Parallel Query Processing on a Cluster-based Database System

By

Kenji Imasaki

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

September 30, 2004

Your file Votre référence
ISBN: 0-612-97826-5
Our file Notre référence
ISBN: 0-612-97826-5

Acknowledgment

I would like to acknowledge the following people for their contributions:

Prof. Sivarama P. Dandamudi	For being my adviser and never giving up on me
Ms. Hong Nguyen	For helping me with part of the implementation
Mr. Peter Taillon	For giving me support and encouragement and most importantly for being my best friend
Proof Readers	For putting up with my English
Mr. Fred Potts	For correcting my initial draft
Mr. Sylvain Pitre	For checking Chapters 1 and 2
Dr. Jason Morrison	For checking Chapters 3 and 5
Prof. Irwin Reichstein	For checking Chapter 4
Prof. Pat Morin	For checking Chapter 6
Mr. Steve Gruneau	For correcting my final draft
Technical Staff	For giving technical support
Mr. Andrew Miles	For giving me the required computer privileges
Mr. Gerardo Reynaga	For solving our printer problems
Administrative Staff	For making my stay pleasant
Ms. Linda Peiffer	For being there for me all the time
Mr. Shunsuke Morinishi	For being my best friend and giving me a lot of support during his life
Ms. Zheyin (Grace) Li	For being my girlfriend and providing invaluable assistance and moral support
My brother and sister-in-law	For supporting me 100% no matter what happened
My parents	For giving me the courage to follow my dream

I would like to say thank you all from the bottom of my heart.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the availability of Giga-hertz processors, Giga-byte memory, and Giga-bit bandwidth communication networks, huge parallel processing power from parallel computers can be used for various types of scientific computing. Good examples of such dedicated parallel computers are the SUN SunFire and IBM SP2. However, this power does not come for free; generally, parallel systems are very expensive. Also, with the fast pace of technological advances, constituent components of the parallel computers quickly become obsolete. As an alternative to such parallel systems, *cluster computing* has been introduced. Database query processing can also benefit from parallel execution on such cluster systems.

This thesis focuses on query processing algorithms on cluster systems. This chapter describes the motivation and the need for parallel query processing on a cluster. First, we present the background and motivation. Then, we state the problem that this thesis deals with. Finally, the contribution and the organization of the thesis are presented.

1.1 Background and Motivation

Cluster computing systems are generating enormous interest within the research community. In a cluster, which can be found in almost any computer network, workstations or workstation-class PCs are connected by LAN. These clusters offer large amounts of aggregate memory and computational power [3, 7]. The system can keep up with the technological advances in processors, memory, and networking, as the system uses off-the-shelf commodity components.

This also implies that a cluster tends to be heterogeneous in nature (different numbers of processors when there is Symmetric Multi-Processor (SMP) nodes in the cluster, different types of processors, different processor speeds, and different memory sizes). Furthermore, there are interactive users logging in and logging off a node from time to time. Thus, the performance of parallel programs run on a cluster is affected by their activity; the load and the available memory on each node fluctuate dynamically.

As a result, in the worst case, the node with the heaviest background load and the largest job determines the execution time for cluster parallel processing. To avoid this scenario, it is important to adopt a good load balancing/sharing[1] algorithm. One survey found that up to 80% of nodes are idle depending on the time of day [64]. Fault tolerant mechanisms are also required, since processing nodes (PNs) of clusters are more vulnerable to failures than parallel computers. For example, users can turn off nodes independently in clusters.

In the field of database systems, several parallel database system architectures have been proposed [20] to satisfy the growing demand for high transaction processing rates. Among these, shared-nothing architectures[2] are attractive from the standpoint of scalability and reliability [80]. In shared-nothing systems, each processor has its own private disk and private memory, and the processors communicate by a message passing mechanism. A Parallel Database Management System (PDBMS) with shared-nothing architecture is called a shared-nothing PDBMS. Generally, parallel execution of a query is beneficial for the following reasons:

- Query processing can be improved by exploiting intra-operator (single-join), inter-operator (multiple-join), and inter-query (multiple-query) parallelism [20].

- The large amount of aggregate memory can be exploited for parallel query processing. The size of today's databases reaches the tera-byte order. In that

[1]In this thesis, the term "load balancing" is used for data distribution among processing nodes at the initial stage and "load sharing" is used for sharing work during the processing stage after the initial data distribution.

[2]This is also called distributed memory architecture in the literature.

case, it is important to reduce the number of I/O operations by increasing the size of main memory because of the access speed gap between the main memory and the disk.

- I/O parallelism can be achieved in the case of shared-nothing architecture in which each node has a local disk.

With the advent of cluster computing environments as described above, parallel query processing on a cluster system has been proposed as an alternative to parallel database systems. In general, there are two approaches to implementing a PDBMS on a cluster system. One is the same as a traditional PDBMS: the data are de-clustered and a query is executed in parallel. Most of the recent commercial PDBMSs (IBM DB2 Universal Database [5, 9, 36, 37, 38, 53, 85], Oracle 9i [70, 72] and Grid-enabled 10g including Real Application Cluster [74], Compaq NonStop SQL/MP [16, 17], Microsoft SQL Server [25, 62], and MySQL Cluster [66, 67]) use this approach. The other approach is to use an existing dedicated DBMS and PNs in clusters for parallel processing to take the query load off the DBMS [10, 18, 19, 22, 23]. We call this system a cluster-based PDBMS (*cPDBMS*) to distinguish the two approaches. This thesis focuses on query processing in a cPDBMS.

The advantages of a cPDBMS over a PDBMS are:

- It is less expensive than a new PDBMS.
- It matches the flexibility of cluster architecture in which the memory and CPU availability of each PN changes from time to time.
- It is relatively easy to recover in the case of node failure since the original data are preserved in a DBMS.

Architecturally, a cPDBMS is close to a shared-nothing PDBMS. There are, however, some significant differences between the two systems. As described previously, one major difference is that the relations are not de-clustered as in the shared-nothing PDBMS. This is because PNs in a cluster might have individual owners, who can control the behavior of their PNs, which are not under the control of a central administrator (e.g. a user can turn off his/her machine). This lack of central control of the system makes it undesirable to distribute relations across the nodes as done in a shared-nothing PDBMS. The database is kept under the control of a traditional database system, which provides security and maintains the integrity of data. However, we can use the additional processor cycles and the memory available on these PNs for query processing on a temporal basis [18]. In particular, we restrict our attention to query processing, which requires only read-only access to data.

Parallel query processing in a cPDBMS has been studied in [11, 26, 40, 41, 47, 81]. In such a cPDBMS, every time a query is executed, the PNs involved must receive the required relations before query processing can proceed. This overhead is not present in the shared-nothing PDBMS, where the required data usually reside at the PN. Thus, there is overhead in transferring data from the DBMS to PNs. The overhead, however, can be amortized by the join execution time and local disk access, which is necessary for join processing when the size of the main memory is not enough for relations [18].

Another interesting research area is *data integration* systems. A data integration system has to process queries over databases spread around the world. It is called data integration since the data to be processed are spread around the world and loosely integrated. These studies on data integration focus on sequential query processing algorithms that can cope with the dynamic changes of the Internet. Several papers [2, 6, 44] describe such problems associated with query processing on the Internet (data may come from remote sources), comparing it to traditional query processing as follows:

- Absence of data statistics: Since the data are from autonomous external sources, the system has relatively few and often unreliable data statistics.
- Runtime selectivity fluctuation: These fluctuations commonly arise due to correlations between predicates and the order of tuple delivery.
- Unpredictable data arrival behavior from remote databases: initial arrival delay and arrival rate fluctuations exist.
- Redundancy among sources: The query processor needs to be able to efficiently collect related data from multiple sources, minimize the access to redundant sources, and respond flexibly when some sources are unavailable.

This thesis deals with the runtime selectivity fluctuations and unpredictable data arrival behaviour from the remote databases (henceforth, it is called *the Internet transfer delay*).

In processing a query, choosing an efficient parallel query processing algorithm is important. Query processing can be improved by exploiting intra-operator (single-join), inter-operator (multiple-join), and inter-query (multiple query) parallelism [20].

Among these three types of parallelism, the single-join operation has attracted a lot of attention, since it is the most expensive operation in query processing [14, 25, 32]. Three basic techniques are used for join processing: nested-loop, sort-merge and hashing [63]. Several studies have proposed parallel versions of nested-loop, sort-merge, and hash join [1, 20, 56]. Of the three, nested-loop and hashing are commonly used for join processing. Schneider et al. [75] claim that hash join algorithms are clearly superior. However, hash join-based algorithms suffer from various kinds of skew [60, 87]. Thus, the choice of load balancing/sharing algorithms becomes important since the slowest PN which has the heaviest data skew dictates the performance of the overall system.

Many researchers have proposed load sharing/balancing algorithms for the hash join algorithm [21, 27, 33, 40, 41, 52, 75, 83]. These load balancing/sharing algorithms for PDBMSs on parallel computers do not work for cPDBMS for the following reasons:

- These algorithms only deal with PNs with pre-partitioned data. However, PNs in clusters are dynamically determined and usually do not have any of the data needed for join processing. The data should be sent from DBMSs to a PN prior to the join processing. This phase is not considered in these algorithms. This situation can be seen as the extreme case of tuple placement skew [60, 87]. In tuple placement skew, some PNs read more tuples while others wait for them to finish reading the whole relation.

- These algorithms do not consider the effect of non-query background load. Even with the adaptive approach, it is difficult to know how much work a PN should transfer to another. Besides, it is not clear whether the transfer is effective or not in the case of clusters in which the load on each PN changes very frequently. In addition, no algorithm considers the effect of the combination of background load and data skew.

- Correction of a wrong prediction by either dynamic or incremental algorithms about bucket distribution has not been studied fully.
- Input data arrival delay and data transfer rate fluctuation are not considered. This is very important, especially in the case of data integration.

Therefore, a new load balancing/sharing algorithm is needed to improve the performance of join processing on clusters. This thesis will propose new load balancing/sharing algorithms for cPDBMSs.

The next parallelism that this thesis deals with is multiple-join algorithms for a cPDBMS. For a PDBMS, there are two kinds of multiple join algorithms: non-pipelined and pipelined algorithms. In non-pipelined algorithms, each join is processed one by one. In pipelined algorithms, all joins are executed at the same time and the results of one join are passed to the next join in a pipelined manner. Generally, non-pipeline algorithms are good for bushy query trees and pipelined algorithms are good for right-deep query trees [47]. A bushy tree is a tree in which some input relations are inner relations and others are outer relations. A right-deep tree is a tree in which all input relations are inner relations. In the context of operations implemented using hash-based algorithms, there is a convention that the left child of an internal vertex denotes the inner relation of the join, i.e., the one used to build the hash table. Correspondingly, the right child denotes the outer relation whose tuples are used to probe the hash table.

In Non-Pipelined Hash Join (NPHJ) algorithms, each join is processed one by one; the result of each join is saved to the disk every time the join processing is finished. In pipelined hash join (PHJ) algorithms, there are two kinds of algorithms: Simple Pipelined Hash Join (or Non-Symmetric PHJ (NSPHJ)) [12, 14, 55] and Symmetric PHJ (SPHJ) [89]. In NSPHJ, there are two phases: table building and table probing. In this algorithm, a building relation (the left child in the query execution tree) and a probing relation (right child) are distinguished. On the other hand, in SPHJ, there is no such distinction. A hash table is built for each relation. Every time a tuple from one relation arrives at a PN, it is probed against a hash table for the other relation and inserted into its hash table. The disadvantage of both pipelined algorithms is that either one (NSPHJ) or both of the relations (SPHJ) should be in the main memory.

However, there is no research on the performance of pipelined hash join algorithms (NSPHJ and SPHJ) for a cPDBMS. Therefore, this thesis focuses on a detailed study of their performance under various conditions. The effect of the Internet transfer delay of input relations is also considered.

1.2 Thesis Focus and Problem Description

This thesis focuses on the design, implementation, and evaluation of query processing algorithms (single-join and multiple-join) that can optimize the performance of cluster-based PDBMSs (cPDBMSs).

The goal is to minimize response time under the following conditions:

- The size of the resources (the number of CPUs, CPU speed, memory, disk speed, etc.) at each PN is statically different.
- The size of the resources at each node changes dynamically due to local interactive users.
- Data are stored in one of the PNs and other PNs do not have the data prior to query processing.
- Input relations have various types of data skew as described in Section 2.3.1.1.
- Any PN in the cluster can receive a query.
- Dynamic features of input relations, caused by network traffic from data sources to PNs as a result of data integration, exist, such as:
  - Unpredictable data arrival characteristics: initial delay and burst data transfer rate changes.
  - Runtime selectivity fluctuation: the selectivity varies according to the arrival order of the tuples.

1.3 Thesis Contribution

This thesis investigates the query processing algorithms for a cluster-based Parallel Database Management System (cPDBMS). The contributions of the thesis are summarized as follows:

1. We re-examine a parallel nested loop join algorithm for a cPDBMS. We compare its load sharing/balancing algorithms (static and dynamic). We identify the case when each algorithm is better with several kinds of background load.

2. We present two versions of adaptive algorithms for a cPDBMS: a non-symmetric join version (ChunkHJ) and a symmetric join version (SCHJ). ChunkHJ applies when one of the joined relations can be used for the main relation a *priori*. The main relation is used for hash table building in hash join or to predict the skew model. The main relation is chosen if it is smaller than the other relation or if it is known that one relation arrives a lot faster than the other. If this information is absent, SCHJ should be used instead.

3. ChunkHJ is evaluated with two other algorithms (one is the hash-based nested loop join and an adaptive version of GRACE [52]) under the conditions of several kinds of non-query background load and data skew. A heterogeneous cluster consisting of processing nodes with different speeds is used for this evaluation. We identify the cases in which ChunkHJ is superior.

4. SCHJ is evaluated with several traditional algorithms such as VP-RR [21] and dynamic sampling [21] under the same conditions as the experiments with ChunkHJ. A new cluster, which consists of SMP nodes, is used for this evaluation. We identify the cases in which SCHJ is better.

5. SCHJ is evaluated with several incremental hashing algorithms with the Internet transfer delay and transfer rate fluctuation under the same conditions as the experiments with ChunkHJ. We identify the cases in which SCHJ is better.

6. NSPHJ and SPHJ are evaluated with several types of non-query background load. In addition, the Internet transfer delay and transfer rate fluctuation have

been added to the conditions of the evaluation. We identify the cases in which SPHJ is superior.

1.4 Thesis Organization

This thesis is organized as follows. The next chapter presents related work on single-join and multiple-join algorithms, as well as PDBMSs, cPDBMSs and data integration systems. We analyze several major algorithms and compare them with our algorithms. Then, single-join algorithms and their load balancing/sharing algorithms without consideration of arrival delay of input relations, are presented and evaluated on a cluster. Chapter 3 covers non-symmetric version of the algorithm. Chapter 4 studies symmetric versions of the algorithm. This chapter also considers the effect of Internet transfer delay. Chapter 5 investigates multiple-join algorithms. It focuses on non-symmetric pipeline hash join and symmetric pipeline hash join algorithms. We compare these two algorithms on a cPDBMS with and without Internet transfer delay. Chapter 6 gives conclusions and points to future work.

Chapter 2

Related Work

This chapter presents related work. We start by introducing parallel database management systems (PDBMSs) and data integration systems on various platforms. Then, the general execution steps for load balancing/sharing algorithms are presented. Finally single-join and multiple-join algorithms and their load balancing/sharing algorithms are described.

2.1 Parallel Database Management Systems

This section introduces PDBMSs and their variations. One of the variations is the cluster-based PDBMS (cPDBMS) and the other is the data integration system.

2.1.1 The Architecture of Parallel Database Systems

Parallel database systems are database systems that execute queries in parallel to achieve the following goals:

- the parallelization of operations involving large amounts of data (involving I/O operations, called *I/O parallelism*).
- simultaneous service to a growing number of users.
- a high level of fault tolerance by using RAID techniques.

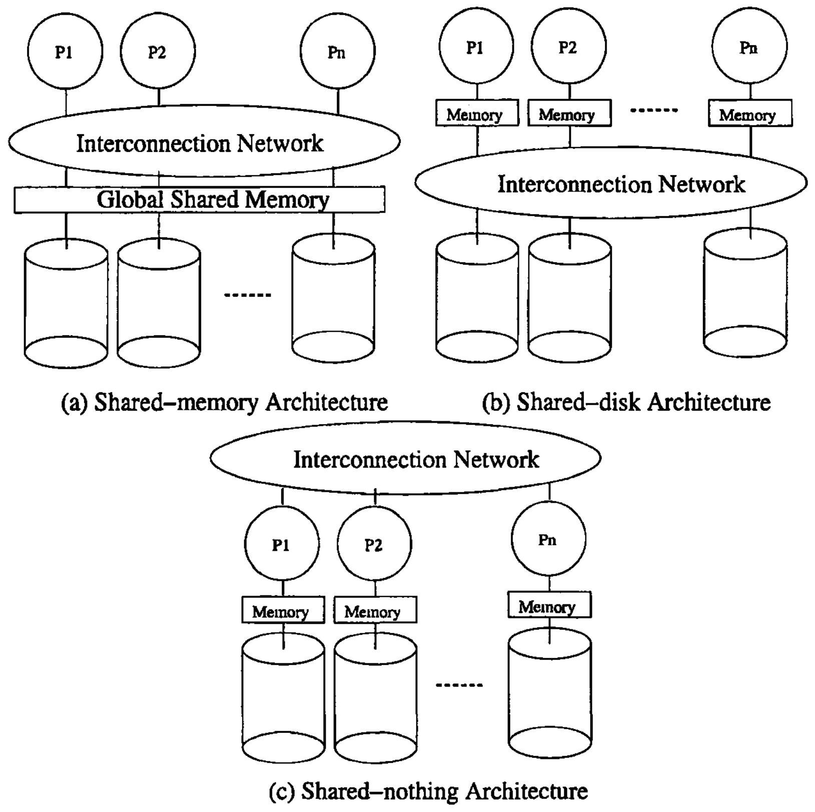

Figure 2.1: PDBMS architectures.

The typical architectures of parallel database systems as classified in [20] are shown in Figure 2.1. In this figure, $P_1..P_n$ denotes a processor. In the shared-memory architecture, all processors have direct access to the global memory and all disks (Figure 2.1(a)).[1] The shared-disk architecture allows all processors to have non-shared, private memory but direct access to all disks (Figure 2.1(b)). Therefore, data do not have to be statically partitioned and mapped [82]. In the shared-nothing architecture, all processors have non-shared, private memory and disks (Figure 2.1(c)). According to [20], the shared-nothing is the most promising among these in architectures terms of scalability.

[1]In the literature, this is sometimes called a SMP(Symmetric MultiProcessor).

In the shared-nothing architecture, the data is partitioned either horizontally or vertically. In *vertical partitioning*, the table is split into multiple smaller tables with the same number of rows, but fewer columns. In *horizontal partitioning*, a table is split into multiple smaller tables containing the same number of columns, but fewer rows. It is also possible to combine both types of partitioning.

Horizontal partitioning is suitable for parallel processing in exploiting intra-operator (single-join) parallelism. The major partitioning schemes are shown in Figure 2.2. All schemes distribute the tuples according to the rule on one of attribute values.

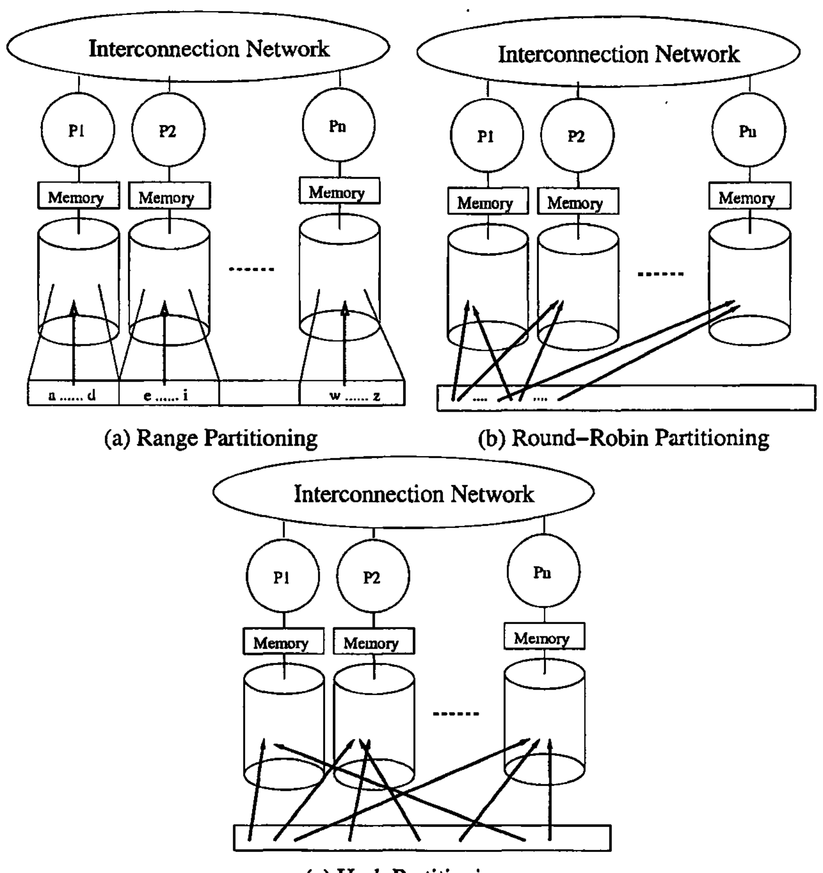

Figure 2.2: Examples of horizontal partitioning schemes in the shared-nothing architecture.

The cPDBMSs that are discussed in this thesis are based on the shared-nothing approach, since the cluster system has a shared-nothing architecture by nature. The difference is that PNs in cPDBMSs are shared by other interactive users. In addition, data (relations) are pre-partitioned in PDBMSs as in Figure 2.2, but not in a cPDBMS.

2.1.2 Parallel Query Processing

Dewitt and Gray summarized the basic techniques for such PDBMSs [20]. They claimed that the relational model is suitable for parallel execution since it consists of uniform operations applied to uniform streams of data. High-level, declarative query languages are also suitable for parallel processing.

In the paper, they described two forms of parallelism. One form is the pipelined parallelism. It streams the output of one operator into the input of another operator and executes them in parallel. The other form is partitioned parallelism. It partitions the input of one operator into several processors, each of which work on a part of data in parallel. They also described several techniques to parallelize relational operations (scan, merge, and join).

There are three types of parallelism mentioned by Lu et al. [56]. The first type is *inter-query parallelism*, in which queries are executed in parallel. The second type is *inter-operator parallelism*, in which operators are executed in parallel. The third type is *intra-operator parallelism*, which is the same as partitioned parallelism.

In this thesis, intra-operator (single-join) parallelism is discussed in Chapters 3 and 4. Inter-operator (multiple-join) parallelism is discussed in Chapter 5.

2.1.3 Research Prototype PDBMSs

In this section, several research prototypes are described for cluster-based and data integration systems. As mentioned in Section 1.1, a cluster-based PDBMS uses an existing dedicated DBMS and PNs in a cluster system for parallel processing to take the query load off the DBMS. Data coherency and consistency are guaranteed by the existing DBMS. The study of data integration systems is related to this thesis, since they also deal with the Internet transfer delay, which is discussed in Chapters 4 and

5.

2.1.3.1 Cluster-based PDBMSs

Recently, cluster-based PDBMSs (cPDBMSs) have been studied to take advantage of the beneficial features of clusters mentioned in Section 1.1.

Enkidu [8, 11, 22, 23] used parallel relational query evaluators, which work in a coupled mode with a sequential DBMS. It uses macro-pipelining for multiple join query processing in multi-user mode. Each join processing is multi-threaded with its priority. The results are shown in the case of Enkidu with and without the Oracle database. However, they did not use intra-operator parallelism and did not consider the effect of background load induced by interactive users.

Dandamudi et al. proposed query processing on centralized [93], distributed [78], and hierarchical [91] architectures. They also analyzed the cost and benefits of parallel processing by clusters [18, 19]. Their idea is to load the data from the database at the beginning of query processing to obtain flexibility. They claimed that the database load time at each query execution can be amortized by the join execution time and local disk access, which is necessary for join processing when the size of the main memory is not enough for relations. Using this idea, Haddad and Robinson [26] implemented a cPDBMS using POSTGRESS and six Pentium IIIs and obtained good speedup results of a TPC-D benchmark database. However, no background load was considered in this paper.

Tamura et al. [81] developed a DBMS on a PC cluster system consisting of 100 PCs connected through an ATM switch. On each PC node, a server program acts as a database kernel to process the queries in cooperation with other nodes. They used the dynamic GRACE and pipeline techniques to process queries. Benchmark results were given but neither background load nor multi-user mode were considered.

MIDAS (MunIch Parallel DAtabase System) [10] is a relational, parallel, shared-disk database system using the cluster method. MIDAS was implemented by Parallel Virtual Machine (PVM). The paper discussed MIDAS communication characteristics and PVM specific optimization aspects. However, no detailed performance results were given.

In summary, there are several cPDBMSs but there is no performance study examining non-query background load on each PN. Furthermore, the Internet transfer delay has not been studied.

2.1.3.2 Data Integration Systems

A data integration system is an automated method of querying across multiple heterogeneous databases in a uniform way [44, 45]. In this section, several research data integration prototypes are discussed.

Several papers [2, 6, 44] describe problems associated with query processing on the Internet (e.g. data may come from remote sources), compared to traditional query processing, as follows:

- Absence of data statistics: Since the data are from autonomous external sources, the system has relatively few and often unreliable data statistics.
- Runtime selectivity fluctuations: These fluctuations commonly arise due to correlations between predicates and the order of tuple delivery.
- Unpredictable data arrival behaviour from remote databases: Initial arrival delay and arrival rate fluctuations exist.
- Redundancy among sources: The query processor needs to be able to efficiently collect related data from multiple sources, minimize access to redundant sources, and respond flexibly when some sources are unavailable.

This thesis deals with the runtime selectivity fluctuations and unpredictable data arrival behaviour from the remote databases.

The systems introduced in this section use the algorithms described later in Section 2.4.2. The purpose of this section is to present a brief overview of current research prototypes of data integration systems. Our emphasis is on systems that use algorithms that handle the Internet transfer delay as summarized in Table 2.1.

Tukwila [44, 45] The Tukwila data integration system aims to scale up to the amounts of data on the Internet with large numbers of data sources. Tukwila handles

Name	Developer	Algorithm to cope with Internet delay and/or load sharing/balancing
Tukwila [44, 45]	University of Washington	double-pipelined hash join [44] re-optimize a query plan
Telegraph [28]	UC Berkeley	Eddies[6] and Flux [76]
Niagara [13]	University of Wisconsin	Rate-based query processing [86]

Table 2.1: Popular research data integration system prototypes.

Internet delay by incrementally re-optimizing a query plan and dynamically choosing materialization points by the optimizer according to their costs and potential benefits as greater knowledge about the data is gained. It uses adaptive double pipelined algorithm, which will be explained in Section 2.4.3, to hide latency. Tukwila itself is not a PDBMS.

Telegraph [28] The *Telegraph* aims to build a global-scale query engine that can execute complex queries over all the data available on-line. It faces challenges in wide-area systems, including the quickly shifting performance and availability of resources on the Internet. It also serves as a plug-and-play shared-nothing parallel query engine.

To balance the load and cope with load imbalance, an algorithm called Flux [76] is used. Flux is inserted between the producer and consumer stage to repartition operators while the pipeline is still executing. The authors addressed input data skew, processing and memory load changes, and heterogeneity in the PNs. They also noted that these factors are severe in parallel systems, as mentioned in Section 1.1.

Another algorithm called Eddy [6] is also used to cope with the Internet transfer delay. Eddy continuously reorders operators in a query plan as it runs. The authors defined the terms for the moments of symmetry during which pipelined joins can be easily reordered so that they can be used according to the arrival of relations. Eddy also reorders operators on a tuple-by-tuple basis. Each tuple can be routed to a different join in the query plan so that each join is busy all the time. The correctness of join execution is ensured by Eddy.

Niagara Project [13] The Niagara is a search engine for XML documents that can support much more powerful and precise searches than HTML documents. To cope with the Internet transfer delay, rate-based query processing [86] is introduced and tested on Niagara. In the paper, the cost model and calculation of the output rate for relational operations (selection, projection, and join) were described. The authors introduced two heuristics: local rate maximization and local time minimization. The Niagara is not a PDBMS.

2.1.4 Commercial PDBMSs

The major commercial PDBMSs are described in this subsection.

IBM DB2 [37] The DB2 UDB Enterprise-Extended Edition (EEE) can run on ccNUMA IBM NUMA-Q [35]. The NUMA-Q architecture combines the linear scalability of parallel processing and the manageability of a single image SMP platform. The benchmark result shows good scalability in 32, 48 and 64 nodes configurations [9, 36, 85].

IBM's DB2 universal database (UDB) for OS/390 and z/OS is implemented on Parallel Sysplex Cluster [39]. Load balancing on Parallel Sysplex is relatively easy since the entire Parallel Sysplex cluster can be seen as a single logical resource like an SMP server and any node can accept and execute a job. There is no need to partition and replicate data across multiple servers as mentioned in Section 2.1.

Oracle Parallel Server and Real Application Server [69, 70, 71, 72, 73, 74] The Oracle 8i Parallel Server (OPS) [69, 70, 71] fully exploits clustered systems for database applications by delivering high availability, cluster scalability, and single system manageability with cluster load balancing and cache fusion clustering architecture. However, OPS uses its shared disk system instead of a network link to communicate among PNs, thus it does not scale well.

Oracle 9i [72, 73, 74] further improves its performance and uses a technology called Real Application Clusters (RAC). It uses partitioned-parallelism and pipelined-parallelism. An adaptive algorithm is used to determine the degree of parallelism that takes the current system status into consideration. Unlike OPS, in RAC, a network is used for communications, although data is still accessed using a shared disk.

Oracle grid-enabled 10g parallelizes its queries [82] using a Parallel Execution (PX) engine and a Parallel Single Cursor (PSC) model. PX consists of a Parallel Execution Coordinator (PEC) and a set of Parallel Execution Servers (PES). The paper [82] describes the advantage of a shared-disk architecture in that the data does not have to be statically partitioned and mapped. Thus, PX is not limited by how users use partitioning to fragment database objects. Also, the paper explains how cluster-aware redistribution is done when the degree of parallelism dynamically changes. It uses dynamic sampling techniques for load balancing.

The HP NonStop SQL/MP [29] The HP NonStop SQL/MP is implemented on NonStop Parallel Servers (shared-nothing architecture), which are fully distributed and optimized for massive parallel systems. The NonStop SQL/MP uses horizontal partitioning and adaptive hash join algorithms. In the case of intermediate results, dynamic repartitioning using hash function is executed. The paper claims that SQL/MP is nearly linear with respect to speedup and scale-up.

Microsoft SQL Server [25, 62] The Microsoft SQL Server introduced parallel query processing capability in version 7.0. The report by Joe Chang [49] summarized its function and performance. Microsoft SQL Server parallelizes a query when its estimation cost is greater than the threshold value. Chang concluded that the unit of cost is time rather than CPU usage relative to a single processor from cost analysis and various experiments. SQL Server executes a query in parallel by using system threads. The degree of parallelism is dynamically determined. It uses horizontal partitioning.

MySQL Cluster [66, 67] MySQL AB released MySQL Cluster [66, 67] in May 2004. It is a main memory database and is implemented on a shared-nothing cluster. MySQL Cluster is an integration of the standard MySQL server with an in-memory clustered storage engine called NDB. Its administrator guide [68] further explains its functionalities. NDB Cluster automatically fragments the data using a hash function and replicates them using the number specified in configuration files. However, the details of the parallelization technique are not illustrated. MySQL Cluster provides a response time of 5-10 millisecond and throughput of 100,000 transactions per second on a typical low-end configuration of a four-node cluster, with two CPUs per node [66].

2.2 General Load Balancing/Sharing Algorithms

Lu and Tan [57] and Märten [60] described the following execution steps for load balancing/sharing algorithms in query processing:

1. Task generation
   (a) Task partition dimension (inter/intra transaction, inter/intra query, and inter/intra operator parallelism) is determined;
   (b) Task partition size (sub-operator, pipeline, several transactions) is determined;
   (c) The number of tasks is determined according to the following criteria:
      i. The number is equal to the number of processors;
      ii. The number is more than the number of processors. The system allocates them to the processors according to some criteria (e.g., the size of the available memory).
   (d) Relation decomposition (full fragmentation, fragmentation and replication, full replication) is determined.
2. Task allocation
   (a) Degree of parallelism is determined. The following factors are considered:
      - load matrices (cardinality of the buckets and their estimated execution time);
      - allocation strategy (static, dynamic, adaptive, demand-driven);
      - resource demand and performance of difference resources.

   (b) PNs are selected from an eligible PN list;
   (c) Task allocation to PNs is done with or without load balancing (e.g., best-fit strategy).

3. Task execution
   (a) The following statistics are collected and maintained:
      - bucket size and result size;
      - load measure (CPU, disk, memory, network).
   (b) Redistribution is done according to some criteria;
   (c) Local join algorithm is selected and executed.

In the algorithms introduced in this thesis, task allocation and task execution overlap. Also, relation decomposition is done at the bucket level (fragmentation and replication), as opposed to at the relation level. Furthermore, task allocation is done with load-balancing and redistribution is done in the task execution phase. Hash join is used for the local join unless otherwise noted.

2.3 Parallel Single-Join Algorithms

To get intra-operator parallelism on parallel computers, parallel single-join algorithms have been extensively investigated, since the join operation is the most expensive operation. Parallel versions of the nested-loop, merge-sort, and hash joins and their variations have been proposed [1, 56].

The purpose of this section is to present various load sharing/balancing algorithms for single-join algorithms, classify them, and elaborate the possibility of their application to join processing on cPDBMSs. First, the various classification criteria are presented and then the proposed load balancing/sharing techniques are described and classified according to the classification criteria.

2.3.1 A Classification

Major classifications for various load sharing/balancing algorithms are discussed in this subsection.

2.3.1.1 Data Skew Type (DST)

The following types of skew may occur while processing joins [87]: *intrinsic skew (IS)* and *partition skew (PS)*. IS is also called *attribute value skew (AVS)* and occurs when attribute values are not distributed uniformly. Furthermore, IS(AVS) does not change between load balancing/sharing algorithms. PS occurs when the workload is not balanced among the PNs. PS can be delineated further as follows:

- *Tuple Placement Skew (TPS)* occurs when the initial distribution of tuples varies among the PNs. This can be treated by static data allocation.
- *Selectivity Skew (SS)* occurs when the selectivity of selection predicates varies among the PNs.
- *Redistribution Skew (RS)* occurs when there is a mismatch between the distribution of the join key values in a relation and the distribution expected by the redistribution mechanism.
- *Join Product Skew (JPS)* occurs when the join selectivities at each PN differ.
- *Query Skew (QS)* occurs when there are predicates that select certain relations, attributes, or values more frequently than others.

2.3.1.2 Data Skew Model (DSM)

Zipf distribution [96] can be used to create skewed data: the domain of the join attribute is assumed to have D distinct values. The probability that the join attribute of a given tuple takes on the ith value in the domain is $p_i = c/(i^{1-Z})$, where $1 \leq i \leq D$, $c = 1/(\sum_{i=1}^{D} 1/(i^{1-Z}))$, where c is a normalization constant, and Z is the Zipf skew factor [96]. The disadvantage of a Zipf like function is that the size of the result relation cannot be controlled easily. Thus, the scalar skew model (*Scalar*) is proposed. In the scalar skew model, in an N tuple relation, in each attribute the constant "1" (or "0") appears in some fixed number of tuples, while the remaining tuples contain values uniformly distributed between two and N. *Normal* is the normal distribution with a certain standard deviation value. This thesis used Zipf and scalar skew since they are commonly used in most experiments, as shown in Table 2.2 on page 27.

2.3.1.3 Load Balancing/Sharing Methods (LBSMs)

The following classification of load sharing algorithms for hash join algorithms is mostly from [34, 56]. As we will show in Section 3.3.1, a hash join algorithm consists of two phases: a table building phase and a table probing phase. A hash table is built using one of the relations (usually the smaller of the input relations) and the tuples of the other relation are used to probe the hash table. Parallelization of a hash join is also straightforward. Both relations are divided into disjoint sets (hash buckets) and scattered. The tuples belonging to the buckets with the same hash value can be joined independently in parallel. Thus, a partition consists of several buckets and is assigned to a node (called a *partition scheme*). The following classification is based on the decision of the partition scheme.

- The *Static (S)* method determines the partition scheme prior to the bucket creation of input relations. This is the simplest algorithm but it does not work well in the case of data skew.
- The *Dynamic (predictive)(D)* method determines the partition scheme after (or during) the bucket creation of input relations. It might work well with data skew.
- The *Incremental (I)* method changes the partition scheme depending on the bucket distribution of input relations while reading the relations. It might work well with dynamic change in the arrival speed of relations.
- The *Adaptive (A)* method changes the partition scheme depending on the bucket distribution of input relations and the actual processing speed of the processing nodes. Even though it has more overhead in communication involving changes to the partition scheme, it might work well in most cases.

2.3.1.4 Data Transfer Methods (DTMs)

There are two ways to transfer the work from a DBMS to PNs. In the *centralized (C)* method, there is a coordinator process for the whole system that determines the work to be assigned to nodes. In the *distributed (D)* method, there is no coordinator process and each processor determines the work to be processed.

For example, let us assume that PN_1 is idle and is looking for work. It is done by looking into the shared memory area where each PN writes its status in. In the case of the centralized method, PN_1 asks the coordinator process for the next job and the coordinator process determines the work for PN_1 and sends the next job (or ask another PN to send it) to PN_1.

In the case of the distributed method, each PN writes its status into the shared memory. When PN_1 looks for an overloaded PN (for example, PN_2), it does not consult the coordinator process. Instead, it checks the shared memory to find an overloaded PN. Then, it asks for the next job from PN_2. Alternatively, PN_1 simply broadcasts its job request to all PNs and waits for a reply from one of the PNs (for example, PN_3). The first reply is granted and PN_3 sends the job to PN_1.

2.3.2 Load Balancing/Sharing Algorithms

In this subsection, a summary of the load balancing/sharing algorithms for single-join processing is presented. A number of join algorithms are surveyed in [63].

The *GRACE Hash Join algorithm (GRACE)* [52] consists of a split phase and a join phase. In the split phase, a hash function is applied to each tuple of input relations to form buckets. After the split phase, small buckets are combined to form optimal size join buckets (called *bucket tuning*). Then, hash join is executed on these buckets.

The *Hybrid Hash Join algorithm (Hybrid)* [75] is the same as GRACE except that some buckets in main memory are joined immediately in the split phase using extra main memory. Thus, the split and join phases overlap. The work reported in [75] compared Simple [75], GRACE [52], Hybrid hash joins, nested-loop and sort-merge join with analysis and experiments on the database machine GAMMA. Their conclusion was that the Hybrid hash join algorithm is the best.

Dewitt et al. [21] proposed the following algorithms to deal with redistribution skew and join product skew: *Range Partitioning (RP)*, *Subset-Replicate (SR)*, *Virtual Processor Range Partitioning (VP-RP)*, *Round-Robin (VP-RR)* and *Processor Scheduling (VP-PS)*. They used sampling techniques and obtained bucket distribution of the input relations. They concluded that the following steps are effective: take a pilot sample of both relations involved in the join and inspect the resulting set of samples to determine which relation is more highly skewed by counting the number of repeated values. If neither relation appears skewed, revert to simple hybrid hash join. If at least one of the relations appears to be skewed, use VP-RR.

Wolf et al. [90] proposed *HJ**, which uses hierarchical hashing: a coarse hash function for storing and a fine hash function for collecting statistics. For very large bucket pairs, they applied a fragment/replicate scheme which divides the inner bucket among several processors and broadcasts the outer bucket to all these nodes.

Hua et al. [33] proposed the following partition schemes. In the *Tuple Interleaving Parallel Hash Join algorithm (TIJ)*, the tuples in each bucket are spread among processors in an interleaved way in the split phase. In *Adaptive Load Balancing Parallel Hash Join (ABJ)*, the partition scheme is decided based on bin-packing on the size of the buckets in the partition phase. In ABJ+, the tuple transfer is delayed until the partition tuning phase. Their conclusion was that the ABJ algorithm should be used in general since it is the most robust ABJ+ should be used if the degree of skew is significant. TIJ should be used when the initial partition skew is serious or the communication and I/O bandwidths are sufficiently large.

Adaptive Parallel Hash Join algorithm (APHJ) [51] uses random scan to read input relations to obtain bucket distribution in the main memory databases and detect skew on the fly. The system monitors the frequencies of values of the join attribute and applies a threshold function to them. If this threshold function is reached by the join attribute, the buckets corresponding to it are fragmented among an appropriate number of nodes. Fragmentation requires some replication of input tuples. To get a good data sample, RIO (Randomization of the Input Order) is used. The experiments were done on a shared-memory parallel computer and no comparison with other algorithms. Their conclusion was that the balancing algorithm they proposed is effective.

Dynamic Balancing Hash Join algorithm (DBJ) [94] checks the distribution at each checkpoint (after reading 10% of the whole relation, after reading 20%, and so on). When the workloads are significantly different at a checkpoint, redistribution is possible and the node that has the largest workload chooses the buckets whose size is closest to half of the difference between the largest and smallest workloads, and transfers these buckets to the node which has the smallest workload. The workload is based on the the size of the buckets from both relations.

Hua et al. [34] compared the performance of the following load balancing algorithms on a shared-nothing parallel computer: (1) no load balancing (similar to GRACE [52]), (2) conventional bin-packing (similar to ADJ [33]), (3) sampling (similar to the sampling in [21]), and (4) incremental (similar to DBJ [94]). They concluded that the sampling method is the best.

Lu and Tan [57] proposed a *Dynamic Load-Balanced Join (DBJ)* algorithm. In DBJ, when a node finishes processing all the allocated tasks, it requests load from a node (called *donor node*) that has not completed the execution of its allocated tasks . It uses the hash-based nested-loop join for the local join algorithm. The work in [57] described how to determine the donor node and how much work should be transfered.

Harada and Kitsuregawa [27] proposed an algorithm that detects and handles join product skew at runtime. It monitors the join processing speed and compares the speed with the speed predicted statically. If the difference is significant, *result redistribution* or *processing task migration* is invoked. Processing task migration is either *hash table migration* or *probe migration* depending on the case. They used two threshold values to detect whether the node is overloaded. From the simulation results and its implementation on GAMMA, they concluded that this dynamic approach is effective when compared to VP-RR. However, they only compared it with VR-RR without any background load.

Distributed Hash Join (DHJ) [83] is an algorithm for the NUMA shared-memory computer. Thus, it is not directly applicable to clusters. However, it uses a distributed scheme (no scalability problem) and tries to avoid any form of locking. It does not do pre-scanning or sampling, which requires synchronization, and does not scale well. During join processing, an under-loaded node (helping node) decides which other nodes to help by checking the status of other nodes and selecting overloaded nodes using a shared-memory mechanism.

Märtens proposed *On-Demand Scheduling (ODS)* [59]. In ODS, join buckets are assigned to the PNs according to the actual progression of work. Only when a PN has finished processing a bucket is it assigned the next one. This study assumed the shared-nothing architecture. The performance of ODS was compared with predictive scheduling (PS), which uses the idea of the longest processing time (LPT) in single and multiple user modes. The conclusion was that ODS is 25% better than PS when the skew is high.

2.3.3 Discussion

A comparison of the algorithms described in the previous section is summarized in Table 2.2. This table includes SS_EQ, SS_PR, DS, ChunkNJ, GRACE+, ChunkHJ, SCHJ, GIHM, and JIHM which are explained in this thesis. "Y" means that the algorithm takes the factor into consideration. "M" means that the algorithm may be modified to deal with this factor. The column "BG" indicates background load disturbance.

A good load balancing/sharing algorithm for a cPDBMS should have the following characteristics:

- It should be simple and require no pre-scan and no knowledge about input relations.
- It should work well under one of the cPDBMS characteristics, which is that one of the PNs acts as a DBMS and has all the input relations.
- It should have the ability to change the partition scheme dynamically and adaptively.
- It should be robust relative to the arrival delay and/or transfer rate fluctuations of input relations.

From Table 2.2 and the above conditions, Kitsuregawa's approach and ODS seem adequate. However, Kitsuregawa's algorithm is too complicated and might not be effective on a cluster. ODS is similar to GRACE+, which is explained in Section 3.3.2.1.

Thus, our recommendation for the load balancing/sharing algorithm is that it should be a combination of static and dynamic load balancing algorithms. Also, it should be an on-demand load sharing scheme.

Algorithm	DST			DSM	LBSM	DTM	BG	machines
	TPS	RS	JPS					
GRACE [52]		Y		Normal	D	C		GAMMA
Hybrid [75]		Y		Normal	D	C		GAMMA
RP,SR [21]		Y		Scalar	D	C		GAMMA
VP-RR,VP-PS [21]		Y		Scalar	D	C		GAMMA
HJ* [90]		Y		Zipf	D	C		
TIJ,ABJ [33]		Y		Zipf	D	C		
APHJ [51]		Y		Zipf	I	C		IBM RP3
DBJ by Zhao [94]		Y		Normal	I	C		ADEPT
DBJ by Lu [57]		Y		Zipf	A	C	M	
Kitsuregawa [27]		Y	Y	Scalar	A	C	M	GAMMA
DHJ [83]		Y	Y	Zipf	A	D	M	TC2000
ODS [59]		Y	Y		A	C	Y	
SS_EQ,SS_PR					S	C		cluster
DS	Y	Y	Y		A	C	Y	cluster
ChunkNJ	Y	Y	Y	Scalar	A	C	Y	cluster
GRACE+	Y	Y	Y	Scalar	A	C	Y	cluster
ChunkHJ	Y	Y	Y	Scalar	A	C	Y	cluster
SCHJ	Y	Y	Y	Scalar,Zipf	A	C	Y	cluster
GIHM	Y	Y		Scalar,Zipf	I	C		cluster
JIHM	Y	Y		Scalar,Zipf	I	C		cluster

Table 2.2: Comparison of load balancing/sharing algorithms for the single-join operation. The algorithms that are not referenced in column 1 are proposed in this thesis.

2.4 Parallel Multiple-Join Algorithms

The focus of this section is multiple-join algorithms. First, several non-pipelined hash join algorithms are discussed. Then, two kinds of pipelined hash join algorithms (Non-symmetric and Symmetric) are described. Finally, pipelined hash join algorithms for data integration systems are presented. A classification of these algorithms is given in Figure 2.3.

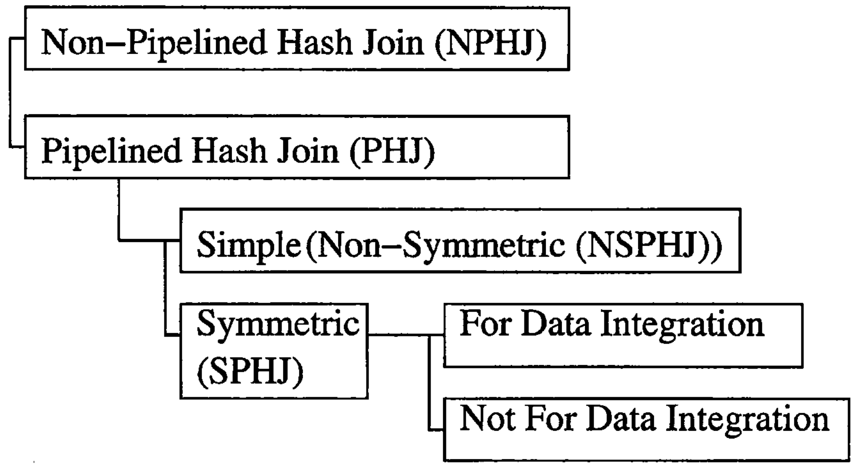
Non–Pipelined Hash Join (NPHJ)
Pipelined Hash Join (PHJ)
Simple(Non–Symmetric (NSPHJ))
Symmetric (SPHJ)
For Data Integration
Not For Data Integration

Figure 2.3: A classification of multiple-join algorithms.

2.4.1 Non-pipelined Hash Join Algorithms

Even though non-pipelined Hash Join (NPHJ) algorithms are not dealt with in this thesis, several algorithms of this type are still worthwhile to introduce since they are used for traditional query processing. In NPHJs, each join execution is completed before the start of the next join execution. These algorithms are not suitable for data integration systems where the Internet transfer delay occurs, as mentioned in Section 1.1. The algorithms presented here determine the number of PNs and which PNs are allocated for the execution of each join in a query tree.

Chen et al. [15] considered the following factors: operational point selection (between *the minimum time point* and *best efficiency point*) in terms of number of PNs, execution dependency in terms of join execution order, and system fragmentation in terms of the utilization of PNs. Alongside a query tree, a bottom-up approach (*sequential execution*, *fixed cluster size*, *minimum-time point*, and *time-efficiency point* are presented) is proposed. Also, a top-down approach (*synchronous execution time*, in which the two input joins to a certain node are completed at approximately the same time) is proposed. *Cumulative execution cost* is used to allocate processors. The conclusion is that the top-down approach is better than the bottom-up approach because of system fragmentation. Another conclusion is that the following steps (called *two phase query optimization*) are effective. In phase 1, the join sequence heuristic is applied to build a bushy tree to minimize the total amount of work required, as if under a single processor system. In phase 2, using a synchronous execution time concept, PNs are allocated to the internal nodes of the bushy tree in a top-down manner.

Liu [54] proposed the following algorithms based on the query cost model considering communication overhead and load imbalance on a shared-nothing architecture connected by a crossbar switch.

The first approach is a non-phased approach, *Dynamic Processor Allocation Algorithm (DPAA)*, in which all ready operations start execution simultaneously by allocating the optimal number of processors to each operation. When the number of PNs is large, this allows for good performance.

The second approach is a phased approach, *Merge-Point Phase Partitioning Algorithm (MPPPA)*, in which all ready operations are grouped in the first execution phase. If an operation's parent is a merge point, in which its children are all in the previous execution phases, then the operation remains in this phase; otherwise, it is moved to the next phase. During the execution, the number of operations are roughly equal in each phase. Within the phase, local optimization (called *time equalization*) is done. In the case where the number of PNs is relatively small, this algorithm is better than DPAA because of the time equalization mechanism.

2.4.2 Pipelined Hash Join Algorithm

There are two types of Pipelined Hash Join (PHJ) algorithms: simple PHJ (or Non-Symmetric PHJ (NSPHJ)) [12, 14, 55] and Symmetric PHJ (SPHJ)[89]. In the NSPHJ algorithm, there are two phases: the table building phase and the table probing phase. A building relation (the left child in the query execution tree) and a probing relation (right child) are distinguished. On the other hand, in symmetric PHJ, there is no such distinction. A hash table is built for each relation.

Lo et al. [55] proposed an algorithm for achieving optimal processor allocation for pipelined hash joins. They used a NSPHJ and a two-phase min-max optimization problem (min is for each stage's execution time and max is for the entire execution time). Their assumptions are as follows:

- The total number of PNs available for allocation is fixed.
- A lower bound is imposed on the number of processors required for each stage to meet the corresponding memory requirement.
- PNs are available only in discrete units.
- Shared-disk architecture is used.
- Each PN has the same size of distributed memory.

Forced constraints are no idling, sufficient memory, and discreet processor allocation. They obtained a solution by incrementally adding these constraints.

Hsiao et al. [32] proposed a top down processor allocation. It transforms a bushy tree into an allocation tree in which each node denotes a pipeline. To allocate the processors to each PN, it uses a cumulative execution cost approach as in [15, 31]. It also uses the concept of synchronous execution to allocate PNs to join sequences judiciously so that inner relations in a pipeline can be made available approximately at the same time. This definition is different in meaning from the non-pipelined case [15, 31].

Wilschut et al. [89] proposed the SPHJ for the main memory parallel database, PRISMA/DB. It consists of only one phase. As a tuple comes in, it is first hashed and used to probe that part of the hash table of the other operand that has already been constructed. If a match is found, a result tuple is formed and sent to the consumer operation. Finally, the tuple is inserted into the hash table of its own operand. Compared to the simple hash join algorithm, this algorithm can produce the result tuples earlier during the join process at the cost of using more memory to store the second hash table. This algorithm was compared with other methods and demonstrated its effectiveness most of the time.

Jalali and Dandamudi [47] studied the performance of the of NPHJ and SPHJ on a cluster with an NFS mechanism. The results showed that in most cases the performance of the NSPHJ with BST is better than the SPHJ.

2.4.3 Multiple-Join Algorithms for Data Integration

This subsection introduces the multiple-join algorithms used in the data integration systems in Section 2.1.3.2. The problems of data integration are as follows [2, 6, 28, 44]:

- absence of statistics on the data in the database;
- unpredictable data arrival characteristics from databases (initial delay, slow delivery, bursty arrival of data);
- redundancy among sources;
- hardware (e.g., data allocation in disk) and workload complexity;
- data complexity and run-time fluctuation (estimation error of selectivity);
- user interface complexity with users' control over their query.

To cope with these difficulties, double PHJ [44] and XJoin [84], which are the modified versions of the symmetric PHJ [89], have been proposed.

Ives et al. [44] proposed a double PHJ, which consists of two stages: a regular stage, which is SPHJ, and a cleanup stage, which matches join pairs of tuples by the use of marking tuples if one of them arrives after the other one has been flushed to disk.

To address the memory overflow problem, two approaches have been proposed. One approach is an incremental left flush, which gradually changes to a hybrid hash algorithm. The other approach is an incremental symmetric flush, which flushes the buckets of both input relations.

The major advantages of double PHJ are that the latency to the first output of the join is minimum and the system uses its CPUs efficiently. However, the disadvantage of double PHJ is that it requires more memory to hold both join relations.

XJoin [84] consists of three stages, each of which is implemented by an independent thread (with an order of priority). One thread executes SPHJ as long as either of the input relations continues to arrive at the PN. One thread executes the join memory portion and disk portion of input relations. The execution of this thread does not depend on the availability of the input relations. The execution is coarse grained and beneficial when both relations are delayed. A portion of the main memory is reserved as the data cache. One thread executes the rest of the join to ensure the correctness of the join.

2.4.4 Discussion

Among the algorithms described in the previous subsection, when the arrival delay of input relations is taken into account, PHJ gives a superior performance since it generates the first result earlier than NPHJ. However, there has been no study that combined background load and data arrival delay.

The advantages of NSPHJ are (1) it uses less memory because only one hash table for one relation is built, and (2) it discards the probing relation once it finishes probing. As a result, it uses less disk space and fewer I/O operations. The disadvantage of the simple PHJ is that it has to wait for the entire hash table to be built prior to the join operation.

The advantages of SPHJ are (1) it is robust relative to data delay and a bursty data arrival rate, and (2) the result is generated early. The disadvantages of SPHJ are (1) it is for the main memory database, so offers no support for join processing with big relations, and (2) it requires more main memory than NSPHJ since SPHJ builds two hash tables.

The advantages of double PHJ and XJoin are (1) it is more robust relative to data delay than SPHJ, since it uses two (double PHJ) or three (XJoin) independent stages which aim at hiding delay of input relations, and (2) it supports join processing with bigger relations. The disadvantage of double PHJ and XJoin is that they are complicated algorithms and have high overheads to ensure the correctness of the join operation.

From these results, it is not obvious whether NSPHJ or SPHJ is more effective for a cPDBMS,with or without background load and with or without Internet delay. Chapter 5 investigates this issue.

Chapter 3

Non-Symmetric Single-Join Algorithm

In this chapter, non-symmetric single-join algorithms on a cluster-based PDBMS (cPDBMS) are studied. As described in Section 1.1, non-symmetric single-join chooses one of the relations as the main relation. It is effective when we can obtain the information on the join relations (e.g., cardinality) and one of the relations is relatively small.

The algorithms are the parallel versions of the nested-loop join and hash join algorithms and their load balancing/sharing algorithms.

In this chapter, as a preliminary experiment, the performance of nested-loop join processing and its load sharing/balancing algorithms are presented and evaluated on a Pentium-based heterogeneous cluster. We then propose a new load-sharing algorithm called *ChunkHJ* for single hash join processing. The novelty of this load-balancing algorithm is that it divides the hash buckets into chunks and uses them for load balancing. This new algorithm is compared with two other algorithms: an adaptive nested-loop join and an adaptive GRACE join. These three algorithms were evaluated on a cluster system with skewed data and various non-query background load.

3.1 Non-Symmetric Single-Join Processing Environment

This section presents the environment in which single-join processing occurs and the notation used in the description of the algorithms in Sections 3.2 and 3.3.

3.1.1 A System Model

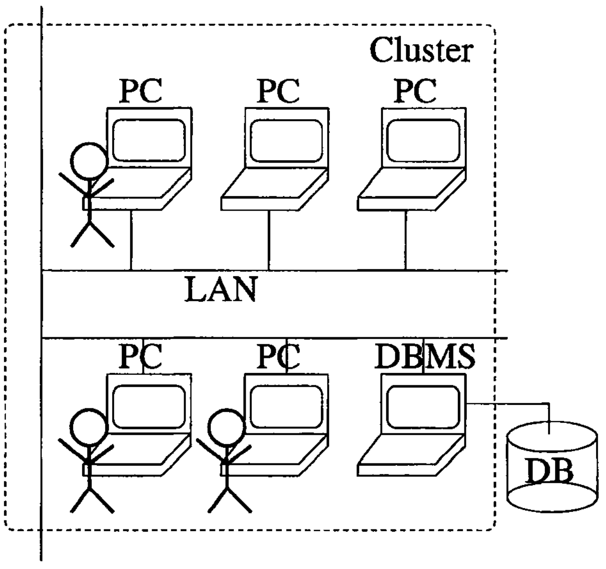

Figure 3.1: A system model without Internet delay.

This system model is a cluster that can be found in many organizations (see Figure 3.1). The model has several PCs (henceforth referred to as processing nodes (PNs)) and many interactive users frequently login and logout. The nodes are used for back-end database query processing and a dedicated database management system (DBMS) is attached to one of the PNs. The key idea of this model is to enhance the existing DBMS for parallel processing using existing clusters rather than replacing a DBMS with a new PDBMS, as described in Section 2.1.3.1, where we called this system a cluster-based PDBMS (cPDBMS).

3.1.2 Non-Symmetric Single-Join Processing System Architecture

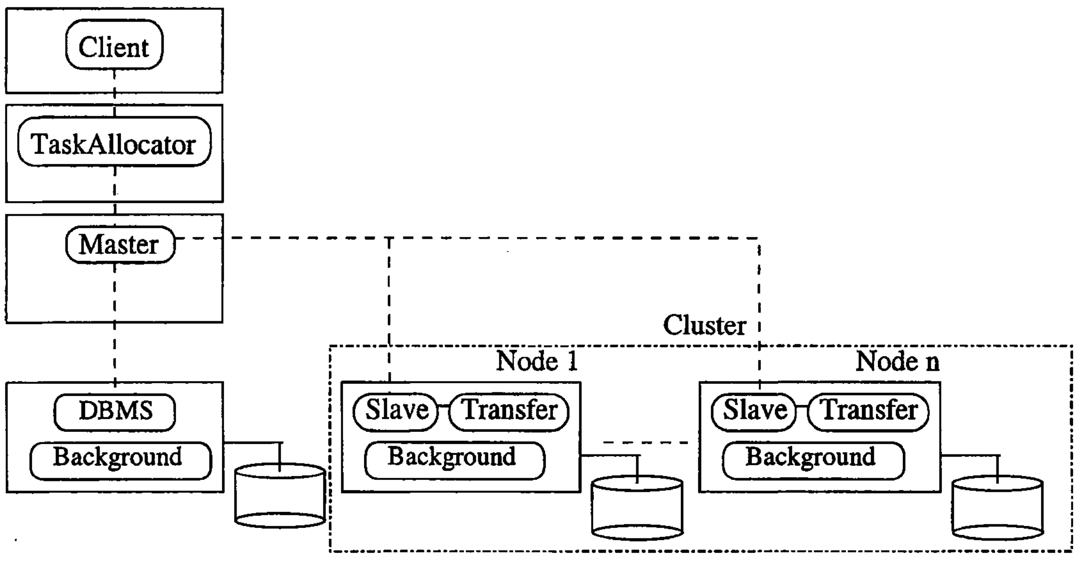

Figure 3.2: Cluster join processing architecture for single-join.

The cluster-based single-join processing system consists of the software components shown in Figure 3.2. The main software components are the *TaskAllocator*, *DBMS*, *master*, *slave*, *background*, and *transfer* processes.

The *TaskAllocator* process is invoked by a client and is responsible for the invocation of all other processes described in this subsection. The allocation of these processes is controlled by a client-specified configuration file. The client can specify parameters, such as a message buffer size, in this configuration file to execute joins with the desired configuration.

The *DBMS* is responsible for data read/write operations to and from its database. These access requests from the master always go through the DBMS. These DBMS read/write operations are implemented using C++ file input/output functions, which is a simple simulation of the read/write operation of a real DBMS.

A *master* process is invoked for each join in a query and is responsible for making load balancing/sharing decisions and sending input relations to slaves based on those decisions. The input data are obtained from a DBMS. This process is also responsible for collecting the results from the slaves and writing them to the local disk.

A *slave* process is responsible for local join processing and uses the join algorithm specified in the configuration file. The local join algorithm is exactly the same as the sequential join algorithm. A message handler for messages from the master and other processes is added to the slave. Its local disk is used to store temporary files.

Background processes are invoked on the slave nodes to simulate the non-query background load. Each background process has an on-time and an off-time parameter. The process is a busy loop followed by a sleep function.

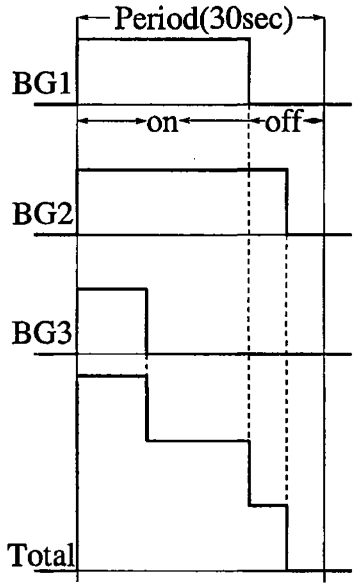

Figure 3.3: Three background loads and their total load on a node.

Two types of background loads are used: proportional background load and inverse background load. In the proportional background load, faster nodes are allocated more load than slower nodes. This is a simulation of real clusters in which faster nodes are used as servers, such as HTTP servers or file servers. Interactive users also tend to run CPU-intensive, time-consuming programs on faster nodes since they expect their programs to execute faster. In the inverse background load, slower nodes are allocated more load than faster nodes. This situation may arise when CPU usage price is proportional to CPU speed. In this case, many users may run their programs on slower nodes. Furthermore, it is interesting to see the results in other extreme opposite case of the proportional background.

This background model is similar to that of Zaki [92], which has maximum load and duration of persistence. This model can achieve more randomness in the load function because of three random parameters as shown in Figure 3.3. The parameters specify the on-time length and are given by a uniform distribution with a maximum value. In the case of a proportional background load, the maximum values for the on-time length on fast, medium, and slow nodes are 15, 10 and 5 seconds, respectively. In the inverse background load, the values are 5, 10, and 15, respectively. These values are reasonable considering the behaviour of the interactive users and the daemon processes [4]. The length of the period is set to 30 seconds. The off-time length is equal to 30 (the length of the period) minus the on-time length.

The *transfer* process is responsible for transferring buckets from one node to another. The details of the transfer process are described in Section 3.3.2.2.

3.1.3 Notation

In the following description, the notation given in Table 3.1 is used.

The functions used in the pseudocode description in the following subsections are summarized in Table 3.2. The *recv*, *send*, and *broadcast* functions are based on the PVM/MPI functions.

Please note that the term *global join* is used to describe the join on the master and the term *local join* is used to describe the join on a slave. The global join algorithm determines how the two relations, R and S, are distributed on the master. The local join algorithm determines how these sub-relations are processed on a slave.

3.2 Parallel Nested-loop Single-Join Algorithm

This section begins by reviewing the parallel version of the nested-loop algorithm. The details of the load sharing methods used in the experiments are then described

Notation	Explanation
p	the number of slaves
nodeProcPowers	the processing power value of each node
R	smaller of the two relations involved in the join
S	larger of the two relations involved in the join
X	either R or S
Y	R∪S-X
$\|R\|$	the number of tuples of R
R^i	the *i*th partition of R
R_h	the *h*th bucket of R
R_h^i	the *i*th chunk of R_h
R_{ALL}	all buckets of R
R[i]	the *i*th tuple of R
R[i..j]	from the *i*th tuple to the *j*th tuple of R
n	the number of buckets
Z	result relation
Z^k	the *k*th partial result
SL_x	a slave
SL_{ALL}	all slaves
idle(SL_x)	return TRUE if SL_x is currently idle
BR(SL_x)	the buckets of R that SL_x has
VR(R_h)	the slaves that have R_h
LT,HT,ST	Low, High, Stop Threshold values

Table 3.1: Notation used in the algorithm description.

[1].

3.2.1 Nested-Loop Join Revisited

The nested-loop algorithm is the oldest join algorithm and is still used due to its robustness relative to data skew. In a nested-loop join involving two relations, the algorithm selects one tuple from one relation (outer relation: R) and scans all tuples

[1]Kenji Imasaki and Sivarama Dandamudi, "Performance Evaluation of Nested-loop Join Processing on Networks of Workstations," *Seventh International Conference on Parallel and Distributed Systems (ICPADS)*, pages 537–544, Iwate, Japan, July 2000.

Name	Arguments	Returns	Description
recv	srcId, tag, msg		receives a *msg* from the master or a slave (*srcId*) with message *tag*
send	destId, tag, msg		sends a *msg* with a message *tag* to the master or a slave (*destId*)
broadcast	tag, msg		broadcasts a *msg* to all slaves
applyHash	relation or bucket, n	buckets	applies a hash function to *relation* (*bucket*) and creates *n* hash buckets
read	relation startId, endId	tuples	reads from DBMS within the specified index range
execLJ	two relations		executes the local join algorithm and stores results in the result buffer; if the buffer is full, it is sent to the master

Table 3.2: Functions used in the pseudocode description.

in the other relation (inner relation: S) to find a match using the join-key attribute. When it finds a match, it produces a result tuple in a relation (output relation: Z) by concatenating the two matched tuples. This process is repeated until there are no more tuples in relation S. The cost of the nested-loop join is the product of the number of tuples in R and the number of tuples in S.

Parallelization of the nested-loop join algorithm is straightforward, as shown by the following description. First, the system broadcasts the entire R (assumes the smaller relation of R and S) and then scatters S to the PNs according to the criteria described in Section 3.2.2. After performing the local join, which is a sequential join operation using the two relations, each PN sends back its result.

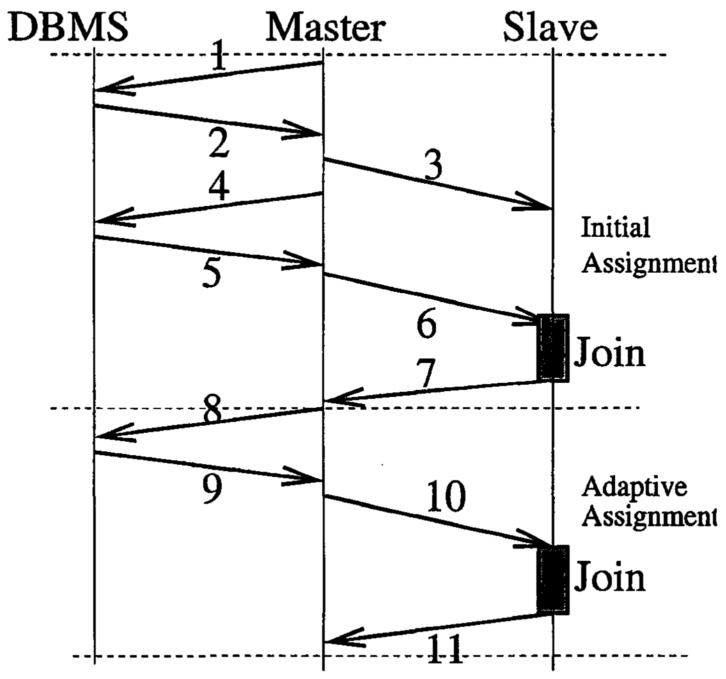

Figure 3.4: Message diagram of load balancing/sharing algorithms.

Algorithm 3.2.1: PARALLELNESTEDLOOPJOINONMASTER(*alg*, *cSize*)

```
{The master executes a global join using relation R and S}
{Input: algorithm name alg (either SS_EQ, SS_PR, DS);
        cSize: chunk size in terms of the number of tuples for DS;
Output:none}
$R \leftarrow read(\text{"R"}, 0, |R| - 1)$   {Message 1 and 2 in Figure 3.4}
$broadcast(\text{"Relation"}, R))$   {Message 3}
$tupleCounter \leftarrow scatter(alg)$   {See Algorithm 3.2.3}
if ($alg$ is DS)
  then while ($tupleCounter < |S|$)   {processing S is not finished}
     do { $recv(SLx, \text{"JobRequest"}, Zx)$   {Message 7}
          $tupleCounter \leftarrow adaptiveJobAssignment(x, tupleCounter, cSize)$
            {this is defined in Algorithm 3.2.4}
$broadcast(\text{"ProcessEnd"}, \text{null})$
```

As shown in Section 3.1.2, the master and the slaves are used to implement this algorithm. Algorithm 3.2.1 shows the pseudocode for the master process. The algorithm to be implemented is given as input. This algorithm can be SS_EQ, SS_PR, or DS, which is described in the next subsection. Algorithm 3.2.2 shows the pseudocode for the slave process. The message diagram is shown in Figure 3.4. The pseudocode also shows the corresponding message number in the diagram.

```
Algorithm 3.2.2: PARALLELNESTEDLOOPJOINONSLAVE()

{The slave($SLx$) executes a local join using relation R and Sx}
{Input:none; Output:none}

$recv(master,$ "Relation", $R)$    {Message 3}
repeat
    $recv(master,$ "Relation", $S^x)$    {Message 6 and 10}
    $execLJ(R, S^x)$    {the result is stored in $Z^x$}
    $send(master,$ "JobRequest", $Z^x)$    {Message 7 and 11}
until $recv(master,$ "ProcessEnd", $null)$    {receive several S tuples in DS}
```

```
Algorithm 3.2.3: SCATTER($alg$)

{scatter relation S to $p$ slaves}
{Input: algorithm name $alg$ (either SS_EQ, SS_PR, or DS)
{Output:tupleCounter}

$tupleCounter \leftarrow 0$
$sumOfNodeProcPower \leftarrow \sum NodeProcPowers$    {used for SS_PR}
for each $SLx$ where $0 \leq x < p$
  do
    if ($alg$ is SS_EQ)
      then $nTuples \leftarrow |S|/p$
    if ($alg$ is SS_PR)
      then $nTuples \leftarrow NodeProcPowers[x]/sumOfNodeProcPowers * |S|$
    if ($alg$ is DS)
      then $nTuples \leftarrow 0$
    if ($nTuples > 0$)
      then $Sx \leftarrow read(``S", tupleCounter, tupleCounter + nTuples - 1)$
                  {Message 4 and 5}
    $send(SLx, ``Relation", Sx)$    {Message 6}
    $tupleCounter \leftarrow tupleCounter + nTuples$
return ($tupleCounter$)
```

3.2.2 Load Balancing/Sharing Techniques for Nested-Loop Join Algorithms

This subsection discusses load balancing/sharing techniques for nested-loop join algorithms. The main topic of this subsection is the *scatter* function, whose pseudocode is shown in Algorithm 3.2.3.

3.2.2.1 Static Scheduling (SS_EQ and SS_PR)

Static scheduling determines how the data is distributed at the compile time. In database query processing, the query is compiled when it is submitted, and the data distribution decisions are made at that time using static scheduling.

There are two initial assignment schemes depending on the data distribution: SS_EQ and SS_PR. These differences are reflected in *scatter* (see Algorithm 3.2.3).

Equal Distribution (SS_EQ) In this distribution scheme, each slave receives the same number of tuples regardless of the processing speeds of the slave nodes. SS_EQ is expected to be slower compared to the other methods since the execution time of the slowest node, which has the same amount of work as the fastest nodes and thus finishes its execution last, dominates the whole execution time. However, SS_EQ might be effective when the processing power of each node is approximately equal from the user's point of view in the case that faster nodes process more work and slower nodes process less work. This situation may occur frequently since users tend to run their programs on faster nodes to finish the programs as soon as possible and system administrators tend to run server programs (e.g., file or HTTP servers) on the faster nodes. In these cases, SS_EQ might be efficient.

Proportional Distribution (SS_PR) In this distribution scheme, each slave receives a number of tuples in proportion to its processing power known prior to the query execution. Node processing power for each node is stored in the array *NodeProcPowers* and is affected by the CPU speed, main memory size, and cache memory size. This node processing power can be obtained from experiments [77] using local join processing with no background load. SS_PR should surpass SS_EQ when there is no background load since all the nodes finish their work at approximately the same time. Its execution time is expected to be shorter than SS_EQ.

3.2.2.2 Dynamic Scheduling (DS)

The pseudocode for this algorithm is shown in Algorithm 3.2.4.

Algorithm 3.2.4: ADAPTIVEJOBASSIGNMENT($SLx, tupleCounter, cSize$)

```
{The master reads the next S tuples and sends them to $SLx$}
{Input: the job-requested slave($SLx$) and $tupleCounter$;
        $cSize$: chunk size in terms of the number of tuples;
Output: tupleCounter}
$Sx \leftarrow read(S[tupleCounter...tupleCounter + cSize - 1])$
                {Message 8 and 9}
$send(SLx,$"Relation"$, Sx)$   {Message 10}
$tupleCounter \leftarrow tupleCounter + cSize$
return ($tupleCounter$)
```

Dynamic scheduling determines the data distribution at run time. Its execution steps are as follows. First, the master assigns initial work to slaves. After a slave finishes its work, it requests more work from the master. Thus, this algorithm uses demand driven (DD) work assignment. As described in Section 2.3.3, this adaptive feature is necessary for the load sharing algorithms on clusters. The data distribution is affected by the run time environment, such as the current load and the size of the available memory of each node. This scheduling is expected to be better than static scheduling when there is node heterogeneity and/or there is background load.

In Section 3.3, a variation of DS called *ChunkNJ* is also introduced in comparison to hash join algorithms. *ChunkNJ* is identical to this algorithm except for its local join algorithm. The local join algorithm is changed from the nested-loop join to the hash join to obtain superior performance.

The advantage of these algorithms is that its execution time does not depend on initial data skew since it is based on the nested-loop join (global join). Join product skew and load changes on a slave caused by background load can be handled by the adaptive feature of this algorithm.

However, its disadvantage is the overhead in broadcasting one of the relations involved in the join to all slaves. Furthermore, the hash table for the broadcasted relation in the local join is larger than those obtained using other hash-based join algorithms.

3.3 Parallel Hash Single-Join Algorithm

This section starts with reviewing a hash join algorithm. Then, the load balancing/sharing algorithms for join processing on clusters are proposed.[2]

3.3.1 Hash Join Revisited

A hash join algorithm consists of two phases: a table building phase and a table probing phase. A hash table is built using one of the relations (usually the smaller of the input relations) and the tuple of the other relation is used to probe the hash table. Parallelization of a hash join is also straightforward. Both relations are divided into disjoint sets (hash buckets) and scattered. The tuples belonging to the buckets with the same hash value can be joined independently in parallel.

3.3.2 Load Balancing/Sharing Techniques for Non-Symmetric Hash Join Algorithms

3.3.2.1 Adaptive GRACE Hash Join (*GRACE+*)

ChunkNJ suffers from the overhead in broadcasting a relation unless clusters have a special broadcast network. To avoid this overhead, *GRACE+* is considered, which is based on the GRACE hash join [52] and includes demand-driven work assignment. The demand-driven work assignment is one of the ways to deal with dynamic changes in the background loads on the slaves.

General Algorithm Description The master and the slaves are used to implement this algorithm. This algorithm first reads relation R and S and builds hash table for each relation. Then, using an adaptive assignment, the master sends a specified number of hash bucket pairs to a slave. The slave works on the join of the hash bucket pairs. After it finishes the join, the slave requests the master for the next hash bucket pairs.

[2]Kenji Imasaki and Sivarama Dandamudi,"An Adaptive Hash Join Algorithm on a Network of Workstations,"*International Parallel and Distributed Processing Symposium (IPDPS)*, on CD-ROM, Fort Lauderdale, Florida, April 2002.

Algorithm 3.3.1 shows the pseudocode for the master process. Algorithm 3.3.3 shows the pseudocode for the slaves.

Algorithm 3.3.1: GRACEONMASTER(*cSize*)

```
{Adaptive GRACE algorithm on master}
{Input: cSize is chunk size in terms of the number of buckets;
Output: none}
$R \leftarrow read(\text{"R"}, 0, |R| - 1)$   {Split phase}
$R_{ALL} \leftarrow applyHash(R, n)$   {See Algorithm 3.3.6 without SendInit}
$S \leftarrow read(\text{"S"}, 0, |S| - 1)$
$S_{ALL} \leftarrow applyHash(S, n)$
$bucketCounter \leftarrow 0$
for $each\ SLx\ where\ 0 \leq x < p$   {Join phase}
    do
        $send(SLx, \text{"Relation"}, R_{bucketCounter..bucketCounter+cSize})$
        $send(SLx, \text{"Relation"}, S_{bucketCounter..bucketCounter+cSize})$
        $bucketCounter \leftarrow bucketCounter + cSize$
while $(bucketCounter < n)$
    do
        $recv(SLx, \text{"JobRequest"}, Z^x)$
        $bucketCounter \leftarrow AdaptiveJobAssign(SLx, bucketCounter, cSize)$
                                  {See Algorithm 3.3.2}
$broadcast(\text{"ProcessEnd"}, null)$
```

Algorithm 3.3.2: ADAPTIVEJOBASSIGN(*SLx*, *bucketCounter*, *chunkSize*)

```
{Adaptive job assignment for GRACE+}
{Input: the job-requested slave (SLx); bucketCounter;
        chunkSize is the number of buckets not the number of tuples
Output: bucketCounter}
$send(SLx, \text{"Relation"}, R_{bucketCounter..bucketCounter+chunkSize})$
$send(SLx, \text{"Relation"}, S_{bucketCounter..bucketCounter+chunkSize})$
$bucketCounter \leftarrow bucketCounter + chunkSize$
return $(bucketCounter)$
```

```
Algorithm 3.3.3: GRACEONSLAVE(chunkSize)

{Adaptive GRACE algorithm on Slave}
{Input: chunkSize; Output:none}

repeat
    recv(master, "Relation", $R_{0..chunkSize}$)   {R buckets are sent first}
    recv(master, "Relation", $S_{0..chunkSize}$)   {R buckets are sent next}
    for each h where $0 \leq h < chunkSize$
        do execLJ($R_h, S_h$)   {the result is stored in $Z^x$}
    send(master, "JobRequest", $Z^x$)
until recv(master,"ProcessEnd",null)
```

Advantages and Disadvantages The advantage of this algorithm is that the master sends only the necessary tuples to the slaves. Thus, there is no overhead associated with broadcasting one of the relations as in ChunkNJ discussed in Section 3.2.2.2.

However, its main disadvantage is that the join start time is delayed because the master has to wait for both relations to be sent from the DBMS. Furthermore, slaves should wait for the data to be sent from the master. Another disadvantage is that the unit of chunking of this algorithm is the number of buckets. Thus, there is no control over the chunk size in terms of the number of tuples. As a result, the algorithm cannot deal with the extreme skew case, where many tuples have the same key value in the input relations. There are also memory restrictions in that the master is required to hold two relations in its main memory prior to the join execution.

3.3.2.2 Chunk-based Hash Join (*ChunkHJ*)

ChunkHJ tries to combine the best features of *ChunkNJ* in Section 3.2.2.2 and *GRACE+*. This algorithm creates chunks of S_h (S_h^i) by setting three threshold values: a Low Threshold (LT), High Threshold (HT), and Stop Threshold (ST) on the hash table in the master, as shown in Figure 3.6(a). LT is determined by the communication overhead involved in sending a chunk. HT is also determined by the communication overhead involved in sending both chunks. ST depends on the amount of memory available.

When the number of tuples in a bucket of S ($|S_h|$) is above one of these threshold values, the master takes the following actions to create a chunk of bucket S (S_h^i): if the number of tuples is greater than LT, then the master sends S_h^i to an idle slave that has the other matching bucket; if it is greater than HT, the master sends S_h^i to an idle slave; if it is greater than ST, the master stops reading tuples of S from the DBMS, waits for a slave to become idle and sends S_h^i to the idle slave.

There are two occasions to check these threshold values. One is when a new tuple is inserted into the hash table on the master, and the other is when an idle slave sends a work request message. The algorithm used in the former case is called the *sender-initiated algorithm* and the algorithm used in the latter case is called the *receiver-initiated algorithm.* Note that "sender" refers to the master and "receiver" refers to a slave. The combination of these two algorithms is needed to obtain strong performance in join processing on clusters.

General Algorithm Description Algorithm 3.3.4 shows the pseudocode for the master process. Algorithm 3.3.5 shows the pseudocode for the slave process.

Algorithm 3.3.4: ChunkHJOnMaster(*partitionSize*)

```
{ChunkHJ algorithm on Master}
{Input: partitionSize for relation S; Output: none}

$R \leftarrow read(“R”,0,|R|-1)$
$R_{ALL} \leftarrow applyHash(R, n)$
invoke Initial Bucket Balancing
$nPartitions \leftarrow |S|/partitionSize$
{Loop (1) and (2) are interleaved}
(1)for $i \leftarrow 0$ to $nPartitions - 1$
    do  $S^i \leftarrow read(“S”, i * partitionSize, (i+1) * partitionSize - 1)$
        $S^i_{ALL} \leftarrow applyHash(S^i, n)$
        {See Algorithm 3.3.6 and SendInit(tuple,h) in Figure 3.5 is invoked}
(2)while (processing R and S is finished)
    do $RecvInit(SLx)$    {See Figure 3.10}
$broadcast(“ProcessEnd”, null)$
```

Algorithm 3.3.5: ChunkHJOnSlave()

```
{Adaptive ChunkHJ algorithm on Slave (SLx)}
{Input: none; Output: none}

repeat
    $recv(master, “Relation”, X)$
    if (X is R)
      then  $R_x \leftarrow X$
            recv(master,“Relation”,$S^i_x$)
      else  $S^i_x \leftarrow X$
    $execLJ(R_x, S^i_x)$
    $send(master, “JobRequest”, null)$
until $recv(master, “ProcessEnd”, null)$
```

Algorithm 3.3.6: ApplyHash(X, n)

```
{apply a hash function to each tuple in X}
{Input: relation (or tuples) X (either relation R and S); Output: none}
for $i \leftarrow 0$ to $|X|$
    do { $h \leftarrow HashFunction(X[i])$
       { $SendInit(X[i], h)$    {See Figure 3.5}
```

Initial Bucket Balancing Scheme During the initial stage, load balancing is done by changing the hash function according to the distribution of the building relation. This is because imbalances in the building relation are worse than imbalances in the probing relation [21]. However, balancing the building relation may introduce skew in the probing relation.

The following balancing schemes are implemented. In the following algorithms [21], the partition scheme, which was explained in Section 2.3.1.3, is changed:

- Load balancing (NB) when the number of buckets is the same as the number of PNs and one bucket pair (one for R and one for S) is assigned to a PN;
- Round Robin load balancing (VP-RR) when the number of buckets is more than the number of PNs and the mapping from a bucket to a PN is done in round-robin fashion;
- Bin Packing load balancing (VP-BP) when the number of buckets is more than the number of PNs and the mapping from a bucket to a PN is done in a bin-packing fashion (balancing the load of each PN).

Sender-Initiated Algorithm The sender-initiated algorithm is explained using the example of Figure 3.6. The main flowchart is shown in Figure 3.5. When a tuple of S is inserted into the hash table, the master checks the number of tuples in each bucket. When the number of tuples in a bucket is greater than ST, the bucket should be sent to any idle slave as soon as possible since it is an emergency situation. The master stops reading tuples from the DBMS and waits for a work request message from SL_x, since the number of tuples in the bucket has reached the limit.

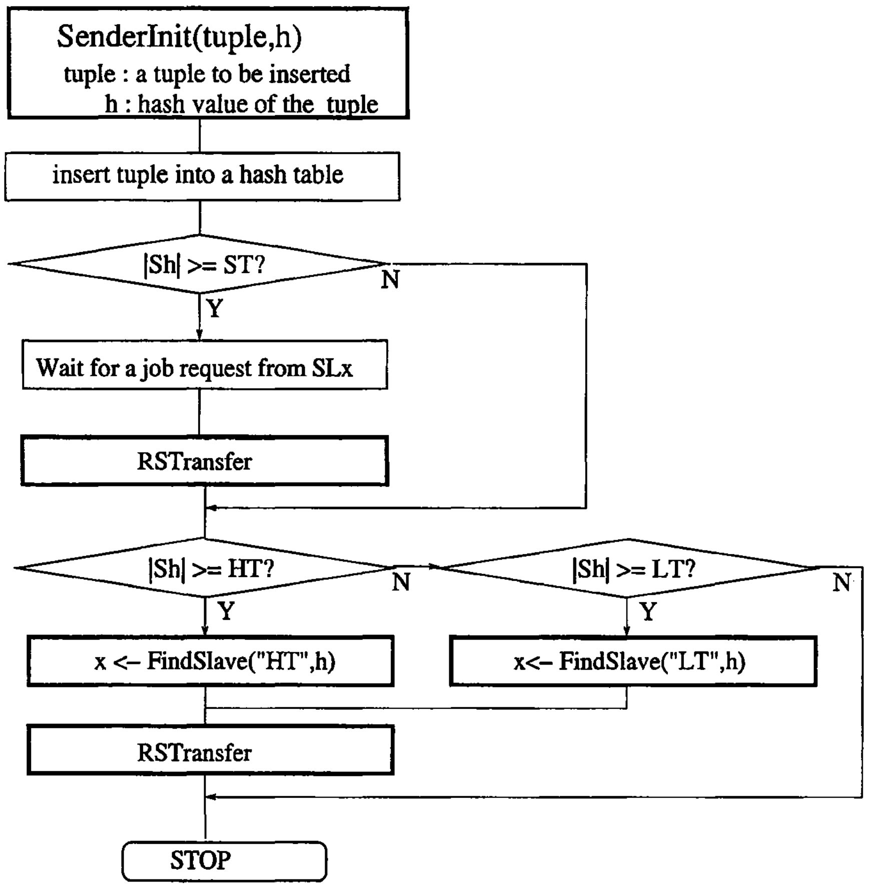

Figure 3.5: Sender-initiated algorithm flowchart.

When the number of tuples of a bucket is greater than HT (in the case of Figure 3.6(a), S_2^2), the master tries to find an idle slave. If the master can find such a slave, it sends the bucket (S_2^2) and the other matching bucket (R_2) if necessary. For example, if SL_2, which has R_2 already (according to Figure 3.6(b)), is idle at that time, the master sends only the bucket (S_2^2). However, if SL_1, which does not have the other matching bucket (R_2), is idle at that time, the master sends the bucket (S_2^2) and the other matching bucket (R_2). The master sends the bucket (R_2) directly or asks a slave with the bucket to send it. The details of this transfer will be described later. If the master cannot find an idle slave, the master does not take any action and continues to read tuples of S from the DBMS.

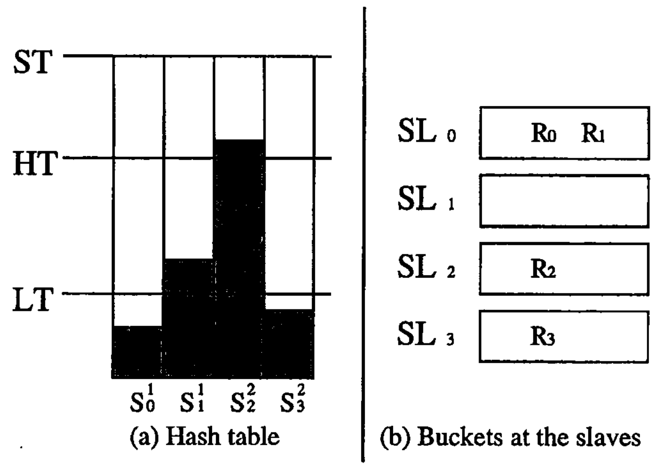

Figure 3.6: An example of a hash table on the master and the buckets at the slaves.

When the number of tuples is greater than LT but less than HT (S_1^1), the master tries to find an idle slave with the matching bucket (R_1). If the master can find such a slave, it will send the tuple (S_1^1). For example, if SL_0 is idle at that time, the master sends the bucket chunk (S_1^1) to SL_0. If the master cannot find such a slave, the master leaves it and continues to read tuples of S from the DBMS.

ChunkHJ looks similar to the Subset-Replicate algorithm given in [21]. However, this algorithm dynamically (reflected by various types of data skew) and adaptively (reflected by the system status) makes subsets (or chunks) and replicates the other matching bucket. This process is executed concurrently when reading the tuples of S from the DBMS as shown in (1) and (2) in Algorithm 3.3.4. As a result, execution of the sender-initiated algorithm and the receiver-initiated algorithms is interleaved. Furthermore, slaves can replicate the bucket among themselves without involving the master.

The *FindSlave* function finds an idle slave with the other matching bucket. The main flowchart is shown in Figure 3.7. In this flowchart, *v1* is calculated first. *v1* is a list of all slaves (in case of HT) or slaves that have R_h (in case of LT). Then, *v2* is calculated. *v2* is a list of idle slaves that are in *v1*. Finally, a slave is randomly selected from *v2* when there are several candidate slaves which satisfy the condition, since the system is rather small (less than 8 slaves) and there are few idle slaves in

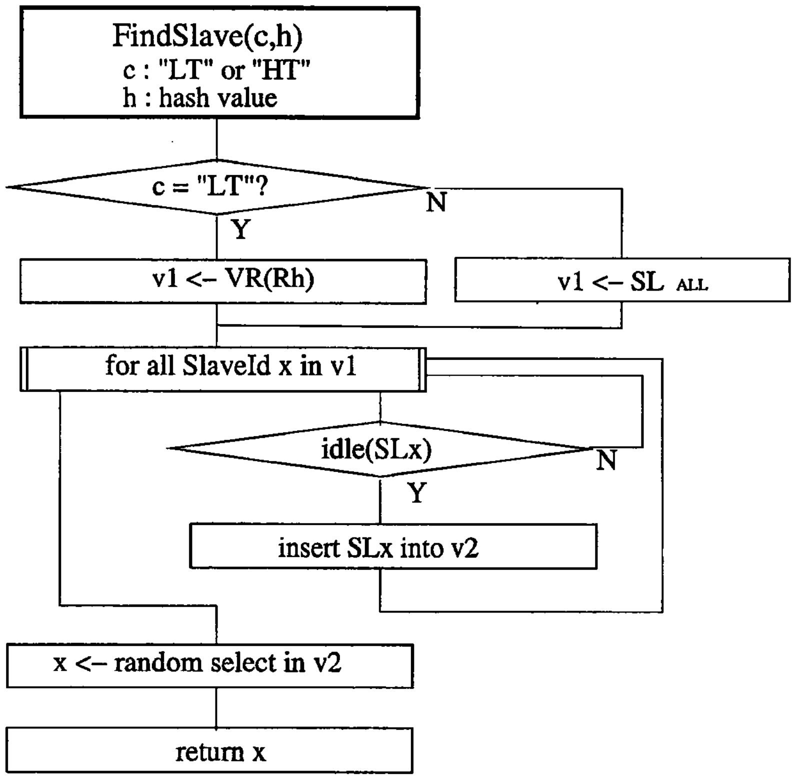

Figure 3.7: FindSlave flowchart.

the system. More sophisticated algorithms should be used for a larger system.

The *RSTransfer* (shown in Figure 3.8) function transfers both R and S buckets. The following Transfer Policies (TP) are designed to transfer the matching bucket (R bucket): (1) the master is always used to transfer the S bucket (*TP-MA*); (2) the master is used when it is idle (no messages in its message queue); otherwise, the master will ask one of the slaves with the R bucket to transfer the bucket (*TP-MI*); (3) the master is used when it is idle; otherwise, the master will ask one of the *Transfer* processes with the R bucket to transfer the bucket (*TP-MIT*); (4) the master always asks one of the transfer processes with the R bucket to transfer it (*TP-TA*). Except for the TP-TA, which never uses the master for such a transfer, the master should keep the buckets in its local disk.

The *Transfer* process (shown in Figure 3.9) is used to transfer an R bucket or S

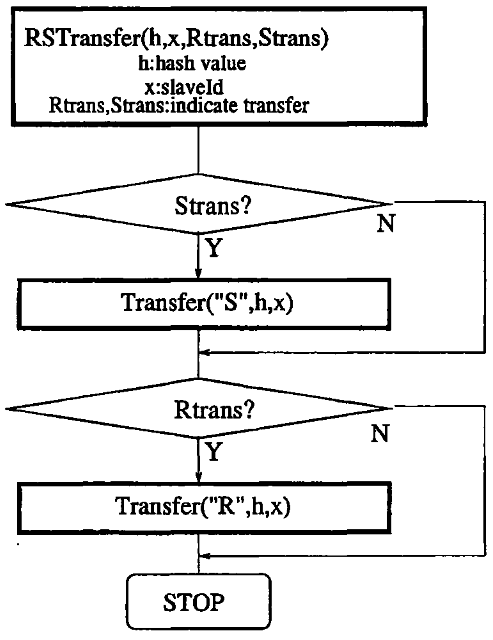

Figure 3.8: RSTransfer flowchart.

bucket from one node to another.

Receiver-Initiated Algorithm The Receiver-initiated algorithm is also explained using the example of Figure 3.6. The main flowchart is shown in Figure 3.10. This algorithm is invoked when a slave (SL$_x$) requests work from the master after it has finished processing the previously allocated chunk. This algorithm is very similar to the sender-initiated algorithm discussed previously.

First, the master tries to find the bucket for which SL$_x$ has the other matching bucket, with the number of tuples in the bucket being greater than LT. For example, in Figure 3.6, if SL$_0$, which has R$_0$ and R$_1$, requests work from the master, the master sends S_1^1 since $|S_1^1|$ is greater than LT and SL$_0$ has R$_1$. If the master cannot find such a bucket, the master tries to find the bucket such that the number of tuples in the bucket is greater than HT. If SL$_1$, which does not have any bucket, requests work from the master, the master sends S_2^2 since $|S_2^2|$ is greater than HT. The master also sends R_2 to SL$_1$ by *RSTransfer*, since SL$_1$ does not have R_2.

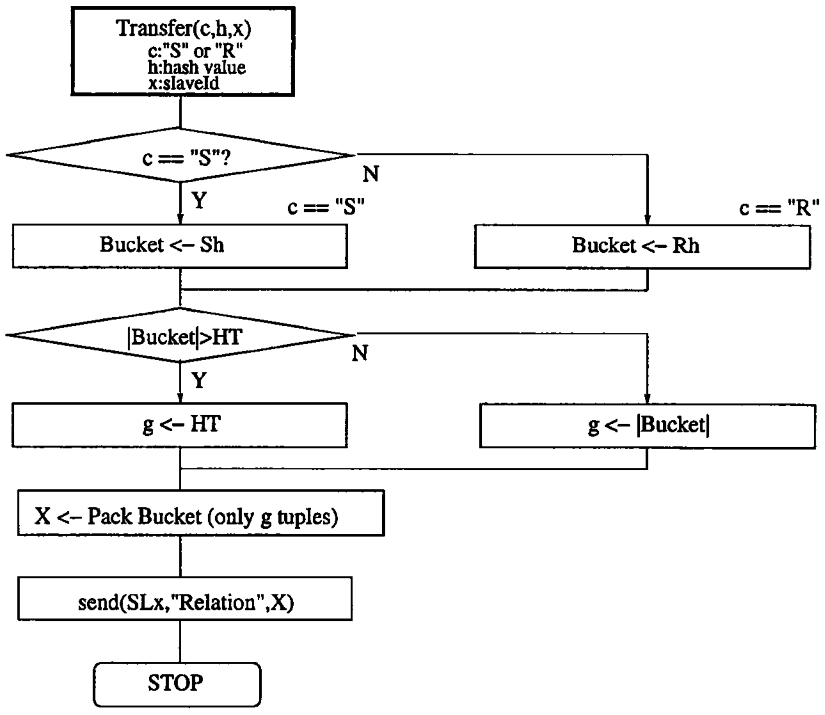

Figure 3.9: Transfer flowchart.

The *FindBucket* function (shown in Figure 3.11) finds a bucket in which the number of tuples is greater than the value specified in the parameter. This value is either LT or HT. In this flowchart, *b1* is a list of all buckets (in the case of HT) or R buckets that SLx has (in the case of LT). Then, *b2* is calculated. *b2* is a list of R buckets in which the number of buckets is greater than HT or LT. Finally, a bucket is randomly selected from *b2* when there is more than one such R buckets.

Advantages and Disadvantages This algorithm is robust with respect to various types of skew since when the bucket is skewed, the number of slaves that work on the bucket changes dynamically and incrementally depending on the data skew and background load. This algorithm also corrects redistribution skew caused by an incorrect prediction of distribution by using an incremental or sampling method. Even a simple distribution scheme such as the round-robin scheme may work with this load sharing

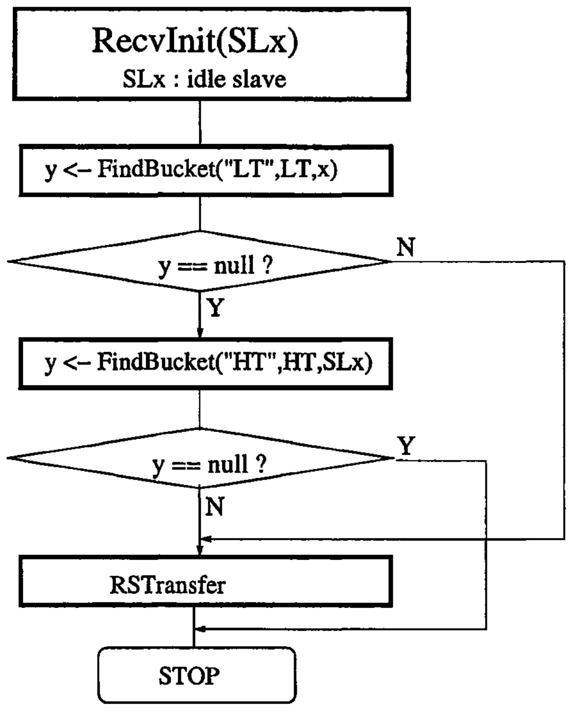

Figure 3.10: Receiver-initiated algorithm flowchart.

algorithm. In addition, the combination of sender-initiated and receiver-initiated algorithms results in a robust algorithm in the presence of data skew and background load on PNs.

However, the disadvantage lies in the overhead in transferring the other matching buckets. Furthermore, join start time is delayed because the master has to wait for one of the relations to be sent from the DBMS. In addition, slaves should wait for the relation to be sent from the master as well. Nevertheless, this delay is not as large as in *GRACE+* in Section 3.3.2.1, in which both relations should be read and sent before starting the join processing.

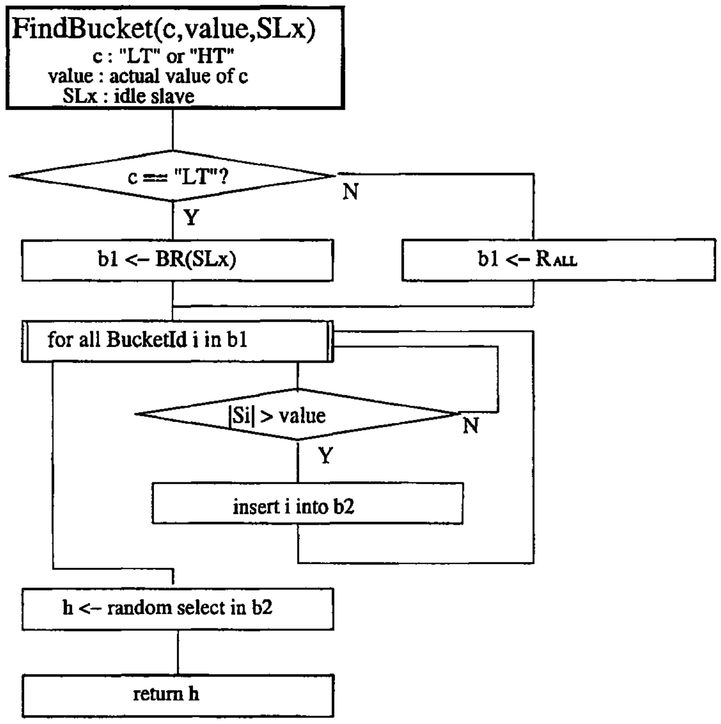

Figure 3.11: FindBucket algorithm flowchart.

3.4 Experimental Environment

3.4.1 Hardware Environment

The hardware system used in this experiment is an inexpensive Pentium P1 based cluster system.[3]

The hardware system that we used consisted of 2 Pentium 133MHz (join processing power is 100, called slow nodes), 2 Pentium 166MHz (125, medium nodes), and 2 Pentium Pro 200 MHz (264, fast nodes). All nodes had 32MB of main memory and were connected by Ethernet. This system simulates a cluster with node heterogeneity found in most cluster environments.

These nodes are interconnected with a 10 Mbps network as well as with a 100 Mbps 100 Base TX high-speed network. The 10 Mbps network is used for booting

[3]These experiments were conducted during 1998-2000.

the system and providing basic communications and remote file service through NFS. The 100 Mbps network is connected with a switch that has a 2 Gbps backplane.

3.4.2 Software Environment

A software system is developed to simulate parallel database processing under various parameters. The overall software system was written in GNU C++ version 2.7.1 and the PVM 3.3 library [24]. As for the PVM parameters, PvmAllowDirect is used for the routing method and PvmDataDefault is used for the data packing method. The parameters used in this experiment are shown in Table 3.3.

Parameter	Value
Input message buffer	140 (KBytes)
Output message buffer	140 (KBytes)
The DBMS node (slow network)	slow node
The DBMS node (fast network)	fast node
The master node (slow network)	slow node
The master node (fast network)	fast node
Slave nodes	2 slow, 2 medium, 2 fast nodes

Table 3.3: Default parameter values used in the experiments.

3.4.3 The Experimental Database

3.4.3.1 Database Schema

The database consists of two relations, R and S. Each tuple has an integer key and 10 byte characters for a total of 14 bytes for each tuple. Each tuple of the result relation has a 24 byte tuple length (20 byte characters from R and S plus the 4 byte integer key).

In the case of nested-loop join experiments, the size of the input relations R and S is 10^4 and the result relation should have 10^4 tuples as well, since there is no skew in input relations.

In the case of hash join experiments, the size of relation R is fixed at 10^5 and the size of relation S varies from 10^4 to 10^6.

3.4.3.2 Skew Generation

In order to change the skew factor of the relations, a scalar skew model is used [21, 27]. In this algorithm, for a relation of size $|R|$, in each attribute the value 1 appears in a fixed number of tuples, while the remaining tuples contain values uniformly distributed between 2 and $|R|$. For example, if $|R| = 200$ and skew factor is 10, relation R has 10 tuples with an attribute of 1 and 190 tuples with attribute values that are uniformly distributed between 2 and 200. In our experiments, we use 10^3, 10^4, and 10^5 skew factors for R and S.

It is worth mentioning that the input order is randomized, thus, even if the data does not have skew, buckets are filled at random.

3.5 Experimental Methodology

Experiments are done in the following way. First, we experiment with the load sharing/balancing algorithms for the nested-loop join algorithm. Skewed relations are not considered since nested-loop is generally robust with respect to the data skew. We compare the algorithms with no background, proportional, and inverse background load cases.

Next, we experiment with the load sharing/balancing algorithms for the hash join algorithm. First, we conduct preliminary experiments to decide the default values, using no data skew or background load cases. Then, we test the algorithms with data skew but without background load. Finally, we test the algorithms with data skew and proportional and inverse background load cases. In terms of data skew, one-sided skew (only one relation has skew) and two-sided skew (both relations have skew) are used with these background load scenarios.

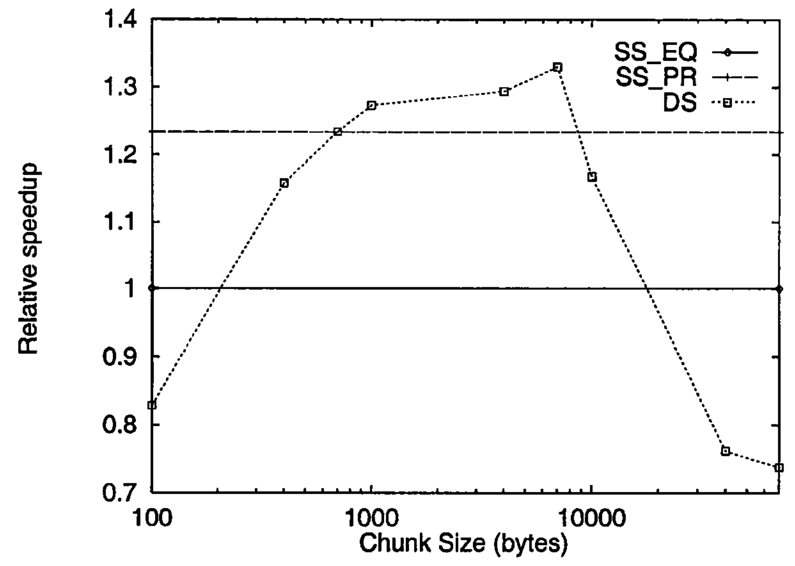

Figure 3.12: The relative speedup without background load using the slow network.

3.6 Experimental Results

3.6.1 Results for Nested-Loop Join

The experimental results of nested-loop join and its load balancing algorithms described in Section 3.2 are presented in this subsection. The graphs show speedup relative to SS_EQ. The results reported here are based on an average of 30 runs.

3.6.1.1 Performance with No Background Load

The relative speedup of SS_PR and DS without the background load on the nodes using the slow (10 Mbps) network is shown in Figure 3.12.

Comparing SS_EQ and SS_PR, SS_PR is 23% better than SS_EQ because the work is distributed in proportion to the processing power of each node obtained prior to execution in SS_PR. In SS_PR, the slow nodes, whose execution time dominates the entire execution time, process less work and finish their execution at approximately the same time as the fast nodes. Accordingly, the overall execution time is reduced. If SS_PR′, which is a variation of SS_RR in which work is distributed

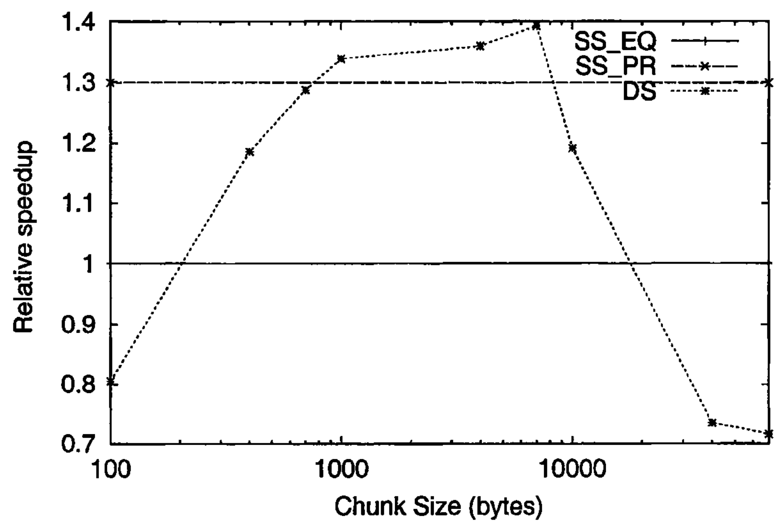

Figure 3.13: The relative speedup without background load using the fast network.

proportionally to CPU speed, had been implemented, its performance would have been intermediate between that of SS_EQ and SS_PR. The reason for this is that SS_PR′ is better than SS_EQ since the slow nodes process less work, as is the case with SS_PR, but the distribution is not as accurate as in SS_PR; accurate distribution reflects the current node processing power affected not only by CPU power, but also by the available memory size, cache size and other factors.

In DS, the performance peaks at a chunk size of 7KB (5% of 10,000 tuples in S). Interestingly, when the chunk size is carefully chosen, namely, between 1KB (0.7% of 10,000 tuples in S) and 7KB (5% of 10,000 tuples in S), DS outperforms SS_EQ by 25%–35% and SS_PR by 5% – 10% even without background load. This demonstrates the effectiveness of DS and the imperfections of the processing power data of each node in this system, obtained by Soleimany [77], due to factors like network load variation. If the data had been perfectly accurate, DS could not have outperformed SS_PR due to the communication overhead of DS, as shown in Figure 3.4.

The relative speedup without background load using the fast (100 Mbps) network is shown in Figure 3.13. Even though the shape of the graph is almost the same as in

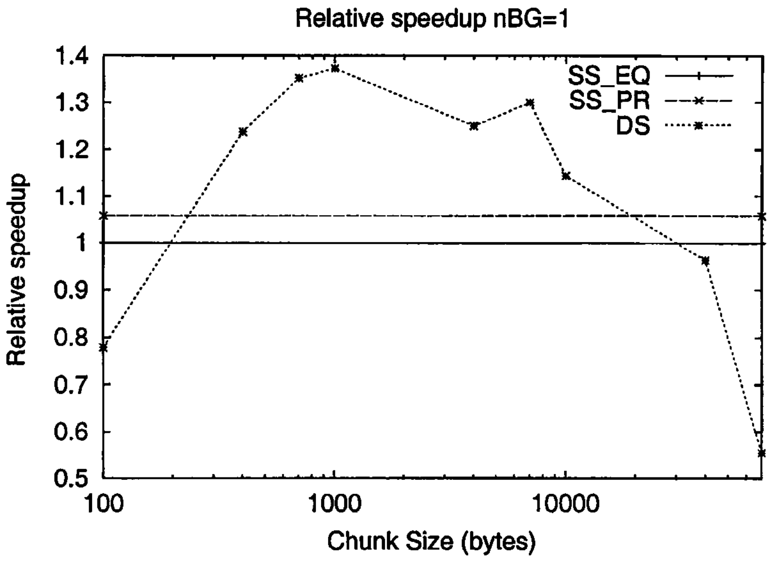

Figure 3.14: The relative speedup with proportional background load using the fast network (the number of background load is 1 process).

Figure 3.12, the relative speedup of DS and SS_PR is higher than the values in Figure 3.12. The execution time shows an improvement (compared to the slow network case) of about 16% (SS_EQ), 20% (SS_PR), and 25% (DS: at the peak using 7KB chunk size). The reason for this DS's better improvement is that many communications are involved in DS, as shown in Figure 3.4, and the fast network reduces the overhead in these communications.

3.6.1.2 Performance with Proportional Background Load

The relative speedup of SS_PR and DS with a proportional background load (1 and 3 processes) using the fast network is shown in Figures 3.14 and 3.15, respectively. As described in Section 3.1, with the proportional background load, faster nodes have greater background load than the slower nodes.

Between SS_EQ and SS_PR (in Figures 3.14 and 3.15), SS_EQ is worse when the background load is 1 process and better than SS_PR when the background load is 3 processes. We also conducted experiments with other numbers of background

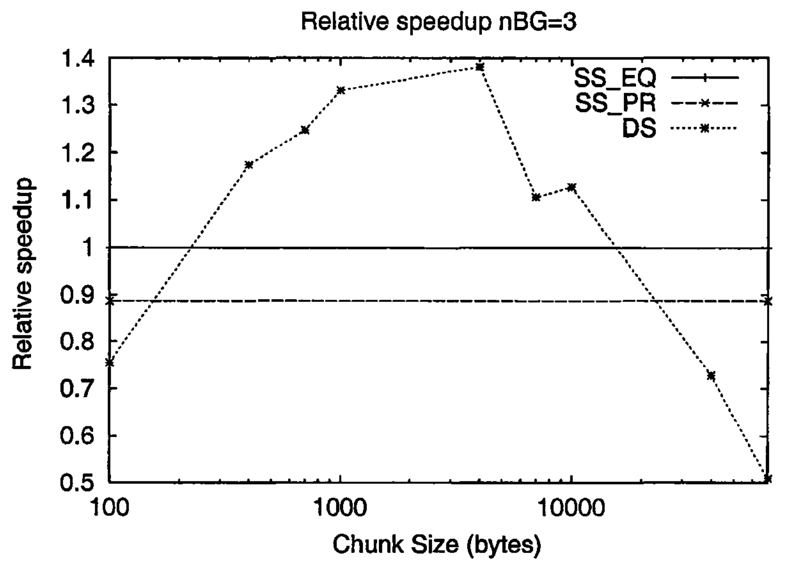

Figure 3.15: The relative speedup with proportional background load using the fast network (the number of background load is 3 processes).

processes. The relative speedups of SS_PR obtained are 1.30, 1.05, 0.82, 0.89, and 0.96 for the background loads ranging from 0 to 4 processes, respectively. When the background load is 0 or 1 processes, SS_PR is faster than SS_EQ, since the data processing power obtained prior to the execution reflects the actual processing power in SS_PR because of little or no interference from background load. However, when the background load is 2 processes, the actual processing power of all processing nodes is roughly equal. As a result, SS_EQ becomes faster than SS_PR. As the background load increases to 3 and 4 processes, SS_EQ becomes slower since the slow nodes have a background load of 3 or 4 and their share of work is equal to that of the fast nodes. Even though the background load is relatively light, the performance deteriorates since processing speed is slow.

When DS and SS_EQ are compared, the relative speedup is 15%–35% between 0.7KB (0.5% of 10,000 tuples in S) and 7KB (5% of 10000 tuples in S). When DS and SS_PR are compared, the relative speedup is 20%–30% (when the background load is 1 process) and 30%–45% (when the background load is 3 processes) with the same chunk size.

3.6.1.3 Performance with Inverse Background Load

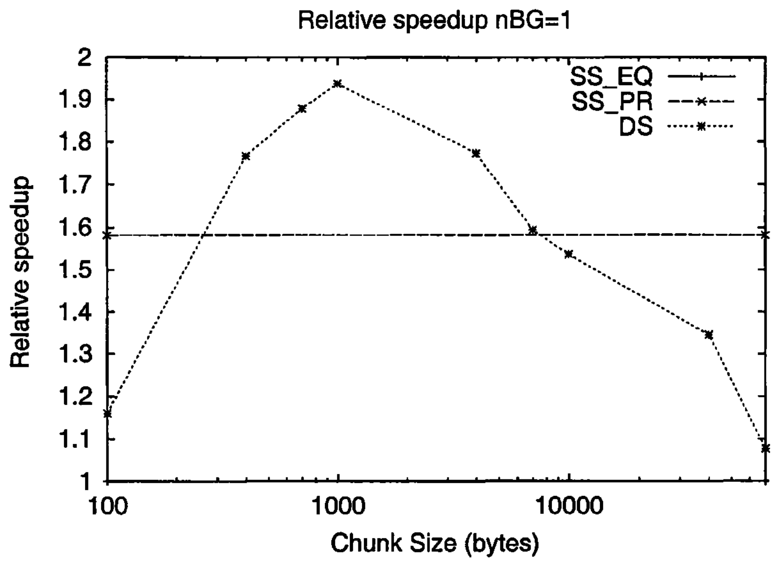

Figure 3.16: The relative speedup with inverse background load using the fast network (the number of background load is 1 process and SS_EQ is 1).

The relative speedup of SS_PR and DS with an inverse background load (the background loads presented here are 1 and 3 processes) using the fast network is shown in Figures 3.16 and 3.17, respectively. As described in Section 3.1, using inverse background load, the faster nodes have a lighter background load and the slower nodes have a heavier background load.

Between SS_EQ and SS_PR, SS_PR is 50%–60% better. We also conducted the experiments with other numbers of background processes. The relative speedups of SS_PR obtained are 1.30, 1.58, 1.55, 1.55, and 1.66 for the background loads ranging from 0 to 4 processes. The reason for this difference is the same as in the case with no background load. However, these values are much larger than in the case with no background load. This is because the slow nodes in SS_EQ become slower than those in SS_PR as a result of the inverse background load.

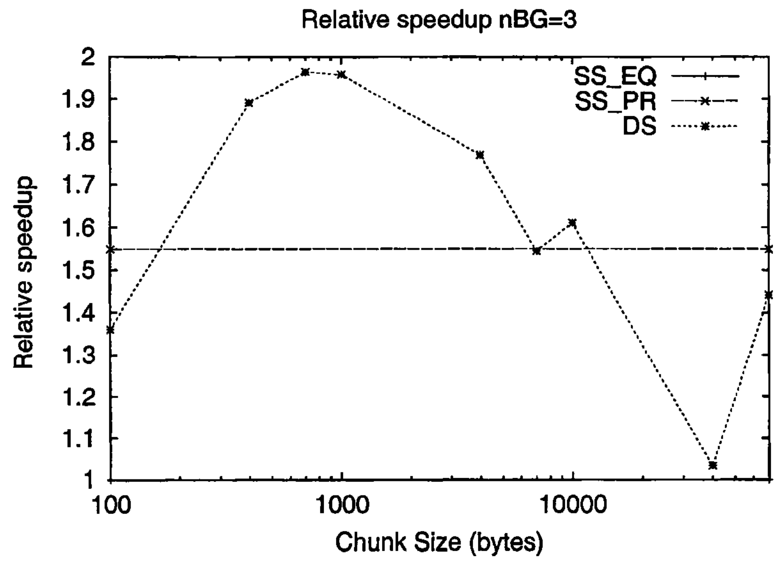

Figure 3.17: The relative speedup with inverse background load using the fast network (the number of background load is 3 processes and SS_EQ is 1).

DS outperforms SS_EQ by 80%-95% between 0.7KB (0.5% of 10,000 tuples in S) and 7KB (5% of 10,000 tuples in S). When these results are compared with the proportional background load used in Figures 3.14 and 3.15, the relative speedup is higher. This is because the slow nodes become slower in SS_EQ due to the inverse background load. In DS, the work that was done by the slow nodes in SS_EQ now goes to the fast nodes. DS also outperforms SS_PR by 20%-40% with the same chunk size range.

3.6.2 Conclusions for Nested-Loop Single-Join Algorithm

In this section, the performance of nested-loop join processing and its load balancing/sharing algorithms on a cluster are evaluated. Three load balancing/sharing algorithms are considered for parallel nested-loop join algorithms: static equal scheduling (the work distribution is equal on all nodes), static proportional scheduling (the work distribution is based on node processing data obtained prior to query execution), and dynamic scheduling with fixed-chunk size. These algorithms were evaluated on a heterogeneous cluster with two kinds of random background loads: proportional (more background load on faster nodes) and inverse (more background load on slower nodes) load using slow and fast networks. The results show that the dynamic scheduling outperforms both static equal scheduling and static proportional scheduling when we select an appropriate chunk size. For most of the cases in the experiments, the effective chunk size was from 0.7KB (0.5% of the S) to 7KB (5% of S). Using dynamic scheduling with this chunk size, a significant improvement over static scheduling, which is independent of the background load and the speed of the network, was obtained.

Even without background load, if the chunk size is appropriate, dynamic scheduling can obtain good improvements (up to 50%) over static equal scheduling. There is also a slight improvement over static proportional scheduling using slow (10 Mbps) and fast (100 Mbps) networks.

With proportional background load and a specific quantity of proportional background load (in this experiment, 2), static equal scheduling performs better than static proportional scheduling, since the processing power of each node can be seen as equal. The performance of dynamic scheduling with proportional background load demonstrates better improvement over static equal scheduling than no background load case. With inverse background load, the improvement over static equal scheduling is much higher than proportional background case.

3.6.3 Results for Hash Join

The experimental results for the hash join and its load balancing/sharing algorithms, described in Section 3.3, are presented in this subsection.

3.6.3.1 Default Parameter Values for *ChunkHJ*

The default values for the algorithm *ChunkHJ* used in the experiments are shown in Table 3.4. These results are obtained from several preliminary experiments. These values are chosen so as to achieve a reasonable performance in all cases.

Parameter Name	Value
Threshold values when $\|S\| = 10^5$ (the number of tuples)	100,000(Stop), 4,000(High), 1,000(Low)
Threshold values when $\|S\| = 10^6$ (the number of tuples)	100,000(Stop), 10,000(High), 1,000(Low)
Transfer policy	the master is always used (TP-MA)
Initial load balancing policy	Virtual Processor Round Robin (VP-RR)
The number of S chunks from the DBMS to the master	10
The number of buckets (virtual processors)	100

Table 3.4: Default parameter values for *ChunkHJ*.

3.6.3.2 Performance with No Data Skew and No Background Load

Figure 3.18 shows the execution time of *ChunkNJ* and *GRACE+* when the number of tuples of S is varied from 10^4 to 10^6. Figure 3.18 shows the execution time of *ChunkNJ* when the chunk size is varied from 1KB to 10KB. These chunk sizes are set according to the results in Section 3.6.1. For the purpose of comparison, the figure also includes the performances of static load balancing algorithms SS-EQ-NJ and SS-PR-NJ from Section 3.2.2. [4]

The results show that *ChunkNJ* is very effective compared to *SS-EQ-NJ* and *SS-PR-NJ* as the size of relation S increases. These results support the conclusion in Section 3.6.2 with the change of the local join algorithm; the dynamic scheduling in Section 3.2.2.2 used a nested-loop join for its local join algorithm and *ChunkNJ* uses a hash join algorithm. The improvement of *ChunkNJ* over *SS-EQ-NJ* and *SS-PR-NJ* is also due to its adaptive nature, which is the same as DS. This improvement is important even with no data skew since clusters are heterogeneous in general as described in Section 1.1. The adaptive algorithm automatically assigns the work according to the real processing speed of each PN reflected by its CPU speed, the size of main memory, and other factors. Another reason for this improvement is that *SS-EQ-NJ* and *SS-PR-NJ* have to wait for both relations to be read completely from

[4]The NJ suffix is added to indicate they are based on a global Nested-loop join.

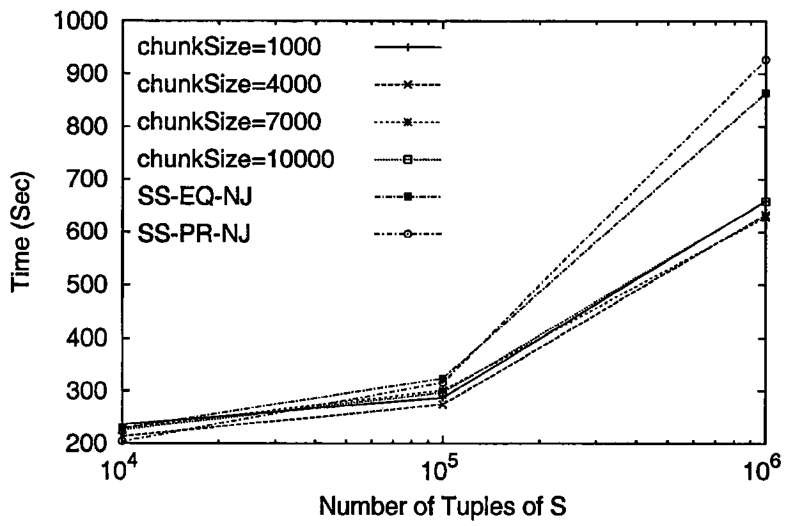

Figure 3.18: The performance of *ChunkNJ* with no data skew and no background load.

the DBMS prior to join execution. On the other hand, *ChunkNJ* can start working on the joins before the two relations are read completely.

As for the appropriate chunk size selection of *ChunkNJ*, 4KB and 7KB are better than other chunk sizes. However, the performance difference when the wrong chunk size is chosen is small (less than 10% of the entire execution time). This means that finding the optimal chunk size is not necessary to obtain good performance.

Figure 3.19 shows the execution time of *GRACE+* when the chunk size is varied from 1 to 10. It should be noted that the chunk size is based on the number of buckets, not the number of tuples as in *ChunkNJ*. The figure also includes the performance of *SS-EQ-HJ*[5]. The results show that *GRACE+* is better than *SS-EQ-HJ*. However, the performance differences are not as large as in Figure 3.18. This is because both *SS-EQ-HJ* and *GRACE+* have to wait for both relations to be read completely prior to the join execution. The waiting time dominates the whole execution time as the number of tuples of S increases.

[5]*SS-PR-HJ* was not tested since it does not make sense in this load sharing algorithm

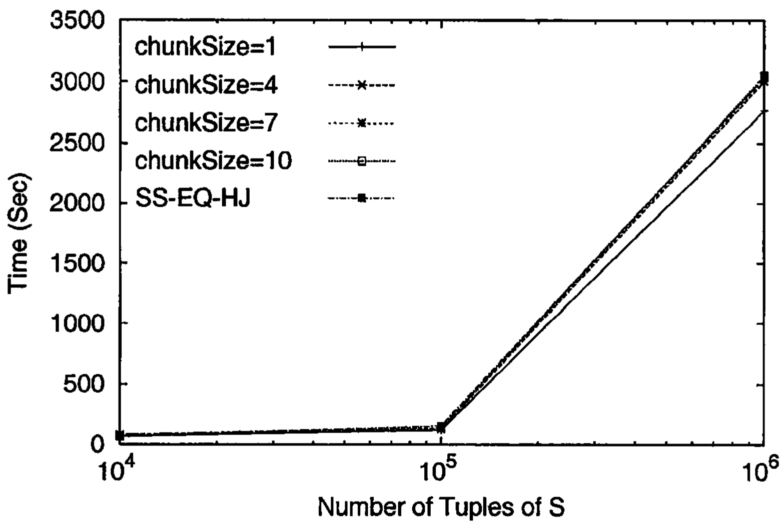

Figure 3.19: The performance of *GRACE+* with no data skew and no background load.

As for the appropriate chunk size selection of *GRACE+*, the chunk size of 1 is the best. When the number of tuples in S is 10^6, each S bucket has 10^4 tuples, since the number of buckets is 100. Other chunk size selections are too large to obtain good performance.

Figure 3.20 shows the execution time of *ChunkHJ* when the number of tuples of S is (a) 10^5 and (b) 10^6. In both figures, the LT value (or HT value) is fixed and the HT value (or LT value) is varied. Figure 3.20 shows that there is no significant difference in performance when the LT value is varied. The LT value is a system dependent value determined by PNs' processing and communication speed. Figure 3.20 also shows significant differences when the HT value is varied. The HT value makes a significant difference in the performance; if it is too high there is no load sharing in the transfer process; if it is too small, too many transfers occur and degrade the performance.

Figure 3.21 compares the performance of the transfer policies described in Section 3.3.2.2. The result shows that transfer policies *TP-MA* and *MP-MI* are superior to other policies. The reason for this is that our experimental system is quite small.

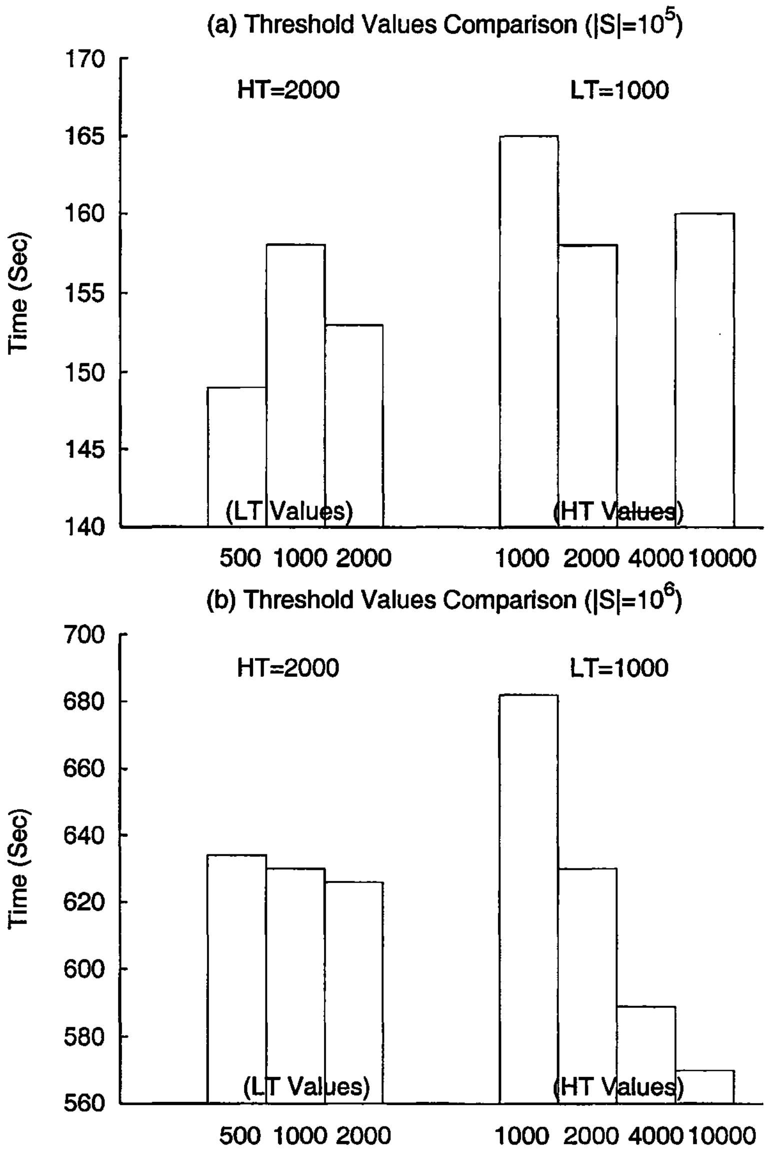

Figure 3.20: Comparison of threshold values with no data skew and no background load.

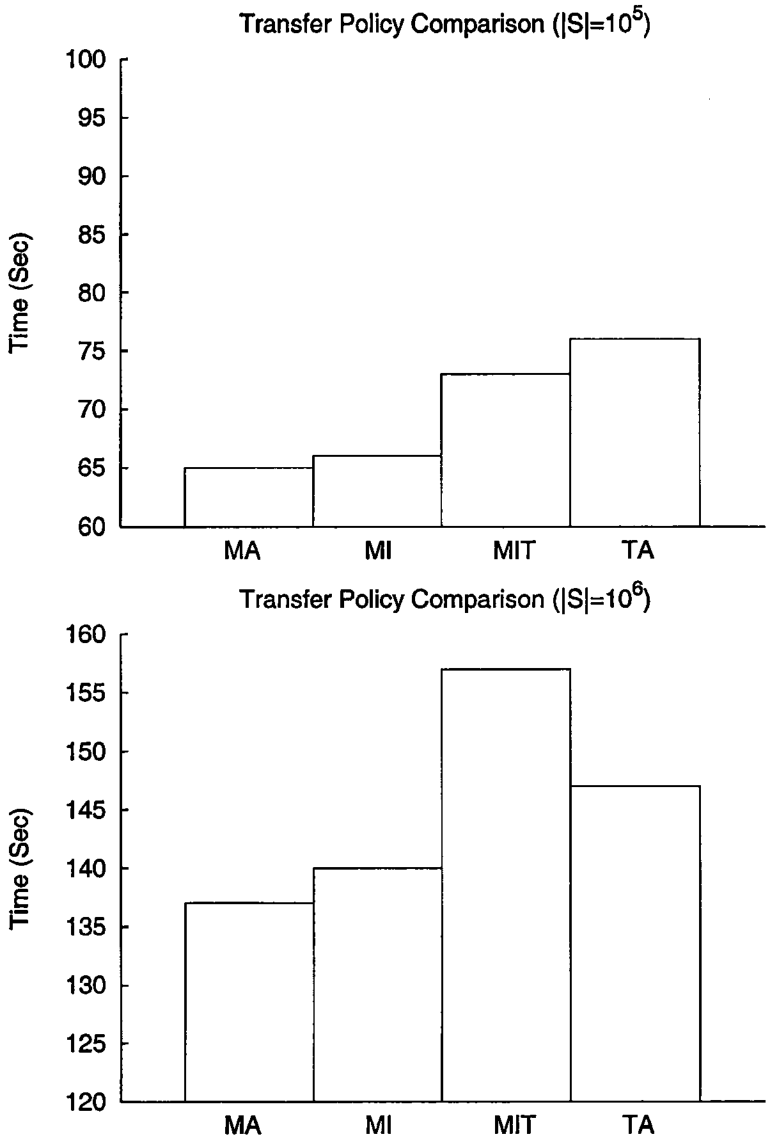

Figure 3.21: The comparison of transfer policies with no skew and no background load. The description of each algorithm is given in Section 3.3.2.2.

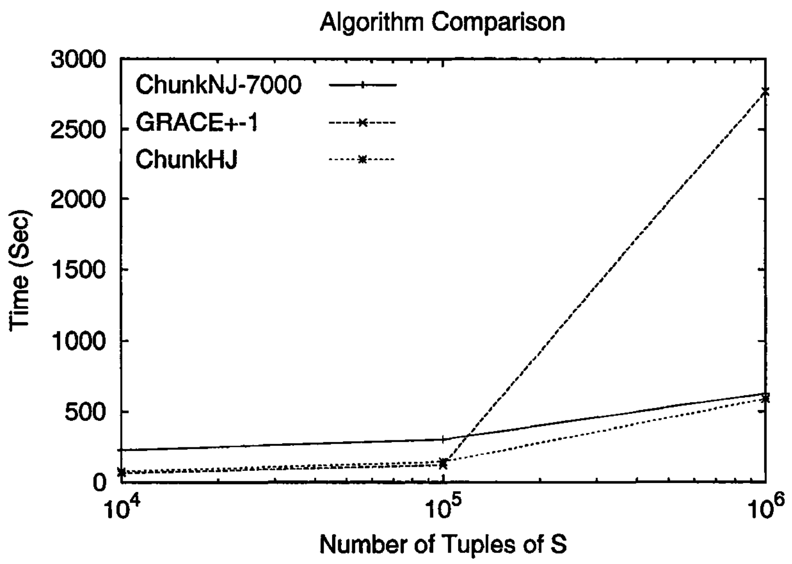

Figure 3.22: The comparison of load sharing algorithms with no skew and no background load.

Figure 3.22 compares the performance of the three load balancing/sharing algorithms. *ChunkHJ* is the best as the number of tuples of S increases. *ChunkNJ* also performs well, since it has an adaptive feature. *GRACE+* performs as well as *ChunkHJ* when $|S| <= 10^5$. After that, the overhead to read the large S relation dominates the overall execution time.

From these preliminary results, the following comparisons are made: (a) comparing *ChunkNJ*, *GRACE+*, *ChunkHJ* in the case when $|S| = 10^5$ and (b) comparing *ChunkNJ* and *ChunkHJ* in the case when $|S| = 10^6$ in the presence of data skew and background load.

3.6.3.3 Performance with Data Skew and No Background Load

In this experiment, the skew factor of one of the relations is fixed at 1 (no skew) and the the skew factor of the other relation is varied. This scenario can occur when one of the join attributes is a foreign key. The case where both relations are skewed is evaluated in Section 3.6.3.5.

Case A: $|S| = 10^5$ case Figure 3.23 compares the performance of the three load sharing algorithms with data skew in (a) the building relation R and (b) the probing relation S without any background load. *ChunkNJ* is the worst of the load sharing algorithms in most of the cases. However, its performance degradation when changing the skew factor and skewed relation (either the building or probing relation) is small because it is based on nested-loop join with an adaptive feature. *GRACE+* suffers as the skew factor increases. However, it is superior to other algorithms when the relation has a modest skew. *ChunkHJ* does not suffer the extreme skew as much as *GRACE+* because of its transfer mechanism, which softens the skew effect on the fly. Thus, its performance is rather flat regardless of the skew factor, except for performance degradation in the case of extreme skew in the building relation. The reason for this degradation is that there are too many transfers in this case; the average amount of the total transfer from the master to slave is $1*10^4$, $1*10^4$, and $3*10^5$ for skew factors 10^3, 10^4, and 10^5 of building relation R and $1*10^4$, $8*10^4$, and $8*10^4$ for skew factors 10^3, 10^4, and 10^5 of probe relation S, respectively. However, the degradation is not as large as in *GRACE+* .

Case B: $|S| = 10^6$ case Figure 3.24 compares the performance of *ChunkNJ* and *ChunkHJ* with data skew on (a) the building relation R and (b) the probing relation S without any background load when $|S| = 10^6$. *ChunkHJ* is better than *ChunkNJ*, and both algorithms exhibit a flat performance. The reason for this is the same as in the case where $|S| = 10^5$. The only exception is the extreme skew in the building relation. The skew factor is 10^5. This means that 10% of the whole relation has the same value, which happens quite rarely in real data. Thus, in the following experiments, the performance effects on the load sharing algorithms of the skew factor on S are focused on.

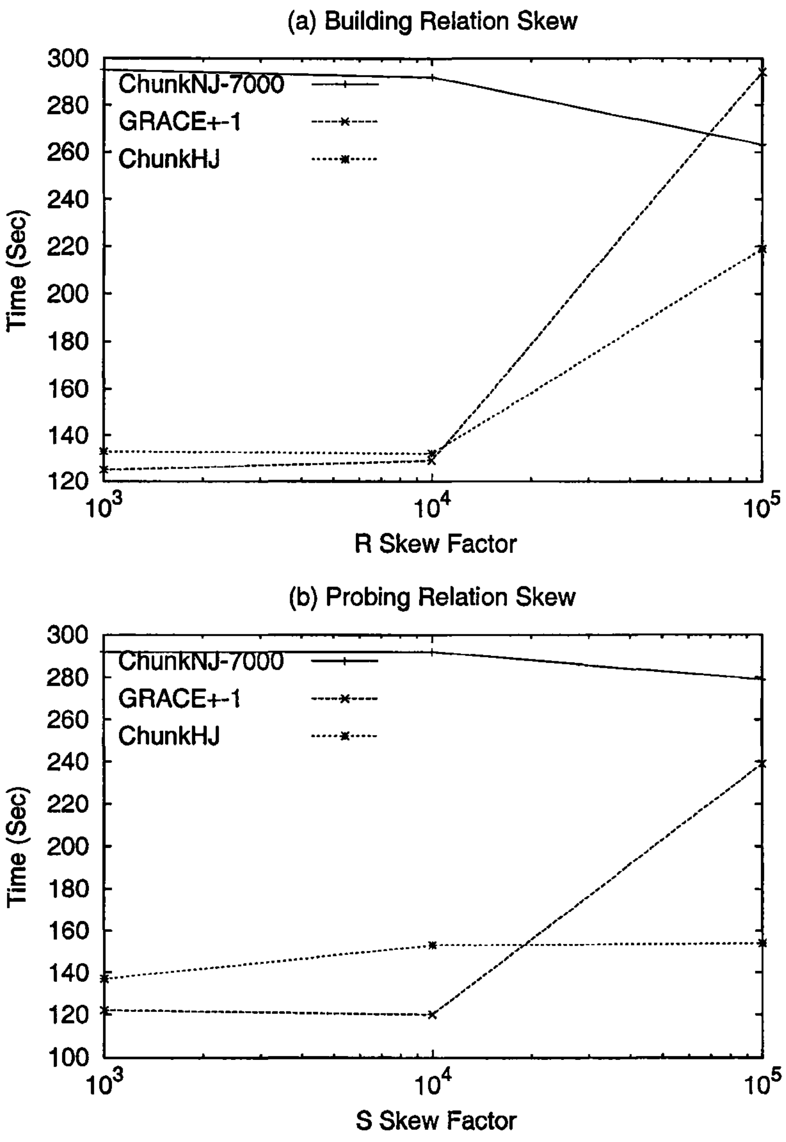

Figure 3.23: The performance of load sharing algorithms with data skew on (a) building relation R and (b) probing relation S, without any background load ($|S| = 10^5$).

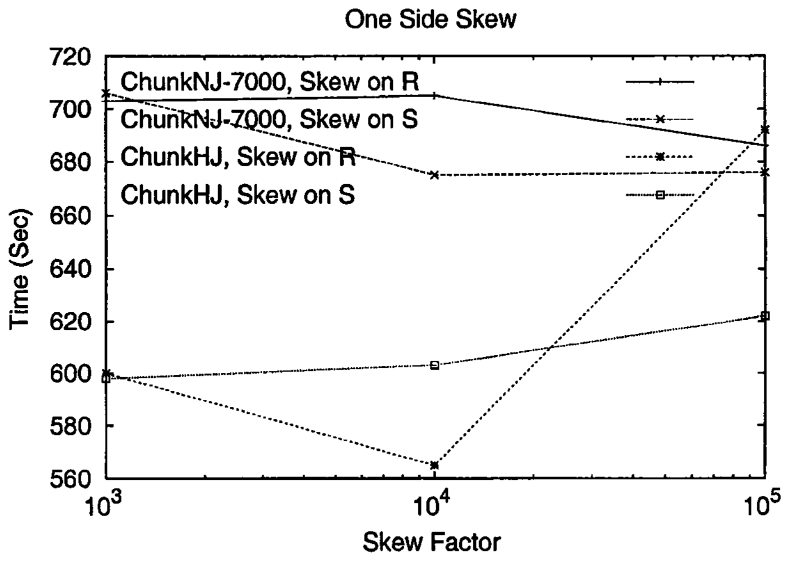

Figure 3.24: The performance of the load sharing algorithms with data skew on building relation R and probing relation S without any background load ($|S| = 10^6$).

3.6.3.4 Performance with Data Skew and Background Load

In this section, in addition to the data skew, background load is added to the slaves. As described in Section 3.1.2, two types of background load were implemented: proportional background load and inverse background load. Using proportional background load, faster PNs receive a higher load than slower PNs. Using inverse background load, slower PNs receive a higher load than the faster PNs.

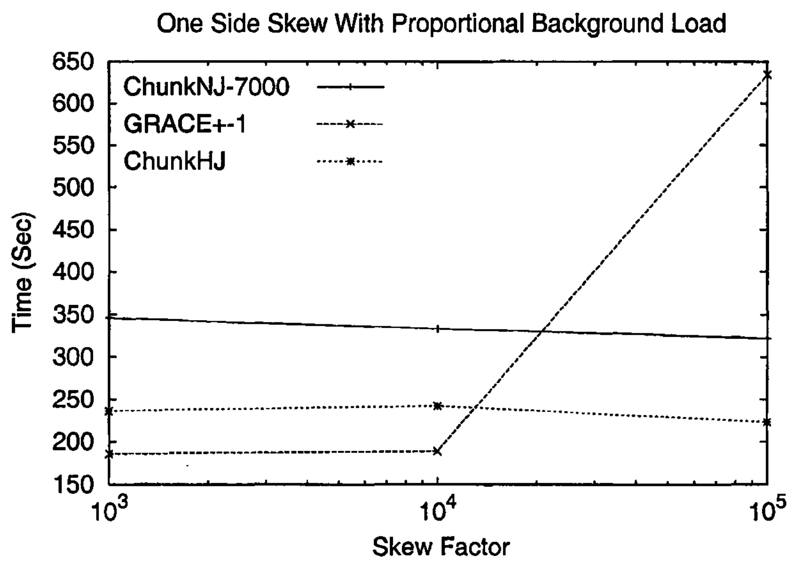

Figure 3.25: The performance of the load sharing algorithms with data skew on the probing relation S and with proportional background load ($|S| = 10^5$).

Case A: $|S| = 10^5$ case Figure 3.25 shows the performance of load sharing algorithms with data skew on probing relation S with proportional background load on the PNs when $|S| = 10^5$. Compared to Figure 3.23(b) in which there is no background load, the slowdown of the extreme skew case of *GRACE+* is larger, since the fast PNs become slower due to the proportional background load and the skewed buckets are likely to be transferred to the fast PNs. Alternatively, *ChunkHJ* helps the PNs by transferring the work to another idle PN. The number of such helping PNs is decided on adaptively. Thus, even with proportional background load and data skew, the performance of *ChunkHJ* is almost the same.

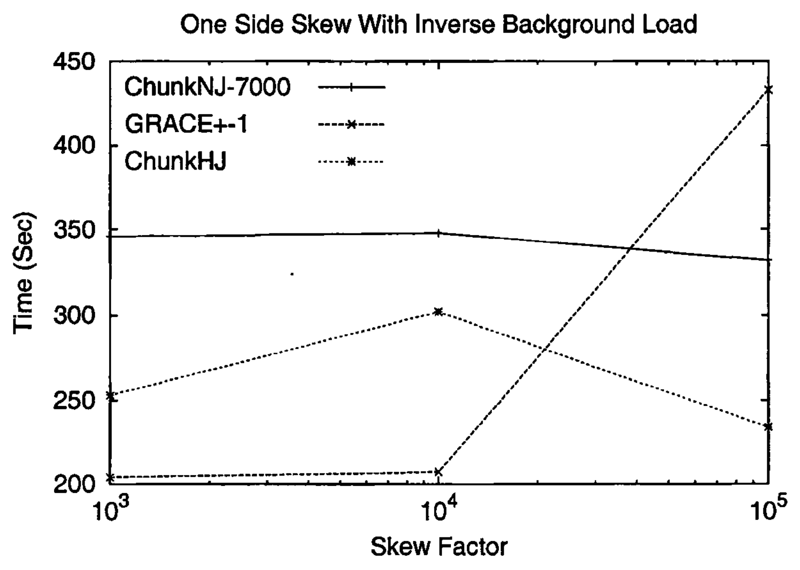

Figure 3.26: The performance of the load sharing algorithms with data skew on the probing relation S and with inverse background load ($|S| = 10^5$).

Figure 3.26 shows the performance of load sharing algorithms with data skew on the probing relation S with inverse background load when $|S| = 10^5$. This figure exhibits the same tendency as when proportional background load is used; *ChunkHJ* has a flat performance even with data skew and inverse background load. *GRACE+* does not suffer as much as in the case shown in Figure 3.25, since the fast PNs, which work on the skewed buckets, are not affected much by the presence of the inverse background load.

Case B: $|S| = 10^6$ case Figure 3.27 shows that *ChunkHJ* is superior to *ChunkNJ* under proportional and inverse background load ($|S| = 10^6$). In this figure, "PBG" means proportional background load and "IBG" means inverse background load.

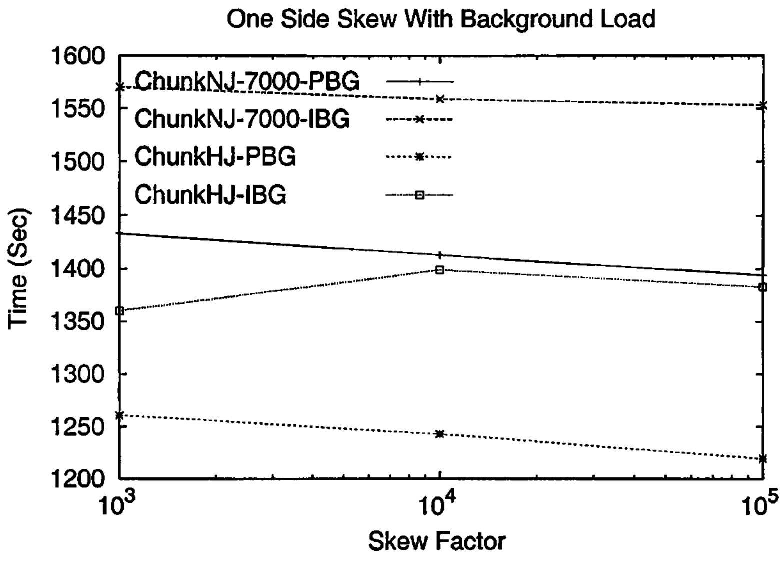

Figure 3.27: The performance of the load sharing algorithms with data skew on probing relations and with background load ($|S| = 10^6$).

3.6.3.5 Performance with Data Skew on Both Relations and Background Load

Figure 3.28 shows the performance of load sharing algorithms with data skew (skew factor is 10^3) on the building and probing relations and with no (on the left), proportional (in the centre), and inverse background load (on the right) when $|S| = 10^5$. When both relations have skew, join product skew occurs. When there is no or proportional background load in Figure 3.28, *ChunkHJ* can deal with the product skew. However, when there is inverse background load, *GRACE+* performs well. Overall, *ChunkHJ* outperforms the other two load sharing algorithms.

3.6.4 Conclusions for Hash Single-Join Algorithm

This subsection proposed and evaluated a new hash join algorithm, called the adaptive chunking hash join algorithm for a cPDBMS. Previous load sharing/balancing algorithms may not work well on clusters since (1) they deal with pre-partitioned

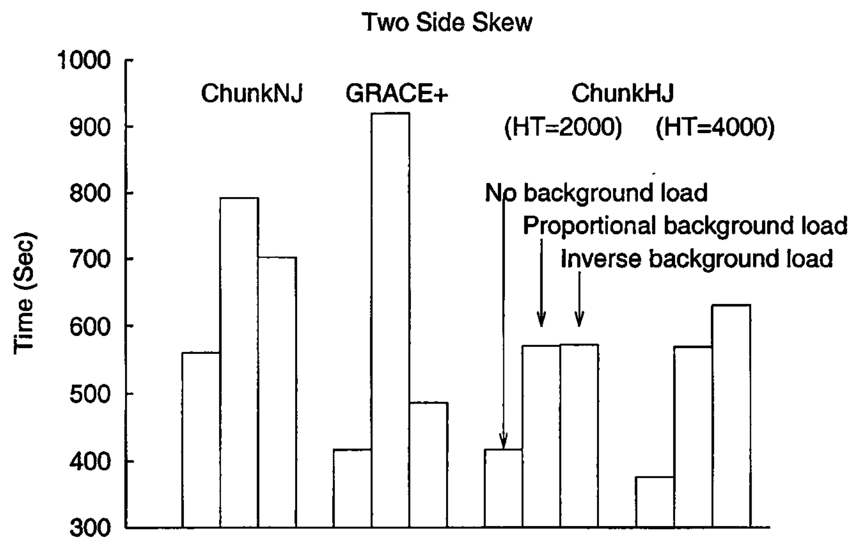

Figure 3.28: The performance of the load sharing algorithms with data skew (skew factor is 10^3) on building relation S and probing relation S and with background load ($|S| = 10^5$).

data, (2) they do not consider the effect of background load, and (3) they do not correct erroneous predictions about the data distribution. The new algorithm uses combinations of chunking and hash join algorithms to deal with such situations. Using threshold values, the algorithm creates several chunks of buckets and assigns them to processing nodes both dynamically and adaptively. This algorithm, along with two other algorithms, namely global adaptive nested-loop join and global adaptive GRACE, were evaluated on a cluster. The conclusion is that the chunk-based hash join is effective in most of the cases, when the size of input relations, the skew of the relations, and the background loads vary. In cases where both relations are small and have modest skew, global adaptive GRACE works more effectively.

Chapter 4

Symmetric Single-Join Algorithms

In this chapter, symmetric single-join algorithms and their load balancing/sharing techniques are studied. One of the problems with the non-symmetric joins proposed in the previous chapter are that these join algorithms require at least one of the join relations (main relation) to be read completely before performing the join. If one of the relations is relatively small and resides locally, these algorithms are effective. However, if the main relation is delayed because of data arrival delay and/or data transfer rate fluctuation due to the Internet characteristics, the delay may cause significant performance degradation. Or if the sizes of the relations are not known, the main relation cannot be selected. Another problem with the non-symmetric joins is that, in the case of continuous query processing (e.g., a search query on the search engine), when non-symmetric join algorithms are used, the time to produce the very first result is delayed.

Due to these reasons, as mentioned in Section 2.4, the symmetric hash join algorithms were proposed by Wilchut et al. [88]. Also, recently developed algorithms for data integration systems or continuous query processing (i.e., XJoin [84] and Ripple Join [58]) are based on symmetric single-join algorithms.

In this chapter, we combine symmetric hash join algorithms with the ChunkHJ algorithm in the previous chapter. Also, we compare its results with other traditional load balancing/sharing algorithms.

Evaluation of these algorithms is done under the following conditions to reflect differences in the target platforms used in this chapter and the previous chapter, namely:

- increase in the number of tuples;
- inclusion of SMP nodes in a target cluster;
- introduction of relation data arrival delay and/or relation data transfer rate fluctuation[1] because of network characteristics on the Internet.

We first explain the environment that we used in this chapter, then, the load balancing/sharing algorithms are described. Next, experimental environments, including the Internet transfer delay model, are discussed along with the differences between these and previous chapter. Finally, experimental results and conclusions are presented.

4.1 Symmetric Single-Join Processing Environment

This section presents the environments for symmetric single-join processing. It focuses on the difference between Section 3.1.1 and this section.

4.1.1 A System Model

The target model is shown in Figure 4.1. The differences between Figure 3.1 in the previous chapter and Figure 4.1 are as follows:

- Some of the PNs within a cluster are dual processors (or SMP).
- Relation data may come from geographically distributed databases. Relation data arrival delay and/or transfer rate fluctuation may occur because of dynamic characteristics of the Internet traffic.

4.1.2 Symmetric Single-Join Processing Architecture

Software Architecture Design The software architecture introduced in this chapter is based on the previous chapter (Section 3.1.2) but extended to be more general

[1]From now on, they are called "the Internet transfer delay" together.

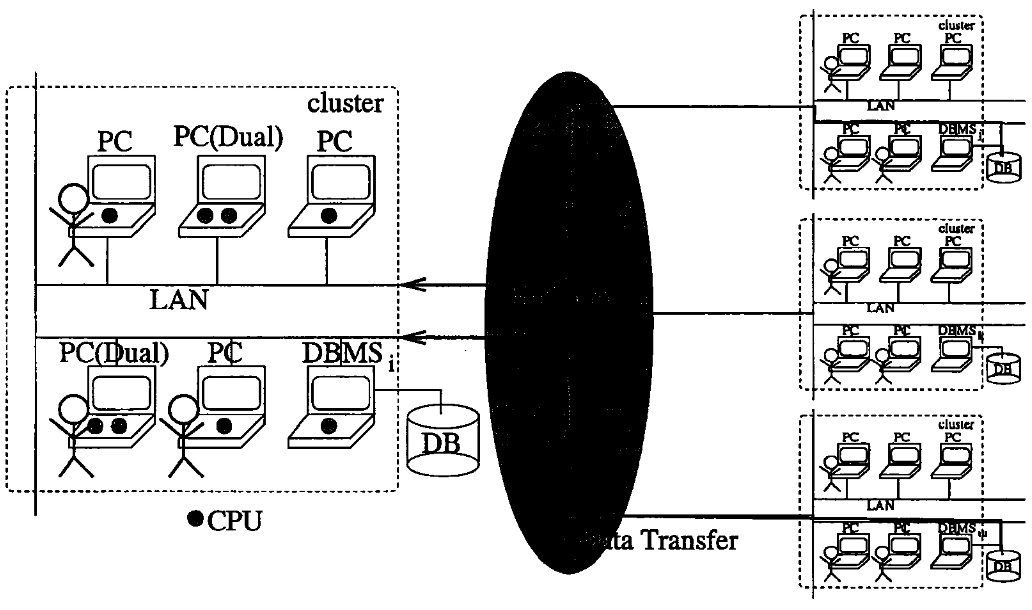

Figure 4.1: A system model with the Internet transfer delay.

purpose (e.g., multiple-join). The following are the major differences from the architecture in Section 3.1.2:

- We rename the components as shown in Table 4.1.
- Relations are sent directly from the Database (DatabaseReader) to the JoinExecutor. This reduces the number of messages instead of going through the JoinManager.
- Relations are read from the Database simultaneously using DatabaseReader threads.

The whole system is designed to simulate query processing on a cluster as in Section 3.4.2. The major components are shown in Figure 4.2 where the numbers between components (classes) show the relationship in terms of the number of instances. A description of each component and its number of instances are shown in Table 4.2.

There are two important tables used for the symmetric hash join algorithms: a join state matrix and hash bucket chunk allocation table.

Chapter 3	Chapter 4
master	JoinManager
slave	JoinExecutor
transfer	TransferExecutor
DBMS	DatabaseManager (DatabaseReader)

Table 4.1: Name changes.

A Join State Matrix (JSM) resides on the JoinManager and is used to ensure the correctness of the join execution. Each entry in the JSM represents the join status of matching pairs for each bucket. Figure 4.3 shows an example of a join state matrix snapshot for R $\bowtie$ S. It is obvious that the number of JSM entries for each hashId varies according to the arrival speed and the degree of skew of the relations.

A Hash Chunk Allocation Table (HCAT) keeps track of the allocation information in the hash table (e.g., to find matching bucket pairs). An HCAT is updated every time a bucket is moved from one PN to another PN. A HCAT also resides on the JoinManager.

Execution Flow The main part of execution flow of the simulation program is shown in Figure 4.4. The details of the communications during the join execution among the components are also shown in this figure. The whole execution flow is as follows:

1. A query is generated by the QueryGenerator according to predefined parameters provided in a configuration file.
2. The query is then sent to the QueryScheduler which makes a processor allocation decision for the query according to the current cluster state.
3. The QueryScheduler invokes a QueryManager for the query.
4. The QueryManager invokes several JoinManagers, one for each join in the query.
5. The JoinManager invokes several JoinExecutors. The number of JoinExecutors is determined by QueryManager according to the current cluster state. A

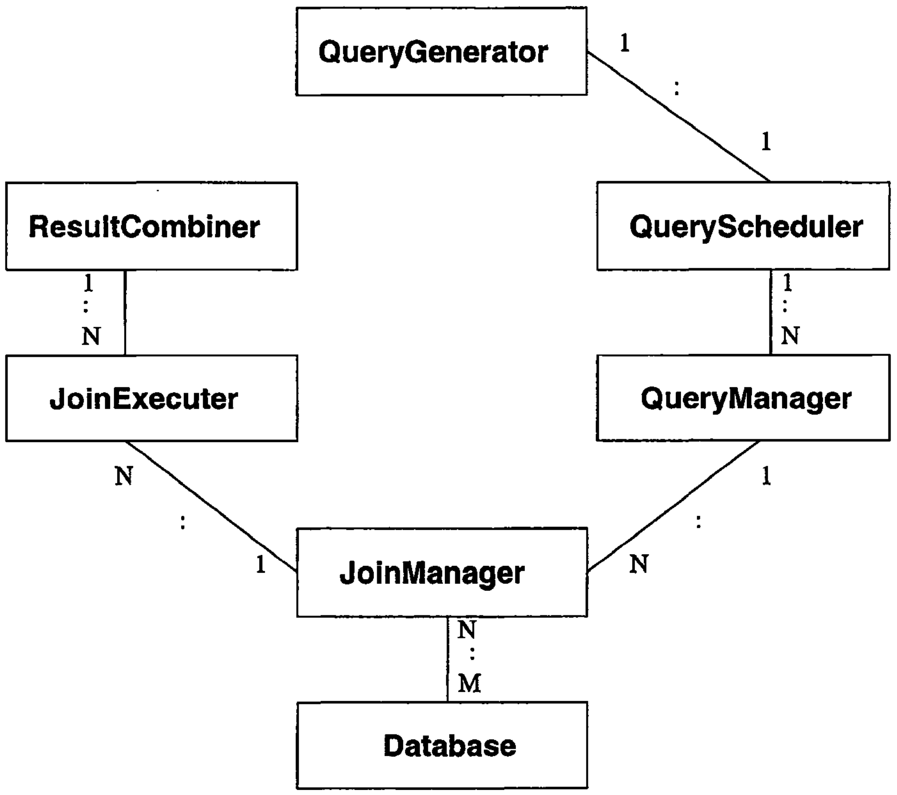

Figure 4.2: Major components.

TransferExecutor is also invoked on the same nodes as the JoinExecutor.

6. The JoinManager consults the Database to read the relations associated with the join.

7. The Database invokes a DatabaseReader, a HashGenerator, a TransferExecutor for each relation needed to execute the join.

8. The DatabaseReader starts reading the relation page by page and puts each page into the read buffer.

9. The HashGenerator reads the page from the read buffer, applies a hash function to each tuple in the page.

HashId0

S \ R	0	1	2
0	F	F	F
1	R	J	R
2	R	J	R

HashId1

S \ R	0	1
0	F	F
1	F	F
2	R	R

HashId2

S \ R	0	1	2
0	F	F	F
1	R	J	R

Figure 4.3: An example of join state matrix snapshot: (R:ready, J:joining, F:finished). The hash value of each hash bucket chunk is given by hashId.

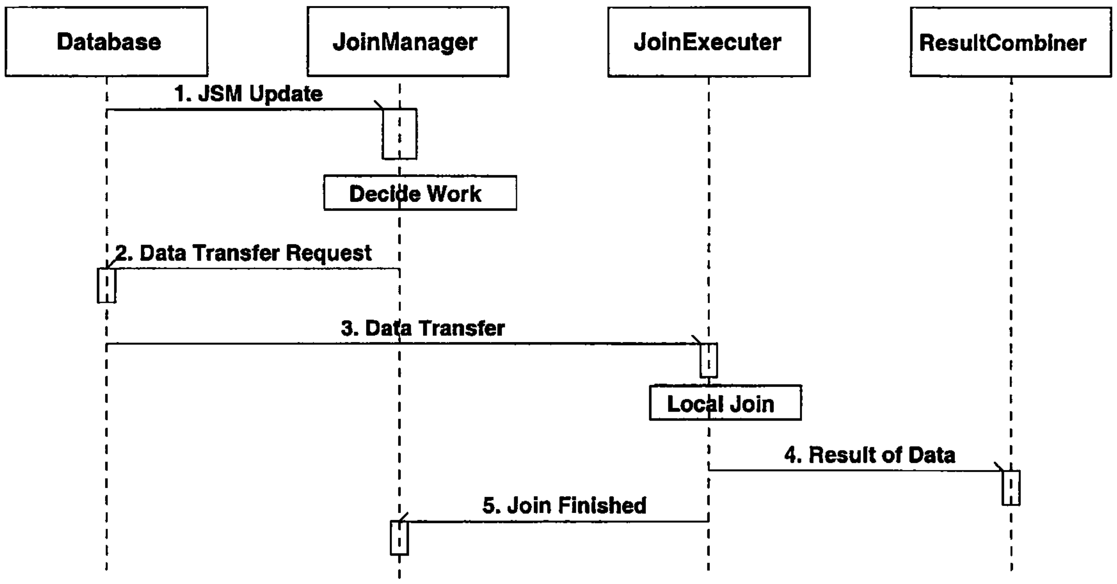

Figure 4.4: Message sequence chart during join execution.

Class Name	Abbr.	Number of instances	Description
TaskAllocator	TA	one for each node	allocates other components
QueryGenerator	QG	one	generates queries at certain interval
QueryScheduler	QS	one	schedules queries
QueryManager	QM	one for each query	manages query makes allocation decision
JoinManager	JM	one for each join	manages join makes load-balancing decision
JoinExecutor	JE	determined by QM	executes local joins
TransferExecutor	TE	one for JE one for DB	transfers chunks
DatabaseManager	DB	one	accepts a read request invokes DRs
DatabaseReader	DR	one for each relation	reads a relation puts it into the buffer
BackgroundGenerator	BGG	one for each node	generates BGs
Background	BG	determined by BGG	executes busy loop simulates a non-query process
HashChunkStore	HCS	one for each node	stores the chunk deals with disk I/O
ResultCombiner	RC	one for each query	collects all results

Table 4.2: Major components descriptions.

10. When a hash bucket size of the hash table reaches the predefined chunk size (refereed as *chunk size*), the HashGenerator sends a JSM update message to the JoinManager (message 1 in Figure 4.4) and puts the hash bucket into the HashChunkStore.

11. The JoinManager inserts a JSM entry for the hash bucket according to the JSM update message. At the same time, it chooses the target JoinExecutor according to the load balancing/sharing algorithms described later in this section and sends the target JE id back to HashGenerator (message 2 in Figure 4.4).

12. The TransferExecutor sends the bucket from the HashChunkStore to the target JoinExecutor chosen by the JoinManager (message 3 in Figure 4.4). Its implementation is described in Figure 4.5.

13. The JoinExecutor starts executing the join after receiving the bucket ("Local Join" in Figure 4.4).

14. The results of the join are sent to the ResultCombiner when the number of result tuples reaches a threshold value (message 4 in Figure 4.4)

15. The JoinExecutor sends a "Join-Finished" (and next job request) message to the JoinManager (message 5 in Figure 4.4).

16. The JoinManager determines the next job and send it to the JoinExecutor using receiver-initiated algorithm similar to the one described in Section 3.3.2.2.

Implementation Details All components described in Figure 4.2 are implemented by Java threads, which run concurrently on PNs.

A HashChunkStore on a PN is shared by several threads (HashGenerator, JoinExecutor and TransferExecutor). It stores all the hash chunks and is used for the local hash join. When the memory is full, a victim hash chunk is selected and flushed into the disk. The Least Recently Used (LRU) algorithm is used to select the victim chunk. A detailed look at the data transfer is given in Figure 4.5. In this architecture, there are two kinds of bucket transfer: one is from Database to JoinExecutor and the other is from JoinExecutor to JoinExecutor.

A BackgroundGenerator on each node generates several idle threads (Background threads) for simulating background non-query processing tasks. It is implemented by Timer Java class. The types of background are the same as in Section 3.1.2. However, we use identical background loads and a shorter time (period and on-time) to reflect the hardware difference.

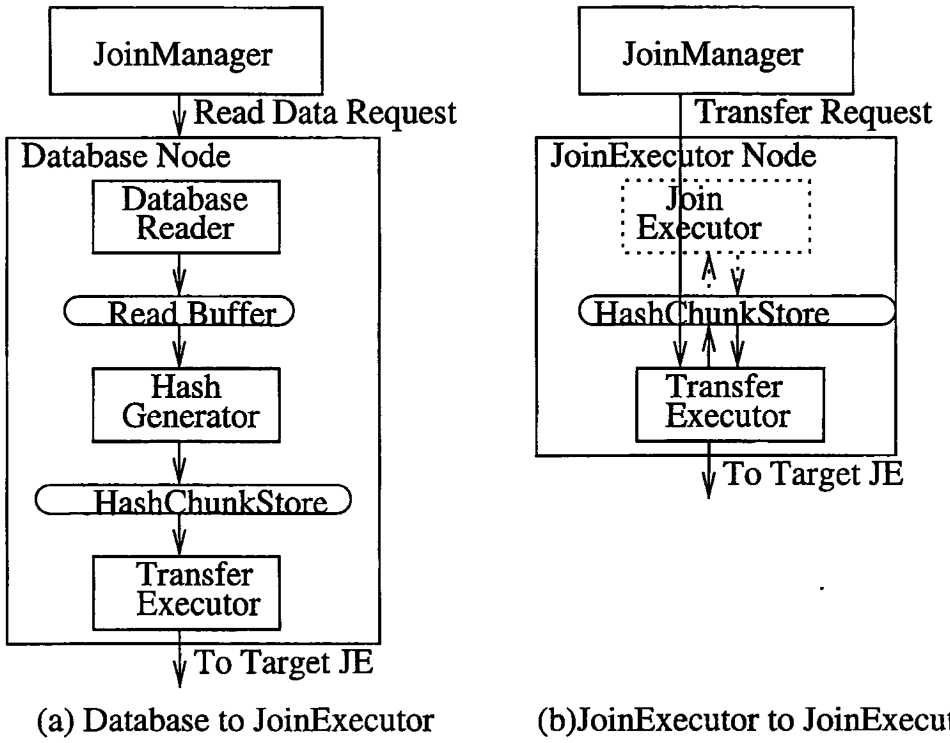

Figure 4.5: A detailed look at the data transfer: (a) from Database (DatabaseReader) to the target JoinExecutor and (b) from a source JoinExecutor to the target JoinExecutor.

4.2 Load Sharing/Balancing Techniques for Symmetric Hash Join Algorithms

We designed and implemented several load balancing/sharing algorithms for symmetric hash joins. The main focus is to decide hash mapping[2] from hashId to JoinExecutorId. It is stored in *hashMappingTable* for load sharing/balancing. We developed five algorithms to the decide the hash mapping. The following subsections explain these algorithms in detail using the notation in Chapter 3 (Table 3.1). The differences among these algorithms are summarized in Table 4.3.

In Table 4.3, the column labeled *Hash mapping* determines the assignment from

[2]In Section 2.3.1.3, it is called the partition scheme.

Abbr.	Name	*Hash Mapping*	*Decision*	*Experiments*
RR	Dynamic Round-robin Hash Mapping	none	none	BG, Skew
SCHJ	Symmetric ChunkHJ	none	none	Delay, BG, Skew
SMP	Dynamic Sampling	Dynamic	tuples	BG, Skew
GIHM	Greedy Incremental Hash Mapping	Incremental	chunk	Delay, BG, Skew
JIHM	JSM-based Incremenatal Hash Mapping	Incremental	JSM	Delay, BG, Skew

Table 4.3: Algorithm summary.

a hash bucket to a JoinExecutor. In the implementation, if *none*, it is not used. In *Dynamic*, hash mapping is determined at one point of the execution all at once. In *Incremental*, the hash mapping is determined incrementally during the join execution.

The column labeled *Decision* describes the factors that are considered in making the decision for hash mapping. In *tuple*, the number of tuples is considered. In *JSM*, the number of ready entries in a JSM ("R" in Figure 4.3) is considered. In *chunk*, hash chunk arrival timing is considered.

The column labeled *Experiments* describes the condition under which these algorithms are tested. These decisions are made according to the features of each algorithm. In *BG*, experiments are done with non-query background load. In *Skew*, experiments are done with database skew (scalar skew and Zipf skew). In *Delay*, experiments are done with the Internet transfer delay, which will be discussed in Section 4.3.4.

The following algorithm description is related to "Decide Work" in the JoinManager component (see Figure 4.4). The pseudocode for JoinManager is shown in Algorithm 4.2.1. The pseudocode for HashGenerator is also shown in Algorithm 4.2.2. *expandJSM* is used to expand the JSM entry according to the argument *JSMInfo* (a pair of hashId and chunkId) sent from *HashGenerator*. It also fills the expanded matrix entries with "R"(Ready) entry. *findJoinExecutor* is the sender-initiated part of the algorithm that tries to find an idle JoinExecutor with the maximum number of matching pairs for this bucket. Each algorithm has a different version of findJoinExecutor as we will explain later. *FindBucketForSymmetricJoin* is used to find a suitable bucket pair with an idle JoinExecutor. For the hash mapping-based algorithm (non-SCHJ algorithms), hash mapping is used to find a suitable bucket pair. For SCHJ, the algorithm described in Section 4.2.2 is used.

RSTransferForSymmetricJoin is used to transfer the bucket among DB and JoinExecutor. The details are shown in Section 4.2.2 as well.

```
Algorithm 4.2.1: SYMMETRICJOINONJOINMANAGER($cSize$)

{manages join and is responsible for load sharing/balancing by JSM}
{Input: cSize for chunkSize; Output: none}

{start reading relations}
$send(HashGenerator,$ "RelationRead", "R")
$send(HashGenerator,$ "RelationRead", "S")
repeat
    $recv(source, tag, JSMInfo)$
      {JSMInfo consists of hashId and chunkId}
    if ($tag$ is "JSMUpdate")   {from HashGenerator}
      then
          $expandJSM(JSMInfo.hash, JSMInfo.chunkId)$;
          $JEx \leftarrow$
          $findJoinExecutor(JSMInfo.h, JSMInfo, chunkId, cSize)$;
          $send(source,$ "JSMUpdateReply", $JEx)$
      else if (tag is "JobRequest")   {from JoinExecutor}
      then
          change corresponding completed JSM entries to "F"
          $chunkPair \leftarrow findBucketForSymmetricJoin(cSize)$
            {chunkPair consists of hashId, chunkId1, chunkId2}
          $RSTransferForSymmetricJoin(chunkPair)$
until ((finished reading relations(R and S)) and (JSM entries are all "F"))
$broadcast($"ProcessEnd", $null)$   {to all JEs and HGs}
```

Pseudocode for JoinExecutor is shown in Algorithm 4.2.4.

Algorithm 4.2.2: SYMMETRICJOINONHASHGENERATOR($X, pS, cSize$)

```
{applies hash function on tuples of X}
{Input: relation X; pS for partition size for relation X;
        cSize for chunk size;
Output: none}

repeat
  $recv(JoinManager, \text{"RelationRead"}, X)$
      {Read relation X}
      {this is done by Database and DatabaseReader}
  $nPartitions \leftarrow |X|/pS$
  for $i \leftarrow 0$ to $nPartitions$
     do { $X^i \leftarrow read(X, i * pS, (i+1) * pS - 1)$
              {actual reading is done by DatabaseManager}
          $X^i_{ALL} \leftarrow applyHash(X^i, cSize)$
until $recv(JoinManager, \text{"ProcessEnd"}, null)$
```

Algorithm 4.2.3: APPLYHASH(X, *chunkSize*)

```
{apply a hash function to each tuple in X}
{Input: relation X (either relation R and S); Output: none}

for $i \leftarrow 0$ to $|X|$
  do
    $h \leftarrow HashFunction(X[i])$
    $X_h.insert(X[i])$
    if $(|X_h| > chunkSize)$
      then
        $JSMInfo.hashId \leftarrow h$
        $JSMInfo.chunkId \leftarrow X_h.getChunkId()$
        $send(JoinManager,$ "JSMUpdate", $JSMInfo)$
        $recv(JoinManager,$ "JSMUpdateReply", destJE)
        if $(destJE >= 0)$
          then $send(destJE,$ "Relation", $X_h)$    {send it directly}
        $hashChunkStore.insert(X_h)$
        $X_h.incrementChunkId()$
```

Algorithm 4.2.4: SYMMETRICJOINONJOINEXECUTOR()

```
{Symmetric Hash Join on a Slave(SLx)}
{Input: none; Output: none}

repeat
  $recv(TransferExecutor,$ "Relation", $X_h)$
  $if$(X is R)
    then $execLJ(X_h, S_h)$
    else $execLJ(R_h, X_h)$
  $send(JoinManager,$ "JobRequest", $null)$
      {to invoke sender-initiated algorithm}
until $recv(JoinManager,$ "ProcessEnd", $null)$
```

4.2.1 Dynamic Round-Robin (RR)

This algorithm is similar to VP-RR [21]. In this algorithm, hash mapping is not used. Each hash bucket is sent in a round-robin manner. The pseudocode of *findJoinExecutor(h,chunkSize)* of *RR* is shown in Algorithm 4.2.5. For example, if the number of JoinExecutors is 8 and hash bucket Id is 10, then the bucket is sent to JE_2 (10 mod 8 = 2). It is a simple algorithm and it may work well when there is no background load and no skew. Otherwise, this algorithm may not perform well. We will conduct experiments in Section 4.5 to verify this.

```
Algorithm 4.2.5: FINDJOINEXECUTOR_RR(h, chunkSize)

{Symmetric Hash Join on a JoinExecutor(JEx)}
{Input: h for hashValue; chunkSize
Output: destination JE index}
return (h mod p)      {round robin: p is the number of JEs}
```

4.2.2 Symmetric ChunkHJ (SCHJ)

This algorithm is based on ChunkHJ in Section 3.3.2.2. Thus, hash mapping is not used. As we described at the beginning of this chapter, we would like to combine advantages of ChunkHJ and symmetric joins. The differences from ChunkHJ are as follows:

- There is a modification for symmetric joins (use of JSM (Figure 4.3)).
- There is just one threshold value (no LT/HT/ST) to simplify the algorithm.
- Chunk transfer on a node is done by an independent thread (instead of a process) and done without involving the master (or JoinManager).

The pseudocode of *findJoinExecutor(h,cId,chunkSize)* of *SCHJ* is shown in Algorithm 4.2.6. FindSlave in Figure 3.7 is used; however, the buckets which have "R" entires in JSM are considered.

Algorithm 4.2.6: FINDJOINEXECUTOR$_{SCHJ}(h, cId, chunkSize)$

```
{find a JoinExecutor(JEx) using Chunk HJ}
{Input: h for hashValue; cId for chunkId; chunkSize;
Output: destination JE index}
$SLx \leftarrow findSlave(\text{“LT”}, h)$        {Figure 3.7 but make use of JSM}
if (SLx is null)
  then $SLx \leftarrow findSlave(\text{“HT”}, h)$
return ($SLx$)
```

Every time a JoinExecutor finishes the job (Message 5 in Figure 4.4), the JoinManager invokes *findBucketForSymmetricJoin* (Algorithm 4.2.7) after receiving a job request message from a JoinExecutor. *FindBucketForSymmetricJoin* selects a chunk pair is selected using *findBucket(LT,chunkSize)* and then *FindBucket(HT,chunkSize)* algorithm in Figure 3.7. However, it is changed for the symmetric join to use JSM shown in Figure 4.3; only buckets with "R" entries in JSM are considered. If several candidates are selected, one of them is chosen randomly.

Algorithm 4.2.7: FINDBUCKETFORSYMMETRICJOIN$(chunkSize)$

```
{Find suitable bucket chunk pair from JSM Entry}
{Input: chunkSize; Output:bucketPair(hashId, chunkId1, chunkId2)}
$bucketPair \leftarrow FindBucket(\text{“LT”}, chunkSize)$
    {find local hash bucket but use JSM}
if (bucketPair is null)
  then $bucketPair \leftarrow FindBucket(\text{“HT”}, chunkSize)$
    {find remote hash bucket but use JSM}
return ($bucketPair$)
```

After a bucket chunk pair is selected, chunk transfer is done using *RSTransferForSymmetricJoin.* If at least one of the pair chunks is not presented on a JoinExecutor, then it is sent from the source node to the destination node by TransferExecutor on the source node. The source node is determined randomly but TransferExecutor on JoinExecutor node has a higher priority than TransferExecutor on Database node.

Figure 4.6 shows the message sequence chart in this chunk transfer (from JoinExecutor A to JoinExecutor B). Figure 4.5(b) also shows details of this transfer.

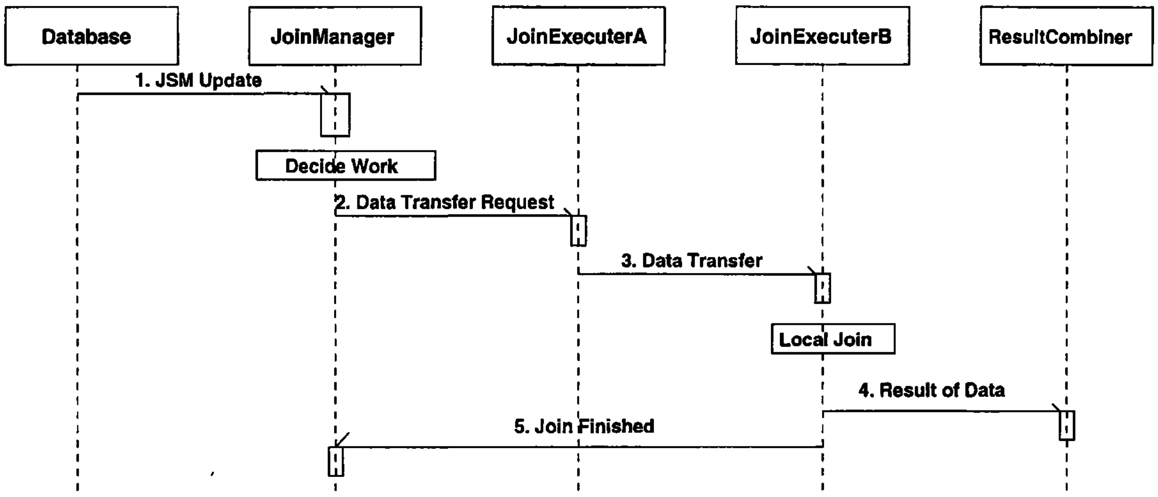

Figure 4.6: Message sequence chart in data transfer from JoinExecutor A to JoinExecutor B.

This algorithm may perform well in the case that background load and/or data skew exist since load balancing/sharing is done in a dynamic and incremental way. However, too many transfers may cause performance degradation. We will conduct experiments in Section 4.5 to verify this.

4.2.3 Dynamic Sampling (SMP)

This algorithm is based on the sampling algorithm proposed in [75]. As we mentioned in Section 2.3.2, sampling is the best algorithm for a pre-partitioned PDBMS. In SMP, there is a pre-defined sampling value (e.g., 10% of the the number of tuples of both relations). The JoinManager continuously receives JSM update messages (message 1 in Figure 4.4) from HashGenerator until the size of both received relations reaches the sampling value without executing joins. Then, the JoinManager uses a bin-packing algorithm to balance the workload and determines the hash mapping based on it. After that, JoinExecutors start executing joins.

The pseudocode of *findJoinExecutor(h,chunkSize,samplingValue)* of SMP is shown in Algorithm 4.2.8.

```
Algorithm 4.2.8: FINDJOINEXECUTOR_SMP(h, chunkSize, samplingValue)

{find a JoinExecutor(JEx) using sampling method}
{Input: hashValue h; chunkSize; samplingValue;
Output: destination JE index (-1 means no destination)}

if ((the sum of amount of read relations)>samplingValue) and (!smpOn))
  then { decide hash mapping using bin-packing algorithm
       { smpOn ← true
  else increase the amount of read relations by the chunk size
if (smpOn)
  then { if (hashMappingTable.containsKey(h))
       {   then return (hashMappingTable.get(h))
       {          { x ← random selection from idle JE list
       {   else   { hashMappingTable.insert(h, x)
       {          { return (x)
return (−1)
```

This algorithm may perform well in the case of data skew because of its load balancing/sharing mechanism. However, it may perform poorly due to dynamic changes such as background load and/or join product skew. We will conduct experiments in Section 4.5 to verify this.

4.2.4 Greedy Incremental Hash Mapping

Greedy Incremental Hash Mapping (GIHM) and JSM-based Incremental Hash Mapping (JIHM) try to deal with the Internet transfer delay that occurs in the data integration systems. The basic idea of these algorithms is to delay JoinExecutors to work on a bucket until there is enough work available in the bucket. The unit of work measurement is the number of tuples (GIHM) or the number of entries in JSM (JIHM).

GIHM uses a greedy algorithm and incrementally determines the hash mapping according to the arrivals of hash chunks.

The pseudocode of *findJoinExecutor(h)* of *GIHM* is shown in Algorithm 4.2.9.

```
Algorithm 4.2.9: FINDJOINEXECUTOR$_{GIHM}(h)$

{Decide destination JoinExecutor}
{Input: h hashValue;
 Output: destination JE index (-1 means no destination)}

if (hashMappingTable.containsKey(h))
  then return ($hashMappingTable.get(h)$)
       { $x \leftarrow$ random selection from idle JE list
  else { $hashMappingTable.insert(h, x)$
       { return ($x$)
```

This algorithm is greedy in the sense that the JoinManager looks for an idle JoinExecutor and assigns the JoinExecutor to the hashId (hash mapping) immediately. When there are several JoinExecutors, one of them is chosen randomly as a target JoinExecutor. After the target JoinExecutor is selected, the JoinExecutor receives the hash chunk from TransferExecutor in the same way as *SCHJ* and starts the join.

It may perform well when the arrival relation is delayed due to dynamic characteristics of the Internet transfer. We will conduct experiments in Section 4.5 to verify this.

4.2.5 JSM-based Incremental Hash Mapping (JIHM)

This algorithm is the same as GIHM except that the JoinManager waits for the hash mapping assignment until the number of JSM ready entries reaches a pre-defined number.

The pseudocode of *findJoinExecutor(h,threshold)* of *JIHM* is shown in Algorithm 4.2.10.

```
Algorithm 4.2.10: FINDJOINEXECUTOR$_{JIHM}$($h$, $threshold$)

{Decide destination JoinExecutor(JEx)}
{Input: h hashValue; threshold value for JSM
 Output: destination JE index (-1 means no destination)}

if (hashMappingTable.containsKey(h))
  then return ($hashMappingTable.get(h)$)
  else
    if (JSM Ready entries of h >threshold)
      then
        $x \leftarrow$ random selection from idle JE list
        $hashMappingTable.insert(h, x)$
        $return(x)$
      else return ($-1$)
```

4.3 Experimental Environment

From this section, we describe the experiments preformed and the results of the load balancing/sharing algorithms discussed in the previous section. We will start with a detailed description of environment.

4.3.1 Hardware Environment

The LINUX cluster we use in the experiments described in this chapter is different from the hardware used for the previous chapter (see Section 3.4.1). It consists of 4 dual CPU nodes and 7 single CPU nodes, one of which is used as a host machine and thus is not used in these experiments. Table 4.4 summarizes the hardware environment. These experiments are done in April 2004. Figure 4.7 shows the topology of the LINUX cluster.

4.3.2 Software Environment

The software environment is summarized in Table 4.5. MPI (Message Passing Interface) [61] is now the de facto standard for programming languages for parallel

4 nodes (n_d)	Dual Xeon 2.4GHz, 1GB Memory, 80GB Disk
7 nodes (n_s)	P4 2.4GHz, 1GB Memory, 80GB Disk
Network	each processing node is connected by 100 Mbps Ethernet
OS	RedHat 8.0

Table 4.4: Hardware environment.

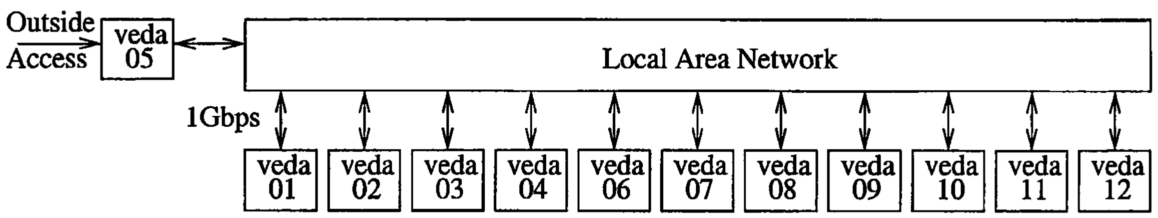

Figure 4.7: Topology of the LINUX cluster.

processing. We use mpich since it has thread-safe architecture and our simulation uses several threads running concurrently on PNs.

We use mpiJava [30, 48] to combine the advantage of MPI and Java. MpiJava is an object-oriented Java interface to the standard MPI. MpiJava itself does not assume any special extensions to the Java language. It is portable to any platform that provides compatible Java-development and native MPI environments. We did not use pure Java parallel processing such as RMI for performance reasons. However, mpiJava does not support the dynamic process (thread) invocation on a remote node. Thus, a TaskAllocator is implemented on each node and is responsible for the dynamic thread invocation. Initially, a TaskAllocator creates a QueryGenerator, a QueryScheduler,

Name	Version
MPI	mpich 1.2.5.2
Java	JDK 1.4.1
mpiJava	1.2.5
JSIM	1.2

Table 4.5: Software environment.

and a Database, then dynamically, QueryManagers, JoinManagers, JoinExecutors, ResultCombiners on specific PNs.

JSIM [50] is a Java-based simulation and animation environment supporting web-based simulation. It is used to generate hyper/hypo-exponential distribution sequences to simulate relation data transfer over the Internet which will be discussed in Section 4.3.4. Other parameters for this experiments are shown in Table 4.6. Among these parameters, node allocation sequence is important in the case of speedup measurement.

Parameter	Value
Memory size for a component	24MB
Node allocation sequence	Interleaved (single CPU, dual CPUs, single, ...)
Background model	Identical
Background loop period	1 sec

Table 4.6: Memory size, allocation, background process parameters.

4.3.3 The Experimental Database

The experimental database used in the experiment is the same as that described in Section 3.4.3.1 except for the following:

- The number of tuples of each relation is increased to 1 million.
- Random key and sequence generation is used for tuple generation for input relations. We will create 10 relations with different random seed for each parameter setting.
- Zipf skew model [96] is used in addition to the scalar skew model.

The Zipf model was originally proposed by Zipf [96]. As shown in Table 2.2, it is commonly used to model and control the degree of data skew. In this section, we used a slightly different model. For each data value in relation R, we determined the number of its occurrences ($|R_i|$) using the following Zipf-like distribution function

$|R_i| = \frac{|R|}{\sum_{j=1}^{N} 1/j^{Z_p}}$ where $|R_i|$ represents the number of times the value $|R_i|$ occurs, $|R|$ represents the number of tuples in R, Z_p is called the skew factor. Thus, using skew factor Z_p we can control of the degree of skew and therefore the imbalance condition. We varied Z_p from 0 percent (no skew) to 100 percent (extreme skew).

4.3.4 The Internet Transfer Delay Model

In this subsection, data transfer delay over the Internet is discussed. The purpose is to approximately model the real world Internet data transfer. The issues include: Is the transfer time linear to the distance and/or the amount of data? If so, what is the slope and overhead? How do we measure the time? Which model do we use?

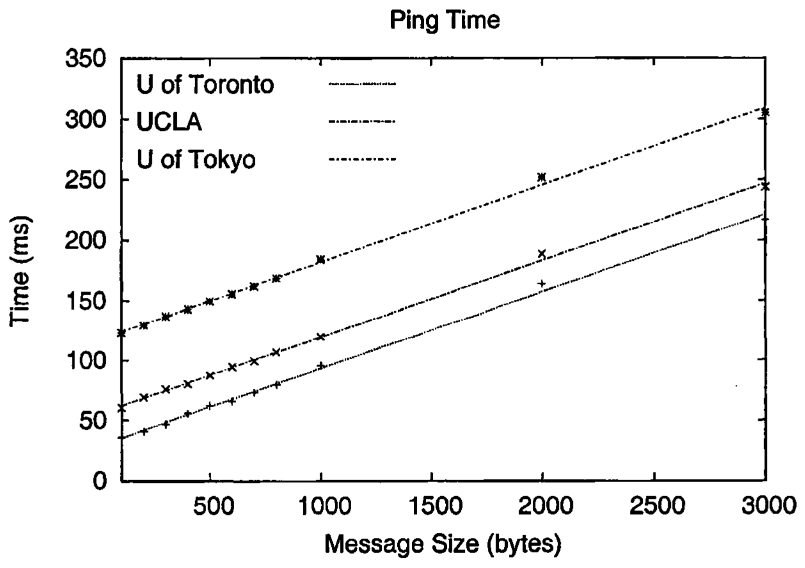

Figure 4.8: Ping transfer time: trend line is shown in column 4 of Table 4.7.

In order to answer these questions, we first measured the transfer time of different message size by UNIX ping command [65, 79] 100 times at 3 different times of the day from our office to several locations spanning a range of distances from University of Toronto, University of California Los Angles (UCLA), to University of Tokyo.

Figure 4.8 shows their average transfer time. Table 4.7 shows approximate ping transfer functions for each location.

Table 4.8 shows ping transfer time, standard deviation, and coefficient of variation (CV) (= standard deviation/average time) when the message size is 3 KB. Since each CV is below 1, we decided to use hypo-exponential distribution to simulate data transfer over the Internet. As a result, we use the following model to get the transfer times (*ActualTransferTime*) of data as a function of *size*:

Name	Location	URL	Approx. Transfer Func. (trend lines in Fig. 4.8)
Toronto	University of Toronto	info.utcc.utoronto.ca	t=0.064×size+28.839
UCLA	UCLA	www.ucla.edu	t=0.0637×size+55.416
Tokyo	University of Tokyo	www.u-tokyo.ac.jp	t=0.0638×size+117.57

Table 4.7: Approximate transfer functions.

Name	Avg. Tran.	Std. dev. (*dev*)	CV
Toronto	216	28.02	0.13
UCLA	243	29.84	0.12
Tokyo	305	30.13	0.01

Table 4.8: Detailed ping transfer times when message size is 3 KB.

- MeanTransferTime = a**size* + b as shown in column 4 of Table 4.7
- ActualTransferTime = Hypo_exponential(*MeanTransferTime,dev*) where *dev* is its standard deviation of the ping time shown in column 3 in Table 4.8.

We insert a sleeping function after reading the relation and before applying the hash function (between DatabaseReader and HashGenerator in Figure 4.5(a)) with duration of the corresponding time according to the above model.

In the experiments, it is assumed that one relation resides on the local area network and the other relation resides in another location (either at Toronto, UCLA, or Tokyo). The reason for this decision is that if both of them reside on remote locations, then it is better to execute the join on one of the remote locations.

4.4 Experimental Methodology

Experiments are done in the following way. First, we compare RR, JSM, and SMP with background load, data skew, and combination of both as in Section 3.5. With the same methodology as the previous chapter, we try to find default parameter values first and then compare algorithms in different conditions. Then we compare JSM, GIHM, and JIHM for the relation arrival delay environment.

For SMP and JIHM, several sampling/JSM entries value combinations are tested; For SMP, 10% and 20% are used with the same condition as [34]. For JIHM, 5, 10, and 20 are used.

We use two-sided skew (both relations (R and S) have data skew).

We repeat each experiment 30 times; ten times for each random seed to generate relations. The 95% confidence interval (CI) for the mean is calculated for each set of execution time according to the following formula [46]: $CI = 1.96 * dev/\sqrt{n}$ where 1.96 is the Z score corresponding to 95% and *dev* is the standard deviation of the result sets and n is the number of runs.

4.5 Experimental Results

The experimental results obtained by executing the symmetric hash join algorithms and their load balancing/sharing algorithms described in Section 4.2 are presented in this section under the data skew, background load, and the Internet transfer delay conditions.

4.5.1 Default Parameter Values

The default parameters for the algorithms used in the experiments are shown in Table 4.10. The results are obtained from some preliminary experiments. These values are chosen so as to achieve good performance for all cases.

Parameter Name	Value
Chunk size	9600 bytes
Database page size	9600 bytes
Output buffer size	12000 bytes
The number of hash buckets	1000
The number of JoinExecutors	8
The DBMS node	a single CPU (n_s) dedicated node
The JoinManager node	a single CPU (n_s) dedicated node
The ResultCollector node	a single CPU (n_s) shared node with other threads

Table 4.10: Default parameter values.

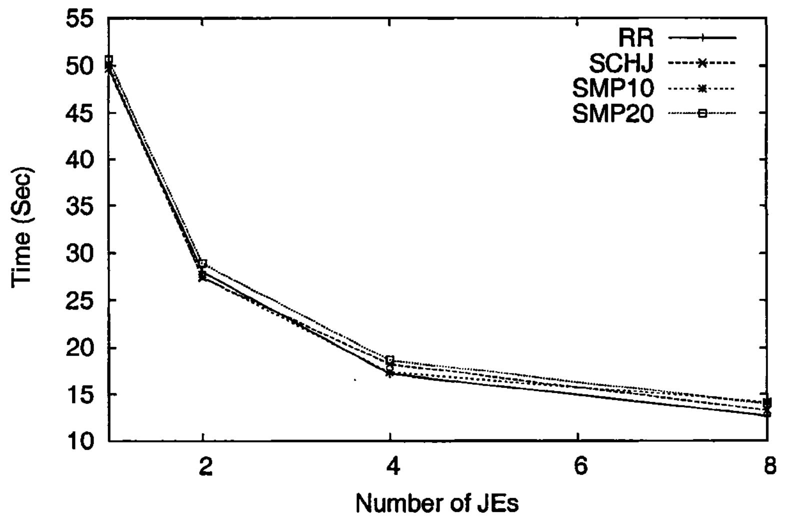

Figure 4.9: Performance of the algorithms with no skew and no background.

4.5.2 Performance with No Data Skew and No Background Load

This section establishes the base case in which there is no data skew in the input relations. Also, there is no background load on the JEs, which represents the case where the cluster is dedicated to database query processing. In Figure 4.9, we have plotted the number of JEs versus the execution time for the algorithms with sampling values of 10 and 20 for SMP. These results are used for base case for comparison. The confidence interval is less than 1%.

These results show that the Dynamic Round-Robin (RR) algorithm is the best for all the number of JEs. For example, when the number of JEs is 8, RR is 5%–11% better than other algorithms. This is because RR performs better when there are no dynamic changes in the system and no data skew since it distributes the work equally.

The Dynamic Sampling algorithms (SMP10 (10% sampling) and SMP20 (20% sampling)) have almost the same performance. However, in comparing SMP with the other two algorithms, we found that they do not compare favorably because of the delay in starting of the join processing; JEs should wait until 10% (20%) of the relations arrive at JM.

The Symmetric ChunkHJ (SCHJ) has the next best performance after RR despite the overhead in transferring the other chunks. The performance difference between them is about 5%.

4.5.3 Performance with No Data Skew and Background Load

Figures 4.10 and 4.11 show the execution time with no skew and with the background load on JEs, which the confidence interval is always less than 5%, in most case it is within 2.9%.

In Figure 4.10, we have plotted the number of JEs versus the execution time when we fix the background load to 6 processes. It shows that SMP10 is best among all algorithm for all the number of JEs. The longer it takes to process data, the smaller the effect of the delay in starting the join processing. The graph also shows SMP10 is almost 20% better than SMP20 since there is no skew, there is no advantage in increasing the sample value. The early start of join processing is better since the data has no skew and thus skew prediction from sampling with small sampling values is close to the real skew.

Compared with the no background case, RR does not perform well because of the interference caused by the background load.

The SCHJ performs next best to SMP10. However, the difference is less than the

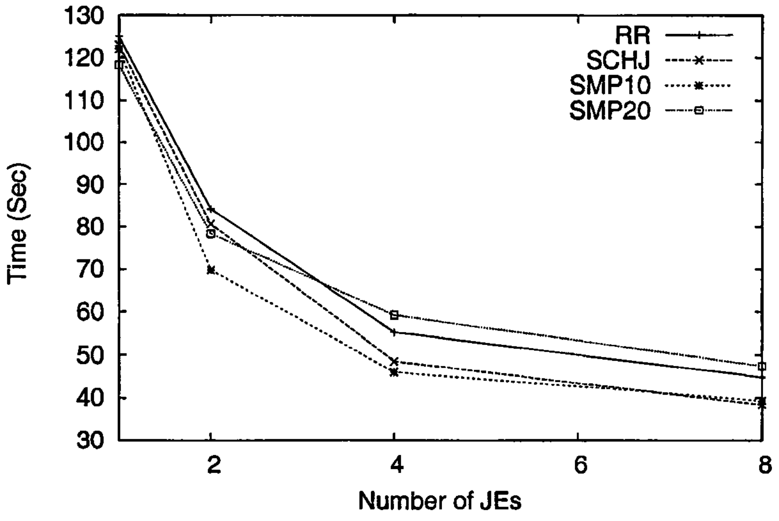

Figure 4.10: Performance of algorithms with no data skew and 6 identical background load.

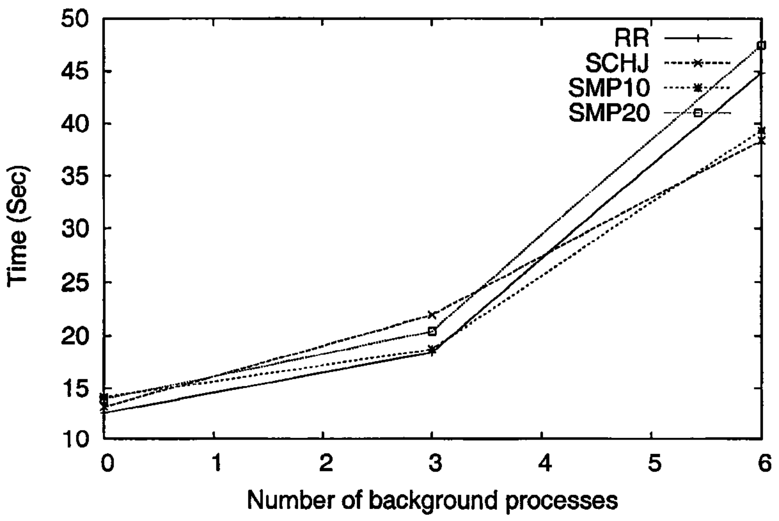

Figure 4.11: Performance of algorithms with no data skew and 8 JEs.

error margin for most of the cases. Compared to RR and SMP20, performance of SCHJ improves as the number of JEs increases. This shows the effectiveness of SCHJ as the number of JEs increases.

In Figure 4.11, we have plotted the number of background load processes versus the execution time with the same conditions but the number of JEs is fixed at 8. In this figure, RR still performs as well as SCHJ when the background load is relatively low (up to 3 processes). It is almost the same as the default case. However, as the background load becomes higher, SCHJ performs relatively better than the others.

4.5.4 Performance with Data Skew and No Background Load

This section presents the performance with data skew (scalar skew and Zipf skew) and no background on JEs.

4.5.4.1 Performance with Scalar Skew

In these experiments, we used the relations with the scalar skew factor of 0, 10,000 and 20,000, which means there are 0, 10,000 and 20,000 tuples with attribute 0, respectively. This model is the same as the model introduced in Section 3.4.3.2.

Figure 4.12 and Figure 4.13 show the execution time with scalar skew and no background load on JEs. The confidence interval is less than 1.5%.

In Figure 4.12, we have plotted the number of JEs versus the execution time when the skew factor is fixed at 20,000. It shows that SCHJ performs the best among these algorithms when the number of JEs changes. The difference increases with the number of JEs. SCHJ gives 12% improvement over SMP10 for 2 JEs and 15% improvement when number of JEs is 4 and 8, respectively.

Figure 4.12 also shows that the speedup of this graph is not as good as Figure 4.9 because of the join product skew caused by scalar skew. SMP10 and SMP20 are both better than RR. RR performs the worst because of the join product skew. However, both SMP algorithms are not as good as we expected because of the delay in starting the join.

In Figure 4.13, we have plotted the skew factor versus the execution time when

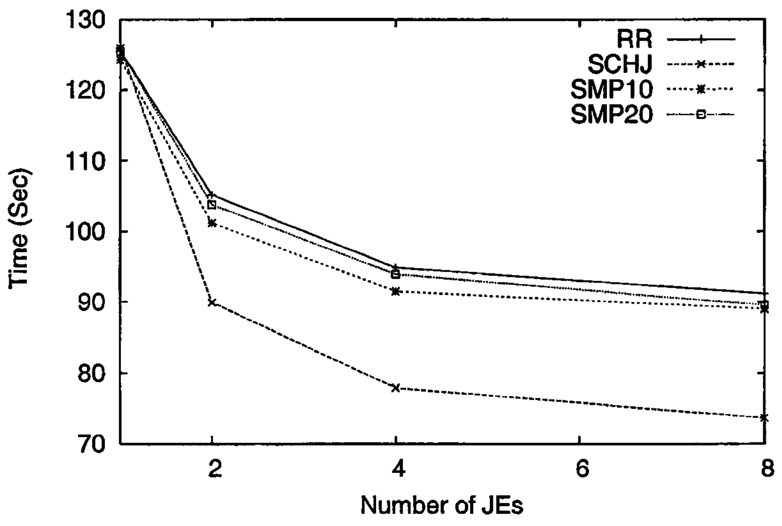

Figure 4.12: Performance of algorithms with no background load on JEs and a scalar skew factor of 20,000.

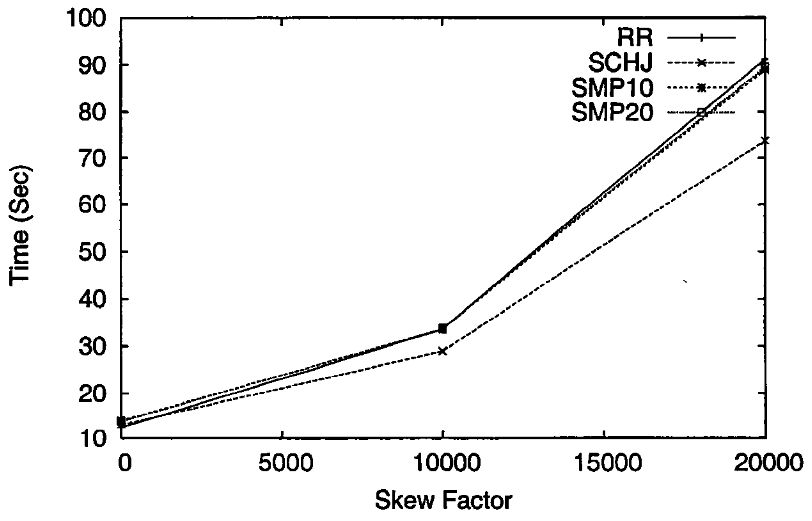

Figure 4.13: Performance of the algorithms with no background load and with 8 JEs.

we fixed the number of JEs to 8. It also shows that SCHJ performs the best among the algorithms for all the degrees of skew. The more skewed the data are, the better the SCHJ performs over the other algorithms. It is at least 14% and 18% better than other algorithms when the skew factor is 10000 and 20000. respectively.

From these results, we can conclude that SCHJ performs the best in the case of scalar skew.

4.5.4.2 Performance with Zipf Skew

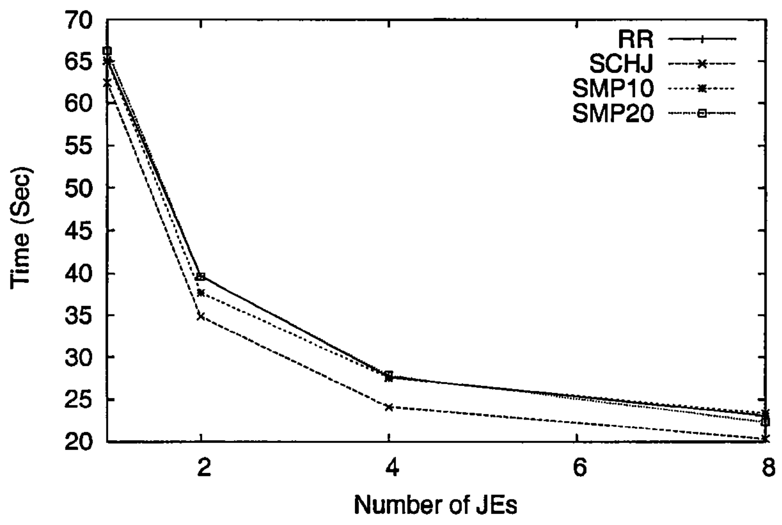

Figure 4.14: Performance of algorithms with Zipf skew factor of 60 and no background load.

Figures 4.14 and 4.15 show the execution time with Zipf skew and with no background load. The confidence interval is less than 1.7%.

In Figure 4.14, we have plotted the number of JEs versus the execution time when we fixed the skew factor to 60. It shows that the SCHJ is 8% to 13% better than SMP10. As the number of JEs increases, the greater performance gain over SMP10 is obtained.

The other two algorithms (SMP20 and RR) have the same performance (within

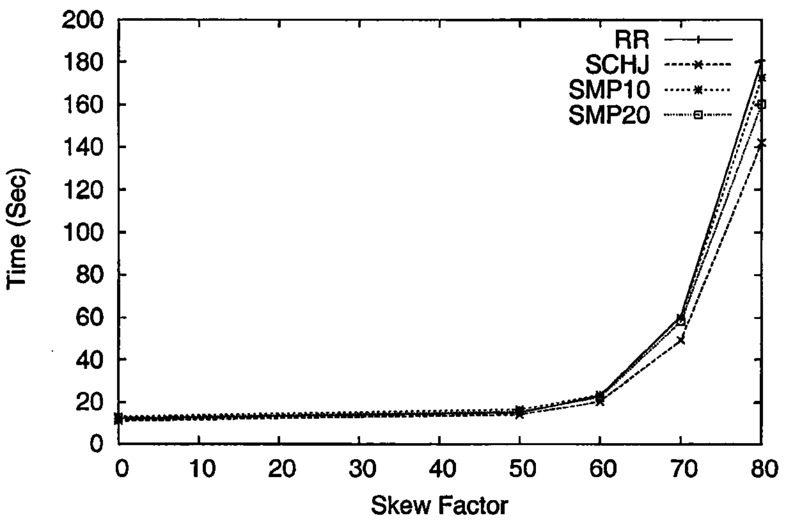

Figure 4.15: Performance of the algorithms with Zipf skew factor and no background load and 8 JEs.

the error range). This is because load imbalance in RR is not as severe as the extreme skew case and it is canceled out by the join start delay in SMP20.

In Figure 4.15, we have plotted the skew factor versus the execution time when the number of JEs is 8. It shows that SCHJ is 15% better than RR when the Zipf factor is 60, 18% better with the Zipf factor is 70, and 21% better with the Zipf fator is 80. The greater the skew, the better SCHJ performs over RR. Also, the performance improvement over SMP20 is 13% when the skew factor is 80.

SMP20 is better than RR and SMP10 as they predict better work distribution than the others.

From these results, we conclude that SCHJ performs the best in the case of Zipf skew as well as the case of scalar skew.

4.5.5 Performance with Data Skew and Background Load

In this section, we add background load on JoinExecutors with data skew (scalar skew and Zipf skew).

4.5.5.1 Performance with Scalar Skew and Background Load

Figures 4.16 and 4.17 show the execution time when there is both scalar skew (skew factor is 10,000 and 20,000, respectively) as well as the background load. The confidence interval is less than 2.8%.

Figure 4.16 shows that SCHJ is better by approximately 10% compared to RR and SMP20 in all the cases. Figure 4.17, where the skew factor is increased to 20,000, shows that SCHJ is better than RR by 15% with no background load, remains 15% with 3 processes, and increases to 19% with 6 processes. The more background load there is, the better the performance of SCHJ.

The SMP algorithms as well as RR do not perform well because of the background load disturbance. Also, the poor performance of SMP10 is due to inadequate sampling for the skewed data. The more background loads that are present, the worse the wrong prediction gets.

From these results, we can conclude that with the combination increased scalar skew and background load, SCHJ is better than the other algorithms. The more scalar skew and background are present, the better SCHJ performs.

4.5.5.2 Performance with Zipf Skew and Background Load

Figures 4.18 and 4.19 show the execution time when both Zipf skew and background load are present. The confidence interval is always less than 5%. However, for most results, it is less than 3.2%.

Figure 4.18 shows the performance of algorithms with the skew factor of 60. When the degree of skew is modest (skew factor is 60) and background load is modest (3 processes), RR and SCHJ perform the same. However, when the background load increases to 6 processes, the performance gain of SCHJ over RR reaches 19%.

Figure 4.19 shows the performance of these algorithms with the skew factor is 80. It shows that SCHJ has a stable performance gain over the other algorithms. The gains are 21% when the background load is 0, 22% with 3, and 27% with 6 processes.

In conclusion, SCHJ outperforms the other algorithms with the Zipf skew and various background load.

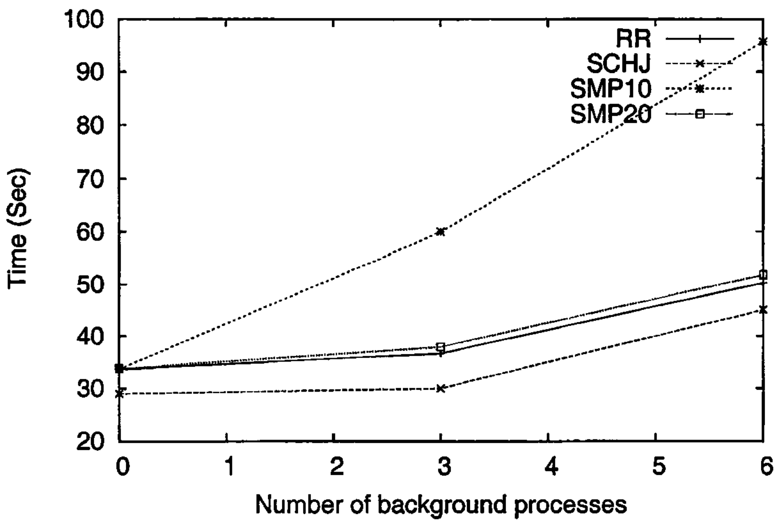

Figure 4.16: Performance of algorithms with scalar data skew variance of 10,000 and background load.

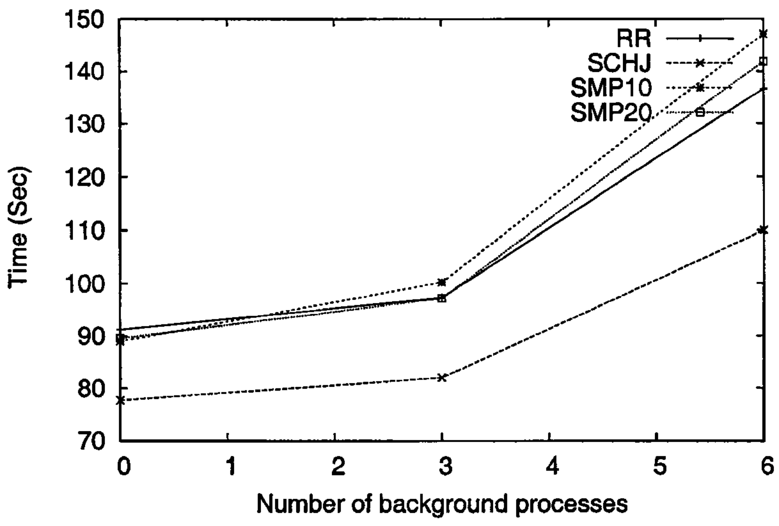

Figure 4.17: Performance of algorithms with scalar data skew variance of 20,000 and background load.

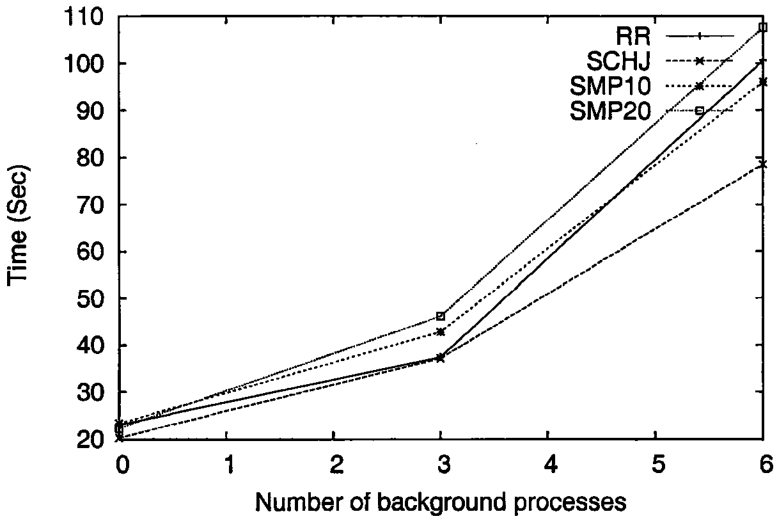

Figure 4.18: Performance of algorithms with the Zipf data skew factor of 60.

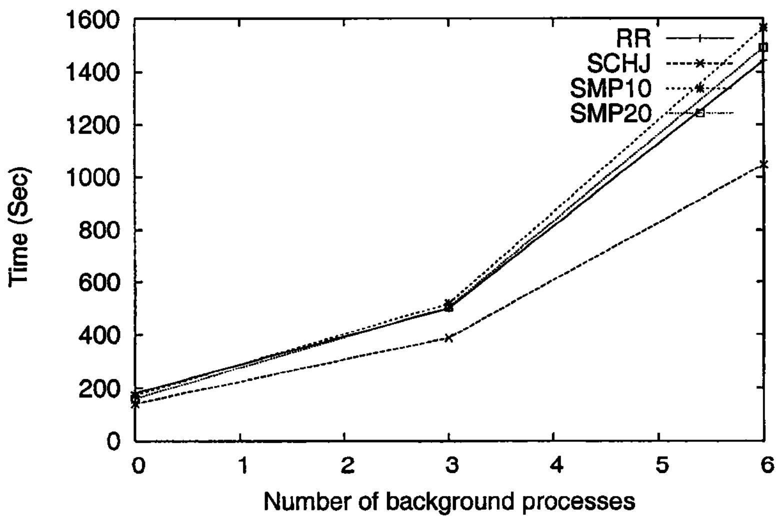

Figure 4.19: Performance of algorithms with the Zipf data skew factor of 80 and 8 JEs.

4.5.6 Performance with the Internet Transfer Delay

As explained in the experimental methodology in Section 4.4, this subsection focuses on performance of SCHJ, GIHM, and JIHM under the effect of the Internet transfer delay using the model described in Section 4.3.4. First, we isolate other factors and just investigate the effect of the Internet transfer delay on the performance of these algorithms. Then, we add data skew and background load as in the previous section. We focus more on scalar skew rather than Zipf skew because the skew control is easier. For GIHM, we use 5, 10, and 20 as its threshold values.

4.5.6.1 Performance with the Internet Transfer Delay Alone

Figure 4.20 shows the performance of the algorithms with no data skew and no background load but with the Internet transfer delay. The confidence interval is less than 0.1%.

In this figure, SCHJ is marginally better than the other algorithms for all the DB locations. The improvement is only about 1% because it has no background load and no skew.

Another remark about this figure is that the delay caused by the Internet transfer delay from one location to another location is 4% (from Toronto to UCLA) and 25% (from UCLA to Tokyo). Table 4.8 shows the delay is 13% (from Tortoto to UCLA) and 26% (from UCLA to Tokyo). Thus, when the delay is small the algorithms can absorb the delay. However, as the distance becomes long, the algorithms are affected by the delay.

4.5.6.2 Performance with the Internet Transfer Delay and Data Skew

Figures 4.21 and 4.22 show the performance of the algorithms there is no background load and the scalar skew factor is 10000 and 20000, respectively. The confidence interval is less than 0.5%.

SCHJ is marginally better than the other algorithms for all DB locations and all the degrees of skew.

Another interesting point is that the difference between Figure 4.21 and 4.22 are not as large as the case without the Internet transfer delay (Figure 4.13). This

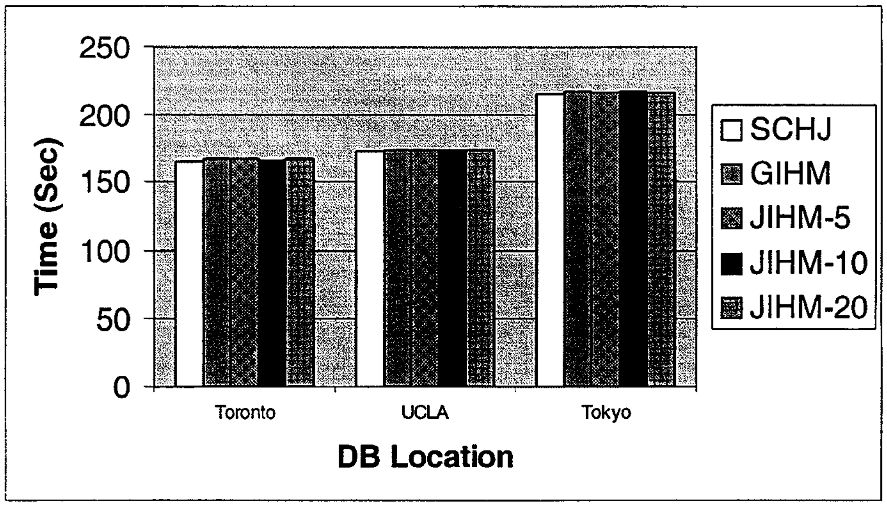

Figure 4.20: Performance of the algorithms with no data skew and no background load and the Internet transfer delay.

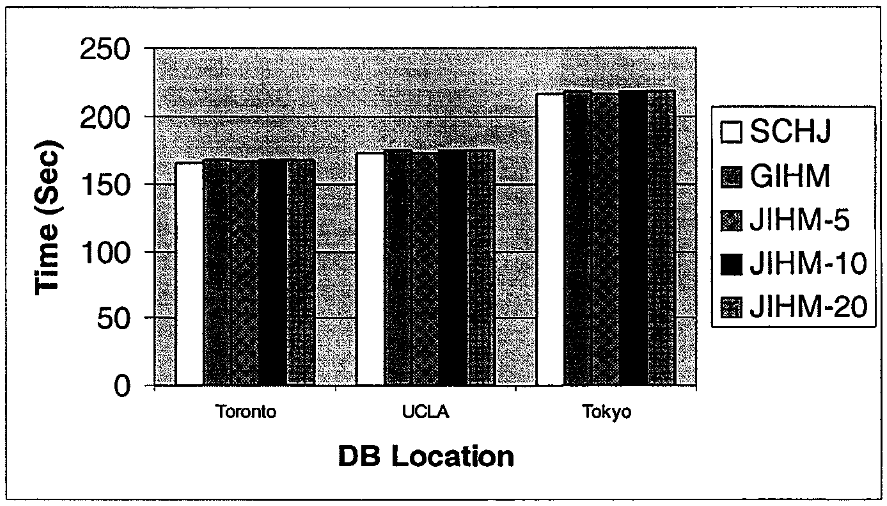

Figure 4.21: Performance of algorithms with scalar data skew factor of 10,000.

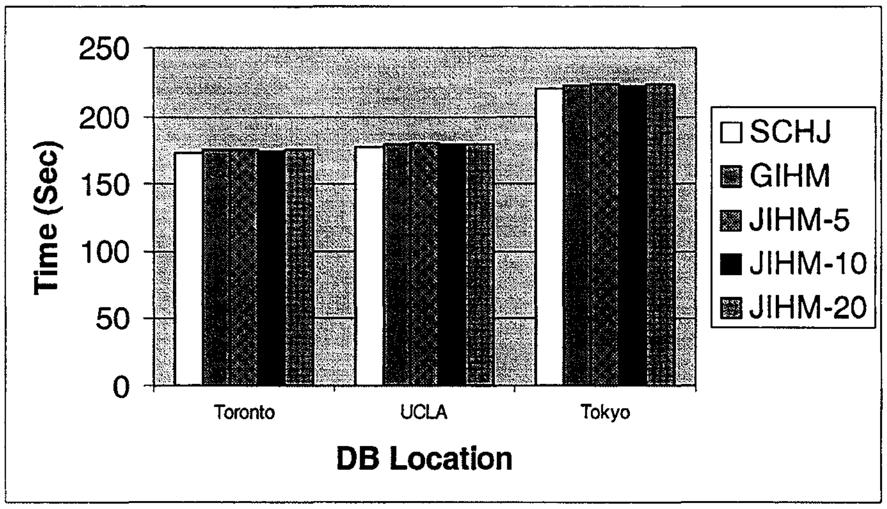

Figure 4.22: Performance of algorithms with scalar data skew factor of 20,000.

is because JoinExecutors can work on the join processing on skewed bucket while waiting for the relations to arrive as long as there is no background load on them. If there is background load, it delays the join execution as we will see in Section 4.5.6.4.

4.5.6.3 Performance with the Internet Transfer Delay and Background Load

Figures 4.23 and 4.24 show the performance of the algorithms as the function of background load when the DB location is Toronto and Tokyo, respectively. The confidence interval is less than 1%.

Both figures show that the effect of background load is small in SCHJ comparing to other algorithms because of its adaptive load balancing/sharing mechanism. Thus, the higher background load, the higher its improvement over the other algorithms. For example, in Figure 4.23, SCHJ improvements over GIHM are 2% and 5% when the background load is 3 and 6 processes, respectively.

Both figures also show that JIHM5 is better than GIHM, especially when the background load is 6 processes. GIHM is based on a greedy algorithm. Once it fixes the hash mapping, it cannot change later. With higher background load, it is not effective because of frequent change in background load. On the other hand, JIHM waits to fix the hash mapping until the enough work is available and the timing of fixing the hash mapping is totally depends on the load of the PNs. Thus, it is effective in the case of higher background load.

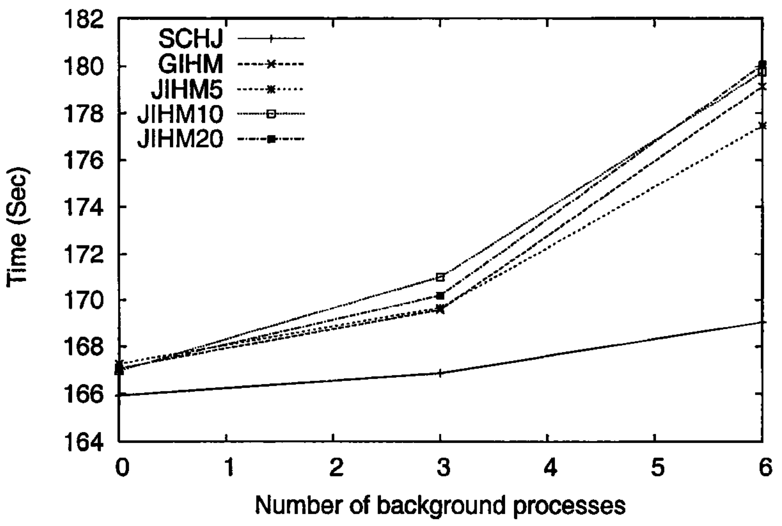

Figure 4.23: Performance of the algorithms when the DB location is in Toronto.

4.5.6.4 Performance with the Internet Transfer Delay, Data Skew and Background Load

Figures 4.25 and 4.26 show the performance of the algorithms as a function of the number of background load processes when the skew factor is 10000 and the DB location is Toronto and Tokyo, respectively. The confidence interval is less than 3%.

Both figures show that SCHJ is the least affected by the change of background load. Thus, the higher the background load, the more SCHJ improves compared to other algorithms which are affected by the change in background load.

Comparing GIHM with JIHM in these figures, GIHM is better than JIHM. GIHM is based on greedy algorithm which assigns skewed buckets as soon as they arrive.

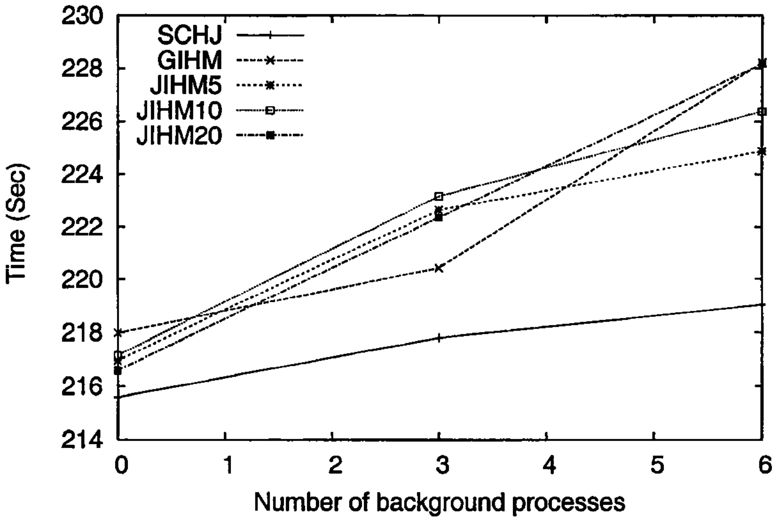

Figure 4.24: Performance of algorithms when the DB location is in Tokyo.

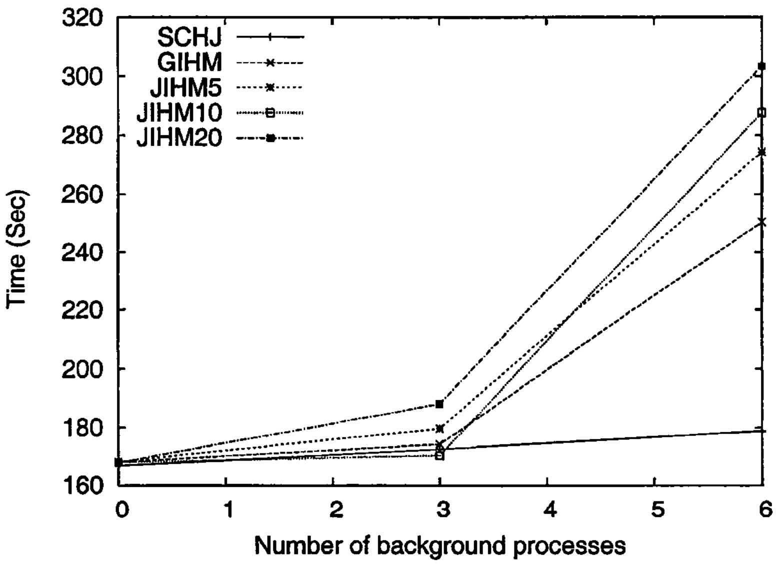

Figure 4.25: Performance of algorithms with the DB location as Toronto and the skew factor of 10,000.

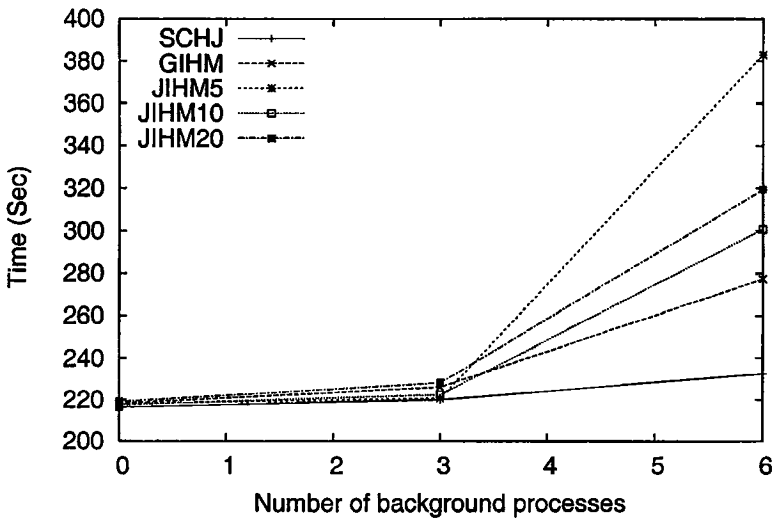

Figure 4.26: Performance of algorithms with the DB location as Tokyo and the skew factor of 10,000.

Figures 4.27 and 4.28 show the performance of the algorithms as a function of the number of background load processes when the skew factor is 20000 and the DB location is Toronto and Tokyo, respectively. The confidence interval is always less than 5.5%. In most cases, it is less than 3.1%.

The effect of the background load is more obvious than in the previous graphs (e.g., Figure 4.23 – 4.26) because of the extreme skew.

Performance of SCHJ is 11% better than JIHM10 in both cases, which shows the effectiveness of SCHJ. JIHM10 is better than other JIHMs and GIHM. When the skew factor is high, the effect of the background load is severe on GIHM. GIHM is good for moderate skew case and low background load. In case of high skew and high background load, it does not perform well because of its greedy algorithm which result in poor hash mapping decision. On the other hand, JIHM waits for more data to arrive before it makes a decision. Among the various JIHM algorithms, smaller number (5 or 10) of ready JSM entries is better in these cases. If it is 20, it waits too long and keeps JEs idle long.

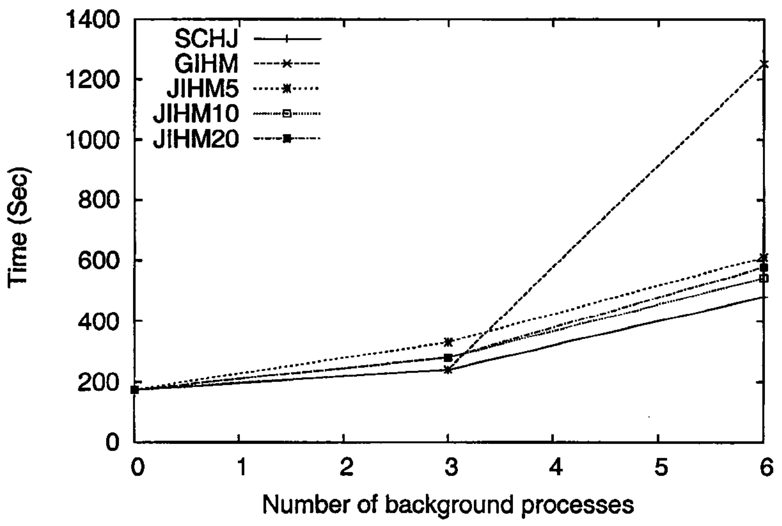

Figure 4.27: Performance of the algorithms with the DB location as Toronto and skew factor of 20,000.

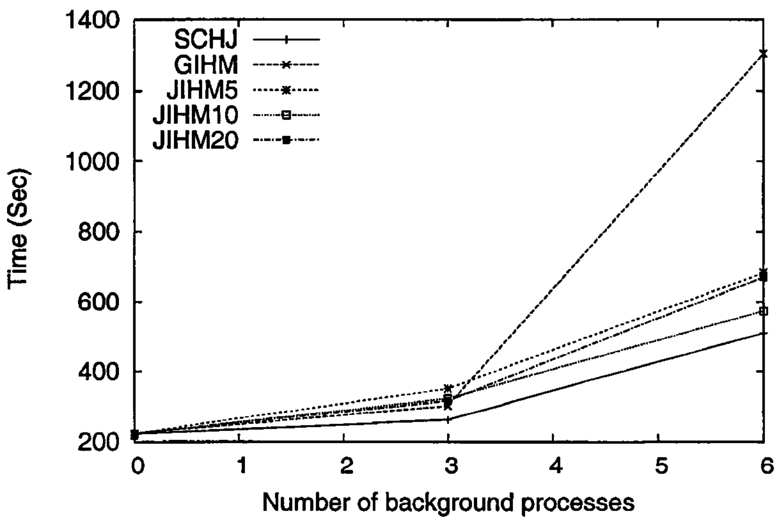

Figure 4.28: Performance of the algorithms with the DB location as Tokyo and the skew factor of 20,000.

From these results, we can conclude that SCHJ performs better than other algorithms under the Internet transfer delay, background load, and data skew.

4.6 Conclusion

Non-symmetric join algorithms are not effective when we do not know which relation to choose as the main relation (i.e. which one is smaller or which one arrives faster than the other). Thus, in this chapter, we first proposed symmetric versions of ChunkHJ (SCHJ). They are evaluated with dynamic round-robin and dynamic sampling algorithms on a cluster. We reached the following conclusions from these experiments without the Internet transfer delay.

- Dynamic round-robin algorithm is better than other algorithms when there is no data skew and no background load.
- Dynamic sampling is better when there is no data skew and background load which cancels out the delay of start of join processing. But, it is just marginally better than SCHJ.
- SCHJ is better than the other algorithms when there is data skew (scalar skew and Zipf skew) and background load. In most cases, the more the background load, the bigger the improvement. In the extreme case, when the background load is 6 processes and Zipf skew (skew factor is 80) exists, it is 27% better than the other algorithms.

We also proposed Greedy Incremental Hash Mapping (GIHM) and JSM-based Incremental Hash Mapping (JIHM) mainly for the Internet transfer delay case. They are compared with SCHJ for the Internet transfer delay model. In the model, one of the relations resides locally and the other resides at a remote location (University of Toronto, UCLA, or University of Tokyo), which has a dynamic transfer delay according to the geographical distance. We drew the following conclusions:

- With only the Internet transfer delay or with data skew (scalar skew), SCHJ is marginally better than the other algorithms. The greater the background load, the better the SCHJ performance because of the load balancing mechanism. In this case, data skew can be absorbed by the arrival delay if there is no background load.

- The improvement of SCHJ becomes greater when increasing skew or background load.
- When there is modest skew and background load, GIHM is better than the JIHM algorithms but worse than SCHJ
- When there is extreme skew and background load, JIHM is better than the GIHM algorithm but worse than SCHJ.

In conclusion, for a cPDBMS, where there is background load along with data skew, Symmetric ChunkHJ, performs better than incremental mapping hash algorithms with the Internet transfer delay.

Chapter 5

Multiple-Join Algorithms

In this chapter, we compare the performance of two pipelined join algorithms in a cPDBMS. First, pipelined hash join algorithms are reviewed. Then, two hash join algorithms, simple (non-symmetric) pipelined hash join (NSPHJ) [43] [1] and symmetric pipelined hash join algorithm (SPHJ) are explained [42] [2]. Then, hardware and software environments are explained. Finally, the experimental results are presented.

5.1 Pipelined Hash Join Algorithm Revisited

This section describes the pipelined hash join algorithms used in our experiments. The two relations participating in a join are traditionally called inner and outer relations. To facilitate pipelined execution, we use a right-deep query tree shown in Figure 5.1. In this tree, the inner relation of a join is shown as the left relation and the outer relation is shown as the right relation. For example, relations R_2, R_3, R_4, R_5, and R_6 are inner relations. Relation R_1 is the outer relation for join J1. The output produced by J1 will be the outer relation to J2 and so on.

In a hash join algorithm, the inner relation is first partitioned into disjoint subsets

[1]Kenji Imasaki, Hong Nguyen and Sivarama Dandamudi,"Performance of Pipelined Nested Loop and Hash Joins on a Workstation Cluster," *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, June 2002.

[2]Kenji Imasaki, Hong Nguyen and Sivarama Dandamudi,"Performance Comparison of Pipelined Hash Joins on Workstation Clusters," *9th International Conference on High Performance Computing (HiPC2002)*, pp. 264–275, Bangalore, India, December 2002.

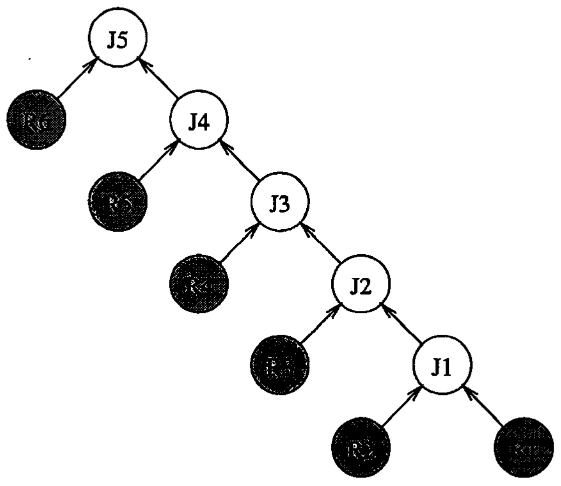

Figure 5.1: Right-deep query tree.

called buckets, using a hash function applied on the join attribute. This creates a hash table for the inner relation, against which tuples from the outer relation are matched to join the tuples.

Thus, a hash join algorithm normally involves two phases:

1. *Table-Building Phase:* During this phase, a hash table is built for the inner relation. This step involves reading the complete inner relation.

2. *Tuple-Probing Phase:* Once the hash table for an inner relation is available, tuples from the outer relation are used to probe the hash table to find matches on the join attribute. If a match exists, the two tuples are joined to produce the result tuple. Thus, to complete a join operation, complete scan of the outer relation has to be made.

The following notations are used in the description.

- R_i = inner relation; R_{ik} = the k_{th} partition of the inner relation
- R_o = outer relation; R_{ok} = the k_{th} partition of the outer relation
- R_r = result relation; R_{rk} = the k_{th} partition of the result relation.

Also, *join pattern* at each node is discussed. Join pattern is a sequence of arrived partition name (inner and outer relation) and join execution denoted by "J". For example, join pattern "$R_{11}R_{21}JR_{12}$" means that first the node receives partition $R_{11}R_{21}$ and executes the join and then receives R_{12}. In the following description, each slave node is responsible for each join in a query.

5.2 Non-Symmetric Pipelined Hash Join Algorithm

Non-Symmetric Pipelined Hash Join (NSPHJ) algorithm works as follows:

1. Each slave node waits for the partitions of R_i to arrive. The tuples of each received partition of R_i are inserted into the hash table as they arrive. Once all of R_i has been received and the hash table is complete, the hash join operation begins if a partition of R_o is available at the slave node.

2. The intermediate join result R_r is shipped to the next slave node in partitions when the buffer becomes full. These shipped partitions act as the partitions of R_o for the next join process.

3. After the intermediate result has been sent to the next slave, the current node waits for another incoming partition of R_o. When this partition arrives, it performs the join as before. This process is repeated until the last partition of R_o has been received (with an end-of-R_o message).

For this join algorithm, the join pattern at each node is as follows: R_{i1}, R_{i2}, R_{i3}, $\cdots$, end-of-R_i, R_{o1}, J, R_{o2}, J, R_{o3}, J, $\cdots$, end-of-R_o, J.

5.3 Symmetric Pipelined Hash Join Algorithm

NSPHJ is asymmetric in that only the inner relation is hashed. On the other hand, in Symmetric Pipelined Hash Join (SPHJ) algorithm, both relations are hashed and works as follows:

1. Each slave node waits for the partitions of either R_i or R_o to arrive. If a partition of $R_o(R_i)$ arrives, the join begins between the tuples of partition $R_o(R_i)$ and the hash table of $R_i(R_o)$, respectively.
2. The intermediate join result is sent to the next slave node as a partition of R_r when the buffer becomes full. This partition becomes the R_o partition in the next join process.
3. After the intermediate result has been sent, the tuples of the partition of $R_o(R_i)$ are inserted into the hash table of $R_o(R_i)$.
4. The process is repeated until the last partitions of R_i and R_o have been processed.

For this join algorithm, an example of the join pattern at a slave node is as follows: R_{i1}, R_{i2}, R_{o1}, J, R_{i3}, J, $\cdots$, end-of-R_i, J, R_{o2}, J, R_{o3}, J, $\cdots$, end-of-R_o, J.

5.4 Experimental Environment

This section presents the environment that we used for multiple-join processing.

5.4.1 Hardware Environment

The following experiments were conducted on the same cluster used in the experiments in Chapter 4. Its detailed description of the cluster was given in Section 4.3.1.

5.4.2 Software Environment

The software architecture is shown in Figure 5.2, and is based on the software architecture used in Chapter 3. A detailed description was given in Section 3.4.2. The Database (DBMS) node maintains the actual database. The master node acts as the coordinator as well as the interface between the slave and DBMS nodes. The slaves actually perform the join operation. More details on the software architecture will be given in the next subsection.

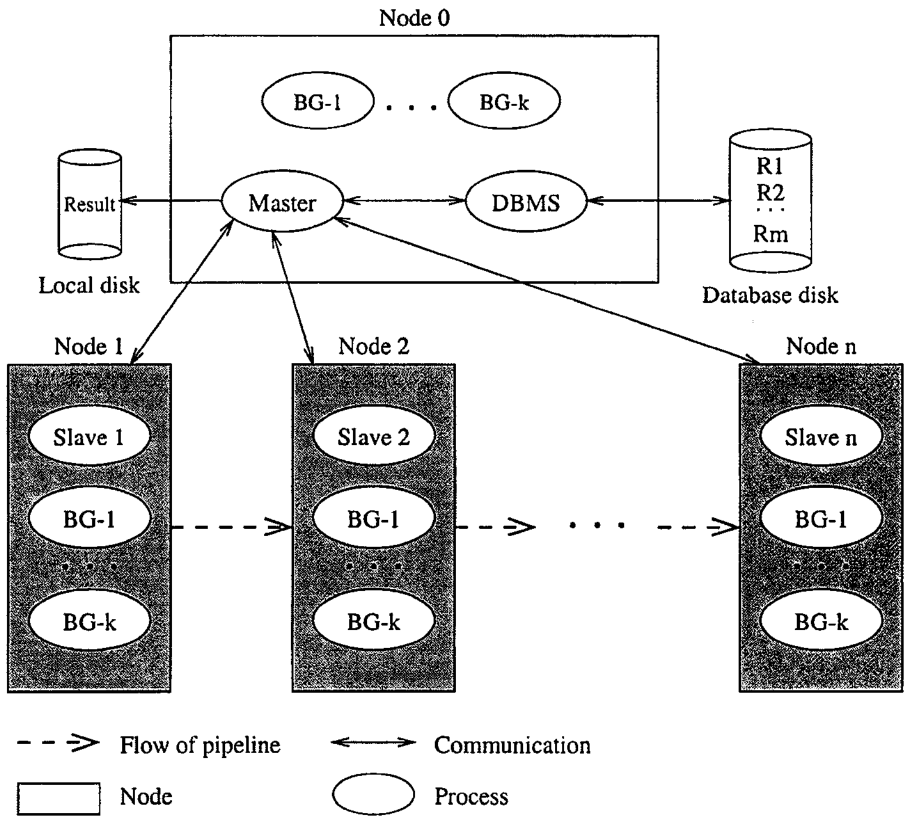

Figure 5.2: Software architecture for pipelined hash joins.

The relation sizes used in our experiments are 1 and 2 million. Database data is the same as the data in Section 3.4.3.1.

5.4.3 Implementation Detail

In these experiments, one node is used to run the master and DBMS processes, while the remaining nodes are used for the slave processes. Unless otherwise stated, the results reported here were obtained using 8 slave nodes.

Our implementation consists of four major software components: a master, a slave, a DBMS, and a background process. The relationship among these components is shown in Figure 5.2. The master process acts as the controller of all processes. When invoked, it spawns the other processes (i.e., slave, background, and DBMS processes).

The master process monitors the join processes and gathers the final join results, which are then written to a local disk.

Slave processes perform the local join operations. The DBMS process is responsible for reading input relations and passing them on to the master. Background processes control the non-query loads that simulate the local user's workload. When configured, background processes are invoked on the same nodes as the slave processes and are run during the join operations, which is the same as in Section 3.1.2.

At the beginning of the execution, the master process reads in the parameter settings (like buffer size, number of tuples, etc.), spawns the child tasks (slave and background processes), and depending on the configuration, passes the necessary information to the children. It then initiates the join process by making requests to the DBMS process for data inputs and sends the received inputs to the slave processes.

After receiving requests from the master, the DBMS obtains the relation information and reads the appropriate input relations from the disk. These relations are then sent to the master in partitions. The master node then sends them to the corresponding slaves.

A slave process is responsible for the execution of a join; each slave is assigned to a pipeline stage (no intra-operator is used in our experiments), where it receives partitions of the inner relation (or a base relation) from the master. However, the first slave of the pipeline receives two input relations from the master. In NSPHJ, the join process begins after receiving all partitions from the inner relation and a partition from the outer relation. In SPHJ, the join process starts immediately after one partition from either outer relation or inner relation arrives. The resulting tuples are sent to the next slave in the pipeline. The next slave receives the intermediate result and uses it as input for its local join. The number of tuples to be sent to the next slave is limited by the buffer size set for the outer relations. If the number of tuples of the result relation exceeds the buffer size, they are sent to the next stage.

Algorithm 5.4.1 shows the pseudocode for the master process.

```
Algorithm 5.4.1: PHJONMASTER($pS, bRs, nBRs, nStages$)

{Master reads the relations and distribute them to slaves}
{Input: partitionSize $pS$;
        base relations for this query $bRs$;
        the number of base relations $nBRs$;
        the number of pipeline stage $nStages$;
Output: none}

{base relations are read in parallel}
for $i \leftarrow 0$ to $nBRs$
    do { $X \leftarrow bRs[i]$;
         $nPartitions \leftarrow X/pS$
         for $j \leftarrow 0$ to $nPartitions$
             do { $X^j \leftarrow read(\text{“X”}, j * pS, (j+1) * pS - 1)$
                  $SLx \leftarrow assign(X)$;   {according to the query tree}
                  $send(SLx, \text{“Relation”}, X^j)$
$send(SL_0, \text{“ProcessEnd”}, null)$   {first slave on the pipeline}
$recv(SL_{nStages-1}, \text{“ProcessEnd”})$   {last slave on the pipeline}
```

The pseudocode for slave$_i$, which works on ith join of the query, of NSPHJ is shown in Algorithm 5.4.2.

```
Algorithm 5.4.2: NSPHJONSLAVE_i(n, pS, Ri, Ro)

{slave process for NSPHJ that works on join}
{Input: the number of buckets n;
        partitionSize pS;
        inner relation Ri;
        outer relation Ro;
Output: none}
{table building phase}
$nPartitions \leftarrow |Ri|/pS$
for $j \leftarrow 0$ to $nPartitions$
  do  $recv(master, \text{"Relation"}, Ri^j)$
      $Ri^j_{ALL} \leftarrow applyHash(Ri^j, n)$   {for inner relation}
      $Ri_{ALL} \leftarrow Ri_{ALL} \bigcup Ri^j_{ALL}$   {insert into the existing hash table}
{table probing phase}
{prev is the previous stage}
if $(i == 0)$
  then $prev \leftarrow master$
  else $prev \leftarrow SL_{i-1}$
{next is the next stage}
if $(i == (nStages - 1))$
  then $next \leftarrow master$
  else $next \leftarrow SL_{i+1}$
repeat
  $recv(prev, \text{"Relation"}, X)$   {For outer relation}
  $execLJ(Ri_{ALL}, X)$
      {send it to the next stage ($SL_{i+1}$) when the buffer is full}
until $recv(prev, \text{"ProcessEnd"})$   {from the previous stage}
$send(next, \text{"ProcessEnd"}, null)$   {to the next stage slave}
```

The pseudocode for slave$_i$, which works on ith join of the query, of SPHJ is shown in Algorithm 5.4.2.

Algorithm 5.4.3: SPHJONSLAVE(n)

```
{slave process for SPHJ that works on join}
{Input: the number of buckets $n$;
Output: none}
{table building and probing phase}
{$prev$ is the previous stage}
if ($i == 0$)
  then $prev \leftarrow master$
  else $prev \leftarrow SL_{i-1}$
{$next$ is the next stage}
if ($i == (nStages - 1)$)
  then $next \leftarrow master$
  else $next \leftarrow SL_{i+1}$
repeat
  $recv(prev, \text{"Relation"}, X)$
  $execLJ(X, Y_{ALL})$
  $X^p_{ALL} \leftarrow applyHash(X, n)$
  $X_{ALL} \leftarrow X_{ALL} \cup X^p_{ALL}$
until $recv(prev, \text{"ProcessEnd"}, null)$    {from previous stage}
$send(next, \text{"ProcessEnd"}, null)$    {to the next stage JE}
```

5.5 Experimental Methodology

In our experiments, we study the impact of background loads on the performance of NSPHJ and SPHJ algorithms with various buffer sizes given the following background load settings:

- No background load;
- Background load on the DB node;
- Identical background load on the slaves.

First, we test these cases without the Internet transfer delay. Then, we add the Internet transfer delay using the model specified in Section 4.3.4.

5.6 Experimental Results

This section discusses the results of our experiments conducted on a cluster described in Table 4.4. The following parameters are used in the experiments. We used 8 slave processes on 8 nodes (one slave per node) during the experiments (except for the speedup experiments; in that case, the number of slave nodes is changed). The number of buckets is set to 1,000 and the number of tuples is fixed at 1 or 2 millions. The experiments were conducted with variable buffer sizes.

5.6.1 Performance without the Internet Transfer Delay

This subsection reports the results of the experiments of NSPHJ and SPHJ without the Internet transfer delay. First, we assume that there is no background load on any of the nodes. Then, identical background load is added on the DB node. Finally, identical background load is added on the slave nodes.

5.6.1.1 Performance with No Background Load

This subsection examines the base case in which there is no background load on the slave as well as on the DB nodes. This represents the case where a cluster is dedicated to database query processing.

Figures 5.3 and 5.4 show the execution time of NSPHJ and SPHJ when the buffer size varies and the number of slaves is 8 for 1 and 2 million tuples, respectively. The confidence interval is less than 1%. We did not include the case that the number of slaves is 1 in Figures 5.4 since it took too much time to execute.

While the performance superiority of NSPHJ is marginal, it is important to note that in other environments, SPHJ provided substantial performance improvement over NSPHJ, as mentioned in Section 2.4.2. When there is no background load to slow down the incoming partitions of the relations, the majority of the partitions of the base relations (or inner relations) R_i arrive at the join processes earlier than those

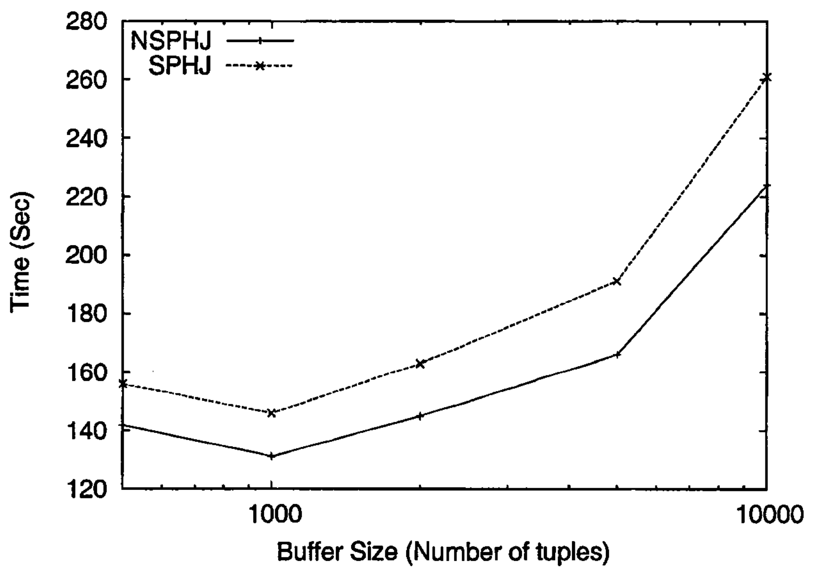

Figure 5.3: Performance of the NSPHJ and SPHJ when the number of slaves is 8 and the number of tuples is 1 million.

of the outer relations. As a result, the behaviour of the joins performed under SPHJ is similar to that performed under NSPHJ. That is, the execution patterns of the two join algorithms are similar, which results in similar execution time. The SPHJ takes more time, as it has to spend additional time to build the second hash table.

These figures show that the execution time increases with the buffer size beyond 1,000 tuples. This is the result of the trade-off between the communication time to send the tuples and the join time. We can reduce the communication time if we use larger buffers (thus fewer messages). However, larger buffer also means slaves have to wait for the whole buffer to arrive before working on the joins. Thus, we would like to use smaller buffers to reduce the join time. However, setting the buffer size small results in an increase in the number messages and induces more overheads.

As the size of the relations increases, the difference in the execution time of NSPHJ and SPHJ decreases as shown in Figure 5.4. This slight improvement in the performance of SPHJ (compared to the 1 million-tuple relations case) is due to the fact that, when more tuples are involved, there are more partitions participating in the

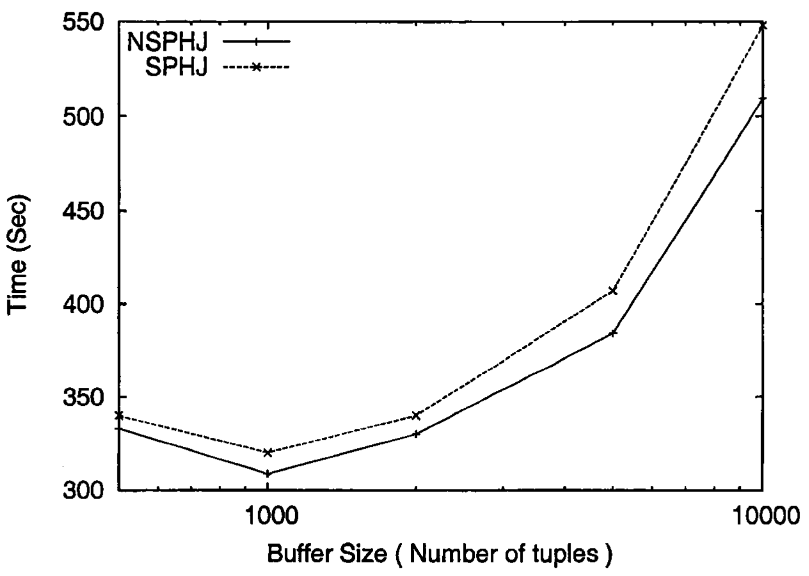

Figure 5.4: Performance of the NSPHJ and SPHJ when the number of slaves is 8 and the number of tuples is 2 million.

join processes for the same buffer size. Therefore, NSPHJ takes longer to receive the whole inner relation R_i. Alternatively in the SPHJ algorithm, the join process takes place earlier, as soon as both partitions of R_i and R_o have been received, and therefore the smaller the buffer size, the sooner the join process can start. This leads to better performance with SPHJ as the buffer size decreases. This observation implies that, with very large relations, the larger the ratio of the relation size to buffer size, the larger the benefits of using SPHJ in place of NSPHJ.

Our results are in contrast to the substantial performance advantages achieved by using SPHJ in data integration systems that were explained in Section 2.1.3.2. There are several reasons for this performance difference: first, in data integration systems, data come from different sources and their query processing may experience long, highly variable delays in delivering the tuples. However, in a cPDBMS, all the data comes from a single source (the database node). We will study this case in Section 5.6.2. Also, in NSPHJ which waits to receive the complete inner relation, it is important to identify/predict which of the two relations is smaller. This is because its performance is sensitive to the selection of inner relation. NSPHJ gives better performance if we select smaller of the two relations as the inner relation. SPHJ is symmetric and is not sensitive to the assignment of the inner and outer relation. In data integration systems, it is difficult to predict the size of the relations. In a cPDBMS, the relation size information is available in the system catalog. Our results suggest that, when using a cPDBMS, NSPHJ performs as well as or better than SPHJ. Implementing NSPHJ is also more efficient because it demands less memory by building only a single hash table.

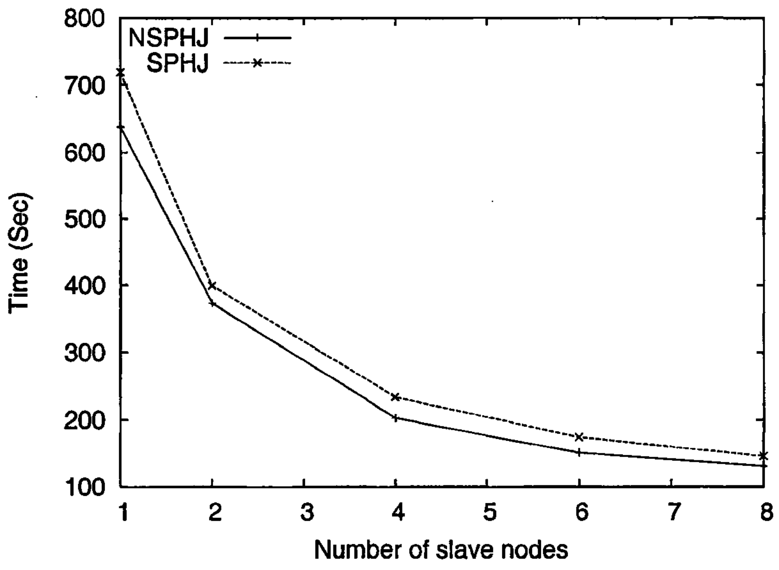

Figure 5.5: The execution time of the NSPHJ and SPHJ when buffer size is 1000 (14 KBytes) and the number of tuples is 1 million.

The execution time of NSPHJ and SPHJ algorithms for 1 and 2 million tuples are shown in Figures 5.5 and 5.6, respectively. The confidence interval is less than 0.1%.

This data also show that NSPHJ provides better performance than SPHJ when the number of slaves is varied. The performance difference between NSPHJ and SPHJ in Figures 5.5 and 5.6 is 10% and 4% when the number of slaves is 8 for 1 and 2 million tuples, respectively.

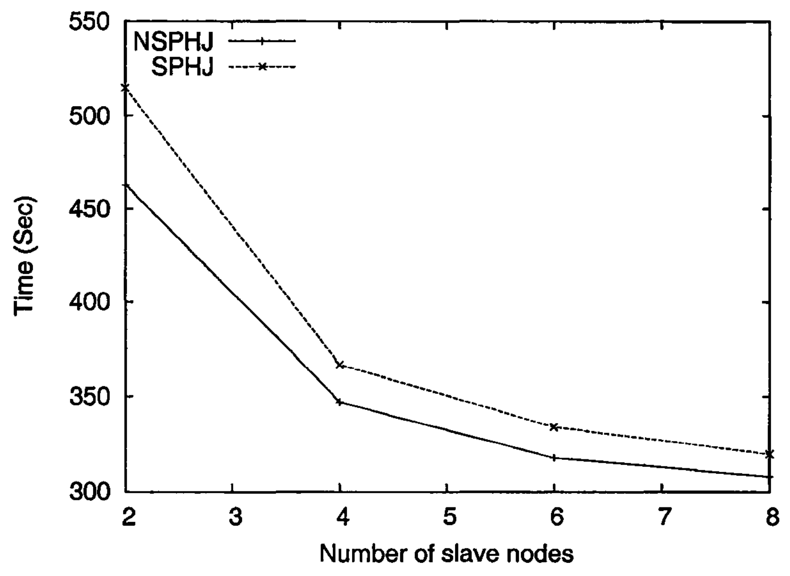

Figure 5.6: The execution time of the NSPHJ and SPHJ when buffer size is 1000 (14 KBytes) and the number of tuples is 2 million.

5.6.1.2 Impact of Background Load on Database Node

In this section, we consider the effect of having a background load only on the DB node. The results for the 1 million tuple relations are shown in Figures 5.7 and 5.8 when background load on database node is 5 and 10 processes, respectively. From the data presented in these figures, it can be seen that the presence of the background load on the DB node favours SPHJ. As shown in these figures, SPHJ outperforms NSPHJ when the buffer size is small. This is due to the fact that the presence of a background load on the DB node slows down the flow of R_i partitions to the slave nodes, causing a mixed execution pattern to likely occur. This kind of execution pattern improves the performance of SPHJ. Small buffer size also helps boost the performance of SPHJ as smaller buffer sizes means slower NSPHJ performance. The reason that for this is that setting small buffer size requires more messages to transmit the whole inner relation. This makes the improvement of SPHJ significant enough to outperform NSPHJ. However, peek performance can be obtained when buffer size is 2000 tuples at which NSPHJ performs better. The performance difference between

NSPHJ and SPHJ is about 7% in Figure 5.7.

Figure 5.8 shows the peak performance can be obtained when the buffer size is 5000 at which NSPHJ is marginally better than SPHJ. Thus, SPHJ performs better as the background load increases. However, the performance difference between them is within 6%. Also, large buffer size is better for both algorithms when these in higher background load on the DB node because the relation arrival is delayed and more tuples in the buffer compensate for the delay.

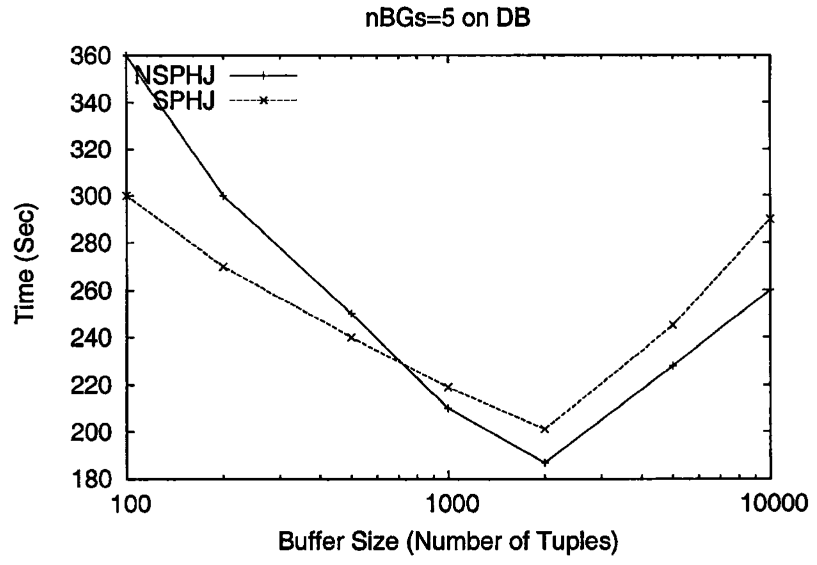

Figure 5.7: Performance of the NSPHJ and SPHJ when the background load on the DB node is 5 processes.

5.6.1.3 Impact of Identical Background Load on Slave Nodes

In this subsection, we consider identical load on all slaves nodes. Performance of the NSPHJ and SPHJ algorithms when identical background load of 5 and 10 processes is shown in Figures 5.9 and 5.10, respectively.

NSPHJ provides better performance than SPHJ as the case of no background load and background on the DB node case. The performance difference between NSPHJ and SPHJ is 5% when buffer size is 500 tuples are shown in Figure 5.9. The marginal

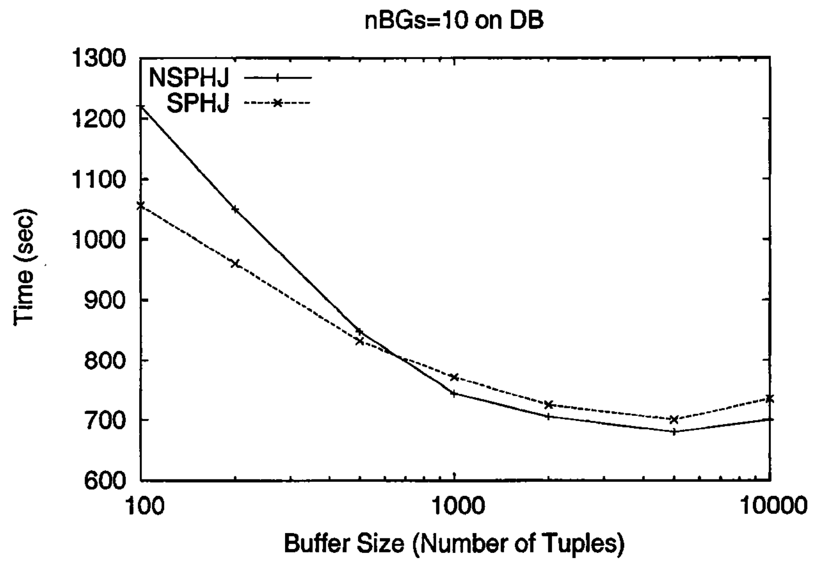

Figure 5.8: Performance of the NSPHJ and SPHJ when the background load on the DB node is 10 processes.

performance superiority of NSPHJ is discussed in Section 5.6.1.1.

Figure 5.10 shows that the difference in the performance of NSPHJ and SPHJ is almost negligible but NSPHJ is as good as SPHJ around the peak performance buffer size (500 tuples).

As opposed to the background load on the DB node case where larger buffer is favorable, the smaller buffer size is favorable. If it is too large, the pipeline stalls because of the interference during join operation on slave nodes.

5.6.2 Performance with the Internet Transfer Delay

This subsection reports the results of the experiments of NSPHJ and SPHJ when there is the Internet transfer delay. We use the same conditions as the previous section and add the Internet transfer delay while varying the number of delayed relations to be 1, 2 or 4. We have chosen joined relations from the end of the pipeline as delayed

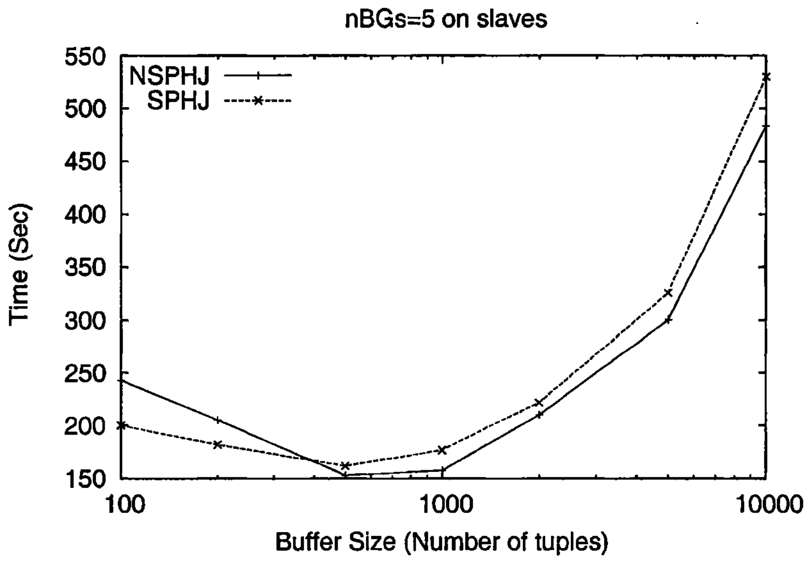

Figure 5.9: Performance of the NSPHJ and SPHJ when identical background load on slave nodes is 5 processes.

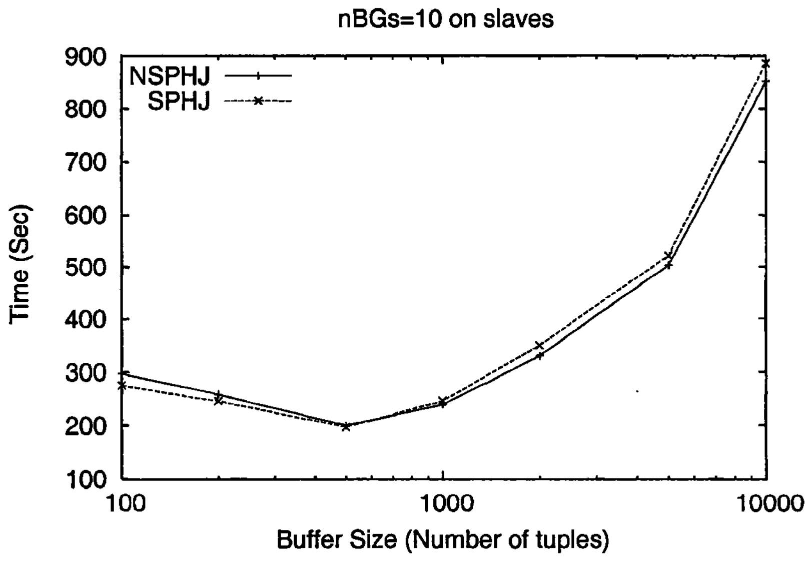

Figure 5.10: Performance of the NSPHJ and SPHJ when identical background load on slave nodes is 10 processes.

relations as shown in Figure 5.11[3]. We also changed the DB location as described in Section 4.5.6.

Please note only one DB is used when the DB location is local (no Internet transfer delay); otherwise, two DBs are used (one is for local and the other is for remote).

In this section, first, we present the performance results with the Internet transfer delay alone and later we add background load on the DB nodes and slave nodes as in the previous section.

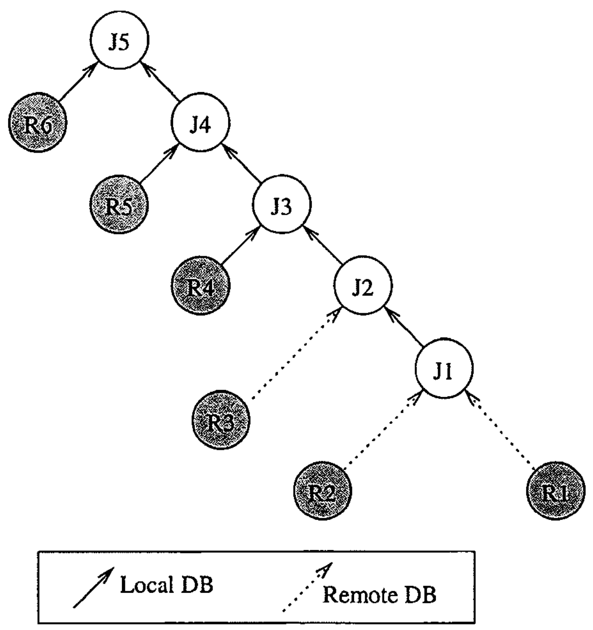

Figure 5.11: Right-deep query tree with the Internet transfer delay when the number of delayed relations is 3.

5.6.2.1 Performance with the Internet Transfer Delay Alone

Effect of DB location Figure 5.12 shows the execution time with no background load when we change the DB location. The confidence interval is less than 1%.

[3]We tried different combinations for choosing delayed relations but we did not observe any significant difference in results.

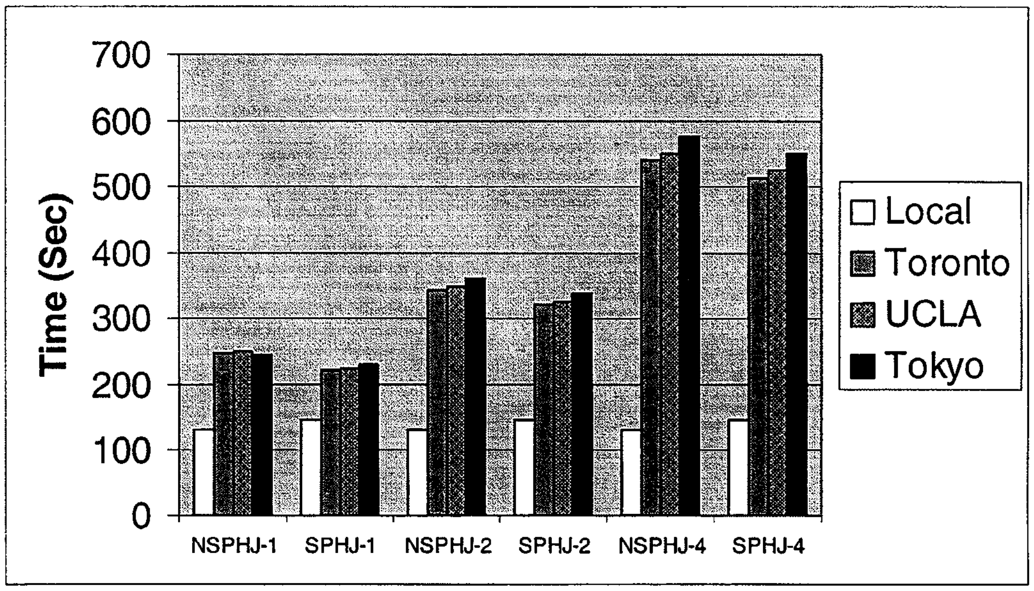

Figure 5.12: Performance of the NSPHJ and SPHJ with no background load when the DB location is changed (the number in the x-axis labels indicates the number of delayed relations).

It is surprising that the DB location does not affect the performance of both algorithms. Both algorithms can absorb the relation arrival delay and rate fluctuation by working on joins while waiting for the relation.

However, the more delayed relations there are, the more the performance degradation. In the case when the number of delayed relations is 4, the slowdown rate from Toronto to Tokyo is 6% (NSPHJ) and 7% (SPHJ). Figure 5.12 also shows the effectiveness of SPHJ over NSPHJ when varying the DB location and the number of relations. The performance difference is almost the same for all the DB locations.

Effect of the number of delayed relations Figure 5.13 shows the execution time with no background load and the Internet transfer delay when we vary the number of delayed relations. This figure shows the execution time is almost proportional to the number of delayed relations. When the DB location is Tokyo (NSPHJ-Tokyo), NSPHJ is more affected by the change in the number of delayed relations than the

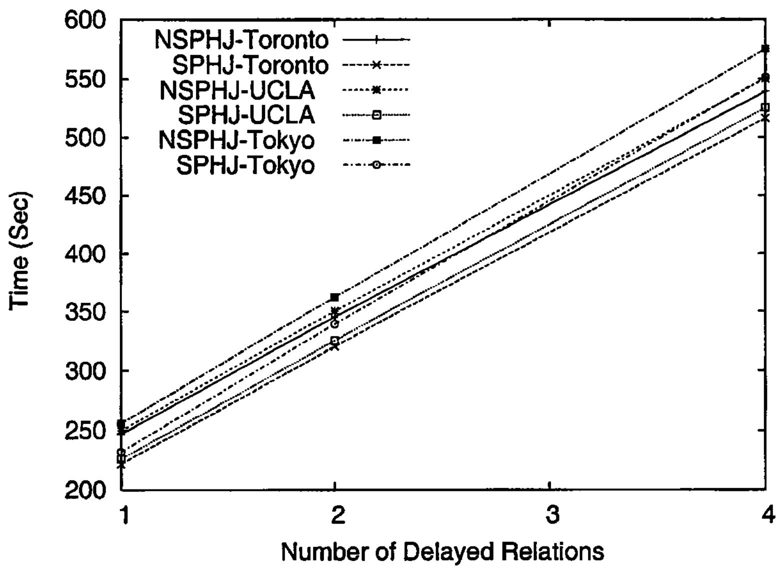

Figure 5.13: Performance of the NSPHJ and SPHJ with no background load when the number of delayed relations is varied.

others since the line is steeper than the others.

5.6.2.2 Impact of Background Load on Database Node

Effect of DB location Figures 5.14 and 5.15 show the execution time when we change the DB location and the background load on the DB node of 5 and 10 processes, respectively. The confidence interval is less than 2.6%. Their execution time is much larger than the no background case (shown in Figure 5.12).

These figures show that performance is not as largely affected by the DB location change as no background load. Besides, the difference between NSPHJ and SPHJ is almost the same as in the no background case. It means that a combination of the Internet transfer delay and background load on the DB node together has an effect on both the algorithms.

Comparing Figure 5.15 with Figure 5.14 and 5.12, we notice that the difference between local and other three DB locations is much smaller in Figure 5.15. This is because higher DB node background load causes extreme delay in the pipeline and the

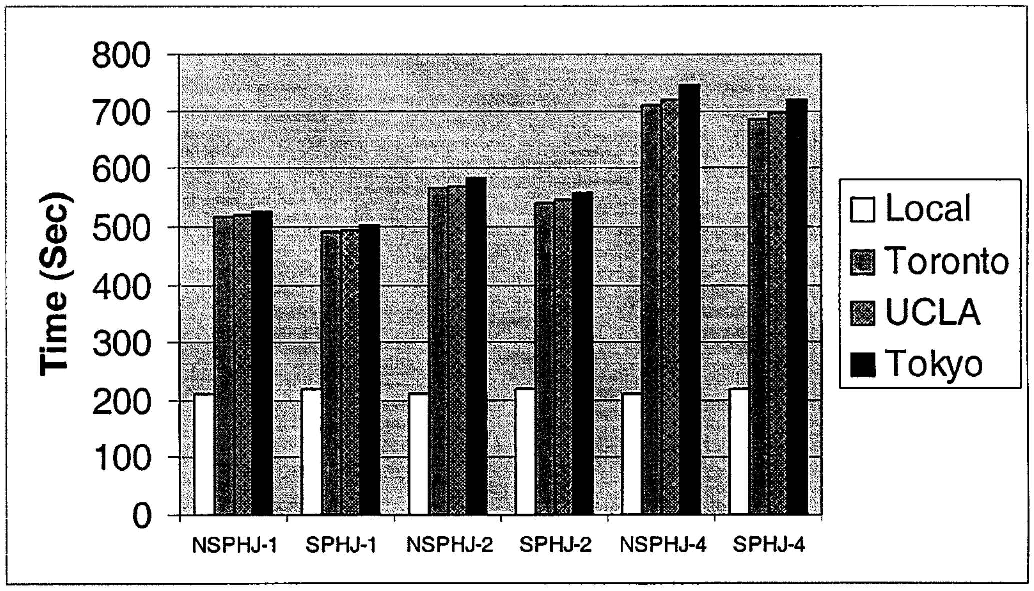

Figure 5.14: Performance of the NSPHJ and SPHJ when background load on the DB node is 5 processes.

Internet transfer delay does not make much difference. Both the algorithms exhibit this effect.

Effect of the number of delayed relations Figures 5.16 and 5.17 show execution time with background load on database node of 5 and 10 processes, respectively, when the number of delayed relations changes. The performance difference between NSPHJ and SPHJ is almost the same as the difference with no background load.

Comparing Figure 5.17 with Figure 5.13, we notice that the slowdown rate is smaller and it is sub-linear. The reason for this is the same as that we discussed for the database location change scenarios. This is because higher DB node background load causes extreme delay in the pipeline and the Internet transfer delay does not make much difference.

5.6.2.3 Impact of Identical Background Load on Slave Nodes

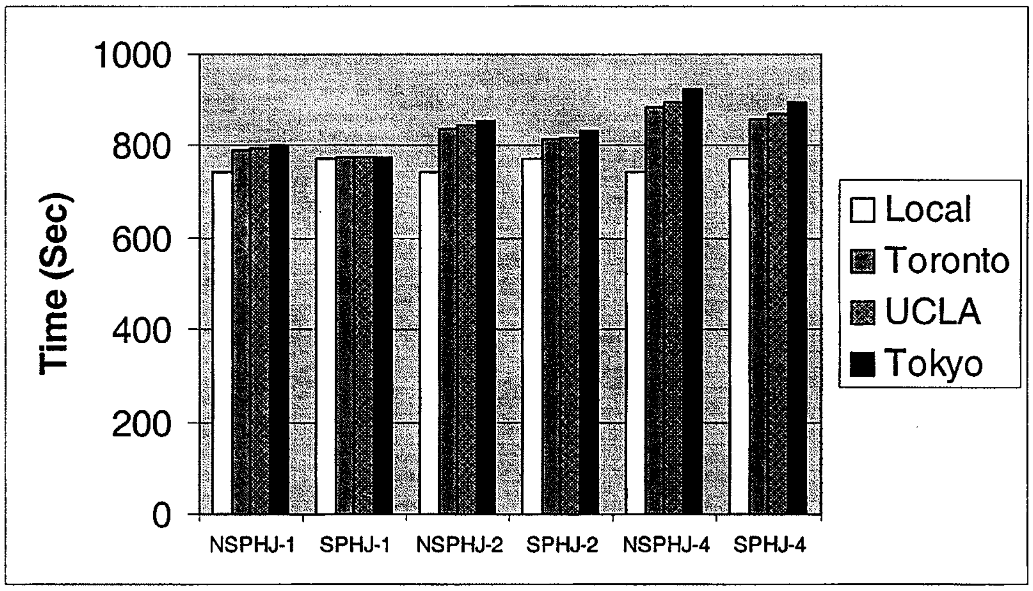

Figure 5.15: Performance of the NSPHJ and SPHJ when background Load on DB node is 10 processes.

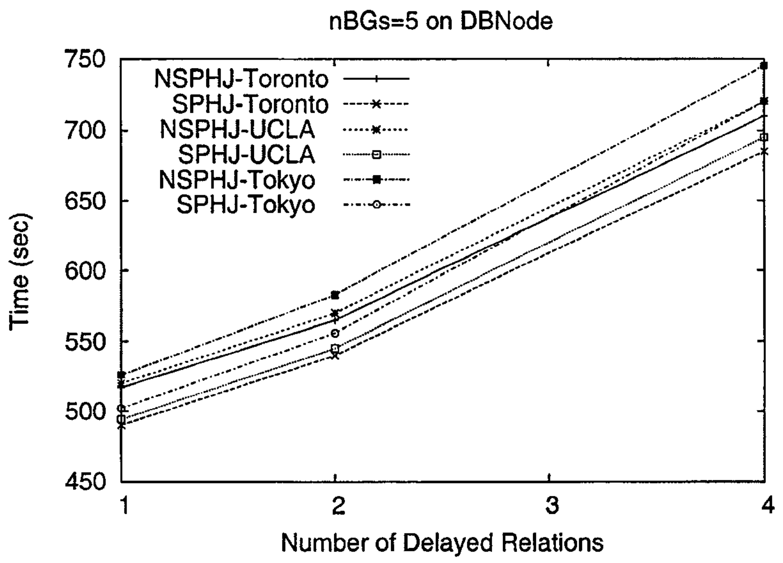

Figure 5.16: Performance of the NSPHJ and SPHJ when background load on the DB node is 5 processes.

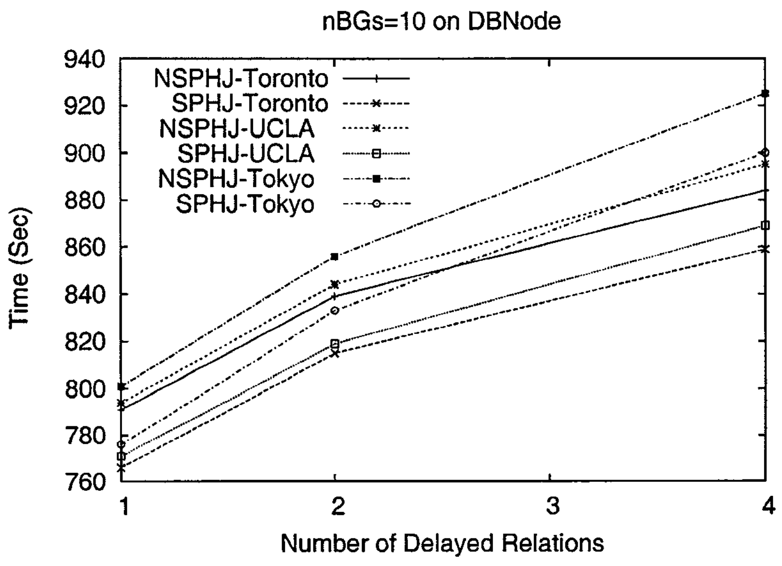

Figure 5.17: Performance of the NSPHJ and SPHJ when background load on the DB node is 10 processes.

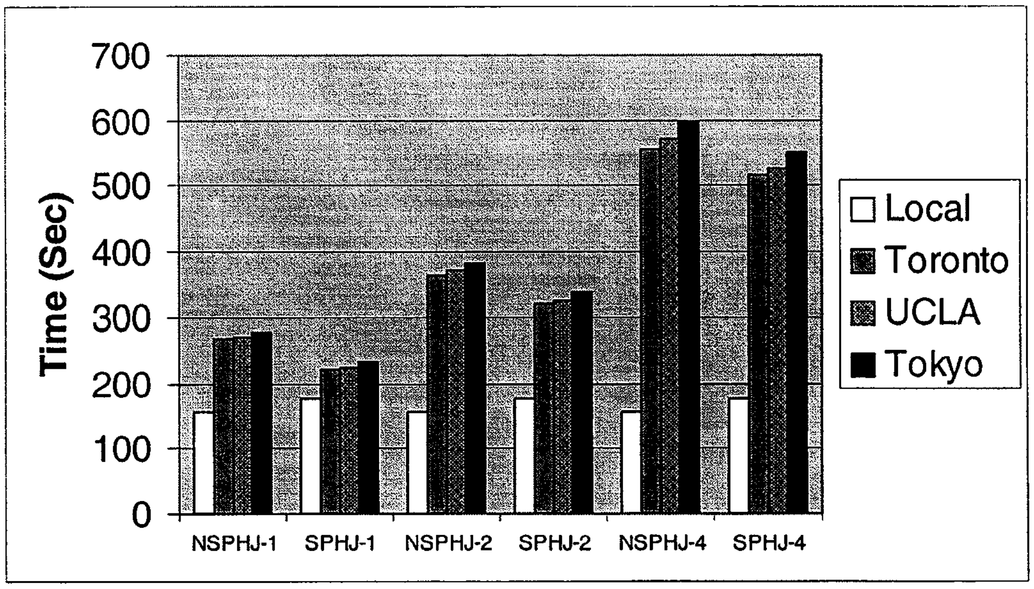

Figure 5.18: Performance of the NSPHJ and SPHJ when background load on slave nodes is 5 processes.

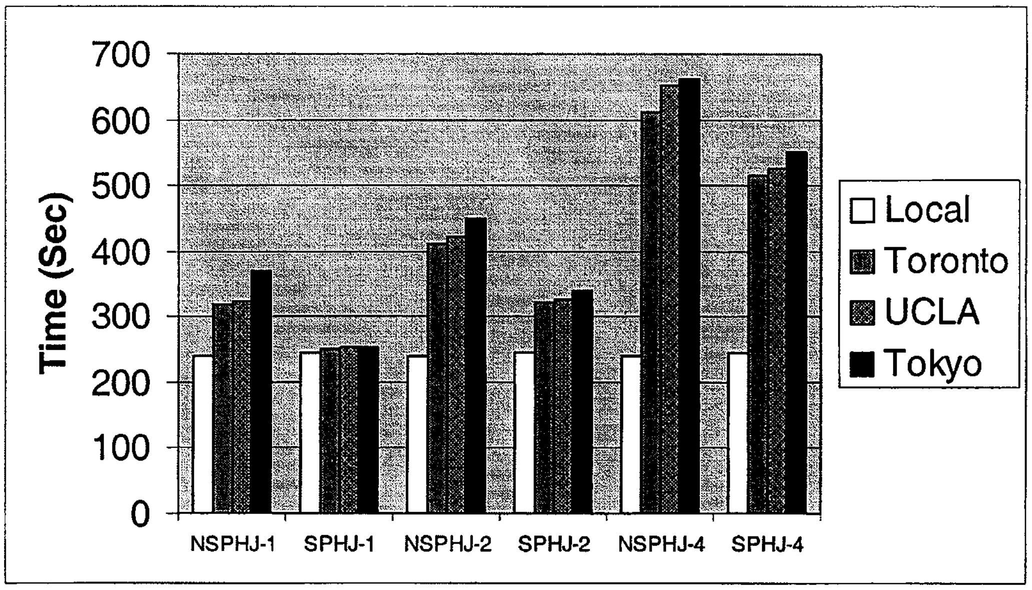

Figure 5.19: Performance of the NSPHJ and SPHJ when background load on slaves is 10 processes.

Effect of the DB location Figures 5.18 and 5.19 show the execution time when we change the DB location and the background load on the slaves is 5 and 10 processes, respectively. The confidence interval is less than 2.9%. The performance difference between NSPHJ and SPHJ gets larger as the background load increases as shown in Figure 5.19. For example, when the number of delayed relations is 4 and DB location is Tokyo, the difference is 8% (5 processes) and 20% (10 processes).

As for the DB location, NSPHJ is more affected by the DB location changes than SPHJ. Besides, the higher the background load the slave has, the more it is affected by the DB location. However, SPHJ is not affected much because of the adaptive nature of the algorithm. This is different from the previous case (Figures 5.14 and 5.15) where, when background load is on DB, both algorithms were affected.

Effect of the number of delayed relation Figures 5.20 and 5.21 show the execution time with the background load on slaves of 5 and 10 processes, respectively,

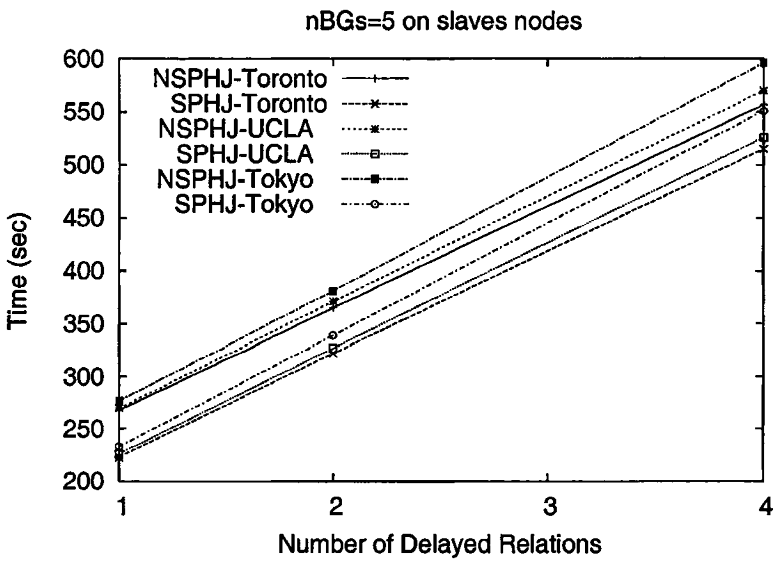

Figure 5.20: Performance of the NSPHJ and SPHJ when background load on slave nodes is 5 processes.

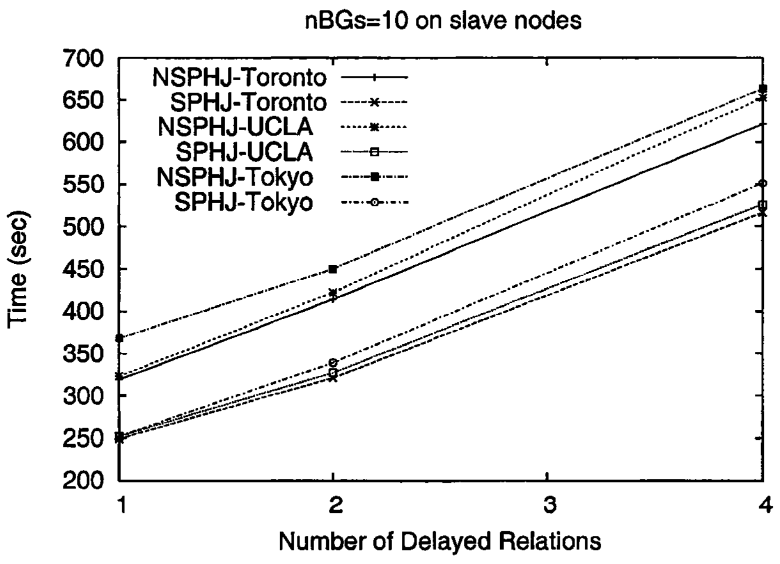

Figure 5.21: Performance of the NSPHJ and SPHJ when background load on slave nodes is 10 processes.

when the number of delayed relations is varied. The execution time of both algorithms is proportional to the number of delayed relations. NSPHJ is more affected by the change in number of delayed relations (e.g., NSPHJ-Tokyo). It is the same as that in the no background load case.

5.7 Conclusion

We studied the performance impact of non-symmetric and symmetric hash joins with and without Internet transfer delay. Previous studies have shown that the symmetric pipelined hash join (SPHJ) is superior to non-symmetric pipelined hash join (NSPHJ) for systems like the data integration systems for Internet.

Without the Internet transfer delay, the results indicate that the NSPHJ performs as well as or marginally better than SPHJ. Some of the reasons for this difference are due to the system characteristics. For example, in a cPDBMS, all data come from a single source as the database node supplies the tuples needed for performing the join operations of a query. We have presented performance sensitivity to a variety of factors including the background load on the database node and the slave nodes.

Regarding the buffer sizes, a larger buffer size is favorable when there is background load on the DB node. When the slave nodes have a background load, a smaller buffer size is beneficial.

However, with the Internet transfer delay, SPHJ is better than NPHJ, which supports the previous study results for a single-node data integration system. We did experiments with and without background load on database nodes and slave nodes. Also, the larger background load on either the database node or slave nodes, the better SPHJ performs. The database location does not have much effect on either of the two algorithms in most of the cases. However, when the DB node has high background load, there is an effect on both algorithms. When the slave nodes have a high background load, SPHJ can absorb the data arrival delay and/or data rate fluctuation caused by the database location while NPHJ cannot.

Chapter 6

Conclusion and Future Direction

This thesis investigated the query precessing algorithms for cluster-based Parallel Database Management Systems (cPDBMSs). Nodes in a cPDBMS experience dynamic load variations due to locally generated load. Furthermore, all data reside on a traditional DBMS node and processing nodes are used for query processing.

Under these conditions, there are three areas we examined in this thesis: non-symmetric single-join, symmetric single-join, and multiple-join algorithms.

6.1 Non-Symmetric Single-Join Algorithms

Non-symmetric single-join algorithms, which choose one of the relations to be joined as the main relation, have been studied without considering the Internet transfer delay of input relations. The algorithms studied here are the parallel versions of the nested-loop join and hash join algorithms and their load balancing/sharing algorithms.

First, as a preliminarily experiment, the performance of nested-loop join processing and its load sharing/balancing algorithms were evaluated on a Pentium-based heterogeneous cluster. The results prove the effectiveness of a dynamic load sharing/balancing algorithm on a cPDBMS.

Then, a new load-sharing algorithm, called *ChunkHJ* for single hash join processing is introduced. The novelty of this load-balancing algorithm is that it divides the hash buckets into chunks and uses them for load balancing. This new algorithm is compared with two other algorithms: an adaptive nested-loop join and an adaptive

GRACE join. These three algorithms were evaluated on the same cluster with skewed data and various non-query background loads. The results show that the new algorithm is the best among the three and should be used for non-symmetric single-join processing on clusters.

6.2 Symmetric Single-Join Algorithms

The symmetric single-join algorithms, which treat both relations equally, are studied with and without considering the Internet transfer delay of input relations.

ChunkHJ is modified for symmetric joins. This new algorithm, called SCHJ, is compared with dynamic round-robin algorithm, sampling algorithm (without the Internet transfer delay), incremental hash mapping algorithms (with the Internet transfer delay). These algorithms were evaluated on a Pentium 4 and Xeon based cluster. We tested them with background load and skew (scalar and Zipf) types. The results without the Internet transfer delay show that SCHJ is the best among these algorithms except when there is modest skew and no background load. Even with the Internet delayed data obtained from three different locations, SCHJ still performs better than the other incremenatal hash mapping based algorithms. DB location has little impact on the performance of SCHJ.

6.3 Multiple-Join Algorithms

We also studied the performance impact of non-symmetric and symmetric pipelined hash joins (NSPHJ and SPHJ) with and without the Internet transfer delay. Previous studies for a single-node DBMS, have shown that the SPHJ is superior to NSPHJ for systems like the data integration systems for the Internet. Without the Internet transfer delay, the results indicate that the NSPHJ performs marginally better than the SPHJ, which is in contrast to the results from the previous studies. We have identified the reasons for this behavior. Some of the reasons for this difference are due to the system characteristics. For example, in a cPDBMS we consider here, all the data come from a single source as the database node supplies the tuples needed for performing join operations of a query. With the Internet transfer delay, SPHJ is better, which supports the previous study results for a single-node DBMS. In addition, increasing background load on either the DB node or slave nodes favors SPHJ. As for the DB location, when there is a high background load on slaves, SPHJ can absorb the data arrival delay and/or data rate fluctuation caused by database location level.

6.4 Thesis Conclusion

In this thesis, we have identified the advantages of cPDBMSs over general PDBMSs. From these experimental results, we reach the following conclusions on join processing algorithms on a cPDBMS.

- For a single-join algorithm, when the information on both relations is known or one of the relation is smaller than the other or one of the relations arrives much faster than the other, the non-symmetric version of ChunkHJ should be used. Otherwise, the symmetric version of ChunkHJ should be used no matter what kind of skew and background load exist.

- For a multiple-join algorithm, if all relations come from local sources, then the non-symmetric version of pipelined hash join should be used. If any of the relations in the query come from other sources, the symmetric version of a pipelined hash join should be used.

6.5 Future Direction

This section gives a few directions for future research.

- *Algorithm implementation using mySQL or Oracle 10g*
  The proposed algorithms can be combined with the existing commercial database system. MySQL runs on our cluster and members in our research group implemented pipelined and non-pipelined algorithms on the same cluster we used in our experiments using MySQL, mpiJava, and JDBC. They obtained good speedups. We can combine ChunkHJ or SCHJ with these programs.

- *Algorithm evaluation (single- and multiple-joins) on a bigger cluster*
  Due to the limitation of the cluster size, our experiments were limited to 8 nodes or less. We would like to verify the effectiveness of our algorithms when we increase the number of processing nodes.

- *Combining SCHJ and symmetric pipelined hash join algorithm*
  Due to the limitation of the cluster size, we could not explore the possibility of combining single-join and multiple-join algorithms. With larger cluster sizes, it becomes important to combine the two algorithms. The number of nodes for each pipeline stage needs to be decided. At that time, it would be also good to consider the arrival speed of the input relation.

- *Application for database processing on a grid*
  The Internet transfer delay model we introduced can be used for grid computing. Global eCommerce companies like Amazon have databases spread all over the world, e.g. Amazon operates in six countries (Canada, USA, UK, Germany, France, and Japan). They also use horizontal scaling (parallel) techniques to keep up with huge volume of customers. In addition, most of the queries are read-only. Under these circumstances, the algorithms proposed in this thesis could be useful to speed up query processing.

Appendix A

Glossary

BG	Background load
ChunkNJ	Chunk-based Nested-loop Join
ChunkHJ	Chunk-based Hash Join
cPDBMS	cluster-based PDBMS
DD	Demand-Driven
DS	Dynamic Scheduling
GIHM	Greedy Incremental Hash Mapping
HT	High Threshold
IBG	Inverse Background load
JIHM	JSM-based Incremental Hash Mapping
JSM	Join State Matrix
LT	Low Threshold
NSPHJ	Non-Symmetric Pipelined Hash Join
PBG	Proportional Background load
PDBMS	Parallel Database Management System
PN	Processing Node
RR	Dynamic Round-Robin
SCHJ	Symmetric Chunk-based Hash Join
SMP	Sampling method

SMP	Symmetric Multiple Processor
SPHJ	Symmetric Pipelined Hash Join
SS_EQ	Static Scheduling with Equal distribution
SS_PR	Static Scheduling with Proportional distribution
ST	Stop Threshold

Bibliography

[1] Mahdi Abdelguerfi and Kam-Fai Wong, editors. *Parallel Database Techniques.* IEEE CS Press, Los Alamitos, CA, 1998.

[2] Laurent Amsaleg, Michael J. Franklin, Anthony. Tomasic, and Tolga Urhan. Scrambling Query Plans to Cope with Unexpected Delays. In *Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, pages 208–219, Miami Beach, Florida, USA, December 1996. IEEE Computer Society Press.

[3] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995. available at http://www.cs.berkeley.edu/.

[4] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[5] Aslam Nomani and Lan Pham. *UB2 Universal Database for Windows High Availability Support Using Microsoft Cluster Server - Overview.* Technical report, IBM Corporation, May 2001. available at http://www-4.ibm.com/software/data/pubs/papers/db2mscs/db2mscs.pdf.

[6] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In Weidong Chen, Jeffery Naughton, and Philip A. Bernstein, editors, *The 2000 ACM SIGMOD International Conference on Management of Data,*

volume 29(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 261–272, Dallas, Texas, May 2000. ACM Press.

[7] Mark Baker and Rajkumar Buyya. Cluster Computing at a Grance. In *High Performance Cluster Computing*, volume 1, chapter 1. Prentice Hall, 1999.

[8] Mostafa Bamha and Matthieu Exbrayat. Pipelined parallelism for multi-join queries on shared nothing machines. In *Proceedings of the International Conference on Parallel Computing (ParCo 2003)*, September 2003.

[9] Chaitanya K. Baru, Gilles Fecteau, Ambuj Goyal, Hui-I Hsiao, Anant Jhingran, Sriram Padmanabhan, and Walter G. Wilson. An Overview of DB2 Parallel Edition. In Michael J. Carey and Donovan A. Schneider, editors, *the 1995 ACM SIGMOD International Conference on Management of Data*, pages 460–462, San Jose, California, 22–25 May 1995. ACM Press.

[10] Giannis Bozas, Michael Fleischhauer, and Stephan Zimmermann. PVM Experiences in Developing the MIDAS Parallel Database System. In *European Conference on Parallel Computing (EURO-PAR)*, volume 1332 of *Lecture Notes in Computer Science*, pages 427–434, Crakow, Poland, November 1997. Springer-Verlag.

[11] Lionel Brunie, Matthieu Exbrayat, and Andre Flory. *Parallel Evaluation of Relational Queries on a Network of Workstations.* Technical-Report RR1999-22, Inria, Institut National de Recherche en Informatique et en Automatique, LIP, ENS Lyon, Lyon, France, March 1999. Also available as Research Report RR-3638, INRIA Rhone-Alpes.

[12] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. Scheduling Problems in Parallel Query Optimization. In ACM, editor, *PODS '95. Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1995*, volume 14, pages 255–265, San Jose, California, May 1995. ACM Press.

[13] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ : A Scalable Continuous Query System for Internet Databases. In *The 2000 ACM*

SIGMOD International Conference on Management of Data, volume 29(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 379–390, Dallas, Texas, 2000. ACM Press.

[14] Ming-Syan Chen, Mingling Lo, Philip S. Yu, and Honesty C. Young. Applying Segmented Right-Deep Trees to Pipelining Multiple Hash Joins. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):656–668, August 1995.

[15] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Optimization of Parallel Execution for Multi-Join Queries. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):416–428, June 1996.

[16] Compaq Computer Corporation. Compaq NonStop Himalaya K2000SE Server, 2000. available at http://www.compaq.com/.

[17] Compaq Computer Corporation. NonStop SQL/MP – Reliable, Parallel, Scalable Database Services, 2000. available at http://www.compaq.com/.

[18] Sivarama P. Dandamudi. Using Workstations for Database Query Operations. In *International Conference of Computers and Their Applications*, pages 100–105, Tempe, Arizona, October 1997.

[19] Sivarama P. Dandamudi and G. Jain. Architectures for Parallel Query Processing on Networks of Workstations. In *International Conference of Parallel and Distributed Computing Systems*, pages 444–451, New Orleans, Louisiana, October 1997.

[20] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High-Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[21] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and Srinivasan Seshadri. Practical Skew Handling in Parallel Joins. In *The 18th International Conference on Very Large Data Bases (VLDB)*, pages 27–40, Vancouver, Canada, August 1992.

[22] Matthieu Exbrayat and Harald Kosch. Offering Parallelism to a Sequential Database Management System on a Network of Workstations Using PVM. In Marian Bubak, Jack Dongarra, and Jerzy Wasniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, pages 457–462, Crakow, Poland, November 1997. Springer-Verlag.

[23] Matthiew Exbrayat and Lionel Brunie. A PC-NOW Based Parallel Extension for a Sequential DBMS. In *International Parallel and Distributed Processing Symposium Workshops PC-NOW*, pages 91–100, Cancun, Mexico, May 2000.

[24] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference Manual.* Oak Ridge National Laboratory, May 1994.

[25] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *The 24th International Conference on Very Large Databases (VLDB)*, pages 86–97. Morgan Kaufmann Publishers, 1998.

[26] Mohammed Al Haddad and Jerome Robinson. Using a Network of Workstations to Enhance Database Query Processing Performance. In Yannis Cotronis and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting*, volume 2131 of *Lecture Notes in Computer Science*, pages 352–359, Santorini, Thera, Greece, September 2001. Springer-Verlag.

[27] Lilian Harada and Masaru Kitsuregawa. Dynamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems. In *The 4th International Symposium on Database Systems for Advance Applications (DASFAA)*, pages 246–255, Singapore, April 1995.

[28] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul Shah.

Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin: Special Issue on Adaptive Query Processing*, 23(2):7–18, June 2000.

[29] HP. Query processing using HP NonStop SQL/MP software white paper , 2004. available at http://h71033.www7.hp.com/object/QUERYTB.html.

[30] HP Java Project. mpiJava Home Page. available at http://www.javagrande.com/ mpiJava.html.

[31] Hui-I Hsiao, Ming-Syan Chen, and P. S. Yu. Parallel Execution of Hash Joins in Parallel Databases. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):872–883, August 1997.

[32] Hui-I. Hsiao, Ming-Syan Chen, and Philip S. Yu. On Parallel Execution of Multiple Pipelined Hash Joins. In Richard T. Snodgrass and Marianne Winslett, editors, *the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 185–196, Minneapolis, Minnesota, 1994. ACM Press.

[33] Kien A. Hua, Chiang Lee, and Chau M. Hua. Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):968–983, December 1995.

[34] Kien A. Hua and Wallapak Tavanapong. Performance of Load Balancing Techniques for Join Operations in Shared-nothing Database Management Systems. *Journal of Parallel and Distributed Computing*, 56(1):17–46, January 1999.

[35] IBM Corporation. IBM DB2 Universal Database Enterprise-Extended Edition (EEE) on IBM NUMA-Q Hardware Platforms. available at http://www-4.ibm.com/ software/data/db2/udb/numa-q/wp.pdf.

[36] IBM Corporation. Linear Scalability for Business Intelligence using DB2 on NUMA-Q, September 2000. http://www-4.ibm.com/software/data/db2/benchmarks/ 090500.html.

[37] IBM Corporation. DB2 Product Family, February 2001. available at http://www-4.ibm.com/software/data/db2/.

[38] IBM Corporation. A Quick Reference for Tuning DB2 Universal Database EEE, May 2002. available at http://www-106.ibm.com/ developerworks/db2/library/techarticle/ 0205parlapalli/0205parlapalli.html.

[39] IBM Corporation. Parallel Sysplex Cluster Technology, 2004. available at http://www-1.ibm.com/servers/eserver/zseries/pso/sysover.html.

[40] Kenji Imasaki and Sivarama Dandamudi. Performance Evaluation of Nested-loop Join Processing on Networks of Workstations. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS)*, pages 537–544, Iwate, Japan, July 2000.

[41] Kenji Imasaki and Sivarama Dandamudi. An Adaptive Hash Join Algorithm on a Network of Workstations. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages on CD–ROM, Fort Lauderdale, Florida, April 2002.

[42] Kenji Imasaki, Hong Nguyen, and Sivarama Dandamudi. Performance Comparison of Pipelined Hash Joins on Workstation Clusters. In *9th International Conference on High Performance Computing (HiPC2002)*, pages 264–275, Bangalore, India, December 2002.

[43] Kenji Imasaki, Hong Nguyen, and Sivarama Dandamudi. Performance of Pipelined Nested Loop and Hash Joins on a Workstation Cluster. In *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, Las Vegas, Nevada, June 2002.

[44] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data Integration. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *The 1999 ACM SIGMOD International Conference on Management of Data: SIGMOD '99*, volume 28(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 299–310, Philadelphia, PA, USA, June 1999. ACM Press.

[45] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive Query Processing for Internet Applications. *IEEE Data*

Engineering Bulletin: Special Issue on Adaptive Query Processing, 23(2):19–26, June 2000.

[46] Raj Jain. *The Art of Computer System Performance Analysis.* Wiley, 1991.

[47] Susheel Jalali and Sivarama Dandamudi. Pipelined Hash Joins Using Network of Workstations. In *14th International Conference on Parallel and Distributed Computing and Systems*, August 2001.

[48] Java Grande Forum. Java Grande Forum. available at http://www.javagrande .com/.

[49] Joe Chang. SQL Server Parallel Execution Plans, 2004. available at http://www.sql-server-performance.com/jc_parallel_execution_plans.asp.

[50] John A. Miller. JSIM: A Java-Based Simulation and Animation Environment. available at http://chief.cs.uga.edu/~jam/jsim/.

[51] Arthur M. Keller and Shaibal Roy. Adaptive Parallel Hash Join in Main-Memory Database. In *The First International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 58–67, Miami Beach, Florida, December 1991.

[52] Masaru Kitsuregawa, Masaya Nakayama, and Mikio Takagi. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *The 15th International Conference on Very Large Data Bases (VLDB)*, pages 257–266, Amsterdam, The Netherlands, August 1989.

[53] Bruch Lindsey. SMP Intra-Query Parallelism in DB2 UDB, February 1998. Database Seminar at U.C. Berkeley.

[54] Kevin Hao Liu. Performance Evaluation of Processor Allocation Algorithms for Parallel Query Execution. In *The 1997 ACM symposium on Applied Computing*, pages 393–402, San Jose, California, February 1997. ACM Press.

[55] Ming-Ling Lo, Ming-Syan Chen, Chinya V. Ravishankar, and Philip S. Yu. On Optimal Processor Allocation to Support Pipelined Hash Joins. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 69–78, Washington DC, 1993. ACM Press.

[56] Hongjun J. Lu, Beng-Chin Ooi, and Kian-Lee Tan, editors. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society Press, Los Alamitos, 1994.

[57] Hongjun J. Lu and Kian-Lee Tan. Load-Balanced Join Processing in Shared-Nothing Systems. *Journal of Parallel and Distributed Computing*, 23(3):382–398, December 1994.

[58] Gang Luo, Curt J. Ellmann, Peter J. Haas, and Jeffrey F. Naughton. A Scalable Hash Ripple Join Algorithm. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of data*, pages 252–262, Madison, Wisconsin, 2002. ACM Press.

[59] Holger Märtens. Skew-Insensitive Join Processing in Shared-Disk Database Systems. In *the International Workshop on Issues and Applications of Database Technology (IADT'98)*, pages 17–24, Berlin, Germany, July 1998.

[60] Holger Märtens. A Classification of Skew Effects in Parallel Database Systems. In *European Conference on Parallel Computing (EURO-PAR)*, pages 291–300, Manchester, United Kingdom, August 2001.

[61] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tennessee, Knoxville, Tennessee, June 1995.

[62] Microsoft Corporation. Microsoft SQL Server. available at http://www.microsoft .com/SQL/.

[63] Priti Mishra and Margaret H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[64] Matt M. Mutka and Miron Livny. The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, 12(4):269–84, July 1991.

[65] Mike Muuss. The Story of the PING Program. available at http://ftp.arl.mil/~mike/ping.html.

[66] mySQL AB. Introduction to MySQL Cluster for MySQL Users, 2004. available at http://www.mysql.com/.

[67] mySQL AB. MySQL AB Launches MySQL Cluster - the First Open Source Database Clustering Solution, April 2004. available at http://www.mysql.com/.

[68] mySQL AB. MySQL Cluster Administration Guide, February 2004. available at http://www.mysql.com/.

[69] Oracle Corporation. Oracle Parallel Server: Solutions for mission critical computing, 1999. available at http://oracle.com/database/options/parallel.html.

[70] Oracle Corporation. Extreme Scalability With Oracle Applications:XS Benchmarking and Oracle Parallel Server, 2000. available at http://otn.oracle.com/deploy/performance/pdf/extreme_scalability.pdf.

[71] Oracle Corporation. Oracle8i Parallel Server – An Oracle Technical White Paper, January 2000. available at http://otn.oracle.com/deploy/availability/pdf/Oracle8i_Parallel_Server_Whitepaper.pdf.

[72] Oracle Corporation. Oracle9i Real Application Clusters, 2001. available at http://otn.oracle.com/products/oracle9i/pdf/appclusters_cache.pdf.

[73] Oracle Corporation. Scalability and Performance with Oracle9i Database, June 2001. available at http://otn.oracle.com/products/oracle9i/pdf/scalability_performance_9i.pdf.

[74] Oracle Corporation. Oracle Grid Control Reduces the Complexity and Cost of Managing Mission-Critical Business Services, April 2004. available at http://www.oracle.com/enterprise_manager/docs/Summit_Strategies_Oracle_Grid_Control.pdf.

[75] Donovan A. Schneider and David J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment. *SIGMOD Record*, 18(2):110–121, June 1989.

[76] Mehul A. Shah, Joseph M. Hellerstein, and Sirish Chandrasekaranand Michael J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering*, pages 25–36, Bangalore, India, March 2003. IEEE Computer Society.

[77] Cyrus Soleimany. Distributed Parallel Query Processing on Network of Workstations. Master's thesis, Carleton University, Ottawa, Canada, 1998.

[78] Cyrus Soleimany and Sivarama P. Dandamudi. Distributed Parallel Query Processing on Networks of Workstation. In *High-Performance Computing and Networking, 8th International Conference (HPCN Europe)*, volume 1823 of *Lecture Notes in Computer Science*, pages 427–436, Amsterdam, The Netherlands, May 2000. Springer-Verlag.

[79] W. Richard Stevens. *UNIX Network Programming; Volumne I, Networking APIs: Sockets and XTI, Second Edition.* Prentice Hall PTR, 1998.

[80] Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, March 1986.

[81] Takayuki Tamura, Masato Oguchi, and Masaru Kitsuregawa. High Performance Parallel Query Processing on a 100 Node ATM Connected PC Cluster. *IEICE Transactions on Information and Systems*, 1(1):54–63, January 1999.

[82] Thierry Cruanes and Benoit Dageville and Bhaskar Ghosh. Parallel SQL Execution in Oracle 10g. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 850–854. ACM, 2004.

[83] Walid R. Tout and Sakti Pramanik. Distributed Load Balancing for Parallel Main Memory Hash Join. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):841–849, August 1995.

[84] Tolga Urhan and Michael J. Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical Report CS-TR-3994, University of Maryland, College Park, February 1999.

[85] Shivakumar Venkataraman and Tian Zhang. Heterogeneous Database Query Optimization in DB2 Universal DataJoiner. In *The 24th International Conference on Very Large Data Bases (VLDB)*, pages 685–689, New York, New York, August 1998. Morgan Kaufmann.

[86] Efstratios Viglas and Jeffrey F. Naughton. Rate-Based Query Optimiation for Streaming Information Sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM Press, 2002.

[87] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *The 17th International Conference on Very Large Data Bases (VLDB)*, pages 537–548, Barcelona, Catalonia, Spain, September 1991.

[88] Annita N. Wilschut, Peter M. G. Apers, and Jan Flokstra. Parallel query execution in PRISMA/DB. In Pierre America, editor, *Proceedings of Parallel Database Systems*, volume 503 of *LNCS*, pages 424–433, Berlin, Germany, September 1991. Springer.

[89] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel Evaluation of Multi-join Queries. In ACM, editor, *1995 ACM SIGMOD International Conference on Management of Data*, volume 24(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 115–126, San Jose, California, May 1995. ACM Press.

[90] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. New Algorithms for Parallelizing Relational Database Joins in the Presence of Data Skew.

IEEE Transactions on Knowledge and Data Engineering, 6(6):990–997, December 1994.

[91] Boquan Xie and Sivarama P. Dandamudi. Hierarchical Architecture for Parallel Query Processing on Networks of Workstations. In *The 5th International Conference on High Performance Computing*, Chennai, Madras, India, December 1998.

[92] Mohammed Javeed Zaki, Wei Li, and Srinivan Parthasarathy. Customized Dynamic Load Balancing for a Network of Workstations. Technical Report TR602, University of Rochester, Computer Science Department, December 1995.

[93] Sijun Zeng and Sivarama P. Dandamudi. Centralized Architecture for Parallel Query Processing on Networks of Workstations. In *High-Performance Computing and Networking, 7th International Conference (HPCN Europe)*, volume 1593 of *Lecture Notes in Computer Science*, pages 683–692, Amsterdam, The Netherlands, May 1999. Springer-Verlag.

[94] Xiaoding Zhao, Roger G. Johnson, and Nigel J. Martin. DBJ — A Dynamic Balancing Hash Join Algorithm in Multiprocessor Database Systems. In *The 4th International Conference on Extending Database (EDBT)*, pages 301–308, Cambridge, United Kingdom, March 1994.

[95] Stephan Zimmermann. *Evaluation and Tuning of parallel Database Systems.* PhD thesis, Technische Universitat Munchen, 2000. available at http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2000/zimmermann.html.

[96] George K. Zipf. *Human Behavior and the Principle of Least effort : An Introduction to Human Ecology.* Addison-Wesley, 1949.