is, then, $3M-\frac{4M}{N}$.

In the revised version of Algorithm 4, merging task is further distributed. The pivot finding overhead is counted as part of the preprocessing cost. As before, the cost of the partition of relations into $\frac{N}{2}$ equal part is $\frac{N}{2}$, which is negligible. The pivot finding cost is obviously $M\log(\frac{2M}{N})$, since the subrelations are already sorted and binary searching can be used. Before the part exchange phase, the size of each subrelation is approximately $\frac{2M}{N^2}$. During the part exchange phase, each site takes turn to transmit its subrelations to the corresponding sites. The communication cost of each turn will be $(N-1)\frac{2M}{N^2}$. Hence, the total cost of this phase is $2M-\frac{2M}{N}$.

For the merging phase, all subrelations are sorted. Each site contains two sets of subrelations, one consisting of those from S_{rst}'s group and the other consisting of those from S_{rmt}'s group. The subrelations from S_{rst}'s group and those from S_{rmt}'s group are first merged together respectively. Since there are $\frac{N}{2}$ subrelations from each group, $\log[\frac{N}{2}]$ steps are needed to merge subrelations from each group. For the first step, each subrelation is merged with one other subrelation to form a relation of double size. In the subsequent steps, pair-wise mergings between resulting relations from the last step are performed. Since the relations are originally of size $\frac{2M}{N^2}$, the cost of first merge step, as well as subsequent steps, is $\frac{M}{N}$. Therefore, the merge of subrelations from each group costs $\frac{M}{N}\log[\frac{N}{2}]$. After merging, each site contains two sorted relations of size $\frac{M}{N}$. The subsequent merge join then costs $\frac{2M}{N}$. Hence, the total expense of the merging phase is $\frac{M}{N}(\log[\frac{N}{2}]+1)$. The total number of tuples (keys) for each set of subrelations is approximately $\frac{M}{N}$. Therefore, the result relation size will be $c\times\frac{M}{N}$, where c is the average matching tuples for each tuple and is about 1 in our experiments. Therefore, the result collection expense is $M-\frac{M}{N}$. Plus the data distribution cost in phase 2, which is $M-\frac{2M}{N}$, the total communication expense is $4M-\frac{5M}{N}$. Note that the pivot distribution overhead is negligible. Table 5-1 gives the summary of cost analysis of Algorithm 4 and its modified version. Notice that Algorithm 4a gives globally sorted results, whereas Algorithm 4 does not.

Algorithm 5 is slightly different. The data exchange cost is reduced due to the sophisticated preprocessing technique. But the preprocessing cost is increased accordingly because of the additional scanning of the relations. Since the size of the relations is M keys, the hash-based partition of the partition phase costs M according to our convention. Nested loop algorithm with hash indices is used for the local joining. As in the case of Algorithm 2, the parallel cost of preprocessing for local joining operations, i.e. index set up, is $\frac{M}{N}$; and the parallel local joining cost is $c'\times\frac{M}{N}$. Recall c'=4, which is the average length of list associated with each table entry. The time to distribute subrelations to the two groups and to exchange data between the result site and the remote site is $2(\frac{N}{2}-1)\frac{M}{N}+2\frac{M}{N}=M$. The time for the result site to collect result data is $\frac{M}{N}(N-1)$. The total cost of data exchange is then $2M-\frac{M}{N}$.

Load Sharing Sort Merge Algorithm:

Preprocessing I (Partition)	Preprocessing II Parallel Sort	Communication	Parallel Local Merge
$\frac{N}{2}$ (negligible)	$O(\frac{2M}{N}\log(\frac{2M}{N}))$	$3M-\frac{4M}{N}$	2M

Load Sharing Sort-merge Algorithm: modified version

Preprocessing I Partition and Pivot Finding	Preprocessing II Parallel Sort	Communication	Parallel Local Merge
$M\log(\frac{2M}{N})$ (negligible)	$O(\frac{2M}{N}\log(\frac{2M}{N}))$	$4M-\frac{5M}{N}$	$\frac{2M}{N}(\log[\frac{N}{2}]+1)$

Table 5-1: Cost Analysis of Algorithms 4 & 4a

In the revised version of Algorithm 5, the partition task is further distributed to achieve more parallelism. The preprocessing cost now includes the partition of the original relations into $\frac{N}{2}$ equal parts (phase 2) and the further partition of the subrelations into N nonoverlapping sets (phase 3). The cost of the first component is $\frac{N}{2}$, which is negligible. The cost formula for the second component is proportional to the subrelation sizes, which is $\frac{2M}{N}$. Now, the size of each subset of a subrelation is approximately $\frac{2M}{N^2}$.

The data exchange phase is the same as that for Algorithm 4a. Hence, it costs $2M-\frac{2M}{N}$. At the end of this phase, after all sites combine the subsets properly, we have two relations at each site, being approximately of size $\frac{M}{N}$, respectively. It follows that the index setup cost is $\frac{M}{N}$, and the local join cost is $\frac{4M}{N}$. As for Algorithm 5, the result collection overhead is $\frac{M}{N}(N-1)$. On the other hand, the task distribution cost in phase 2 is $M-\frac{2M}{N}$, the total communication overhead is then $4M-\frac{5M}{N}$. Table 5-2 shows the cost analysis for Algorithm 5 and Algorithm 5a.

Load Sharing Hash Join Algorithm:

Preprocessing I Partition	Preprocessing II Hash Index Setup	Communication	Parallel Local Join
M	$\frac{M}{N}$	$2M-\frac{M}{N}$	$\frac{4M}{N}$

Load Sharing Hash Join Algorithm: modified version

Preprocessing I Partition	Preprocessing II Hash Index Setup	Communication	Parallel Local Join
$\frac{2M}{N}$	$\frac{M}{N}$	$4M-\frac{5M}{N}$	$\frac{4M}{N}$

Table 5-2: Cost Analysis of Algorithms 5 & 5a

The analysis for the load sharing tree-merge is straightforward. The tree setup and tree traversal costs are the same as Algorithm 3. However, the tree setup cost is not included as part of preprocessing expense due to the earlier assumption that the tree indices are most likely to exist. Therefore, the preprocessing includes only tree traversal and relation partition. The tree traversal complexity is M, as for Algorithm 3, and the relation partition is $M\log(\frac{2M}{N})$, as for Algorithm 4a.

The cost of phase 4, the data transmission phase, is M, the same as for Algorithm 5. At the end of the phase 4, each site will contain one subrelation from S_{rst} and one from S_{rmt}. They are of both approximately size $\frac{M}{N}$. Therefore, the merge join will be $\frac{2M}{N}$, as for the simple sort merge algorithm. And the result collection cost is the same as for Algorithm 5, which is $M-\frac{M}{N}$. Table 5-3 summarizes the analytical results for Algorithm 6.

Algorithm 6 Load Sharing Tree Merge.

Preprocessing	Communication	Parallel Local Merge
$M + M\log(\frac{2M}{N})$	$2M - \frac{M}{N}$	$\frac{2M}{N}$

Table 5-3: Cost Analysis of Algorithm 6

Chapter 6
Experimental Validation II

6.1. Test Results

We have implemented the load sharing algorithms and actually tested on our experiment environment. However, because of the limitation on the number of machines available, we have run the programs on only four machines(Sun-3's). The results can be well predicted for a reasonable number of machines. Appendix A gives the collection of results obtained from our experiments. Figure 6-1 shows the performance comparison among them. Note the tree setup cost is not included for the tree merge algorithm.

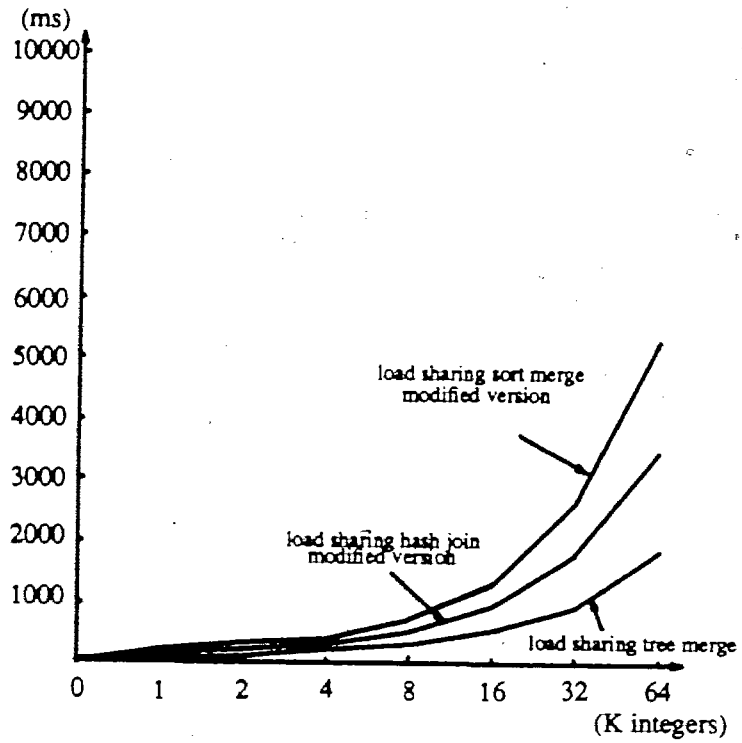

Figure 6-1: Comparison of Load Sharing Algorithms

6.2. Load Distribution

Load distribution for increased performance is the central idea behind the load sharing algorithms. The costly processing tasks in the conventional algorithms are decomposed and distributed over the network to reduce the total processing cost by making use of additional machines.

For load sharing sort merge algorithms, the local sort and merging components of the simple sort merge algorithm are distributed and processed in parallel. Hence, the (parallel) cost is much less than before, which is consistent with our theoretical analysis although minor deviation may exist.

For load sharing hashing methods, the index setup and local join jobs are decomposed and processed in parallel. However, some preprocessing effort is introduced in order for the hashing method to be efficient. The preprocessing task is further distributed in the modified algorithm (Algorithm 5a) based on the same idea. Similarly to the load sharing tree merge algorithm, the merging task is shared among multiple processors by partitioning the sorted relations into a number of subrelations and distributing them.

All the processing time reductions are achieved at the expenses of higher data volume for transmission. However, our experiments show that the communication is relatively cheap and data exchange cost can be further reduced by well planned data transmissions to increase communication parallelism. The experimental results have confirmed that load sharing strategies can effectively reduce the total processing cost. Table 6-1 gives the comparison of total response time, in milliseconds, of the algorithms when the relation size is 32K (for a total of 64K) integers. It also shows the percentage of improvement of the load sharing algorithms over their respective counterparts.

As we can see, load sharing strategies can effectively reduce the total cost of join tasks. We have mentioned that our load sharing approach towards relational joining operation is more like parallel processing procedure in the sense that the primary concern is to speedup the application execution by making use of additional machines. The distributed system is used to improve performance in the way a parallel system does. In a parallel system, we can often expect such features as direct

Algorithms	Response Time	Improvement
1	4608	
2	3197	
3	1464	
4a	2670	42.0%
5a	1781	44.3%
6	928	36.6%

Table 6-1: Comparison of Algorithms

sharing of memory and fast bus interconnections. However, there are two important features in our test environment: (1) the multiple-machine system is connected by a fast local network and is supported by a sophisticated communication software system; (2) it is a main memory system. They imply that the data exchange is cheap and the large amount of data can be processed locally without incurring costly disk operations. Our distributed environment is more favorable to data intensive applications and our approach will certainly be advantageous for a small number of machines.

The results of this research and of [LUK 87] suggest that parallel processing strategies may be employed for a distributed environment to increase performance.

6.3. Four Machines vs Eight Machines

Due to the equipment limitation, we have tested the load sharing algorithms with only four machines. However, since the empirical data are very consistent, we can predict, without much deviation, plausible results for a similar environment with more, specially eight, machines.

For the algorithms described in Chapter 3, both processors are synchronous and no other simultaneous transmissions are likely to occur. As more machines are involved, simultaneous messages are very likely to be initiated. Specially in our load sharing algorithms, transmissions among group members happen concurrently. Therefore, the concept of communication parallelism is particularly important. We say the *maximum communication parallelism* is achieved when the time needed to transmit a message in the presence of simultaneous messages is equal to the time to transmit the same message alone. The *minimum communication parallelism* transmission is the situation when the timing for simultaneous messages is equal to that for serial transmission of

them. Recall that in our theoretical cost analysis, the maximum communication parallelism is assumed. It implies that the time required to transmit messages initiated at the same time is equal to the time needed to transmit the longest one. However, in actual data transmission, contentions are likely to occur. The amount of network contention, which is difficult to predict theoretically, determines the degree of communication parallelism that can be achieved.

Let us consider Algorithm 5a (the modified load sharing hash join) with the assumption that there are eight machines available and the relation size is of 64K integers. The plausible results can be derived for the experimental data in the four machine case (they can be found in Appendix A.3). For the distribution phase, the cost of partition of the relations into equal parts is still negligible. In the partition phase, the relation at each machine will be half the size of the one in the four machine situation. Therefore, the presumable preprocessing cost will be approximately the same as that for the four machine case with the relation size of 32K integers, which is 404 ms. After the component exchange phase, each site will have two relations of approximately 8K integers. Hence, the hash index setup cost should be 319 ms. Similarly, the local join expense will be 485 ms, the same as that in the four machine situation when the relation size is of 32K integers.

The component most difficult to analyze is the data exchange expense. As we know from the previous chapter, the total communication process consists of three parts: load distribution, component exchange and result collection, and only the load distribution employs parallel transmission. As before, the sum of component exchange and result collection expenses is $3\times(M-\frac{M}{N})$. To calculate the communication upper bound, the minimum communication parallelism is assumed. Hence, the load distribution cost will be $2M-\frac{4M}{N}$, which is doubled the expense if maximum communication parallel is achieved. Therefore, the upper bound is $5M-\frac{7M}{N}$, which is $\frac{33M}{8}$ if N = 8, whereas the lower bound is $4M-\frac{5M}{N}$, which is $\frac{27M}{8}$ if N = 8. The lower bound for the load distribution expense (when maximum communication parallelism is assumed) is $M-\frac{2M}{N}$, which is $\frac{3M}{4}$ if N = 8. Since we know the time needed to transmit 64K integers is 402 ms(from Algorithm 1), the lower bound for the total data exchange cost will be 1356 ms; and the upper bound will be 1658 ms. It is 1492 ms if 10% contention rate is allowed. Therefore, the total cost for the algorithm is then 2700 ms, achieving 21.8% improvement over the four machine situation.

Similarly, we can get the upper bounds for Algorithms 4a and 6. They are $5M-\frac{7M}{N}$ and $3M-\frac{3M}{N}$, which are $\frac{33M}{8}$ and $\frac{21M}{8}$ respectively when N = 8. The estimated results can then be derived. Notice that the merging cost of Algorithm 4a will not be cut in half when the number of machines is doubled. Table 6-2 shows the data obtained.

	Algorithm 4a	Algorithm 5a	Algorithm 6
Upper bound	1658	1658	1055
Lower bound	1356	1356	754
Total Cost with 10% contention	3831	2700	1611
Actual Cost (4 machines)	5431	4520	1840
Improvement	29%	21.8%	12.5%

Table 6-2: Eight Machine Results

6.4. Optimal Number of Machines

Because of the facility limitation, we can only predict the optimal number of machines based on our previous theoretical and empirical study. The cost component most difficult to analyze is the communication overhead, since it is hard to predict the degree of communication parallelism that can be achieved. However, as the number of machines increases and more contentions occur, we can eliminate the parallel transmissions and employ serial transmissions for all data transfer process. The behaviors of serial transmissions are quite predictable and the upper bound complexity gives the precise estimation.

With this assumption in communication, we proceed to study the optimal number of machines for the load sharing algorithms. Let us consider Algorithm 5a. As we know from Chapter 5, the total elapsed time can be expressed as follows:

$$T_{total} = a_1 \times T_{preproc} + a_2 \times T_{setup} + a_3 \times T_{comm} + a_4 \times T_{join}$$

where $T_{preproc}$, the preprocessing cost component, is $\frac{2M}{N}$; T_{setup}, the cost for hash index setup, is $\frac{2M}{N}$; T_{comm} is the communication overhead and T_{join}, the expense of local joining, is $\frac{4M}{N}$. Since minimum communication parallelism is assumed, T_{comm} is $5M-\frac{7M}{N}$ as in the last section. Since

our experimental data are consistent with the cost analysis, the coefficients a_1, a_2 and a_4 can be determined by them. As we can see, the performance data are already stable, we can take the data obtained when the relation size is of 32K integers to calculate the coefficient values. Since the timing for preprocessing component is 404 ms, a_1 is equal to $404 \times \frac{4}{2 \times 32768} = 0.025$. Similarly, $a_2 = 0.019$ and $a_4 = 0.015$. To evaluate a_3, the time complexity upper bound has to be calculated. We know from Algorithm 1 that the time needed to transmit 32K integers is 192 ms. The upper bound timing when N = 4 is then 624 ms. Hence, a_3 will be 0.006 and we have the following formula:

$$T_{total} = 0.106 \frac{M}{N} + 0.03M.$$

As we can see, T_{total} is decreasing as N, the number of machines available, is increasing. The trend is shown in Figure 6-2. Therefore, there is no theoretically optimal number of machines for the load sharing algorithms. However, as the number increases, the percentage of actual performance improvement, in terms of timing, is decreasing. It also shows the performance improvements as a result of doubling the number of machines. This is because the local processing cost can no longer be reduced to greatly cut the total processing expense while the data exchange costs are likely to increase. Therefore, we claim that 16 to 20 machines would be optimal in this case if cost/effect factor is considered.

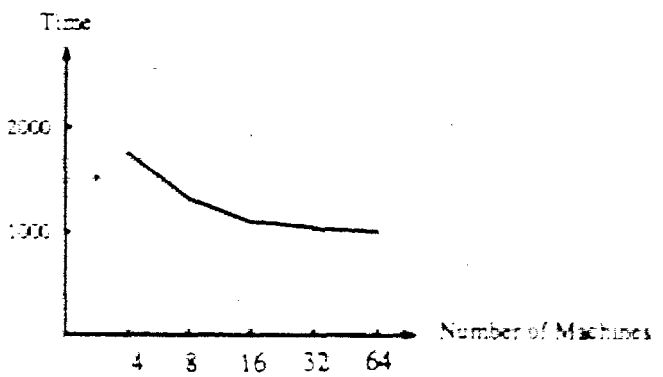

Figure 6-2: Trend of the Elapsed Time

We have done a similar analysis for Algorithm 4a and Algorithm 6. Notice, however, the tree traversal cost of Algorithm 6 will not be affected greatly by the number of machines available. The total local processing expense will not be cut in half if the number of machines is doubled. We estimate that 10 to 16 machines would be optimal in this situation.

Machines	Local Processing	Communication	Total Cost	Improvement
4	1208	573	1781	
8	604	792	1396	22%
16	302	876	1178	15.6%
32	151	918	1069	9%
64	75.5	939	1011.5	4%

Table 6-3: Trend of Algorithm 5a

The optimal number varies from the processing strategies used and the size of the relations involved.

6.5. Trend of Algorithm Performance

6.5.1. Improvement of the Network Performance

We consider two kinds of network improvement. One is the straightforward hardware improvement such as faster transmission media. The other is the software implementation improvement such as new protocals with less transmission overhead. Both will improve the overall performance at the same rate. However, they will not affect the cost of local processing. Table 6-4 shows the percentage of communication expense out of the total cost when relation size is of 32K integers, and the corresponding algorithmic performance increment if the network performance is increased by 100%.

Algorithms	Local Processing	Communication	Percentage	Improvement
1	4416	192	4.2%	2.1%
2	2998	199	6.2%	3.1%
3	1264	200	13.7%	6.85%
4a	2103	567	21.2%	10.6%
5a	1208	573	32.2%	16.1%
5	524	404	43.5%	21.75%

Table 6-4: Performance with Improved Network

As we can see, the improvements are not substantial, due to the small percentage of the data exchange expense. However, it favors the load sharing strategies which encourage local processing communication tradeoffs.

6.5.2. Improvement of Processor Capacity

The improvement in the processor capacity has two impacts. First, it directly reduces the local processing cost. Second, it speeds up data exchange as well. As we have discussed, in a data transmission process, a large amount of time is spent on the local processing of the transmitting and receiving machines. The communication cost also includes the local processing expenses, which will be reduced by more powerful processors.

Sun-2 workstations use MC68010 processors which are less powerful than MC68020 processors. We also ran some of our programs on Sun-2's in a similar environment. Table 6-5 gives the performance difference of Algorithm 1 in the two situations. It shows that several times of improvement has been achieved.

Sun-2's:

Size	Local Sort	Comm	Local Join	Total Cost
32K	9990	350	5340	15680

Sun-3's:

Size	Local Sort	Comm	Local Join	Total Cost
32K	3388	192	1028	4608

Table 6-5: Performance Difference with Sun-2's and Sun-3's

6.5.3. Relation Cardinality and Join Selectivity

Our experimental data are consistent for all algorithms in the sense that if the size of the relations increases, all the cost components also increase in proportion. We can reasonablely predict that in the similar environment, if the size of the relations becomes greater than 64K integers, the total cost, as well as each cost component, will change accordingly, in a linear fashion, as long as the entire database is still memory resident.

If we have a higher join selectivity, the resulting subrelation of each processing phase will be larger and hence the absolute exchange cost will be increased. Notice, however, that in the situation, the cost of merging two sorted relations will be increased because each key now has more matching ones; and the cost of joining two hash indexed relations will also be increased because there are, on average, more keys associated with each table entry. Therefore, the communication

cost is comparatively low even in this situation, and load sharing strategies are still expected to have good performance.

6.6. Multi-backend Database Systems: An Application

For the load sharing strategies we have discused previously, the multi-backend database systems can be an applicable environment. A prototype hardware organization is shown in Figure 6-3. A controller and a number of general-purpose backend machines are connected by an Ethernet-like broadcasting bus, with the controller being in turn attached to a host computer. All backend machines run the same system software and the entire database is distributed among the storage of individual machines. When a query regarding the database is received, the host passes it to the controller which broadcasts it to all the backend machines where the query is executed, in parallel, with the local database portion. As soon as a backend processor finishes the current query, it can start with the next one [HHKOS 83]. The overall system performance is increased and more concurrency is allowed.

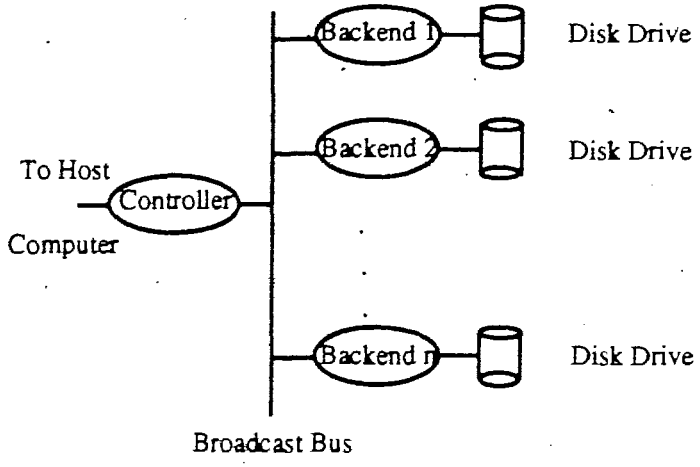

Figure 6-3: Multi-backend System Architecture

Our load distribution strategies can well fit the environment for the following two reasons. First, our experiment environment is very similar to the backend architecture. Second, since the entire backend system activities are supervised by the controller, the process procedures can be finely tuned such that maximum communication parallelism is likely to be achieved.

In a multi-backend database system, tuples are grouped into clusters. Therefore, the execution time of a query varies from machine to machine. Some machines may finish earlier and they have different workload throughout the processing period. Load sharing strategies are likely to be beneficial. As we have argued previously, increasing communication parallelism can also effectively reduce the data exchanging overhead. The overall system performance can be greatly increased if the load sharing strategies are applied and processing procedures are well planed.

Chapter 7
Conclusion

General query processing strategies for distributed main memory database systems have been investigated in this study.

An analytic system model and a cost model have been built for a local area network environment. The study shows that both local processing expense and communication overhead should be considered as major cost components. However, communication expense is cheap compared even with local main memory processing cost and it will become cheaper with improvements of processor speed, network bandwidth, and system software.

Two sets of algorithms are designed and analyzed based on the analytical models. Simple sort merge, hashing nested loop and tree merge are algorithms derived from those for conventional (disk-based) distributed systems.

There is no single algorithm that is the best in all aspects. Each algorithm can be the best for a specific environment. If both relations are already in sorted order, or one of the indices is missing and the result relation is required to be in sorted order, the sort merge strategy is the best. T-tree traversal is found to be a cheap operation, and the tree merge algorithm is the best of the three if both tree indices on the join column(s) exist. Hash join is preferred otherwise. Hashing can be effective in a main memory environment.

The load distribution concept has been developed. It differs from the conventional "load sharing" concept (although we call our algorithms load sharing ones) in that the otherwise "indivisible" tasks are decomposed and distributed to increase system performance by means of reducing application response time. The fact that communication overhead in a local area network is low makes the strategy feasible. Three algorithms, called load sharing sort-merge, load sharing hash join and load

sharing tree merge, are designed and analyzed. They are the load sharing versions of the previous set of algorithms. The algorithms are implemented in our experimental environment. Both our cost analysis and empirical data show that the load distribution strategy can effectively reduce the total processing cost. As an example, load sharing sort-merge improves simple sort-merge by 42.0% with four machines. Local processing/data transmission tradeoff is exploited. Although the load distribution concept is developed in the main memory environment, the results can be applied equally well to disk-based systems.

The experimental data are consistent with our cost analysis. For this reason, we can predict that the trend will be kept for larger relations as long as they can fit well into the main memory. The behaviors of the load sharing algorithms in the case of eight machines are discussed. They are expected to have better performance since all the costly local processing components are further distributed or shared. It is expected that 16 is the optimal number of machines for the load sharing algorithms, as further increase will result only in marginal performance improvement (<10%).

Our study shows that in a similar distributed environment, parallel processing strategies may be employed to improve system performance. The strategies may work well in the distributed environment for data intensive applications.

Appendix A

Experimental Results

A.1. Algorithms 1, 2 & 3

Algorithm 1
Simple Sort-Merge:

Size	Local Sort	Comm Cost	Merging	Total Cost
1K	82	11	31	124
2K	172	16	64	252
4K	365	25	129	519
8K	768	52	268	1088
16K	1614	98	513	2225
32K	3388	192	1028	4608
64K	7068	402	2048	9518

Algorithm 2
Nested Loop with Hashing Index:

Size	Table Setup	Comm Cost	Local Join	Total Cost
1K	38	10	56	104
2K	74	21	113	208
4K	150	31	231	412
8K	293	59	462	814
16K	577	99	924	1600
32K	1147	199	1851	3197
64K	2304	394	3699	6397

Algorithm 3
T-tree Merging Algorithm.

Size	T-tree Setup	Tree Traversal	Comm	Merging	Total Cost	Pure Join
1K	446	7	15	36	504	58
2K	943	17	19	65	1044	101
4K	1932	29	33	130	2124	192
8K	3966	63	61	254	4344	378
16K	8147	120	101	511	8879	732
32K	16615	246	200	1018	18079	1464
64K	33962	490	392	2035	36879	2917

A.2. Algorithms 4 & 4a

Algorithm 4
Load Sharing Sort Merge

Size	PreProc Cost	Local Sort	Comm	Merging	Total Cost
1K	0	39	34	28	101
2K	1	83	48	56	188
4K	1	171	78	113	363
8K	1	356	144	229	730
16K	2	761	250	443	1456
32K	0	1605	477	890	2972
64K	0	3364	942	1784	6090

Algorithm 4a
Load Sharing Sort Merge: modified version

Size	PreProc Cost	Local Sort	Comm	Merging	Total Cost
1K	16	43	82	17	158
2K	14	84	104	32	234
4K	15	174	132	59	380
8K	12	359	197	120	689
16K	12	765	316	243	1336
32K	12	1606	567	485	2670
64K	12	3372	1086	961	5431

A.3. Algorithms 5 & 5a

Algorithm 5
Load Sharing Hash Join

Size	Preproc Cost	Hash Index Setup	Comm	Local Join	Total Cost
1K	22	13	49	16	100
2K	48	23	58	31	160
4K	91	45	90	63	289
8K	182	82	136	119	519
16K	376	160	225	242	1003
32K	765	316	402	484	1967
64K	1538	636	777	950	3901

Algorithm 5a
Load Sharing Hash Join: modified version

Size	Preproc Cost	Hash Index Setup	Comm	Local Join	Total Cost
1K	14	10	79	16	119
2K	30	19	103	32	184
4K	55	39	138	60	292
8K	102	79	192	117	490
16K	205	160	316	245	926
32K	404	319	573	485	1781
64K	802	639	1060	952	3453

A.4. Algorithm 6

Algorithm 6
Load Sharing Tree-Merge

Size	T-tree Setup	Tree Traversal	Comm	Local Merge	Total Cost	Pure Join
1K	458	7	47	12	524	62
2K	939	15	64	18	1036	97
4K	1934	35	90	36	2095	161
8K	3977	62	130	75	4244	267
16K	8176	120	212	150	8658	482
32K	16670	242	404	282	17598	928
64K	34008	486	794	560	35848	1840

Appendix B

Pseudo-Code for the Join Algorithms

Algorithm 1. Simple Sort-Merge

Suppose there are two relations R_1 and R_2 at sites S_1 and S_2 respectively.

Site S_1:

1. Create an array index for R_1;
2. Sort the index using internal QuickSort;
3. Receive R_2 (and its index) from S_2;
4. Merge R_1 and R_2 to produce the result relation.

Site S_2:

1. Create an array index for R_2;
2. Sort the index using internal QuickSort;
3. Send R_2 (and the index) to S_1.

Algorithm 2. Nested Loop Join with Hash Index

Suppose there are two relations R_1 and R_2 at sites S_1 and S_2 respectively.

Site S_1:

1. Create a hash index for R_1;
2. Receive R_2 from S_2;
3. Local joining using nested loop method.

Site S_2:

1. Send R_2 to S_1.

Algorithm 3. Tree Merge Join

Suppose there are two relations R_1 and R_2 at sites S_1 and S_2 respectively.

Site S_1:

1. Create a T-tree index for R_1;
2. Traverse the index to get a sorted array index for R_1;
3. Receive R_2(the array index) from S_2;
4. Merge the two relations.

Site S_2:

1. Create a T-tree index for R_2;
2. Traverse the index to get a sorted array index for R_1;
3. Send R_2(the array index) to S_1.

Algorithm 4a. Load Sharing Sort Merge.
(modified version)

1. Suppose there are n sites on a local network with light loading:

   $S_1, S_2, \ldots, S_n$;

   where n is an even number, and the sites are so numbered that the original relations are at sites S_1 and $S_{\frac{n}{2}+1}$. S_1 is the result site.

2. Partition Phase

   Sites S_1 and $S_{\frac{n}{2}+1}$ partition their relations into $\frac{n}{2}$ equal subrelations in parallel. Then, site S_1 sends $\frac{n}{2}-1$ of the $\frac{n}{2}$ subrelations to sites $S_2, \ldots, S_{\frac{n}{2}}$ respectively; and site $S_{\frac{n}{2}+1}$ sends $\frac{n}{2}-1$ of its $\frac{n}{2}$ subrelations to sites $S_{\frac{n}{2}+2}, \ldots, S_n$ respectively.

3. Local Sorting Phase
   Each site sorts its subrelation using internal QuickSort.

4. Pivot Finding Phase

   a. S_1 finds the n-1 mean values which divide its subrelation into n (almost equal) parts.
   b. S_1 sends the n-1 mean values to each of $S_2, \ldots, S_n$.
   c. $S_2, \ldots, S_n$ divide their subrelations into n parts by the n-1 pivot values.

5. Part Exchange Phase
   For i = 1 to n do
   {
   S_i sends n-1 of its subrelations to the other sites, retaining the i-th subrelation for itself.
   }

6. Merging Phase
   For each of the sites $S_1, \ldots, S_n$, do
   {
   All subrelations received from $S_1, \ldots, S_{\frac{n}{2}}$ (including the one retained for itself) are to be merged together.
   All subrelations received from $S_{\frac{n}{2}+1}, \ldots, S_n$ (including the one retained for itself) are to be merged together.
   Merge join is performed between the two resulting relations.
   }

7. Result Collection Phase
   Sites $S_2, \ldots, S_n$ transfer resulting relations to the result site S_1.

Algorithm 5a. Load Sharing Hash Join.
(modeified version)

1. Suppose there are n sites with light loading over a local network:

   $S_1,\ldots,S_n$

   where n is an even number, and the sites are so numbered that the original relations, R_a and R_b, are at S_1 and $S_{\frac{n}{2}+1}$ respectively. S_1 is the result site.

2. Distribution Phase

   a. S_1 and $S_{\frac{n}{2}+1}$ partition, respectively, the two original relations into $\frac{n}{2}$ equal sized subrelations.

   b. S_1 sends $\frac{n}{2}-1$ of the subrelations to sites S_2, ..., $S_{\frac{n}{2}}$, and $S_{\frac{n}{2}+1}$ sends $\frac{n}{2}-1$ of the subrelations to sites $S_{\frac{n}{2}+2}$, ..., S_n.

3. Partition Phase

   a. Choose a hash function.
   b. All sites partition, using the same hash function, their subrelations into n subsets.

4. Component Exchange Phase
   For i = 1 to n do
   {
   S_i sends its subsets to the corresponding sites, i.e. the j^{th} subset is to be sent to site S_j.
   }
   All sites combine the $\frac{n}{2}$ subsets from sites S_1, ..., $S_{\frac{n}{2}}$ into one relation, and the $\frac{n}{2}$ subsets from sites $S_{\frac{n}{2}+1}$, ..., S_n into another relation.

5. Index Setup Phase
   Each site sets up, in parallel, a hash index for one of its two relations.

6. Local Join Phase
   All sites perform the join operation on the two relations residing on them.

7. Result Collection Phase
   Sites S_2, ..., S_n send result relations back to S_1.

Algorithm 6. Load Sharing Tree Merge.

1. Suppose there are n sites with light loading over a local network:

   $S_1,...,S_n$

   where n is an even number, and the sites are so numbered that the original relations, R_a and R_b, are at S_1 and $S_{\frac{n}{2}+1}$ respectively. S_1 is the result site.

   Assume there are T-tree indices for both relations.
2. Tree Traversal Phase

   S_1 and $S_{\frac{n}{2}+1}$ traverse the tree indices to get the sorted array indices.
3. Partition Phase

   a. S_1 partitions its relation into n almost equal subrelations.

   b. S_1 sends the partitioning key values to $S_{\frac{n}{2}+1}$.

   c. $S_{\frac{n}{2}+1}$ partitions its relation into n subrelations according to the povot values.
4. Data Transmission Phase

   a. S_1 sends $\frac{n}{2}-1$ subrelations to $S_1,...,S_{\frac{n}{2}}$ and $S_{\frac{n}{2}+1}$ sends $\frac{n}{2}-1$ subrelations to $S_{\frac{n}{2}+1},...,S_n$.

   b. S_1 sends another $\frac{n}{2}-1$ subrelations to $S_{\frac{n}{2}+1},...,S_n$ and $S_{\frac{n}{2}+1}$ sends another $\frac{n}{2}-1$ subrelations to $S_1,...,S_{\frac{n}{2}}$.

   c. S_1 and $S_{\frac{n}{2}+1}$ exchange a subrelation.

   At the end of this phase, all sites contains two corresponding subrelations to be joined.
5. Merging Phase

   All sites perform merging of their subrelations.
6. Result Collection Phase

   S_1 receives results from all other sites.

Appendix C

Examples for the Load Sharing Algorithms

C.1. An Example for Load Sharing Sort Merge (Algorithm 4a)

The data file in each site is shown as follows:

S_{rst}: 1, 5, 10, 12, 22, 18, 28, 36, 2, 7, 11, 12, 4, 24, 29, 35

S_{rmt}: 3, 4, 17, 10, 14, 19, 20, 9, 2, 9, 13, 15, 7, 16, 25, 30

Note that the data file is not initially sorted as shown. There are two additional machines available. They are S_1 and S_2 respectively.

1. **Partition**: sites S_{rst} and S_{rmt} partition their data files into 2 equal parts. S_{rst} sends part2 to S_1 and S_{rmt} sends its part2 to S_2.
   S_{rst}: part={1, 5, 10, 12, 22, 18, 28, 36}
   S_1: part={2, 7, 11, 12, 4, 24, 29, 35}
   S_{rmt}: part={3, 4, 17, 10, 14, 19, 20, 9}
   S_2: part={2, 9, 13, 15, 7, 16, 25, 30}
2. **Local Sorting Phase**: all sites sort their own parts in parallel.
   S_{rst}: part={1, 5, 10, 12, 18, 22, 28, 36}
   S_1: part={2, 4, 7, 11, 12, 24, 29, 35}
   S_{rmt}: part={3, 4, 9, 10, 14, 17, 19, 20}
   S_2: part={2, 7, 9, 13, 15, 16, 25, 30}
3. **Pivot Finding Phase**: S_{rst} further partitions its part into 4 parts and sends the three pivot values to S_1, S_{rmt} and S_2 which then partiton their data sets into 4 parts according to the pivot values. The pivot values are 10, 18, and 28.
   S_{rst}: part1={1, 5}, part2={10, 12}, part3={18, 22}, part4={28, 36}
   S_1: part1={2, 4, 7}, part2={11, 12}, part3={24}, part4={29, 35}
   S_{rmt}: part1={3, 4, 9}, part2={10, 14, 17}, part3={19, 20}, part4={}
   S_2: part1={2, 7, 9}, part2={13, 15, 16}, part3={25}, part4={30}
4. **Part Exchange Phase**: S_{rst} receives all part1's, S_1 receives all part2's, S_{rmt} all part3's and S_2 all part4's.
   S_{rst}: group1 contains {1, 5} and {2, 4, 7}; group2 contains {3, 4, 9} and {2, 7, 9}

S_1: group1 consists of {10, 12} and {11, 12}; group2 contains {10, 14, 17} and {13, 15, 16}
S_{rmt}: group1 consists of {18, 22} and {24}; group2 consists of {19, 20} and {25}
S_2: group1 contains {28, 36} and {29, 35}; group2 contains {30}

5. **Parallel Merge**: all sites merge their group1 parts together and group2 parts together.
   S_{rst}: {1, 2, 4, 5, 7}, {2, 3, 4, 7, 9, 9}
   S_1: {10, 11, 12, 12}, {10, 13, 14, 15, 16, 17}
   S_{rmt}: {18, 22, 24}, {19, 20, 25}
   S_2: { 28, 29, 35, 36}, {30}
   Then, all sites perform merge join on the two resulting relations.
   S_{rst}: result relation is {2, 4, 7}.
   S_1: result relation is {10}.
   S_{rmt}: result relation is {}.
   S_2: result relation is {}.
6. **Result Collection**: S_1, S_{rmt}, S_2 send the joining result to S_{rst}.

C.2. An Example for Load-Sharing Hash Join (Algorithm 5a)

The data subfiles are the same as in the case of the previous algorithm.

1. **Distribution**: S_{rst} and S_{rmt} partition their data files into two equal parts.
   S_{rst}: part1={1, 5, 10, 12, 18, 22, 28, 36}, part2={2, 4, 7, 11, 12, 24, 29, 35}
   S_{rmt}: part1={3, 4, 9, 10, 14, 17, 19, 20}, part2={2, 7, 9, 13, 15, 16, 25, 30}
   S_{rst} sends part2 to S_1 and S_{rmt} sends its part2 to S_2. After the distribution, each site consists of the following:
   S_{rst}: {1, 5, 10, 12, 18, 22, 28, 36}
   S_1: {2, 4, 7, 11, 12, 24, 29, 35}
   S_{rmt}: {3, 4, 9, 10, 14, 17, 19, 20}
   S_2: {2, 7, 9, 13, 15, 16, 25, 30}
2. **Partition**: each site partitions its data set into 4 parts by a hash function (assuming h(key) = key/4)
   S_{rst}: part1={1, 5}, part2={10, 18, 22}, part3={}, part4={12, 28, 36}.
   S_1: part1={29}, part2={2}, part3={7, 11, 35}, part4={4, 12, 24}.
   S_{rmt}: part1={9, 17}, part2={10, 14}, part3={3, 19}, part4={4, 20}.
   S_2: part1={9, 13, 25}, part2={2, 30}, part3={7, 15}, part4={16}.
3. **Subrelation Re-distribution**: part1 of each site is transferred to S_{rst}; part2's are sent to S_1; part3's are sent to S_{rmt} and part4's are transmitted to S_2. The parts from S_{rst} and S_1 are combined into one set and the parts from S_{rmt} and S_2 to another.
   S_{rst}: {1, 5, 29}, {9, 17, 9, 13, 25}
   S_1: {10, 18, 22, 2}, {10, 14, 2, 30}
   S_{rmt}: {7, 11, 35}, {3, 19, 7, 15}
   S_2: {12, 28, 36, 4, 12, 24}, {4, 20, 16}
4. **Hash Index Setup**: All sites set up a hash index for one of their two sets of data.
5. **Local Hash Join**: all sites do the nested loop join between their sets of data.
6. **Result Transmission**: S_1, S_{rmt}, S_2 send the joining reesult to S_{rst}.

References

[AHU 74] Aho,A., Hopcroft,J. & Ullman,J.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[BERG 86] Berglund,E.J.
An Introduction to the V-System.
IEEE MACRO , August, 1986.

[BERN 81] Bernstain,P., et al.
Query Processing in a System for Distributed Databases(SDD-1).
ACMTODS 6(4), December, 1981.

[BIT 86] Bitton,D.
The Effect of Large Main Memory on Database Systems.
In *Proceedings of SIGMOD, ACM.* , 1986.

[BLA 77] Blasgen,M., & Eswaran,K.P.
Storage and Access in Relational Databases.
IBM Syst. J. 16(4), 1977.

[CAR 85] Carey,M.J. & Lu,H.
Some Experimental Results on Distributed Join Algorithms in a Local Network.
Technical Report 587, Comp Sc Dept, U. of Wisconsin-Madison, March, 1985.

[CEP 84] Ceri, S. and Pelagatti, G.
Distributed Databases: Principles and Systems.
McGraw-Hill, 1984.

[CHER 84] Cheriton,D.
The V Kernal: A Software Base for Distributed Systems.
IEEE Software , April, 1984.

[COM 79] Comer, D.
The Ubiquitous B-Tree.
Computing Surveys 11(2), June, 1979.

[DEW 84] Dewitt,D., Katz,R., Olken,F., Shapiro,L., Stonebraker,M., & Wood,D.
Implementation Techniques for Main Memory Database Systems.
In *Proceedings of SIGMOD, ACM.* , New York, 1984.

[ELH 84] Elhard,K. & Bayer,R.
A Database Cache for High Performance and Fast Restart in Database Systems.
ACMTODS 9(4), December, 1984.

[HHKOS 83] He, Xin-Gui, et al.
The Implementation of a Multi-backend Database Systems: Part 2.
In *Advansed Database Machine Architecture*. Prentice-Hall, New Jersey, 1983.

[KNU 68] Knuth,D.
The Art of Computer Programming: Fundamental Algorithms.
Addison-Wesley, 1968.

[KNU 73] Knuth,D.
The Art of Computer Programming: Sorting and Searching.
Addison-Wesley, 1973.

[LANT 85] Lantz,K.A., Nowicki,W.I. & Theimer,M.M.
An Empirical Study of Distributed Application Performance.
IEEE Trans on Software Engineering SE-11(10), October, 1985.

[LEH 85] Lehman,T.J. & Carey,M.J.
A Study of Index Structures for Main Memory Database Management Systems.
Technical Report 605, Comp Sc Dept, U. of Wisconsin-Madison, July, 1985.

[LEH 86] Lehman,T.J. & Carey,M.J.
Query Processing in Main Memory Database Management Systems.
In *Proceedings of SIGMOD, ACM.* , 1986.

[LU 85] Lu,H.
Distributed Query Processing with Load Balancing in Local Area Network.
Technical Report 624, Comp Sc Dept, U. of Wisconsin-Madison, December, 1985.

[LUK 87] Luk,W.S. & Ling,F.
An Analytic/Empirical Study of Distributed Sorting.
Manuscript under preparation , 1987.

[MEB 76] Metcalfe. R.M. & Boggs, D.R.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of ACM 19(7), July, 1976.

[PAGP 85] Page Jr.,T.W. & Popek, G.J.
Distributed Data Management in Local Area Networks.
In *Proc 3rd ACM Symp. on Princ. of Database Systems.* ACM-SIGACT-SIGMOD, March, 1985.

[PARK 86] Park,A.
Massive Memory Means Massive Performance.
Technical Report 036-86, Dept of CS, Princeton Univ, May, 1986.

[SAL 86] Salem,K. & Garcia-Molina,H.
Crash Recovery Mechanisms for Main Storage Database Systems.
Technical Report 034-86, Dept of CS, Princeton Univ., April, 1986.

[SHAP 86] Shapiro,L.D.
Join Processing in Database Systems with Large Main Memory.
ACMTODS 1(3):239-264, Sept, 1986.

[TAN 85] Tanenbaum,A. & Renesse,R.V.
Distributed Operating Systems.
Computing Surveys 17(4), December, 1985.