INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Pipelined Hash Joins using Network of Workstations

by

Susheel Jalali, B.E. (Hons.) Computer Science

A thesis submitted to

The Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

December 1999

© Copyright 1999, Susheel Jalali, B.E (Hons.) Computer Science

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48442-4

Canada

The undersigned hereby recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis,

Pipelined Hash Joins using Network of Workstations

submitted by

Susheel Jalali, B.E. (Hons.) Computer Science

in partial fulfillment of the requirements
for the Degree of Master of Computer Science

Dr. Evangelos Kranakis
(Director, School of Computer Science)

Dr. Sivarama P. Dandamudi
(Thesis Supervisor)

Carleton University

December 1999

Abstract

Demands for high transaction processing rates from database systems led to the use of parallel processing techniques for transactions and queries. Most commercial parallel database systems suffer from exceptionally high hardware and software costs. Availability of public domain software (such as PVM and MPI) that harnesses several workstations on a LAN into a virtual parallel machine provides a welcome alternative for parallel query processing. This thesis exploits such a Network of Workstations for intra-query (pipelined) and intra-operation (partitioned) parallelism using the Hash Join algorithm in a centralized query processing architecture. A comprehensive performance evaluation suggests that Pipelined Hash Joins exhibit significant improvement in response time for complex queries and for queries consuming very large input relations.

Acknowledgements

I am grateful to my thesis supervisor, Dr. Sivarama P. Dandamudi, for his research advice, ongoing guidance and personal encouragement throughout this thesis. I also thank the administration, technical and academic staff in the school for their professional conduct.

This thesis would have been incomplete without the unlimited understanding, co-operation, and good wishes of my family. I thank my parents Shri Hriday Nath and Shrimati Santosh for providing the inspiration. I thank my sisters Subhadra, Sunaina and Sukriti for their love and encouragement. I owe a debt to my wife, Naina, and children Shruti and Samdarsh for putting up with the long hours that made this work possible.

I also dedicate this thesis to the tens of thousands of Kashmiris who fell victim to international terrorism in Kashmir and the hundreds of thousands of Kashmiri Pandits who silently endured decades of discrimination and persecution in their own aboriginal homeland, followed by terrorist threats to their life and religion. Their Gandhian response of withdrawal rather than violence has caused them untold suffering in exile for over ten years. They are yet waiting to return to their homes, lands and livelihoods.

Table of Contents

Chapter 1	Introduction	1
1.1	Database Primer	3
1.2	Parallel Processing Architectures	6
1.3	Query Processor	8
1.3.1	Query Execution Time	9
1.3.2	Parallel Query Processing	10
1.4	Parallel Database System Architectures	12
1.5	PVM: A Parallel Programming Environment	18
1.6	Contributions of Thesis	20
1.7	Overview of Thesis	21
Chapter 2	Background	23
2.1	Query Trees	24
2.1.1	Left Deep Trees	26
2.1.2	Right Deep Trees	27
2.1.3	Bushy Trees	28
2.1.4	Segmented Right Deep Trees	29
2.2	Join Algorithms	31
2.2.1	Nested-Loops Join	32
2.2.2	Sort-Merge Join	33
2.2.3	Hash Join	34
2.3	Query Optimization Strategies	35
2.4	Choice of Join Algorithm	37
2.5	Choice of Query Trees	39
2.6	Skew	41
Chapter 3	Parallel Hash Joins using NOW	43
3.1	Introduction to NOW	43
3.2	Query Architecture on NOW	44
3.2.1	Centralized architecture	45
3.2.2	Distributed architecture	47
3.2.3	Hierarchical architecture	48
3.3	NOW System Setup	49
3.4	Database Model	53
3.5	Query Workload Model	55
3.6	Parallel Virtual Machine (PVM)	56
3.7	Description of Algorithms	60
3.7.1	Building Blocks	61
3.7.2	Scheduler	70
3.7.3	Reader	71
3.7.4	Worker	73
3.7.5	BST	74
3.7.6	RDT	79
3.8	Analysis of a Generic Query	84
3.8.1	Table Building (TB) phase	86
3.8.2	Tuple Probing (TP) phase	88
3.9	Example Comparison of RDT and BST	91
3.9.1	BST Analysis	94
3.9.2	RDT Analysis	95
3.9.3	Comparison of RDT and BST	96

Chapter 4	Performance Analyses	98
4.1	Introduction	98
4.1.1	Input Parameters	99
4.1.2	Description of Experiments	100
4.2	Effect of Bucket Size	102
4.3	Slow Network Experiments	102
4.3.1	Effect of Query Complexity	103
4.3.2	Effect of Input Selectivity	104
4.3.3	Effect of Background Load	107
4.3.4	Summary of Slow Network Experiments	109
4.4	Fast Network Experiments	111
4.4.1	Effect of Number of Join Nodes	111
4.4.2	Effect of Selectivity	113
4.4.3	Effect of Large Table Size	115
Chapter 5	Conclusions and Future Work	119
5.1	Summary	119
5.2	Contributions of the Thesis	121
5.3	Future Work	122
Chapter 6	List of Abbreviations	125
Chapter 7	References	126

Table Of Figures

Figure 1	The ANSI/SPARC 3-Schema Architecture [based on TK78]	5
Figure 2	A shared-memory system	7
Figure 3	A distributed -memory system	7
Figure 4	Common Architectures for Parallel Database Systems [D97]	13
Figure 5	Types of Query Trees	26
Figure 6	Segmented Right Deep Tree	30
Figure 7	Network of Workstation System	50
Figure 8	Output of Sysinfo command on Triton	52
Figure 9	Comparison of RDT and BST for example query	92
Figure 10	Effect of BKTSIZE	102
Figure 11	Effect of Query Size under Slow LAN	104
Figure 12	Effect of Single Join Selectivity in Slow Network	105
Figure 13	Effect of Avg. Selectivity in Slow Network	106
Figure 14	Experiment with Low Background Load	108
Figure 15	Experiment with High Background Load	109
Figure 16	Effect of Query Size under Fast LAN	112
Figure 17	Effect of Single Join Selectivity in Fast Network	114
Figure 18	Effect of Average Selectivity in Fast Network	115
Figure 19	Effect of Query Size with Large Tables under Fast LAN	117

Table Of Tables

Table 1	Nodes of Triton System.....	51
Table 2	Power Units of each Triton Node.....	51
Table 3	Join Selectivities for Normal Sized Relations [X88].....	54
Table 4	Cost Variables for Analytic Comparison	85
Table 5	Output tuples for example scenario.....	91
Table 6	Analytical Comparison of RDT and BST for example query	93
Table 7	Input Relations for Single Join Selectivity	105
Table 8	The Input Relations for Average Selectivity Experiment.....	106
Table 9	The Input Relations for Large Input Experiment	116

Chapter 1 Introduction

The past decade has witnessed a proliferation of very large databases following the success of the relational database management systems (RDMS) and the successful introduction of advanced data models as in object oriented (ODMS) and semantic/knowledgebase (KBMS) databases.

On a parallel note, stupendous progress in the speed and cost of CPU MIPS, main memory as well as disk speeds has fueled even greater expectations for new application areas like just-in-time inventory, consumer preferences tracking, consumer credit, stock-market transactions, web-databases, telecommunications network management, genome databases, etc. These expectations typically involve the computation of sophisticated derived knowledge from multiple distributed instances of information base or database systems. Other examples are self-healing communications networks, flow-through service-and-network-provisioning, and flow-through travel-and-hotel ticketing.

Very large databases are often combined with artificial intelligence, expert system or real-time applications to partially automate repetitive logic-intensive tasks. An example is the delegation of self-diagnostic and performance gathering scripts to millions of network elements distributed across a large telecommunications network. Another

scenario scans millions of patient records to determine the occurrence or otherwise of certain complications due to given combinations of medications. Stock market, decision-support, banking and credit processing applications also have very high transaction processing rates.

The business environment of the late 90's, with massive amalgamations, rapid restructurings and takeovers, has caused scale up of even traditional applications as inventory management, production control and credit transactions.

The result is that demands for database performance, reliability and scalability are increasing at the same rate as the increase in speeds and capacities of CPU, disk and main memory. Once the limitations of mainframe-based transaction processing systems were realized, parallel database systems were invented as an initial solution, but their specialized and expensive nature has generated a need to look at other alternatives.

To remain ahead of the game, researchers have exploited multiprocessing and parallel processing techniques [DG92]. The initial flavor of parallel processing involved specialized supercomputing architectures, but a new research focus is to exploit commodity hardware by using the concept of Networks of Workstations. The rest of this chapter will define the foundations of the thesis work, including parallel query processing, architectural approaches to parallel transaction systems and an overview of this study.

1.1 Database Primer

When a user needs to answer a complex question against a “database”, a “query” is issued to answer that question. The processing necessary to translate such queries into a machine-understandable form, termed query processing, constitutes the most frequently used and expensive functionality in a database system.

A database is any organized collection of raw data, managed according to certain rules of access, which determine an acceptable level of integrity, timeliness, accuracy and durability for the collection. It could be as simple as a logical cluster of files on the flash ROM of a single-board computer or many such clusters in one or more formats spread across a LAN, MAN or WAN. The latter case would form a distributed heterogeneous database.

In its most common usage, the term database refers to a collection of data organized in one format and one set of access rules under one Database Management System (DBMS). An excellent account of the early history of DBMS occurs in the March 1976 issue of the ACM Computing Surveys. It includes a history and overview of the various data models [FS76] and a detailed discussion of the relational data model [C76].

Three major file organizations for databases were recognized until the late 1980's: hierarchical, network and relational which have been well defined in traditional textbooks [e.g. U82, D86]. Subsequently, Object-oriented [Kim89] and Semantic or knowledge-based [B80, C90] data models have been found useful for handling many complex applications.

Most DBMSs provide the benefit of data independence to applications via the 3-schema architecture finalized by ANSI/SPARC [TK78, Ch.4 of OzVa91]. It consists of:

- Internal Schema
- Conceptual Schema
- External Schema

A conceptual schema holds the logical data definitions and relationships such that the design and implementation of the schema presented in the graphical front-end (external schema) is independent of the internal schema representation on disk (see Figure 1).

By far the most prevalent and intensely used model continues to be the Relational Database Management System (RDMS) model [D86]. This model is not only used in many business-critical and large-scale applications, but has also been the longest surviving, widely researched, well standardized, best-supported and best-formalized model to date. The availability of nonprocedural languages for RDMS has significantly

improved application developer and end-user productivity, but at the same time allowed users to pose very complex queries involving billions of records at the touch of a button.

Figure 1 The ANSI/SPARC 3-Schema Architecture [based on TK78]

A good description of various issues related to RDMS, distributed DBMS architecture and design, query processing and optimization and multidatabases occurs in [OzVa91]. All of these areas are touched in the present work or the future work following from the present work.

1.2 Parallel Processing Architectures

Parallel processing systems are either shared-memory or distributed-memory systems. Shared-memory systems, also called multiprocessors, provide a single address space much like that of a traditional uniprocessor system. All memory modules are shared by all processors via multiple ports or bus arbitration. Processors communicate through shared memory variables (see Figure 2). These systems can also be categorized as a supercomputing architecture, and do not scale well due to network bandwidth considerations. Examples of this kind include BBN Butterfly, Sequent Balance, Stanford DASH, Illinois Cedar and multiprocessor versions of mainframes and minicomputers such as IBM/370, Honeywell-Bull DPS8 and DEC Vax.

Distributed-memory systems pair each processor module with its private memory module(s), as shown in Figure 3. Also termed multicomputers, they communicate by means of message-passing. Their advantage lies in flexibility, fault tolerance and scalability. Several commercial systems sport hundreds and thousands of processors. The systems from Intel and nCube are good examples, as are machines based on Transputers. These systems are potentially the best exploiters of the latest develop-once deploy-many Agent and Applet technologies because message passing is their core paradigm.

Figure 2 A shared-memory system

Figure 3 A distributed-memory system

1.3 Query Processor

Relational database languages such as Structured Query Language (SQL) allow the expression of complex queries in a concise and simple form, without specifying the actual procedure by which the results are to be obtained. The detailed procedure is actually contained in or built on the fly by a database software component called the Query Processor (QP). This allows the concentration of query optimization strategies into the QP where a large amount of information about the “data” is potentially available, such as its location, skew, table sizes, selectivity, and access patterns.

The function of QP is to transform a SQL query (i.e. relational calculus) into an equivalent lower level set of query operations, typically some form of relational algebra, while optimizing the consumption of CPU, I/O, communications and perhaps memory costs. Communications costs are usually ignored for centralized DBMS, but form the most important factor for distributed DBMS.

The most common query operations are Select, Project and Join. Of these, the Join operation has a complexity of $O(n \cdot \log n)$, while the rest are $O(n)$. Therefore this research, like many before, has focussed on improving the performance of Join operations. However, the mechanisms, algorithms and results developed here are

generally applicable to all complex query operations in both centralized and distributed DBMSs, as long as a network of workstations is available for query processing.

1.3.1 Query Execution Time

The core issue of database performance for the majority of users is query performance, which is not surprising since ease of querying is the most important selling point for DBMS and RDMS software. The ratio of queries to updates in most application domains usually follows the famous 80-20 rule of computer science in more ways than one. About 80% of the operations fall into the query (read-only) category both for development effort and for customer usage scenarios. Again, query operations are more likely to touch around 80% of the data in a given time, while update operations are prone to touch 20% of the data. Due to the increasing demands on RDMS, QP has become a critical performance issue for both centralized and distributed DBMSs.

Query response time (QRT) is defined as the duration of time from query submittal by the user, to the time when user sees the results. QRT has two major components - waiting time in the input queue of QP (QWT) and the actual time spent inside the QP.

The prime measure of database performance is query execution time (QET) defined as the duration of time actually spent executing the query operations, as opposed to waiting in the database input queue. QET includes the time to read/write original and

intermediate operands between disk and processor memory. Parallel processing of query operations can reduce the QET, particularly for relational database queries as these involve uniform operations on uniform streams of data.

1.3.2 Parallel Query Processing

The goal of parallel query processing is to reduce QET by subjecting the streams of relational data tuples to three kinds of parallelism:

- Inter-query parallelism, which involves concurrent execution of multiple queries
- Intra-query parallelism, where various operations of a single query are performed in parallel
- Intra-operation parallelism, which partitions the operands of each operation so that multiple copies of the operation are executed in parallel

An introduction to these three kinds of parallelism is given in [D97]. With the availability of inexpensive CPU MIPS and memory expansions, the exploitation of these parallelisms is the best way to gain price/performance advantages and response time improvements [DG92, D97].

Consider the following SQL query on a customer database:

```

SELECT cust_name, cust_address, cust_phone
FROM   customer
WHERE  cust_city = 'Srinagar' AND cust_credit > 500

```

Suppose we have six Processing Nodes (PNs), where each PN consists of a processor, memory, and a disk unit. The disk units may be non-shared, but are often shared in a LAN environment by cross mounting. If the customer table is large (say 600,000 customers), we can partition the table into 6 disjoint fragments and distribute them to the six PNs. Then, instead of a single PN working on all 600,000 entries, six of them will be working concurrently on only 100,000 entries each. This is referred to as partitioned parallelism [DG92].

Intra-query parallelism for RDMS data has the unique potential to exploit pipelined streaming of the uniform data tuples between the concurrently executing operations. In the above example, it is possible to allocate one set of PNs for the equality operation, and another set for the '>' operation, the output of one of them being pipelined to the output of the corresponding PN in the other set. This is called pipelined parallelism [Rich87, HCY94, CLYY95], and uses an algorithm where comparison keys with same values from different input relations go to the same PN. This is usually achieved with value-ranges or via hashing algorithms. The latter approach is used in this thesis since it is the most common and generic approach.

1.4 Parallel Database System Architectures

A parallel database system consists of several processors, memory units, and disk devices connected through a global interconnection network. Figure 4 illustrates some parallel database system architectures [CK89, Stone86]. Each comprises processors (p), memory units (m), disk units (d), local buses, and a global interconnection network. In all three, the interconnection network is an important determinant of the system performance. A detailed survey of interconnection networks is available in [F81].

When all processors have access to both the global shared memory as well as all disks, it is termed as shared-everything (SE). In the shared-disk (SD) architecture, each processor has its own local memory but all processors have direct access to all disks.

SE and SD can be termed as supercomputing or data-server architectures, since they are more tightly knit via low-cost communication. Their penalty is that most algorithms using such architectures acquire a dependency on such communication, reducing the portability and scalability of those programs. Reliability could also be an issue since the interconnection network becomes a single point of failure in some network architectures. Bandwidth limitations in the interconnection network restrict their use for very large database query applications, even though they may be superior for large scale computing (e.g. scientific computing) or even medium-scale database applications. SE data servers are exemplified by XPRS [Stone88] and Sequent [Stone86].

Figure 4 . Common Architectures for Parallel Database Systems [D97]

Each PN in the shared-nothing (SN) architecture has its own (private) disk and memory units. A processor communicates with other processors through explicit message passing. SN database systems have the advantages of flexibility, scalability and reliability [B88, CL89, Stone86]. Several commercial and experimental systems such as Non-Stop SQL from Tandem [TPG88], DBC/1012 from Teradata [Tera85], nCube-Oracle system, DEC VAXcluster, GAMMA at the University of Wisconsin [DeW86, DeW88], and Bubba at MCC [JTK89] belong to this type of architecture.

In general, SE architecture provides better performance than SN, but has poor scalability. On the other hand, SN architecture provides good system scalability but is sensitive to data skew. Besides, algorithms to counter data or access skew are harder to implement in the SN scene. A hybrid approach called hierarchical architecture has been proposed in [XD97] to incorporate the best features of the SE and SN architectures.

In the special case where shared-nothing PNs in a LAN have cross-mounted disks, some of the characteristics of a shared-disk approach are obtained. This occurs also with the network of workstations (NOW) architecture, which is described in more detail in Section 1.5 on page 18, Section 3.1 on page 43 and Section 3.3 on page 49.

In a typical parallel database system, all relations are horizontally partitioned across all disk units to allow parallel access from the PNs. Their use hinges on the solution of two sub-problems:

- A way to partition the database across the PNs to facilitate parallel processing
- A method of distributing the tuples of each involved relation in a consistent way to the various PNs

Common ways of distributing the tuples of a relation are [DeW89]:

- (a) Round robin strategy in which tuples are loaded into a relation, which is distributed across all the disk drives in the system (equal to the number of processors);

- (b) Hashed strategy, which applies a randomizing function to the "key" attribute of each tuple to select a disk drive;
- (c) Range partitioned with user-specified placement by key value, i.e., user specifies a range of key values for each processor and then the tuples are distributed to processors according to the specified range of key values;
- (d) Range partitioned with uniform distribution, i.e., the system horizontally partitions relations with a user-specified (non-key) attribute.

Pipelining approaches are unviable in the case of supercomputing architectures because pipelining relies on a continuous minimum availability of interconnection bandwidth as well as the used processing elements (PEs) during the processing of a query. This may be unacceptable since the primary applications (supported by the supercomputer) may demand all or most, of bandwidth and also of the PEs. For example, consider a vector processor as a PE, which is running a finite-element program, using large matrix operations and expected to run for many hours. Supercomputing is usually characterized by super-fast processors running many orders of magnitude faster than the speed of interconnection networks. Therefore, it is no surprise that most early studies of parallel database systems focussed on partitioned parallelism.

Pipelining ameliorates peaks in data communications and disk I/O by overlapping these two in time; which is attractive as interconnection speeds approach or exceed those of the processing elements. Under suitable conditions, as we shall see, pipelining can be

gainfully combined with partitioned parallelism, since pipelining does involve partitioning of the “program” (i.e. separation of the individual operations in the query).

In both kinds of parallelism, data partitioning strategy has a major bearing on the database performance. This is because of the risks of introducing or accentuating data skew (i.e., non-uniform distribution of data), leading to execution skew i.e. imbalance in the distribution of processing workload. In case of pipelining, such skew can have cascading effect on all downstream query operations in the common pipeline.

Use of supercomputing parallel database systems for query processing is hampered by:

- The huge expense of specialized hardware such as vector processors, instruction-pipelines and hardware-assisted data-flow architectures
- Equally huge cost of buying or developing software that is suitably designed and tuned to take advantage of the hardware features
- Even bigger costs of maintaining such software

The above problems have led to the popularity of software such as Parallel Virtual Machine (PVM) [GS92] and Message Passing Interface (MPI) [GLS94, SOH96] that configure a virtual parallel machine over a network of workstations (NOWs). An important component of DBMSs is their query capability. Due to the factors mentioned earlier in the opening, queries today consume increasing amounts of resources due to:

- Large table sizes as well as larger number of joins
- Intensive logic due to complex multi-attribute joins

- Complex joins involving large numbers of tables of different sizes
- Volatility in the processing and memory demands of any particular query.

These factors have caused the query performance bottleneck to shift from disk I/O to CPU processing for a large class of critical database processing.

To summarize the discussion of this section, pipelined query execution has the following interesting properties:

- a) Ameliorates peaks in data communications and disk I/O, so that data is “streamed” through a partitioned program
- b) A viable strategy as interconnection speeds exceed those of the PNs
- c) Combined with dynamic and adaptive load balancing, promises enhanced query performance with little or no hardware upgrades,
- d) Can be combined with partitioned data parallelism to harness available computing power, so that each operator is evaluated by a “team” of PNs
- e) Program parallelism (i.e. pipelining), is limited by the number of operators used in a query and the dependencies among those operators. It is useful for complex queries with many independent operators.

This thesis will examine the issues of performance related to pipelining of query operations to characterize the benefits of this approach in the NOW environment.

1.5 PVM: A Parallel Programming Environment

PVM is a software package to configure a single virtual cluster of computers (i.e., a distributed-memory system) from a heterogeneous network of parallel, serial and vector computers. This has made it quite popular in the parallel processing community during the last couple of years. PVM has two parts:

- A concurrent daemon process that any user can install on a machine; and
- A user library that contains routines for things like:
 - initializing tasks on other machines,
 - communication and
 - changing configuration of the PVM machine.

PVM supports applications written in Fortran77, C or C++. Transparency at the application, machine, and network level is provided. For example, PVM handles all data conversion that may be required for different integer, character or floating point representations. If the system used by a particular program run contains machines with different architectures, that program needs to be compiled on each of the different architectures. A failed host is automatically detected by PVM and deleted from the virtual machine. Although multiple users can configure overlapping virtual machines this is not useful during performance studies for QET of an individual query.

Using common message-passing constructs, a sequential program can be parallelized into a set of PVM programs. A PVM task is a unit of computation analogous to a user process in UNIX. PVM supplies the functions to automatically start up and terminate tasks on the virtual machine and allows the tasks to find, know about, communicate, group and synchronize with each other. An application in PVM usually comprises a set of logically related or co-operating PVM tasks that are spawned by some kind of centralized, distributed or hierarchical scheduling algorithm.

A PVM application can be started from a UNIX prompt on any of the hosts or from within another PVM application. Each user can execute several independent and/or related PVM applications simultaneously. For example, four PVM tasks could work on a query while another two form the background load for performance scenarios. PVM provides this flexibility in the spawning of tasks, while retaining the ability to exploit particular strengths of individual machines (e.g. CPU power) on the network.

The scheduling support provided by PVM works as follows:

- By default PVM tries to spawn a task on the least loaded machine. This allows a static call to PVM spawn function to still benefit from some dynamicity in the host selected at run-time. But loads can vary, and even the least loaded machine might not be the fastest one for the task.

- From the PVM console or a PVM application, it is possible to explicitly specify the architecture to spawn a task on or even a particular host by name. This allows dynamic selection of a host by the “scheduling” program.
- By maintaining information about loads, memory and other details about all PVM nodes, a “scheduler” program can actually invoke varying levels of adaptive algorithms in the selection of hosts for spawning the next task. This allows fairly good use of adaptive load-balancing algorithms.

The Berkeley NOW project looks at effectively using networks of workstations for parallel processing [AC95]. Complete details on PVM are found in [GS92, G93, G94].

1.6 Contributions of Thesis

Parallel database systems currently in the market have been used extensively for parallel query processing, but for many organizations these are prohibitively expensive to acquire, use and maintain. Besides, they suffer from an interconnection bandwidth bottleneck that limits the size and throughput of database transactions supportable. Dandamudi [D97] has shown the performance benefits of parallelizing a four-way join query using the nested-loop algorithm on a network of workstations.

This thesis extends the research on parallel query processing to the domain of NOWs for hash joins, an important class of query operations. It substantiates previous claims that NOWs can be gainfully employed for database processing in general and query processing in particular. It proves through experimental results that NOWs can be exploited for both pipelined (or inter-operation) and partitioned (or intra-operation) parallelism despite their varying local loads and presence of data skew. It develops a analytic cost model common to the three major query tree structures used in parallel query processing, and provides pointers to how this model can be further refined and combined with other models.

1.7 Overview of Thesis

This thesis is concerned with parallel query processing on a NOW. The focus is on the performance evaluation of pipelined and partitioned parallelism using two major kinds of query-execution structures called Right Deep and Bushy Tree executions. These are defined in Chapter 2, which reviews the previous work on parallel query processing, with special reference to join processing. This chapter also discusses the rationale for selection of query trees for the experiments.

The NOW architecture, the experimental setup and details of the implemented algorithms are given in Chapter 3, as are the fundamentals of a generic cost model

approach for query trees, and the analytical comparison of the two query trees used in this thesis. The performance results and their interpretation are presented in Chapter 4.

Finally, Chapter 5 provides a summary of the work, the main results, and some pointers to future work. Chapter 6 is the list of acronyms used in this thesis. while Chapter 7 forms the list of references.

Chapter 2 Background

In RDMSs, joins are the most expensive operations to execute, particularly in the face of increased database size and query complexity. Even though this thesis focuses on join operation for the above reason, the results are applicable in general to other database operations. It has been shown that all other binary operations can be handled very much like join [Br84]. The performance of join operations in a multi-join query can be improved in two complementary ways:

- Intra-operator parallelism [SD90], where several PNs work concurrently on each two-way join operation. This was the initial focus of most research due to reliance on multiprocessor systems, data partitioning and smaller size/complexity of queries
- Inter-operator parallelism [CYW92], where several joins within a query are executed in parallel.

The second approach has become increasingly important as we hit the limits of the previous approach and shared-nothing architectures with better interconnect bandwidth became available.

The next section describes the concept of query trees as it applies to query operations. The following section describes the three major classes of join algorithms, while Section 2.3 covers some background regarding query optimization. The final two sections of this chapter describe the rationale for the selected query tree and join algorithms.

2.1 Query Trees

The execution of a query is usually modeled in the form of a query graph [U82], which includes query relations as nodes and the joins between relations as undirected edges. This work focuses on tree-structured query graphs, which are different ways of scheduling the various operations of the query for sequenced or concurrent execution. The leaves of these trees represent input relations, while each internal vertex depicts the intermediate relation resulting from a join (or other query operation) applied to its two children (operands). The query tree is executed bottom up. 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 which is used to build the hash table. Correspondingly, the right child denotes the outer relation whose tuples are used to probe the hash table [Br84, CLYY95]. In the rest of this chapter, the hash join operation will be used without loss of generality for all illustrations and examples.

Query tree structure has a significant influence on the benefit derived from parallelization. The major kinds of query trees reported in the literature are:

- Left-deep tree (LDT)
- Right-deep tree (RDT)
- Bushy tree (BST)
- Segmented right-deep tree (SRT)

The term linear tree refers generically to both LDT and RDT. Also, the distinction between these two vanishes for sort-merge join algorithms (see next section) since the base relations can be scanned/sorted concurrently in both LDT and RDT. This is in contrast with most versions of hash-joins, where the left child input must be completely consumed before the right input can be used [SD90 pp. 475].

The various kinds of query trees are demonstrated in Figure 5. Each vertex in those trees represents the set of tuples output by a virtual process (which implements either a read-file or a join operation). Each internal vertex in a query tree is termed a join node (JN), and assumed to run on one or more processing nodes (PNs). Each PN comprises a CPU along with an operating system, main memory, filesystem and (in the context of NOW) some kind of a parallel virtual machine daemon. An operation can be performed with multiple PNs by implementing intra-operation or partitioned parallelism.

Figure 5 Types of Query Trees

2.1.1 Left Deep Trees

In a left-deep tree (LDT), all intermediate relations are stored, since they form the “inner” relation for the next join. Hash joins require the entire inner relation before any tuple from the outer relation (e.g. R3 in Figure 5) can be accepted. LDTs are easily generated by greedy methods that (for example) choose the smallest relation as the first inner relation. The successive outer relations are chosen to minimize the size of intermediate relations. Most traditional query optimizers generate only LDTs due to

historical reasons. LD trees impose a smaller strain on memory because at most the hash tables corresponding to the two adjacent join operations are required to be in memory at any time [SD90]. However, the downside is that the next hash-table up the linear chain cannot be built before the first join operation is completed, since it supplies the inner relation to the next join. Due to this reason, LDTs are not easily amenable to pipelining in multi-database (MDBS) and NOW environments [DSD95], if the goal is to reduce response time.

Another problem is that the size of intermediate relations staged into memory is unpredictable since they result from intermediate join computations, whose selectivities are unknown. If this uncertainty is tolerable, then LDTs may be suitable for NOW-based parallelization where the goal is to utilize free CPU cycles whenever and wherever they occur. This may benefit long-duration queries if QET is not an issue.

An improvisation on LDTs could keep statistics on size of intermediate join computations, and store them in hash tables for use in adjusting the number of buckets for join operators higher in the query tree.

2.1.2 Right Deep Trees

Right-deep trees (RDTs) consume all the transient intermediate relations in a producer-consumer pipeline. This is possible because each join output forms the outer relation

for the next join. RDTs share with LDTs the unpredictability of the cardinalities of intermediate relations, but do offer the possibility of processing small number of tuples at a time in a pipeline. This reduces the importance of correctly estimating join selectivities [SD90].

RDTs need to store in main memory the hash tables of at least one of the input relations for all N join operations throughout the duration of query. It is expected that for many queries, RDTs would more than make up by being very amenable to pipelining [SD90].

2.1.3 Bushy Trees

A bushy tree (BST), also called balanced tree, is obtained when half the input relations are chosen to be inner and the other half outer relations. These offer the best opportunities to minimize the size of intermediate results [She93] and to exploit all kinds of parallelism [Val93].

Bushy trees can be made amenable to synchronous execution, by grouping the bushy tree nodes (BS-nodes) into allocation tree nodes (A-nodes), where each A-node is a pipeline [HCY94]. Another study [CHY93] uses the concept of hash filters (bit-vector filters), to improve the performance of complex multi-join queries with large number of relations. This could also be classified, perhaps, as SRT with inter-segment parallelism.

For assigning the CPUs to join nodes, 2 schemes were studied in [HCY94], where the cost of a leaf node is measured by the size of the base relation to be scanned:

- Based on Original Tree (BOT) - consider the cumulative cost for every node of original query plan tree.
- Based on Allocation Tree (BAT) - consider the cumulative cost for all A-nodes.

BAT was found to be superior in most cases.

Three schemes were compared for hash filter (HF) generation:

- Early generation (EG) coincides with the availability of the BS tree
- Complete generation (CG), where HF generation accompanies the hash join operations: for inner relations during the table-building phase, and for outer relations during the tuple-probing phase
- Inner generation (IG), where HFs are generated only for the inner relations

IG was found to be the winner. In general, CPU allocation had the dominant effect, and HFs had an incremental effect as the number of relations in a query increased.

2.1.4 Segmented Right Deep Trees [CLYY95]

A segmented right deep tree (SRT) is a BST which is composed of a set of right deep subtrees. Each right deep subtree forms a pipeline segment, consisting of pipeline stages (joins). The result of each stage is pipelined directly from its memory to the

memory of the next stage within the pipeline segment. The result of a pipeline segment is written to disk. Figure 6 shows an SRT with three right deep subtrees.

Figure 6 Segmented Right Deep Tree

Certain “greedy functions” are minimized to select each relation of a segment. This is done in the framework of one of 2 heuristics [CLYY95]:

- **Minimal Work (MW)** - reduce the total work by selecting the smallest inner relations first using a cost-only approach. This leads to high variance, instability and high memory requirement in later stages/segments.
- **Balanced Consideration (BC)** – consider the Benefit and Penalty of each segment. A tunable weighting factor w is assigned to the benefit term in the objective function to be minimized: $Y = \text{Penalty} - w \cdot \text{Benefit}$

In that study, BC appeared to be the method of choice for the general case. Another name for SRT seems to be Zigzag Tree [ZZB93], described as “intermediate between LDT and RDT”, and comprising “right-deep subtrees executed in sequence”.

2.2 Join Algorithms

The join operation finds use in combining tuples from two different relations based on comparison of values in certain columns or attributes. For example, the most common type of join operation, called equi-join, requires the equality test. Tuples having the same value in the specified column are then said to “satisfy the join condition”. The result tuples are output with the combined attributes of both input relations. The join operation and its different types are described in [Codd70, Codd72, ME92]. A number of different join algorithms, including hash-based joins, are surveyed in [ME92].

A summary of the major join algorithms is illustrated here in terms of the following simple form of the join operation:

$$R \bowtie_{r(a) \theta s(b)} S$$

where $r(a) \theta s(b)$ specifies the join condition that must hold true between the attributes $r(a)$ and $s(b)$ of R and S respectively. It is assumed without loss of generality in this thesis that θ is the equality operator.

If two relations R and S with numbers of attributes n and m , respectively, are joined, the result is a new relation Q with $(n+m-1)$ attributes, if the common attribute is represented only once. The relation Q has one tuple for each pair of tuples from R and S that satisfies the join condition. The join operation can also be viewed as a select operation composed with the cross product of R and S . From this it is clear that like a cross product, join operation is intensive in terms of data comparisons and I/O staging. If R and S have cardinalities N and M tuples (respectively), which fit into P and Q pages (respectively), then the worst case complexity is $O(N*M)$ for tuple comparisons and $O(P*Q)$ for page I/O (where $P = N/B$, $Q = M/B$, and B is the average number of tuples that occupy a page of main memory). A large number of algorithms, both sequential and parallel, have been devised to reduce the number of comparisons, amount of I/O or both, with the aim of reducing the QET.

2.2.1 Nested-Loops Join

In the simplest implementations of join, every tuple from one relation is compared with every tuple from the other by using two loops, one within the other:

```

for each tuple s
{
  for each tuple r
  {
    if  $r(a) \theta s(b)$  then
       $Q \leftarrow r + s$ 
  }
}

```

Since entire relations may not always fit into memory, they are often read in blocks.

Efficiency is improved if:

- The relation with higher cardinality is picked for the inner loop, as with R here
- An index is available for the inner relation R, or for both relations
- The inner relation is “rocked” with an elevator principle [Kim80]

Nested loops algorithm is attractive for joins with low selectivity factor, because it is simple to implement and amenable to hardware-assisted parallelization. However its inherently high computational complexity at $O(N*M)$ is a barrier for very large queries.

2.2.2 Sort-Merge Join

This algorithm simplifies the matching phase by first sorting both relations on the join attribute, and then scanning both in that order. The result relation is formed by concatenating the matched tuples.

Phase 1: Sort

```
sort R on r(a);
sort S on s(b);
```

Phase 2: Merge

```
read first tuple r from R;
read first tuple s from S;
for each tuple r
{
    while s(b) < r(a)
        read next tuple s from S;
    if s(b) = r(a) then
        Q ← r + s;
}
```

As such, the major contributor to computational complexity of this algorithm is the sorting time, $O(N \log N) + O(M \log M)$, which can be done in parallel on each relation. If they are pre-sorted (or sorted with hardware-assist), and nothing is known in terms of indexes or selectivity, then this is the best choice, particularly for sequential computers or modest-sized queries. Use of indexes significantly reduces the amount of I/O, since only matched tuples are fetched in entirety.

2.2.3 Hash Join

This algorithm takes further the concept of limiting the set of tuples from the two relations that are actually compared. A large number of hash-join methods for sequential computers were reported in [ME92], which as a class are among the most efficient join techniques [G86], with a complexity of $O(N + M)$. The basic version described here forms the foundation for the two parallel versions used in this thesis.

The join attribute values in the smaller relation are hashed using a hash function. These hash values map to entries in a hash table, which may be either entire tuples or only tuple-ids or key values. For simplicity, let us assume that this hash table fits in main memory. Each tuple in the other relation is subject to the same hash function. If it maps to a nonempty bucket of the hash table, all entries in that bucket are compared for the join condition. The algorithm can be clearly described as two phases:

Phase 1: BuildTable

```

for each tuple r in R
{
    hash_value ← hash ( r(a) );
    HashTable[hash_value] ← r
}
    
```

Phase 2: ProbeTuple

```

for each tuple s
{
    hash_value ← hash ( s(b) );
    if HashTable[hash_value] is not empty
    then
        if s matches any r in HashTable[hash_value]
        then
            Q ← r + s;
}
    
```

2.3 Query Optimization Strategies

Traditionally two major strategies were studied in the optimization of a complex query:

1. 2PO (2-Phase optimization). This is a two-phase strategy [HoS91, H96], which separates the parallelization from an initial total-cost optimization phase. This method breaks the problem space into two sub-problems formalized as follows:
 - a) Generate a query tree that has the lowest total execution cost, ignoring the benefits of parallelization itself. This avoids ballooning an already huge search space, and separates join ordering from optimization of physical properties (sort-order, indexing) and parallelization [H96].

- b) Generate a suitable parallelization for this tree; i.e. a schedule of allocation of processing nodes (PNs) to join nodes (JNs) of the query tree.
2. Integrated strategy [SrE93], considers query trees that may have higher total costs, but much lower 'parallelized costs' than those achievable by other (e.g. two-phase) methods. We can term this 1PO (One Phase Optimization).

Each strategy appears to have its merits in specific circumstances. The 1PO approach is attractive in the following context:

- Spontaneous loads are not placed on the PNs
- All PNs are dedicated statically to the parallel query
- Allows consideration of certain tree configurations that are excluded by other approaches
- Size or complexity of the optimization is still limited to tractable levels

The 2PO strategy, used e.g. in XPRS [HoS91], provides the following benefits in the context of a NOW:

- An obvious way to reduce the upfront analysis costs
- Flexibility in dynamic adjustment of the parallelization within a query to account for spontaneous local loads at the PNs
- An approach that ensures that the worst execution plans are avoided, even though there is no guarantee of getting the very best one.

- Separates the join-ordering work from optimal exploitation of physical properties (sort-order, indexing) and parallelization. This is important since only exponential algorithms are currently known for join ordering, whereas good polynomial algorithms can be devised for parallelization and physical property optimization [H96].

This thesis assumes that query trees are produced by a total-cost optimization algorithm and ready for parallelization over a NOW. Combining the strategies of pipelining in a 1PO strategy, while possible, will require additional work that is left for further study.

A crude implementation of the 2PO approach is used in this thesis by ordering the input relations in increasing size order, and joining the smallest available relation with the largest available relation. That this is not always enough was shown in [CLYY95] i.e. size reduction in an RDT segment need not depend solely on small inputs.

2.4 Choice of Join Algorithm

It has been shown in previous work that Hash Joins provide the best opportunities for pipelining as compared to Sort-Merge Join and Nested Loop Join because of the following reasons:

- Demonstrated superiority in utilizing large amounts of aggregate main memory towards better join efficiency over large ranges of parameter values [Swa89].
- Hashing partitions the problem size into smaller chunks, thereby reducing the importance of correctly estimating the join selectivities.
- Sort-merge joins do perform better in some special cases where both join operands are of similar size; but hash-based parallel joins perform best in general [VG84].

Due to the above reasons, only hash joins are studied here. Parallelism of two kinds is considered – partitioned and pipelined. As mentioned previously, partitioned parallelism has received considerably more research interest than pipelined parallelism. This is being corrected in recent reportage, and this same balance is pursued in this thesis. In the NOW environment, pipelining is found to have a niche of its own.

In this regard, it may be mentioned that the earliest use of a pipeline in computer processing appears to be in Planning a Computer System: Project STRETCH [CAM62, p 204-5], where E. Bloch wrote:

The data flow through the computer [...] is comparable to a pipeline which, once filled, has a large output rate no matter that its length. The same is true here. Once the flow is started, the execution rate of the instructions is high in spite of the large number of stages through which they progress.

The application of this concept to data processing evidently occurred in 1975, when Chen wrote [STO75]:

The word "pipeline" and a sister term "streaming" probably originated with the IBM Stretch project (1954-62) which developed both the highly overlapped STRETCH and the radically different 7950 Stream Processor.

It may be interesting to mention that the earliest ever concept of speeding up mathematical calculations via a non-obvious algorithm appears to have been recorded in 4000 year-old Vedic texts of the Indus-Saraswati. A simple example, showing how large numbers could be multiplied mentally in one line is described by S. Haridas on his website at <http://www.keralaastrology.com/vedicmath7.JPG>.

2.5 Choice of Query Trees

Wilshut and Apers [WiA91] found that for parallel hash-joins, bushy scheduling performed better for small operands while linear scheduling was better for large operands. For sort-merge joins, bushy trees can outperform linear trees specially for large number of relations in the query [CYW92]. However, due to unpredictable size of intermediate relations, execution plan scheduling to synchronize the completion of sister branches is very difficult, hampering the benefits of pipelining. This is the major reason most prior work on pipelined joins used hashing with RD Trees.

Parallelization and pipelining approaches to query performance can evaluate various forms of bushy trees that span the gamut between LD and RD Trees. Early studies concentrated on LDT due to historical reasons. System R [Sel79] only supported LDT and most early query optimizers (e.g. in the GAMMA project [DeW86]) produced only LDTs. This simplified the design of the optimizer to allow the use of dynamic programming techniques [Sel79, DeW86, DeW88]. To include better search space for certain queries, BST structures were examined next [CHY93, IoKa91], with the finding that it was easier to optimize both spaces (BST, LDT) together than LDT alone. This led to an interest in exploring the uncharted territory of RDT and SRT [CLYY95, WiA91, Wil95], where the initial approaches were based on simple heuristics and considered only the amount of memory as a constraint. In the NOW environment and other scenarios where commodity processors are harnessed in variable quantities at run-time, CPU power is a valid constraint.

The discussions of the previous paragraphs and sections suggest that if the join operation can be broken into smaller chunks of work scheduled on message-passing PNs, an optimal combination of pipelining and partitioning can be found for different arrangements of PNs, by means of experimental studies. Analytical results for this involve too many unknown factors to yield simple results. An outline of an approach is presented in Section 3.8 that could use SRT as a general case of which RDT and BST are specializations. An illustration of the timing issues is amenable to analytical reasoning and is presented as a comparison between RDT and BST in Section 3.9.

A study of the three tree structures, RDT, BST and SRT, is the best way for exploring the different combinations and granularity of pipelining versus partitioning. In this initial phase, the comparison of RDT versus BST is undertaken, owing to the fact that they form the two endpoints of the range. The study of SRT structures would be a refinement and is left for further study.

In NOWs, the attempt is to reduce the communications costs while trying also to reduce a sudden large expectation of resource availability from a single PN. In this context, it was thought that exploration of the space between bushy and linear scheduling as well as the optimum way of selecting or varying the granularity of intermediate operands could lead to a desirable solution.

2.6 Skew

Any non-uniformity in the data, access pattern or workload significantly impacts the cost estimates and actual QET delivered by a system that was built with uniformity assumptions. Most studies of query processing and join algorithms assume, for the sake of simplicity, that the join attribute values are uniformly distributed over its domain, that queries access all relations in a uniform way and that partitioned relations for parallel processing of an operation are uniformly distributed across the allotted PNs. The violation of each of those assumptions is respectively called data skew, access skew

and distribution skew. Access skew is relevant where relations exist in distributed partitions and when considering inter-query parallelism.

Distribution skew can be avoided with a judicious selection of the (hash) partitioning function. Data skew [WDJ91] occurs frequently in real databases and a number of solutions have been proposed such as bucket-spreading [Kit90], bucket-tuning [KNT89], partition-tuning [HL91] and hierarchical hashing [WYTD93]. The first three are able to handle low skew while the latter works well even for high skew levels. It is also noted in [WYTD93] that all the schemes are very poor in handling highly skewed relations when they are joined using a large number of PNs. The bucket-spreading technique of [Kit90] is applied in the context of specialized hardware and static buckets.

Any of the above concepts can be combined with the two hashing schemes studied here. Hierarchical hashing is particularly attractive due to its good performance despite slightly larger overhead of a scheduling phase.

Chapter 3 Parallel Hash Joins using NOW

3.1 Introduction to NOW

A network of workstations (NOW) follows the SN model and is able to exploit commodity hardware for supercomputing power. In this thesis we assume the database is located on one workstation (called server), which runs DBMS software, and takes care of concurrency and recovery issues. Other workstations simply offload query processing work. This is important because in the NOW environment no central control can be imposed on all the workstations. For example, users might reboot their workstations as often as they like, or start CPU-bound or I/O-bound jobs in an unpredictable way. Because of this lack of central control on the resources, they also differ significantly from the parallel database machines discussed in Chapter 1. The major differences are:

- The database itself is neither distributed among the workstations (as in distributed database systems) nor are relations partitioned as in a parallel database system. This means new costs in sending data to the workstations.
- We simply use the additional processor cycles and memory available on these workstations for database query processing on a time-of-day basis, since queries

only requires read-only access to data. This yields improved performance without additional cost.

- The workstations in a NOW-based system are likely to be heterogeneous.
- In NOW-based systems, having no central scheduler and load balancer, the load on each workstation can vary dynamically even while a query is being processed. This poses the problem of load imbalance across the system.
- NOW may not have a special purpose fast communication network.

3.2 Query Architecture on NOW

Three major steps are involved in the parallel processing of database queries: query interpretation and optimization, processor allocation and operation scheduling.

The first step receives input queries (e.g., in SQL) and generates optimized query execution plans, which specify the order in which the operations of the query will be performed and the sequence in which the intermediate results will be passed from one operator to the next. For the most part, this step is very similar between sequential and parallel versions of QP, although some research implementations may in the future try to exploit the knowledge about speeds of communication networks and specialized algorithms available at certain sites.

Processor allocation forms the second step by determining the set of PNs that is used for processing either an entire query or a part of it. The operation scheduler, as the third step, actually schedules the workload by allocating one or more PNs to each JN of a query. It may schedule an entire operation or can parallelize and load balance the operation. Three query processing architectures have been discussed in [DJ97].

3.2.1 Centralized architecture

The centralized architecture is characterized by a master node that performs all the three steps mentioned above. The other workstations act as slaves who are assigned work by the master node.

Queries submitted at various workstations are forwarded to the master node for interpretation and generation of optimized query execution plans. For this purpose the master node is responsible for maintaining the latest system state information about all the slave nodes.

The processor allocation decisions can be made in three major ways:

- Statically at step two at the beginning of the execution of the entire query
- Dynamically, once at step three while scheduling each operation
- Adaptively, at the beginning of the execution of each operation

The static approach is the only one feasible for RDT, since all the PNs have to be allocated for the pipeline from start to finish. For BST, the dynamic approach is adopted. The complexity of implementing and evaluating the adaptive method is left for further study, because to observe its benefits would require query complexities far in excess of what is achievable in our lab environment.

The centralized architecture is chosen to implement the experiments in this study for the following reasons:

- This initial work focuses on single query performance and not on scheduling of multiple queries
- Load balancing (scheduling) can be done effectively, because allocation decisions are typically made at the operation level.
- The partitioned and pipelined parallelism being studied is simpler to understand and implement from a centralized viewpoint
- Centralized architecture can form a useful starting point for future work in the other architectures

Further description of the processes and algorithms used is given in Section 3.7.

3.2.2 Distributed architecture

An alternative is to distribute the intelligence, so that all the three steps (i.e., query interpretation and optimization, processor allocation and operation scheduling) are done locally at the workstation that received the query. As a result, it eliminates the single-point-of-failure that may exist in the master-slave architecture.

Due to the variability in query complexity and query arrivals, nodes are prone to be overloaded or underloaded. In distributed systems, a threshold on CPU queue length is often used to detect an overload condition. In NOW-based systems an added dynamic is the unpredictable background load. Hence, a weighted combination of the background and query loads is considered as an indicator of system load. Then a simple threshold can be used to determine if a node is overloaded.

This mechanism can classify a node as underloaded or overloaded or normally loaded. Load sharing is achieved by transferring work from overloaded nodes to underloaded nodes. Typical policies on load sharing are sender-initiated, receiver-initiated, adaptive and hierarchical.

This architecture was not selected because of the complexity of implementing the load balancing algorithms in conjunction with the partitioned and pipelined parallelism that is being studied. It would be difficult to isolate the effects of the distribution algorithm

from impacting the QETs being analyzed for the initial comparison of BST and RDT. As follow-on work when data skew and load balancing is selected as the main issue, it would provide a useful area of study.

3.2.3 Hierarchical architecture

The hierarchical architecture utilizes the concept of a hierarchical run queue organization and a hierarchical scheduling policy proposed by Dandamudi [DT97]. A set of queues is organized as a tree with all the PNs attached to the bottom level (L) of the tree (i.e. leaf node). Consider the highest level at the root node to be zero.

The hierarchical architecture is a self-allocation scheme, with no central allocator. All incoming queries are placed in the root queues. A PN looking for work first queries its associated query queue at level ($L-1$). If that queue is empty the PN queries the next parent node at level ($L-2$) and so on up the tree until it finds work to be scheduled (unless the root queue is empty). However, in order to reduce access contention at higher levels, when a queue is queried, a set of work units is moved one level down the tree. The size of the set decreases progressively as one goes down the tree.

It is unclear if this architecture can also be gainfully adapted to schedule individual operations of a query under RDT or BST. The hierarchical scheduling aims to combine

the merits of two types of policies: space-sharing (good at low to moderate system loads and stable job parallelism) and time-sharing (good at high system loads and volatile job parallelism). Due to limited number of PNs in the NOW laboratory, both RDT and BST use significant amount of time-sharing (as opposed to space-sharing PNs exclusively to single jobs). RDT can use more space-sharing if there are sufficient number of PNs because it needs a stable pipeline across all the operations. BST can also use more space-sharing if background load is not too volatile. The hybrid policy was shown, at most system loads of interest, to outperform both space-sharing and time-sharing policies by a wide margin. So it can be conjectured that it would be of benefit under a volatile local load at the members of a NOW.

This architecture has not been used in this study, which uses a single query. Also, even if possible to use a “hierarchical run queue of operations”, it would have been difficult to isolate its overhead from impacting the QETs being analyzed for the initial comparison of BST and RDT algorithms. As follow-on work when data skew and load balancing is the main issue, it might be useful to incorporate a hierarchical run queue.

3.3 NOW System Setup

The NOW system used in the experiments is composed of a number of Pentium processors linked with 10 Mb/s and 100Mb/s networks (Figure 7).

Each node has 32 MB of RAM. The main node, known as 'n0' or 'triton' from the outside, is a P6/200 with a hard disk. This is the only node that can be accessed from the outside, and the only node that supports X11.

Figure 7 Network of Workstation System

The nodes are listed in Table 1. From the table we can see, besides triton (n0), there are 4 different types of nodes in Triton. Nodes 1, 2, 9 are Pentium 133 with slow caches. Node 4 is also Pentium 133 but with fast cache. Nodes 5, 6 are Pentium 166. Nodes 3,7,8 are Pentium 200.

Table 1 Nodes of Triton System

Node	CPU		
	Type	RAM	Cache
n0	P200	32	-
n1	P133	32	Slow
n2	P133	32	Slow
n3	P200	32	-
n4	P133	32	Fast
n5	P166	32	-
n6	P166	32	-
n7	P200	32	-
n8	P200	32	-
n9	P133	32	Slow

The effect of the processor type and speed on their computational capacity was studied earlier [Z98] and the results used in this thesis are depicted in Table 2.

Table 2 Power Units of each Triton Node

CPU	Power Units	Nodes
P200	280	n0, n3, n7, n8
P166	138	n5, n6
P133 (slow cache)	116	n1, n2, n9
P133 (slow cache)	100	n4

Triton being a research system, the status of nodes changes from time to time. The command *sysinfo* displays the status of each node in the system as depicted in Figure 8 below.

```
%sysinfo
triton up 46 days, 23:44, 3 users, load average: 0.16 0.05 0.01
n1 up 34 days, 9:18, ) users,load average: 0.00 0.00 0.00
n2 up 68 days, 19:29, 0 users, load average : 0.00 0.00 0.00
n3 up 0 users, load average: 0.06 0.01 0.00
n4 up 0 users, load average: 0.00 0.00 0.00
n5 up 45 days, 17:57, 0 users, load average: 0.00 0.00 0.00
n6 up 6 days, 1:46, 0 users, load average: 0.00 0.00 0.00
n7 up 6 days, 19:18, 0 users, load average: 0.00 0.00 0.00
fn1 up 34 days, 9:18, 0 users, load average: 0.00 0.00 0.00
fn2 up 68 days , 19:29, 0 users, load average: 0.00 0.00 0.00
fn3 up 0 users, load average: 0.05 0.01 0.00
fn4 down
fn5 up 45 days, 17:57, 0 users, load average: 0.00 0.00 0.00
fn6 up 6 days, 1:46, 0 users, load average: 0.00 0.00 0.00
fn7 up 6 days, 19:18, 0 users, load average: 0.00 0.00 0.00
```

Figure 8 Output of Sysinfo command on Triton

Figure 8 shows that all the 10Mb/s connections are up, i.e. nodes known as nX are connected through 10 Mbps cable and the same nodes connected through 100 Mbps are known as fnX (fast node). The above display shows that n4 is up, but the 100Mbps connection to that node is down i.e., fn4 is down. It is possible for some fnX to be down while nX is up, but if nX is down the node cannot boot, and fnX is not possible. In summary there are currently nine Pentium motherboards, connected through multiple networks, with 1 node with a disk, and all others diskless.

In order to allow multiple users at the same time, when doing time critical experiments, a user can allocate nodes for his/her exclusive use. This allows users to share the system, and have as little impact on each other as possible.

Node allocation is done with command `allocate`. It has five modes of operation

- allocating a number of nodes and creating `/hosts` file: `allocate -c 4`
- as above but node 0 is not included in `/hosts` file: `allocate -c -e 4`
- allocating a number of nodes: `allocate 4`
- allocating specific nodes: `allocate n2 n4 n7`
- displaying allocated nodes: `allocate -s`

Nodes can be given up with the `release` command. It has two modes of operation:

- release selected nodes: `release n3 n5`
- release all nodes: `release -a`

Please note that node 0 (triton) cannot be allocated, as it has to be used by all users.

3.4 Database Model

The model of the database considered consists of a number of tables. Each table contains a large number of tuples. Each tuple has several attributes. In our database model, tables are actually regular Unix files, each file has 5,000 to 9,000 records for the normal experiment and 15,000 to 50,000 records for the large data size experiment. A record is made up of two fields, as two attributes in a tuple. The first field is key

attribute and second field are considered as other attributes. Both fields are actually long integers with size 4 bytes. Thus a tuple has size 8 bytes. In all 16 tables were created for the normal and 8 tables for the big experiment.

The 16 normal tables were re-used from an earlier work [X88] due to certain desirable properties. Among these, two tables are different from the others. The key values in these two tables are unique, sequential long integers. There is no key duplication. In the remaining fourteen tables, however, the key values in each are normally distributed with the same mean value and different variance for the keys in every table. The aim was to obtain different selectivities for the join operations between different tables.

Table 3 Join Selectivities for Normal Sized Relations [X88]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	5.9	1.7	1.4	0.6	1.5	1.6	0.7	1.6	1.2	1.1	1.7	2.0	0.9	1.1	1.6	1.6
1	1.7	4.9	1.4	0.5	1.4	1.5	0.7	1.5	1.2	1.1	4.9	1.7	0.8	1.1	1.5	1.5
2	1.4	1.4	2.9	0.5	1.2	1.3	0.7	1.3	1.1	2.2	1.4	1.4	0.8	1.0	1.3	1.3
3	0.6	0.5	0.5	2.9	0.5	0.6	0.4	0.5	0.5	0.5	0.5	0.6	0.5	0.5	0.6	0.5
4	1.5	1.4	1.2	0.5	3.5	1.3	0.7	1.3	1.1	1.0	1.4	1.5	0.8	1.0	1.3	1.4
5	1.6	1.5	1.3	0.6	1.3	3.9	0.7	1.4	1.2	1.1	1.5	1.7	0.8	1.1	1.5	1.4
6	0.7	0.7	0.7	0.4	0.7	3.4	0.7	0.6	0.7	0.7	0.7	0.7	0.6	0.6	0.7	0.7
7	1.6	1.5	1.3	0.5	1.3	1.4	0.7	4.7	1.1	1.1	1.5	1.6	0.8	1.0	1.4	1.4
8	1.2	1.2	1.1	0.5	1.1	1.2	0.6	1.1	5.0	1.0	1.2	1.3	0.7	0.9	1.1	1.2
9	1.1	1.1	2.2	0.5	1.0	1.1	0.7	1.1	1.0	2.2	1.1	1.1	0.7	0.9	1.1	1.1
10	1.7	4.9	2.4	0.5	1.4	1.5	0.7	1.5	1.2	1.1	4.9	2.7	0.9	1.1	1.5	1.5
11	2.0	1.7	1.4	0.6	1.5	1.7	0.7	1.6	1.3	1.1	1.7	4.8	0.9	1.1	1.6	1.7
12	0.9	0.8	0.8	0.5	0.8	0.8	0.6	0.8	0.7	0.7	0.8	0.9	4.6	0.7	0.8	0.8
13	1.1	1.1	1.0	0.5	1.0	1.0	0.6	1.0	0.9	0.9	1.1	1.1	0.7	3.0	1.1	1.1
14	1.6	1.5	1.3	0.6	1.3	1.5	0.7	1.4	1.1	1.1	1.5	1.6	0.8	1.1	4.8	1.4
15	1.6	1.5	1.3	0.5	1.4	1.4	0.7	1.4	1.2	1.1	1.5	1.7	0.8	1.1	1.8	4.3

The selectivities of join operations among the 16 normal sized tables are shown above in Table 3. The first row and first column give the identifying number of the database tables. The rest are the selectivities of join operations between the tables and the matrix of selectivities is symmetric around the diagonal. The unit of selectivity values in the table is 0.01%. The selectivities are from 0.0043% to 0.059% and the average value of all the selectivities is 0.029%. Data tables 2 and 9 have unique key values and other tables have their key values normally distributed. The variance of the key values in tables 3, 6, 12 and 13 are higher than those in the rest of the tables, resulting in lower probability that the keys will match. Notice that the join selectivities in rows 3, 6, 12, 13 are relatively smaller.

The eight big tables are of increasing size, and only one has unique key values. The other seven have normally distributed keys. The pair-wise selectivities between tables range from 3 E-5 to 5 E-5 %.

3.5 Query Workload Model

Based on the database model and given selectivities among tables, different types of queries were generated as jobs. Each query contains the information of which specific tables in the database are used. Each query has 2, 4, 6 or 8 tables involved. The same

set of queries is subjected to both the studied algorithms, BST and RDT, under different sets of conditions to study their QET.

Performance sensitivity to query structure is given in the next chapter. Inter-operation and intra-operation parallelism can be exploited in different ways by different query structures. Linear queries can exploit inter-operation parallelism by pipelining, whereas the bushy tree can do the same by concurrent execution of sibling JNs at the same level. Both structures can partition a single join operation by hashing to exploit intra-operation parallelism. The number of original tables in the query determines the number of operation levels in the query. There could be 1-3 levels of operations in each balanced query and 1-7 levels of operations in each linear query.

3.6 Parallel Virtual Machine (PVM)

The experiments were implemented on Linux operating system with the help of Parallel Virtual Machine software package. PVM allows a heterogeneous network of parallel, serial and vector computers to appear as a single concurrent computational resource (i.e., a distributed-memory system). PVM consists of two parts:

- A daemon process that any user can install on a machine; and
- A user library that contains routines for initializing tasks on other machines, communication and changing configuration of the PVM machine.

Applications using PVM can be written in Fortran77 or C/C++ and parallelized by using message-passing constructs common to most distributed-memory computers. PVM supports heterogeneity at the application, machine and network level. In other words, PVM allows application tasks to exploit the architecture that is best suited to their solution. For example, PVM may include a parallel machine and a graphical workstation. Computations can be performed on the parallel machine and the results may be displayed in graphical form on the workstation.

PVM handles all data conversion that may be required if two computers use different integer or floating point representations. If PVM is to contain machines with different architectures, programs need to be compiled on each one of different architecture.

A major motivation in using NOW is that pipelining can obviate the need to write intermediate results to disk through a judicious allocation of estimated workloads to the NOW members, particularly if minimum MIPS preservation is available as described in [DZ97], and an analogous scheme could be used perhaps even to reserve main memory. Such preservation is simulated in the experiments by using the notion of relative power of the PN and allotting PNs to JNs in proportion to the amount of work to be done by each JN (vis-à-vis the total work).

Another feature of the presented environment is that we need not expend resources in elaborate scheduling algorithms because conditions at the workstation are liable to change without notice due to local events. With the addition of dynamic and adaptive load-balancing, the algorithm can handle not only static skew, but also dynamic (access) skew. Some dynamic load balancing is introduced in this initial study, and based on the results further refinements can be added in the future.

A simple way to adapt to such surprise conditions is to give more work to those nodes that finish it early and less to those that finish late [D97]. Whenever the processing for a JN is completed, the PNs are reclaimed for use by other JNs in the query. Up to two JNs are allowed to share a PN, thereby ensuring fair availability of a PN's power to any processes running on it. For using such a scheme, more decision points are available in a bushy execution tree than in RDT or SRT. On the other hand, the presence of unpredictable local load at any one node can slow down all consumer nodes downline from it. This applies to both BST and RDT, but is more disruptive to RDT, which expects the established pipeline to function continuously for the smooth flow of tuples.

Early research in parallel execution of complex queries focussed on linear trees to restrict the search space of the optimization algorithms used [KBZ86, LVZ93], and also because conventional query optimization algorithms usually generate Left-Deep (LD) Trees. LD trees can at low cost be mirrored to RD Trees, but generating good Bushy trees is more involved.

Here we study Single Complex (i.e. Multi-join) Queries, as are frequent in many applications due to use of views [YCHL92]. Opportunities for exploiting parallelism, specifically via pipelining, have been studied by simulation experiments using PVM on a network of Linux workstations.

Due to the common cross-mounting of disks across all workstations in the LAN, this could qualify as a Shared-Disk (SD) environment, provided the join processes could access different files concurrently. However, data files for the relations are all stored at the single server, and the results are also all sent to a single workstation that initiated the query. The tuples have to be read and written sequentially because they all belong in one file. Due to this NOW is best classified as Shared Nothing (SN) environment with centralized data storage.

The centralized data store model has the following characteristics and consequences:

- There is no pre-distributed data, so more opportunities for load balancing are available, since data has to be distributed to different PNs anyway.
- It is easier to maintain security and integrity of the data. Data sent to other nodes temporarily for query processing is destroyed soon after the query results are derived. Queries involve no updates or on-line transactions (OLTP). On most live databases, 80% of the operations are usually query operations.
- Ordinary desktop PCs in any workplace are seldom used to their capacity even during the day, but they cannot be trusted to store fragments or partitions of live

database, which might be subject to updates or OLTP (the user may decide to switch it off, overload it, or crash it with a risky program anytime). However, such PCs can always be sent QP workload, which can easily be moved and recalculated in the event of a crash.

- In a centralized data scenario, it is easier to maintain meta-data such as selectivities, relation sizes, tuple-size histograms etc. for use in optimizing the load balancing and query partitioning/pipelining strategies.

3.7 Description of Algorithms

Three kinds of processes are used:

- Scheduler, which spawns all the worker and reader tasks in accordance with a given tree structure, monitors their completion and records the response times.
- Worker, which actually performs a join operation on the inner and outer tuples passed in by its inner and outer Readers.
- Reader, which reads the relation file, hash-partitions the tuples and farms them out to the workers, as identified to it by the Scheduler.

These three processes are described separately below. Some preparatory building blocks are covered first.

Where feasible, algorithmic English language is used. At some places, the complexity or size of the algorithm has necessitated the use of pseudo-code for elucidation of the pertinent parts of the algorithm. Small chunks of C++ code are also shown for certain intricate components.

3.7.1 Building Blocks

3.7.1.1 Tuple

The following structure is used to represent each tuple being processed in memory:

```
typedef struct
{
    long y_idx;
    char y_str[Y_STR_SZ];
} Y_el;
```

3.7.1.2 Hash Tables

Two kinds of classes are used to partition incoming tuples into hashed lists. One is based on arrays, and is used to accumulate a bucket of tuples to be output. As soon as the fixed size bucket is full, it is sent to the next join processor(s) in sequence, or written to a file. Both Readers and Workers use this class, called HashTable_Ar. The relevant parts of this class' interface is shown here:


```

class HashTable_Ar
{
    bool    bkt_full;
    int     numberOfTuples;
    friend ostream& operator<<( ostream& s, const HashTable_Ar& ybe );

public:
    Y_el    Y_tuple[BKTSIZE];

    HashTable_Ar() :
    ~HashTable_Ar() :
    bool isFull() { return bkt_full; }
    int getNtup() { return numberOfTuples; }
    void clear( );

    // Returns True only when this insert causes bucket to be Full
    bool insert( Y_el& Ytup ) :
}
    
```

The other kind of hash table is based on linked lists, and is used by the Workers to perform the actual join operation. A linked list is required in this case because neither the total number of tuples nor the number arriving into a particular bucket is known. The number of lists (buckets) in the hash table can be influenced at setup time based on an input parameter.

```
// List based H.T.
class HashTable_Li
{
    bool    bkt_full;           // the whole chain of buckets is Full
    int     numberOfTuples;     // in the chain of buckets
    int     n_lists;           // no. of buckets
    int     bkt_size;          // An input estimate used to decide the number of buckets

    friend ostream& operator<<( ostream& s, const HashTable_Li & ybe );
    friend ostream& operator<<( ostream& s, const HashTable_Li & ybe );

    list<Y_el> *Y_tuple;
    enum { N_LISTS_tiny =27, N_LISTS_spry =47,
           N_LISTS_big =97, N_LISTS_huge = 151 };

public:

    HashTable_Li() :

    HashTable_Li( int bkt_sz_ )
    {
        numberOfTuples=0; bkt_full = false;
        bkt_size = bkt_sz_ ;
        n_lists = (bkt_size < 100) ? N_LISTS_tiny :
                  (bkt_size < 1000) ? N_LISTS_spry :
                  (bkt_size < 5000)? N_LISTS_big : N_LISTS_huge;

        Y_tuple = new list<Y_el>[n_lists];
    }

    ~HashTable_Li() { delete [] Y_tuple; }
    bool isFull() { return bkt_full; }
    int getNtup() { return numberOfTuples; }

    void insert( Y_el& Ytup ) ;
    void insert( long Y_key, char * Y_str );
    int search( long Y_key, Y_el& Y_found );
}
```

3.7.1.3 Nodes

Two kinds of nodes are considered:

- Physical or processor node (PN), which is an actual host CPU with its memory, operating system and i/o. Conceptually, it runs one worker process, and potentially a reader process. Actually, a second worker process may share the same PN if there are no more PNs left. The PNs are fixed in number, and statically allocated as an array.
- Join node (JN), which is a combination of one or more PNs running one Join operation by partitioning the input relation among the constituent PNs.

The hash-partitioning is done at two different points in the parallel join algorithm:

- first within each Reader process, and then,
- after each join operation within the Worker process just before sending the tuples to the next JN (since the next JN may consist of multiple PNs).

Each PN is described in a compact way as:

```
struct PN_Info_str
{
    char id;
    int rpower;    // relative power
    int memory;    // in MBytes
    int to_JN;     // Number of JNs sharing this PN; 0 to 2
};
```

The Power of each PN is represented by a relative integer value, counting the lowest power workstation in the NOW as 100 units.

The JNs are created and initialized into a dynamic array. A JN goes through some states in its lifetime, captured in the type Rstate:

```
enum Rstate { BORN=0, RDY, RUNNING, FINI };
```

Each query tree comprises a hierarchy of JNs, along which tuples flow bottom up, supplied by Reader processes (see below) at the leaves (level 0) of the tree, and ending at the root to form the result relation. The JNs are initialized with identifiers called JNID in a breadth first bottom up manner, starting with zero. Every JN is a client of two suppliers, one supplying the inner input relation and another the outer relation. A supplier can be either another JN or, for the level one JNs, a Reader. At initialization time, every JN gets its JNID, the JNID of its two “suppliers”, and a best-effort estimate of the cardinalities of the two input and one output relations related to that JN.

Each JN is designed as a class, containing methods to allot PNs, to ‘ready’ itself, to record its completion, etc., as shown in the class definition here:

```

class JN_Info_cla;
queue<class JN_Info_cla > readyQ; // Used in BST

class JN_Info_cla
{
    short coef_a, coef_b, coef_c;
    Rstate run_state;
    static int finished; // Completed Join's

public:
    static int total_JNs;
    static int fin_launched; // Launched Join's
    int JNID, cost;
    int xJN, yJN; // JNs which supply my inner, outer tuples
    int cardiNx, cardiNy, cardiNz; // x,y are inputs, z is result
    short num_bkts, num_fini;
    short got_PNs[MAX_PN_PER_JN];

    JN_Info_cla() ;
    JN_Info_cla( const JN_Info_cla& jni ) :
    JN_Info_cla& operator=( JN_Info_cla& jni ) ;
    ~JN_Info_cla() { }
    static int done() { return finished; } // how many done till NOW
    static int launched() { return fin_launched; } // done + launched
    Rstate howdy() { return run_state; }

    int fini( int cNz ) ; // returns number of PNs finished at this JN
    void ready( ) { run_state= RDY; readyQ.append( *this ); }

    // Initialize the JN, record cardinalities; estimate and return cost
    int init( int jID, int xjn, int yjn, int cardIx, int cardIy, int cardIz );

    int allotPNs( int wantPow ); // allot PNs based on wanted power
    int allot_aPN ( ); // allot first free PN
    void reclaim_PNs(); // called from fini()
}
    
```

3.7.1.4 Cost Estimator

A simple cost function that has low overhead is used to generate execution plans, as reported in [Wil95]. If n_1 and n_2 represent the cardinalities of the join operands and r is the cardinality of the result, then the cost of a main-memory join is estimated with:

$$\text{Cost of JN} = a \cdot n_1 + b \cdot n_2 + c \cdot r$$

Here, a and b are set to 1 if the operand is a base relation, to 2 if it is an intermediate result; while c is always 2. The rationale is explained in [Wil95].

The input cardinalities for the two (inner and outer) relations going into a JN are required for the above calculation; as is the cardinality of the result, particularly for BST where the result is written to disk at every JN. The selectivities between all the input tuples are pre-calculated and stored in a binary file, which is read in at run-time and stored in memory. This information is used to calculate the result cardinality for each leaf-level JN, since both their input cardinalities are always pre-known.

The estimated result cardinalities are also important for RDT, where the cost of all JNs is required at the start of the whole query to establish the pipelines of PNs. For every non-leaf JN of an RDT, the inner cardinality alone is pre-known since the outer tuples are always the result of another JN. For BST, things are simpler because by the time

any JN is scheduled, both its input cardinalities are known. If those inputs are coming from other (lower-level) JNs, the actual result cardinalities of those JNs are stored in the JN_Info instances for that JN as part of the finishing up.

The selectivity between these intermediate results is still unknown, so a weighted factor of the average input selectivities is used to estimate the intermediate selectivities. It is also noted that in the experiments the number of result tuples in the higher-level JNs (i.e the cost) is smaller. Therefore the weights are chosen to reflect that fact. In case of BST, the level one JNs use the average selectivity of leaf-level joins (to calculate JN cost), while the remaining levels use the minimum selectivity value. In RDT, each JN uses a selectivity as 80% of the selectivity of the previous (lower) JN.

3.7.1.5 Measured Time

A class is designed to facilitate the measurement of QET by providing methods to record the start and end time of a query, to accumulate the response times over as many runs as required, and finally to calculate and output the mean as well as the 95% confidence interval. The Scheduler, which controls the runs, uses this class, called MeasTime.

The interface, and implementation of some key methods, of class MeasTime is as follows:

```

class MeasTime
{
public:
    unsigned long dura;    // microseconds of duration
    unsigned long accDura; // microsec of accumulated durations e.g. 30 samples
    long double sqDura; // sum of square of each dura so far
    int nsamp;    // 1 + number of samples in accDura/sqDura

    struct timeval start;
    struct timeval finish;
    friend ostream& operator<<( ostream& s, MeasTime mt );
    friend ofstream& operator<<( ofstream& s, MeasTime mt );

    MeasTime()
    {
        gettimeofday( &start, NULL );
        gettimeofday( &finish, NULL );
        sqDura = nsamp = accDura = dura = 0;
    }
    ~MeasTime() { }

    begin( )
    {
        gettimeofday( &start, NULL );
        dura = 0;
    }

    end( )
    {
        gettimeofday( &finish, NULL );
        // Calculate dur_sec, dur_usec here

        //Discard first run due to caching setup overhead etc..
        if( nsamp > 0 )
        {
            accDura += ( dura = 1000000 * dur_sec + dur_usec );
            sqDura += (dura * dura);
        }
        nsamp++;
    }

    void cumav( ) ; // Calculate QET with 95% Confidence Interval
}
    
```


3.7.2 Scheduler

The responsibility of the scheduler is as follows:

- Read the query and sampleSize as input arguments
- Parse the query string for names of input relations (binary files)
- Find the cardinalities of the relations
- Structure a tree of Readers and Workers to form a query
- Find out selectivities of pairs of relations to calculate the costs of some JNs, and estimate those of the other JNs
- Initialize the Readers and Workers and allot PNs to them based on above costs
- Spawns each Worker at a JN, and each JN's two Readers (if applicable)
- Lets the Readers know about their client Workers
- Run the query sampleSize number of times
- Accumulate the QET values for all runs, and finally
- Output two statistics:
 - QET with 95% confidence interval (CI)
 - Measured selectivities of the join operations performed in the query

The Scheduler performs its work in slightly different ways in the BST and RDT cases, and these details are covered in the specific sections 3.7.5 on page 74 for BST and 3.7.6 on page 79 for RDT.

3.7.3 Reader

A reader is responsible for reading one input relation, partitioning the tuples based on a well-known hash function and sending a different partition to each Worker process in its parent JN. The granularity of partitions is determined by the parameter BKTSIZE, and is constant for the entire join operation. When BKTSIZE tuples accumulate in a result bucket, it is flushed out to next parent in the tree.

A Reader process is also used, if necessary, to read in intermediate relations that have been written to disk (as in BST). In either case, hash partitioning is performed (if applicable) to distribute the tuples to PNs allotted to the next JN up the tree.

The Scheduler provides the following inputs to the Reader at startup time:

- i. Number of input files to read
- ii. Names of input files
- iii. Number of PNs and their respective Relative Power values in its client JN
- iv. TaskIDs (TIDs) of the Workers constituting its client JN

Inputs (iii) and (iv) above are ordered lists, from which the Reader deduces a relationship between a Worker TID and the relative power of the PN available to that Worker. This could allow the Reader as supplier to send more buckets of tuples to the more powerful Workers. This is potentially achieved as follows.

The Reader could establish output buckets, equal in number to the sum of all power units in the client PNs. Then, each client PN is allotted a certain number of the output buckets, depending on its contribution to the total power. Since the tuples are hash-partitioned randomly into these buckets, this scheme has the following properties:

- More powerful client Workers (PNs) get more tuples via more buckets
- Since “relative power” is a comparative integer, it can be safely used by the two JNs that are both independently supplying the two inputs to a common client JN. Tuples with the same index value from both inputs will end up at the same client Worker or PN (call this the “index property” of the hash-partitioning scheme)
- If the “relative power” value is dynamically altered, then the above-defined “index property” will not hold any more.
- Any data skew in the input would be exacerbated in the output, particularly if the relative powers are widely divergent – which is the case where one would really have liked to use it
- This is a dynamic load sharing strategy, which can in future be made adaptive by soliciting dynamic load information from PNs and using it in conjunction with the “relative power” used in the calculations. For this, another way would have to be found to maintain the “index property” e.g. to define a different hash-partitioning scheme
- This method does not counter data skew, access skew or distribution skew, since it does not incorporate load-balancing.

Expressly due to the data skew potential, and the fact that the number of PNs actually available is not very large (to mitigate these ill-effects) this scheme was abandoned after some preliminary tests. However, in a situation where a large number of PNs are in fact available, this load-sharing algorithm can be gainfully employed not only by the Reader process but also by the RDT Worker (that pipelines the output to a client Worker). Another factor was that this could have been an unfair advantage for RDT over BST.

3.7.4 Worker

The Worker process performs the actual hash join operation on the two streams (bucketfuls) of tuples that it receives. One Worker process runs on a single PN, but more than one Worker can co-operate to parallelize a single JN. Up to two Workers from different JNs can share the same PN, if no free PNs are available. The Worker process differs significantly between the RDT and BST experiments, in two important respects:

- It performs the Simple Hash Join (SiHJ) algorithm in BST, whereas in RDT it performs the Pipelining Hash Join (PiHJ) algorithm
- For BST, it writes its output to a file, but for RDT it pipelines its output to a Scheduler-determined Worker at the next (client) JN.

These differences are described in further detail in the specific sections 3.7.5 on page 74 for BST and 3.7.6 on page 79 for RDT.

3.7.5 BST

This section describes the BST-specific parts of the implementation. These parts relate to the Scheduling and SiHJ algorithms.

3.7.5.1 BST Scheduler

The BST Scheduler process for BST performs the following steps:

- Reads in the query type, including the names of input relations
- Sets up global instance QresponseT of class MeasTime to record QET
- Repeatedly calls an internal function called BSTREE.
- Cleans up each run to start the next one, and finally
- Prints out the QET results

BSTREE performs the following tasks:

- Estimates the cost of all concurrent JNs. The cost estimation is progressed level-wise, bottom up, since that is how the JNs are launched as well.

- Maintains a queue, ReadyQ, of JNs, which are ready to execute because their inputs are ready. All the JNs of level 1 are readied first by dividing all the available PNs to them in proportion to their workloads.
- As and when all PNs of a JN signal their completion, calls the fini() function for that JN, where the PNs are freed up and result cardinality is updated to reflect the actual rather than an estimate
- If the just completed JN is the last (root) JN, stops the QET timer by calling QresponseT.end()
- Lets each Worker at a JN know the fileName to write, and also communicates this fileName to the Reader at the next level

The lowest level of JNs are readied first. as follows:

```
#define HNAM_PFX "fn"    // for fast network: would be "n" for slow network
int jnode = 0, level = 0,
    jtids[MAX_JNS][MAX_PN_PER_JN];
    level_JNs = num_input_relations / 2 ;    // #JNs in current level
int cost[ total_number_of_JNs];
int Query_Cost[MAX_LEVELS];    // Query_Cost recorded per level
JN_Info_cla jRdy;

while( jnode < total_number_of_JNs )
{
    for( sibling = 0; ((sibling==0) || (sibling < level_JNs)); sibling++ )
    {
        cost[jnode] = JN_Info[jnode].init( jnode, xjn, yjn, cardX, cardY, cardZ );
        Query_Cost[level] += cost[jnode];
        jnode++;
    }
    level++;
    level_JNs /= 2;
}
```

Following this, the main processing occurs as an iteration that remains active as long as there are JNs to be completed. This consists of a sequence of two subloops. The first one launches the ready JNs:

```
int numPNs = 0,
    PVM_Power = sum of power of available PNs;

while( node < level_JNs )
{
    int wantedPower = MAX( 100,    // At least this power MUST be asked
                           ceil( (cost[node] / Query_Cost[level]) * PVM_Power )
                           );
    numPNs = JN_Info[node].allotPNs( wantedPower );
    JN_Info[node].ready();
    readyQ.append( JN_Info[node] );
    numPNs = 0;
}
```

The second one comprises two nested loops, and is shown on the next two pages. The outer 'while' is active until there are more JNs to be completed. The first nested 'while' looks for ready JNs in the ready queue and launches them by spawning the various Worker and Reader processes. The second nested 'while', triggered by the completion of any JN, looks for JNs which can now be made ready. This loop is executed as long as there are JNs remaining to be launched, and no new ones have been made ready inside this while.

```

while( number of completed JNs < total number of JNs )
{
    while( !(readyQ.empty()) )
    {
        jRdy = readyQ.pop() ;

        // Setup arguments in t_arg, hname, and spawn the Workers
        int buckets = jRdy.num_bkts;
        for( int bkt=0; bkt < buckets; bkt++ )
        {
            char hname [MAX_PN_NAME];
            strcpy( hname, HNAM_PFX );
            strcat( hname, (char *) &PN_Info[jRdy.got_PNs[bkt]], 1 );
            strcat( hname, "\0" );
            pvm_spawn( "simht", (char **) t_arg,
                       PvmTaskHost, hname, 1, &jtids[node][bkt]);
        }
        // Now spawn Readers for the two inputs to above JN
        // All arguments (p_arg, hname) are set up separately for each spawn
        pvm_spawn( "bsread", (char **) p_arg,
                   PvmTaskHost, hname, 1, &pitids[node] );
        pvm_spawn( "bsread", (char **) p_arg,
                   PvmTaskHost, hname, 1, &potids[node] );

        // Send JTID values to POTIDs and POTID value to JTIDs so the
        // JNs can tell the POTID to start sending the outer tuples once
        // the Hash Table for inner tuples is ready.

        if( !node )      // This is first node i.e. node == 0
            QresponseT.begin();
    } // while readyQ not empty
}
    
```



```

// Look for Z_TAG messages until there are more JNs remaining to be launched
while( (! newJNrdy) && remainJNs ) &&
    (bufid = pvm_trecv( -1, -1, &tmout ) )
{
    newJNrdy = false;
    info = pvm_buinfo( bufid, &bytes, &msgtag, &stid );
    switch( msgtag )
    {
        case Z_TAG:
            pvm_upkshort( &jnid, 1, 1 );
            pvm_upkint( &cNz, 1, 1 );

            // Update "num_fini" PNs in this JN
            if( JN_Info[jnid].fini( cNz ) == JN_Info[jnid].num_bkts )
            {
                if( this is the last JN )
                {
                    QresponseT.end();
                    break;
                }

                for( all non-leaf JNs )
                {
                    If ( JN in BORN state, AND
                        its inner and outer supplier JNs are in FINI state )
                    {
                        Ready the JN:
                        newJNrdy = true;
                    }
                }
            } // If fini()
            break;
        }
    }
    if( newJNrdy )
        break: // exit the while
} // while not newJNrdy }
} // while( number of completed JNs < total number of JNs )

```

3.7.5.2 Simple Hash Join

The SiHJ algorithm used in BST experiments is very similar to the basic one described earlier. Each Worker write its output to an intermediate file, which is later read by a Reader to serve the Worker at the next level JN.

Phase 1: BuildTable

```
for each tuple r in R
{
    hash_value ← hash ( r(a) );
    HashTable[hash_value] ← r;
}
```

Phase 2: ProbeTuple

```
for each tuple s
{
    hash_value ← hash ( s(b) );
    if HashTable[hash_value] is not empty
    then
        if s matches any r in HashTable[hash_value]
        then
            Q[result++] ← r + s;
            if( Q is full )
                append it to the intermediate file;
}
```

3.7.6 RDT

This section describes the RDT-specific parts of the implementation. These parts relate to the Scheduling and PiHJ algorithms.

3.7.6.1 RDT Scheduler

The RDT Scheduler process for RDT performs the following steps:

- Reads in the query type, including the names of input relations
- Sets up global instance QresponseT of class MeasTime to record QET
- Repeatedly calls an internal function called RDTREE
- Collects the result tuples from the last JN, and at the same time stops the QET timer by calling QresponseT.end()
- Prints out the QET results

RDTREE performs the following tasks:

- Estimates the cost of all the JNs, since they will all be started concurrently.
- Divides all the available PNs to them in proportion to their workloads
- Lets each Worker at a JN know the identities (TIDs) of the Workers at the next level whom it has to supply the pipelined result tuples of this join operation
- On starting JN-0, starts the QET measurement by calling QresponseT.begin()

RDTREE, has no ready queue, since all JNs are to be started right away in a pipeline.

Query_Cost is calculated as in BST, but for all JNs together, not just for one level. The main processing is an iteration over all JNs. This consists of a sequence of two subloops: the first one launches the ready JNs; while the second one, triggered by the

completion of any JN, looks for JNs which can now be made ready. This processing can be summarized as follows:

```

for( int jnode = 0; jnode < total_number_JNs; jnode++ )
{
    Query_Cost += (cost[jnode] = JN_Info[jnode].init( jnode, NIL_JN.
                                                         (jnode) ? jnode-1 : NIL_JN, cardX, cardY, cardZ ));
} // for jnode

int numPNs = 0.   PVM_Power = sum of power of available PNs;

for( int node = 0; node < total_number_JNs; node++ )
{
    int wantedPower = MAX( 100,    // At least this amount MUST be asked
                           ceil( (cost[node] / Query_Cost[level]) * PVM_Power )
    );
    numPNs = JN_Info[node].allotPNs( wantedPower );
    JN_Info[node].ready();
    numPNs = 0;
}

```

```

for( node = total_number_JNs; node >= 0; node-- )
{
    int count = 0, numt=0, buckets = JN_Info[node].num_bkts;
    if( Last J-node)
        // Setup to File output to scheduler

    // Setup arguments in t_arg, hname, and spawn the Workers
    int buckets = jRdy.num_bkts;

    for( int bkt=0; bkt < buckets; bkt++ )
    {
        // Setup args similarly as in BST, but use PiHJ algorithm
        pvm_spawn( "pipht". (char **) t_arg,
                    PvmTaskHost, hname, 1, &jtids[node][bkt]);
    }

    // Now spawn Readers for the two inputs to above JN
    // All arguments (p_arg, hname) are set up separately for each spawn
    pvm_spawn( "rdread". (char **) p_arg,
                PvmTaskHost, hname, 1, &pitids[node] );

    // Send JTID values to PITIDs

    if( it is leaf-JN ) // Need outer input from file
        pvm_spawn( "rdread". (char **) p_arg,
                    PvmTaskHost, hname, 1, &potids[node] );

    // Send JTID values to POTIDs and POTID value to JTIDs so the
    // JNs can tell the POTID to start sending the outer tuples once
    // the Hash Table for inner tuples is ready.
    if( !node )
        QresponseT.begin(); // If JN-0, start recording time

    // Now deliver to each J-Node the JTIDS of the next J-Node in chain
}
}

```

3.7.6.2 Pipelined Hash Join [Wil95]

The PiHJ algorithm used in RDT experiments is a pipelined version of SiHJ. Both the build and probe phases are merged into one build-and-probe phase by concurrently building two hash tables, one for each of the two input streams of tuples. Each Worker pipes its output directly to serve the Worker at the next level JN.

Phase 1: BuildAndProbeTable

```

for each tuple received
{
    hash_value ← hash ( tuple(a_b) );

    case tuple is
    {
        r from R : HashTable_R[hash_value] ← r
                    if HashTable_S[hash_value] is not empty
                    then
                        if r matches any s in HashTable_S[hash_value]
                        then
                            Q ← r + s;
                            if( Q is full )
                                send it to the Worker at the next JN;
                            break;

        s from S : HashTable_S[hash_value] ← s
                    if HashTable_R[hash_value] is not empty
                    then
                        if s matches any r in HashTable_R[hash_value]
                        then
                            Q ← r + s;
                            if( Q is full )
                                send it to the Worker at the next JN;
                            break;
    }
}
    
```

3.8 Analysis of a Generic Query

A generalized description of the various query tree structures can consider SRT as the general case, with RDT and BST as specializations. This generic approach is introduced here to illustrate the different phases of the general hash-join algorithm. To maintain genericity, the join operations are assumed to perform the Simple Hash Join (SiHJ) algorithm in this section. The use of Pipelining Hash Join (PiHJ) can be incorporated in the formulae as future extension. The generic equations are potentially useful for any study that spans the whole range of query trees from RDT to SRT to BST.

Let R_i be the inner relations, S the outer relation(s), totalling to 'q' relations in the query having N processing nodes, which is executed as m segments of k stages (joins), where k is variable with an estimated upper bound of $\text{ceil}(q/m)$. If M is the memory size of each node, m is estimated as

$$m = \left\lceil \left(\sum_{i=1,q} |R_i| \right) / N.M \right\rceil$$

Each q -join Query is executed as $(m \leq q)$ segments, each segment made of k stages. In case enough memory is available, it may be possible to have a single segment i.e. $m = 1, k = q$. This is the case for RDT. In case of BST, each JN is a segment so that $m = q, k = 1$.

Within each segment, a two-phased algorithm is performed. The two phases are Table Building (TB) and Tuple Probing (TP), which are described below. For each phase, the Total Work (TBW, TPW) and the Elapsed Time (TBT, TPT) are estimated. Table 4 shows the cost parameters used for this purpose.

Table 4 Cost Variables for Analytic Comparison

Cost	Meaning
T_{rd}	Avg cost of reading a tuple from disk
T_{pt}	Cost of applying the partition function to a tuple, usually equal to T_{hs}
T_{sd}	Avg cost of sending a tuple to the network
T_{rc}	Avg cost of receiving a tuple from the network
T_{hs}	Avg cost of applying the hash function to a tuple
T_{in}	Avg cost of inserting a tuple into a hash subtable
T_{cm}	Avg cost of comparing a tuple to entries in a bucket
T_{bd}	Avg cost of building a result tuple, usually equal to T_{in}
T_{wr}	Avg cost of writing a tuple to disk
C1	$T_{rd} + T_{pt} + T_{sd} = T_{rd} + T_{hs} + T_{sd}$
C2	$T_{rc} + T_{hs} + T_{in}$
C3	$T_{rc} + T_{hs} + T_{cm}$
C4	$T_{bd} + T_{hs} + T_{sd}$
C5	$T_{bd} + T_{wr}$

It is safe to assume that the cost of computing a partition or hash function is the same as that of comparing keys or building a tuple, because all of these costs differ by microsecond amounts and anyway are quite small as compared to T_{rd} or T_{wr} . Note that the Simple Hash Join (SiHJ) algorithm is assumed in this section.

3.8.1 Table Building (TB) phase

In this phase the hash tables of all stages are built. If each relation is a file, only one node can read a relation at a time and hash the tuples, passing the tuple to its destination node. Hence, in this phase, only one PN per relation is sufficient for reading the relations. This phase has 3 steps:

- 1) Allocate PNs to Joins. This step is not counted towards query execution time or workload.
- 2) The $k-1$ nodes read inner relations R_i , $i = 1, k$ and use the partition function of stage i to hash the join attribute $r(a)$ of R_i tuples into n_i partitions, where n_i is the number of nodes assigned to stage i of the segment. The partitioned tuples are then sent to their nodes in stage i .

The total work for this step for all stages of a segment j , is approximated as:

$$TBW2_j = (T_{rd} + T_{pt} + T_{sd}) \sum_{i=1}^k |R_i|$$

which is rewritable as:

$$TBW2_j = C_1 \sum_{i=1}^k |R_i|$$

Almost all of this work is performed in the SD environment by the data server, resulting in pseudo-serialization of disk reads, though the other other two (much smaller components) occur in time-overlap. Assuming a MAX function, this brings us to an estimate of the execution time for this phase:

$$TBT2_j \cong T_{rd} \text{MAX}_{i=1}^k |R_i|$$

Alternately, in a strictly SN scenario, the model would have had multiple machines all reading these tuples in parallel.

- 3) Each node in stage i , $i = 1, k$, receives the tuples of its corresponding partition, hashes the tuples with the hash function of the stage (join) and inserts them into hash buckets. The total work for Step 3 for all stages of a segment j , is:

$$TBW3_j = C_2 \sum_{i=1}^k |R_i|$$

This work is performed by the k machines participating in the parallel or pipelined join operation. Hence, the time taken is reduced to:

$$TBT3_j \equiv C_2 \sum_{i=1}^k |R_i|$$

The total work for TB phase of all stages of a segment j , is therefore summarised as:

$$\begin{aligned} TBW_j &= TBW2_j + TBW3_j \\ &= (C_1 + C_2) \sum_{i=1}^k |R_i| \end{aligned}$$

Since the reading overlaps in time the hash-partitioning and transmission of the tuple, the total time for TB phase of all k stages of a segment j , is approximated as:

$$TBT_j \equiv (T_{rd} + C_2) \sum_{i=1}^k |R_i|$$

3.8.2 Tuple Probing (TP) phase

The pipeline segment now starts tuple probing as follows.

- 1) The tuples of outer relation S are read from disk in blocks by the server (or by all nodes, or the k^{th} node for PHJ-see below), and using the partition function of stage 1, they are hashed and routed/broadcast to the corresponding nodes in

stage 1. Tuples are sent to a destination node whenever there are enough to form a communication packet.

The work in this step for all stages of a segment j , is approximated as:

$$TPW1_j = C_1 \sum_{i=1}^k |S_i|$$

This work is performed by the k machines participating in that segment.

- 2) For each node in stage i , $i = 1, k$, whenever a packet of input tuples is received, they are used to probe the hash bucket of R_i using the hash function of stage i . Any matches result in intermediate tuples which are immediately partitioned with the hash function of the next stage $i+1$; and on accumulation are assembled into a communication packet for the next destination node. If σ_i represents the selectivity between relations R_i and S_i then the work in this step for all stages of a segment j , is approximated as:

$$TPW2_j = C_3 \sum_{i=1}^k |S_i| + (C_3 + C_4) \sum_{i=1}^k (\sigma_i |S_i| |R_i|)$$

This work is performed by the k machines participating in that segment.

- 3) Finally, the output tuples at stage k are the result tuples for the whole pipeline segment, and are written to disk block by block. This work for all stages of a segment j , is approximated as:

$$TPW_{3j} = C_5 \sum_{i=1}^k (\sigma_i |S_i| |R_i|)$$

This brings us to the total work for TP phase as:

$$TPW_j = (C_1 + C_3) \sum_{i=1}^k |S_i| + (C_3 + C_4 + C_5) \sum_{i=1}^k (\sigma_i |S_i| |R_i|)$$

The total time for TP phase is a bit difficult to estimate, particularly since some of the factors going into it are not modeled in this simple analysis. The major such factor is pipeline setup and wind-down time, say PST. Then, assuming that the pipeline results in division of the total work among the k stages roughly equally (perhaps not too far from the fact), one can come up with the following estimate:

$$TPT_j \cong 2 PST + (TPW_j - 2 PST) / k$$

From the above components, the total query work and QET can be calculated for the generic case. With appropriate substitutions and summations over all the m stages, the corresponding metrics for the specific query trees can also be estimated.

3.9 Example Comparison of RDT and BST

In this section the cost variables listed in Table 4 above are used to compare the execution time of an example query when the tree used is RDT versus BST.

For this purpose, we assume we have 4 base relations, each containing 5000 tuples. In either case, we have 3 joins or JNs. Let us assume further that the number of output tuples produced by the three JNs is the same in RDT and BST as given by Table 5.

Table 5 Output tuples for example scenario

Tree	JN-0	JN-1	JN-2
BST	3000	2000	1000
RDT	3000	2000	1000

The JNs are numbered bottom-up, left to right, as depicted in Figure 9 below. Input relational tables (and the tasks that read and partition them) are represented with the letter “t”. The comparison is done in Table 6 by calculating the intermediate and total costs using the same general approach as illustrated in Section 3.8.

Figure 9 Comparison of RDT and BST for example query

The suffixes in the Cost column denote the JN to which they relate. The acronyms TBW and TPW stand for the Table Building Work and Tuple Probing Work respectively. We count in terms of 1000's for brevity.

Shown in square brackets is the worst-case overwork (and optimistic time-gain) in RDT due to the use of pipelining hash-join (PiHJ) as against the simple hash-join (SiHJ) of BST. It is analytically estimable how many outer tuples are “built” into the outer hash

table because we stop building it as soon as the inner hash-table is complete – the two would probably have been built to similar size by that time. It is still infeasible to estimate the variable amount of time gain this yields, since selectivities for non-leaf joins are unknown. It is due to this reason that this figure is only shown for illustration, but not used in any subsequent formulae. In the following discussion, D is the depth of the query tree and B is the bucket size (BKTSIZE) in tuples. Its value is used as 100 tuples, to match the experiments performed. The explanations and comparison are covered in separate subsections below.

Table 6 Analytical Comparison of RDT and BST for example query

Cost	RDT	BST
TBW ₀	$5 (C_1 + C_2)$	$5 (C_1 + C_2)$
TBW ₁	$5 (C_1 + C_2)$	$5 (C_1 + C_2)$
TBW ₂	$5 (C_1 + C_2)$	$3 (C_1 + C_2)$
TB Work	$15 (C_1 + C_2)$	$13 (C_1 + C_2)$
TB Time	$5 (C_1 + C_2)$	$8 (C_1 + C_2)$
TPW ₀	$5 (C_1 + C_3) + 3 C_4 [+ 5 T_{in}]$	$5 (C_1 + C_3) + 3 C_5$
TPW ₁	$3 C_3 + 2 C_4 [+ 3 T_{in}]$	$3 C_3 + 2 C_5$
TPW ₂	$2 C_3 + C_5 [+ 2 T_{in}]$	$2 C_3 + C_5$
TP Work	$5 (C_1 + C_2 + C_3 + C_4) + C_5$	$5 (C_1 + C_2 + C_3) + 6 C_5$
TP Time	$2 T_{in} + (TPW - 2 T_{in}) / D$	$13 (C_1 + C_2) + 2 C_3 + 4 C_5$
Total Work	$20(C_1 + C_2) + 5(C_3 + C_4) + C_5$	$18(C_1 + C_2) + C_3 + 6C_5$
Total QET	$6.9 C_1 + 5.2 C_2 + 3.7 C_3 + 2.1 C_4 + 0.5 C_5$	$13 C_1 + 8 C_2 + 7 C_3 + 4 C_5$

3.9.1 BST Analysis

In case of BST, the two join nodes J_0 and J_1 effectively proceed in parallel, so that the total query execution time QET is estimated as:

$$QET_{bst} = TBW_2 + TPW_2 + \text{MAX}((TBW_0 + TPW_0), (TBW_1 + TPW_1))$$

Since in the example the work at J_0 is greater than that at J_1 , the above reduces to:

$$\begin{aligned} QET_{bst} &= [TBW_2 + TPW_2] + [TBW_0 + TPW_0] \\ &= [3(C_1 + C_2) + 2C_3 + C_5] + [10C_1 + 5(C_1 + C_2) + 3C_5] \\ &= 13C_1 + 8C_2 + 7C_3 + 4C_5 \end{aligned}$$

The last term accounts for the fact that output of each join stage is written to disk. While this may appear to be a major overhead, in fact some of it occurs in parallel with the rest of the join computation, but is hard to quantify. This ignored gain is probably of similar magnitude as the gain ignored for RDT due to use of PiHJ algorithm. In both cases, the significance of the ignored gain shows up only when it accumulates over a number of the JN stages, or over a large number of tuples. This is why the actual performance measurements by experimentation are important part of the puzzle.

3.9.2 RDT Analysis

For RDT, the following points are of interest. In the TB phase, the TBW_i all occur in parallel, so that TB Time is:

$$TBT_{rdt} = \text{MAX}(TBW_i)$$

Pipelining is not effective during the processing of the first and last bucket of BKTSIZE tuples. This fringe time is estimated to be at least:

$$T_{fri} = D * B * (C_1 + C_2 + 2C_3 + 2C_4 + C_5)$$

The T_{in} components represent additional work done to build, in each PiHJ node, the second hash table, while the first one is still incomplete. This additional work occurs in parallel, but its small contribution to decrease the QET is inestimable.

The query time for RDT is approximated with the assumptions of :

$$D = 3,$$

$$B = 100 \text{ tuples (or 0.1 K),}$$

$$T_{fri} = 0.3 * (C_1 + C_2 + 2C_3 + 2C_4 + C_5)$$

as:

$$QET_{rdt} = TBT_{rdt} + 2 * T_{fri} + (TPW_{rdt} - 2 T_{fri}) / D$$

The above formula reduces to:

$$\begin{aligned}
 QET_{rdt} &= TBT_{rdt} + 2 * T_{fri} + \\
 &\quad [\{ 5 (C_1 + C_3) + 3 C_4 \} + \{ 3 C_3 + 2 C_4 \} + \{ 2 C_3 + C_5 \}] / 3 \\
 &\quad - 2/3 \{ 2 * T_{fri} \} \\
 \\
 &= TBT_{rdt} + 2/3 * T_{fri} + [5C_1 + 10 C_3 + 5 C_4 + C_5] / 3 \\
 \\
 &= [5(C_1 + C_2)] + 2/3 [0.3 * (C_1 + C_2 + 2C_3 + 2C_4 + C_5)] + \\
 &\quad 1.7 C_1 + 3.3 C_3 + 1.7 C_4 + 0.3 C_5 \\
 \\
 &= 5(C_1 + C_2) + 0.2 * (C_1 + C_2 + 2C_3 + 2C_4 + C_5) + \\
 &\quad 1.7 C_1 + 3.3 C_3 + 1.7 C_4 + 0.3 C_5 \\
 \\
 &= 6.9 C_1 + 5.2 C_2 + 3.7 C_3 + 2.1 C_4 + 0.5 C_5
 \end{aligned}$$

3.9.3 Comparison of RDT and BST

From the coarse analysis of the previous two sections, it is clear that RDT should yield a QET better than that from BST by at least:

$$QET_{diff} = QET_{bst} - QET_{rdt}$$

which amounts to:

$$QET_{diff} = 6.1 C_1 + 2.8 C_2 + 3.3 C_3 - 2.1 C_4 + 3.5 C_5$$

This difference should increase with the sizes of input relations and more so with the depth D of the RDT. As the number of joins in the query tree increases, the RDT is expected to perform better. Conversely, if there are only a few JNs, the difference may not be noticeable much. Most of the time values going into the C_i are of the order of microseconds, except C_1, C_5 which are in the range of tens of milliseconds. In smaller queries, the actual gains are likely to be overshadowed by the overheads of setting up the pipeline and the fact that the benefits are distributed over the lifetime of the query and over the various buckets of flowing data.

Chapter 4 Performance Analyses

This chapter describes the details of the experiments conducted, how they were organized, the parameters used and discusses the results derived from the experiments. The overall conclusions of the thesis, based on these experiments, are provided in the next chapter.

4.1 Introduction

Four major categories of experiments were conducted. The first category deals with establishing the effect (if any) of varying the bucket size, BKTSIZE. The second category concerns the performance of the parallel hash join algorithms BST and RDT when using PVM over a slow (10 Mbps) Ethernet LAN. A third group of experiments repeated the previous group using the same PVM environment but running over fast (100Mbps) Ethernet. The final group of experiments included the same queries and algorithms but using the larger sized input relations. The fast network was used for this.

In each experiment, a set of one or more queries is chosen depending on the purpose of the experiment. The query or queries are executed first with BST query structure using the SiHJ algorithm (see previous chapter), and then the same set of queries is subjected to the RDT execution using the PiHJ algorithm (see previous chapter). For comparison,

one of the experiments also includes a Pure RDT execution, where no intra-operator parallelism is allowed. In BST and RDT, each JN is allowed to parallelize into two PNs by setting the MAX_PN_PER_JN compile-time constant. It would have been desirable to experiment with higher values of MAX_PN_PER_JN, but this was not possible due to paucity of available Triton nodes in the laboratory. and the need to use a consistent value of this constant across all runs in the same experiment to assure an apples to apples comparison. This factor has also restricted the performance of RDT to an extent, since RDT requires the concurrent allocation of all the required PNs throughout the query.

4.1.1 Input Parameters

The following parameters are either kept constant or varied across certain experiments:

- Number of tables
- Names of relations to join
- Number of sample points
- Maximum number of PNs per JN
- Bucket Size (BKTSIZE)
- Pair wise selectivities of base relations
- Number of times to run the experiment

All the following experiments were repeated 30, 50 or 60 times, as required, to get a reasonable 95% confidence interval within $\pm 10\%$.

4.1.2 Description of Experiments

The outline of the simulation experiments is described below by separating the common and variant components. The common basis in all experiments can be described as:

- Single Complex (i.e. multi-join) Query
- Use of NOW, as a parallel virtual machine configured with PVM on a local network
- Each workstation in the NOW provides a virtual Processing Node (PN) to one or more Reader or Worker processes.
- MAX_PN_PER_JN parameter is set to 2, due to limited number of nodes
- The PNs together are considered a Shared Nothing (SN) multiprocessing environment².
- Work-proportional allocation of joins to PNs, using a simple cost-estimator [Wil95], briefly described in section 3.7.1.4.

² Cross-mounted disks accessible across all workstations in the NOW are considered SN because the disks are usually not multi-ported and each relation is a sequential file accessed by one process at a time.

The following variations are examined in the experiments:

- The size of query in terms of the number of joins
- The size of input relations in terms of number of tuples – two sets of tables are used, one representing normal tables (5K to 9K tuples) and another set representing large tables (15K to 50K tuples).
- The pair wise selectivities of input relations
- Presence of increasing workload on the NOW members
- Two versions of Hash-Joins were used, as described in [Wil95]. These two are:
 - Simple Hash Join (SiHJ) where the Hash Table is built only for the inner relation; and
 - Pipelining Hash Join (PiHJ), where Hash Tables are built for both inner and outer operands until one of them runs out of tuples.

It would be interesting to use a formal skew distribution such as Zipf-like distribution, as demonstrated in [HLY93]. This was beyond the scope of the current work.

4.2 Effect of Bucket Size

The first task in performance study was to ascertain whether BKTSIZE had a significant impact on performance. Experiments were performed with six input tables on a slow network while varying the BKTSIZE from 50 tuples to 500 tuples. It was found that if BKTSIZE is 100 tuples or more, it has negligible impact.

Figure 10 Effect of BKTSIZE

4.3 Slow Network Experiments

This section reports the results of experiments to compare performance of BST and RDT using the slower 10 Mbps LAN.

4.3.1 Effect of Query Complexity

This experiment used the set of normal tables to run four different queries – with number of input tables going from 2 through 4, 6, and 8. These queries were run using first the BST query structure, and then the same set of four queries were used in the RDT execution. In this experiment a variation of RDT was also studied. This third query structure is called Pure RDT because it has no intra-operator parallelization. This is obtained by setting the MAX_PN_PER_JN parameter to 1.

The four queries are chosen to have increasing number of join operators and comparable table sizes (between 5K and 9K tuples) to represent increasing amount of query complexity. It is observed that the Pure RDT approach suffers significantly with increasing query complexity, and hence is excluded from further consideration in the remaining experiments.

Figure 11 Effect of Query Size under Slow LAN

It is found that for these smaller table sizes, the BST query structure with SiHJ algorithm outperforms RDT running with the PiHJ algorithm. This is consistent with previous results where BST was found to be better for smaller input relations.

4.3.2 Effect of Input Selectivity

Two queries, one with a single join and another with 3 join operations (or 4 input relations) was selected to study the effect of selectivity. The queries are selected such that effect of other factors similar on both BST and RDT, as shown by the other experiments.

For the first set of experiments, a single join node was selected to guage the effect of PiHJ versus SiHJ algorithms. In this experiment, the inner relation is the same (t0, with 5K tuples, while the outer relation is of increasing selectivity, though constant input size in terms of number of tuples (7K), as shown in Table 7.

Table 7 Input Relations for Single Join Selectivity

Query	Join J0		
	Inner	Outer	Sel.
1	5000 (t0)	7000 (t6)	0.7
2	5000 (t8)	7000 (t2)	1.4
3	5000 (t0)	7000 (t11)	2.0

The QETs are displayed in Figure 12, and discussed at the end of this section.

Figure 12 Effect of Single Join Selectivity in Slow Network

In the second selectivity experiment, four input relations at a time were selected from the normal data set for three related experiments such that the average selectivity of the two pairs of relations is 0.6, 1.3 and 1.9, as shown in Table 8 below.

Table 8 The Input Relations for Average Selectivity Experiment

Query	Join J0			Join J1			Avg. Sel.
	Inner	Outer	Sel.	Inner	Outer	Sel.	
1	5000 (t0)	8000 (t3)	0.6	5000 (t12)	7000 (t6)	0.6	0.6
2	5000 (t8)	8000 (t5)	1.2	5000 (t0)	7000 (t2)	1.4	1.3
3	5000 (t0)	7000 (t11)	2.0	6000 (t10)	7000 (t11)	1.7	1.9

Figure 13 Effect of Avg. Selectivity in Slow Network

The parentheses in each cell of Table 8 show the number of the table used from the normal data set. The results in Figure 12 and Figure 13 show that BST retains its advantage in this region where input tables or query complexity are small, number of

join nodes is one to three and the network speed is slow. The single join data also show that effect of increasing selectivity is very marginal, indicating scalability of the PiHJ and SiHJ algorithms with selectivity in single joins.

4.3.3 Effect of Background Load

A program was written to simulate background load by performing a compute-bound loop for a portion of time, followed by sleeping for another period of time. A low background load was simulated by the program alternating between one second of active busy time and 2 seconds of sleeping. A high background load was simulated by having the program sleep and wake for 2 seconds each in a cycle.

The same queries were used here as in the previous two experiments of sections 4.3.1 and 4.3.2 above. The results are presented in the next two figures.

Figure 14 Experiment with Low Background Load

It is observed that RDT suffers more from periodic background loads as compared to BST. This is explained by the fact that simultaneous availability of the various PNs constituting the pipeline is important for the smooth flow of tuples through the pipeline. Loss of any one PN can disrupt or slow down all the downline PNs for the duration of the disruption. Such is not the case in BST.

The same behavior is exhibited in a slightly more pronounced way during the next experiment (see Figure 15) where background load is increased to twice the previous amount. This clearly shows that a linear tree or pipeline is more vulnerable to stolen CPU cycles.

Figure 15 Experiment with High Background Load

4.3.4 Summary of Slow Network Experiments

The slow network experiments substantiate existing knowledge that BST performs better when the query complexity is small and table sizes are small. For NOW scenario, they also show that BST performs better than RDT when background load is high.

If we look a bit deeper into the issues involved, we find that another reason for the results in these two sections is as follows. With a large number of joins, RDT needs larger number of PNs all at the same time, to be really effective, since under RDT all JNs are active simultaneously. In the event PNs are shared between JNs, the resources of the PN are shared between JNs, impacting (by reduced CPU power and memory for each JN) not only the directly affected JNs but also the other JNs strung in the pipeline.

Similarly the Reader process, with its disk accesses, shares its PN with at least one JN, thereby also affecting the entire pipeline to some extent. In case of BST, this affects only the leaf JNs.

4.4 Fast Network Experiments

This section compares the performance of BST and RDT using the faster 100 Mbps LAN. Query Complexity and Input Selectivity classes of experiments were repeated to see the effect of increasing the speed of the underlying LAN. This is important, since 100 Mbps and even higher (up to gigabit) speeds are becoming common. The market is looking forward to terabit routers in the near future.

The experiment with large data tables was not performed with the slow network, and is only performed with the fast network, due to time considerations.

4.4.1 Effect of Number of Join Nodes

The same experiment as described in Section 4.3.1 was repeated for fast network. The results are shown in Figure 16 below. The comments regarding Pure RDT still hold. Both RDT and BST are seen to benefit from faster network. It is interesting to note that, for the 5-join query, RDT has performed better than BST because it has been able to exploit the combination of intra-operator parallelism and fast network speed to a better level than BST. The prime reason for this is the memory-to-memory communication of tuples between adjacent JNs. RDT could have achieved a similar speedup for the 7-join query if it had a proportionate number of PNs available. BST

does not need all its PNs at once, so is able to harness more PNs per join operation, leading to its better performance in the largest query on the graph.

Figure 16 Effect of Query Size under Fast LAN

It is argued that if more PNs were available, the ability to run the RDT and PiHJ combination with 3 to 6 PNs per JN would have resulted in better performance for RDT in the 7-JN case. The lack of adequate number of PNs inflicts a three-fold penalty on RDT. The first count is that even one JN slowing down in RDT slows down the entire pipeline. The second count is that a smaller number of processors per JN means a smaller amount of memory is available for the JN, resulting in more disk i/o in the middle of hash-join processing, leading to further slowdown. The third count is that parallelization costs are higher, due to larger number of JNs, but the concomitant benefits of parallelization are missing due to lack of enough PNs.

Hasan [H96] reported that some parallelizations can be “worthless” if the resulting cost of communication was higher than the savings. In this experiment, BST has suffered from some extent of higher communication cost, which is apparently caused by data skew, resulting in higher file i/o time for some of the intermediate results. The exact same skew in the RDT execution would have a smaller impact due to in-memory transfer in the pipeline.

As such, higher network speeds have been better utilized by BST in the most complex query, to improve the time taken to store and read from disk the intermediate results of each of the 7 JNs.

These results prove that BST performs better for smaller input relations, smaller queries and for fewer number of available PNs. With increasing query complexity and availability of a fast network, RDT begins to perform better, but it needs all its PNs at the same time to achieve its full potential³.

4.4.2 Effect of Selectivity

The same experiments as described in Section 4.3.2 was repeated for fast network. The QETs for the single join experiment are displayed in Figure 17, and discussed at the

end of this section. As compared with the corresponding slow network case, here PiHJ is seen to have a reduced scalability whereas the execution of SiHJ is still relatively flat. This is because BST involves relatively little overhead for the same selectivity, whereas in case of RDT, the overhead of setting up the second hash table and the various communication paths for the pipelining are still involved, even though there is no benefit gained from it due to non-existence of multiple join nodes. The amortization of the extra costs is zero.

Figure 17 Effect of Single Join Selectivity in Fast Network

The results for the 3-join query average-selectivity experiment are shown in Figure 18 below. It is important to remember that all the queries in this experiment have three join operators and comparable sizes of input relations. The only variant is the

³ This is also the reason why an experiment could not be devised to compare performance of BST and RDT for increasing number of JNs, i.e. 2, 3, 4, 5, 6 JNs. This was feasible for BST but not for RDT, which would need at least 12 PNs for the 4 JN experiment for a meaningful comparison with BST.

selectivity of the base relations. Here RDT shows a smaller QET than BST for the most complex query.

Both RDT and BST are seen to benefit from faster network. BST retains its advantage in the low-complexity region where the average selectivity is 0.6 or 1.3. In the case where average selectivity is 1.9, RDT has been able to exploit the faster network speed to a better advantage.

Figure 18 Effect of Average Selectivity in Fast Network

4.4.3 Effect of Large Table Size

The main target application for parallelization of queries is for very large databases, as mentioned at the beginning in the introduction. Therefore it was important to study how the RDT and BST algorithms would perform when they are exposed to large input relations. These large input relations after going through low selectivities and hash

partitioning into buckets, generate intermediate results of a very variable size. This also simulates some skew across the different join nodes of a query in terms of amount of work.

The input relations sizes and their average selectivities are displayed in Table 9 below from the viewpoint of the BST query structure. These should be seen as an indication of the size and complexity of the input, which remains constant across BST and RDT, even though the actual joins executed are slightly different for the RDT structure.

Table 9 The Input Relations for Large Input Experiment

Query	Join J0 (and J2 etc.)		Join J1 (and J3 etc.)		Avg. Sel.
	Inner	Outer	Inner	Outer	
q07	15K (bt0)	50K (bt7)			3 E-5
q4	15K (bt0)	35K (bt4)	20K (bt1)	30K (bt3)	4 E-5
q6	15K (bt0)	35K (bt4)	15K (bt0)	40K (bt5)	4 E-5
	25K (bt2)	30K (bt3)			
q8	15K (bt0)	40K (bt5)	15K (bt0)	35K (bt4)	5 E-5
	15K (bt0)	30K (bt3)	15K (bt0)	25K (bt2)	

It is found in Figure 19 that in this challenging task RDT performs far better than BST for the largest query. The reason is that in case of BST, even the first hash-table cannot be built for a subsequent stage until the last tuples at the previous stage have been processed and output written to intermediate file. The larger the input file, the larger this wait, and it accumulates quickly for the larger queries.

On the other hand, not only does RDT pass on, as soon as ready, each batch of BKTSIZE output tuples from one join node directly to the next one, but also the next JN right away builds it into a hash table and compares it against tuples already arrived there before. This efficiency adds up for larger queries and larger table sizes, particularly when a fast network is used. Every node speeding up gives a better chance to the whole pipeline.

Figure 19 Effect of Query Size with Large Tables under Fast LAN

On the other hand, the corresponding batch of output tuples in BST is first written to disk. Only after the lowest level joins have completed do these tuples get read by a

Reader process and fed into the next level join node. This entails considerable i/o time which adds up over larger queries and larger table sizes, despite a fast network.

This concludes the exhibits and discussions for the experimental data. A summary of conclusions drawn from the thesis and its overall contribution are delivered in the next chapter.

Chapter 5 Conclusions and Future Work

5.1 Summary

This thesis has studied the performance of two major classes of algorithms for parallelizing query operations, which is very important for large-scale databases.

A comprehensive introduction to the field of database query processing in general and parallel query processing in particular has been provided in the first chapter along with the motivation for this thesis. Chapter 2 presented the overall background information on query trees and join algorithms, including the decision rationale for the selections made for this study. The principal contributions of the thesis are recorded in Chapters 3 and 4. Chapter 3 described the NOW architecture, query algorithms used, a generic cost analysis, and an analytical comparison of the two major algorithms using an example query. Chapter 4 reported the experimental results.

The following conclusions are drawn from this thesis:

- The NOW can be gainfully employed for very large database query processing as a serious alternative to supercomputing architectures

- Both pipelining and partitioning approaches are able to utilize the NOW environment with benefit
- Hash join algorithms are specially amenable to pipelined and partitioned parallelism (jointly and severally) under NOW
- A generic analytical framework is feasible for comparative study of the entire range of query tree structures from BST through SRT and RDT
- For smaller table sizes, the BST query structure with SiHJ algorithm outperforms RDT running with the PiHJ algorithm over the studied ranges of:

≡ query complexity

≡ table selectivities

≡ background workload

- With increasing query complexity and availability of a fast network, RDT begins to perform better, but it needs all its PNs at the same time to achieve its full potential. This means that to achieve the same or better efficiency than BST, there is a need for more minimum number of PNs under RDT
- The advantages of using RDT with PiHJ algorithm in reducing the QET are shown to accrue in complex queries having large tables and many query

operations. This effect is more pronounced under faster LAN speeds due to memory-to-memory communication of data tuples between adjacent JNs of the pipeline. Conversely, it has been confirmed, as previously reported in other situations, that BST performs well for smaller relations and less complex queries.

5.2 Contributions of the Thesis

Most previous studies on parallel query processing have been performed either on specialized architectures or they have used dedicated DBMS software on one of the classical SD, SM, or SN architectures. Also, most early studies looked at balanced trees to reduce the search space, or at left-deep trees because they happened to be the output of conventional query optimizers. Very few studies were reported for right deep trees or their variants. Another important point is that many earlier reports were based on simulations, where subtle interactions and overheads are often overlooked. The experimental results reported here are based on actual implementations and therefore implicitly account for all overheads.

This thesis has extended the study area of parallel query performance to the NOW domain. It has found that RDT executions provide a valuable way of harnessing the available cycles of NOW nodes towards increasing the performance of large and complex queries. It was shown that RDTs are able to support excellent amortization of costs with increasing query size and number of NOW nodes in the system. RDT is also better able to benefit from faster LAN technologies because it relies more on memory-to-memory communication of tuples in the pipeline.

5.3 Future Work

The following aspects are excellent candidates for further study:

- The generic analytical framework (using SRT) may be further expanded. Hasan [H96, particularly chapter 4] provides interesting algorithms and analytic techniques for scheduling of pipelined operator tree. The model of response time given there captures the fundamental tradeoff between parallel execution and its communication overhead. A scheduling algorithm is characterized by its performance ratio, which is the ratio of the response time of the generated schedule to that of the optimal. It uses the concept of “worthless parallelism” to eliminate non-effective parallelization from the schedule.

- Performance of Segmented Right-deep Trees (SRT), which try to combine the characteristics of BST and RDT is the next step. An SRT organizes all operations into suitable pipelined segments and then the segments into a BST of RDTs. Each JN employs the PiHJ algorithm. Such an experiment would allow a finer distinction of the performance ranges in general by varying lengths of pipelined segments allowed. Also, dynamic measurements of available memory and CPU power can yield an algorithm that can dynamically increase or decrease the lengths and number of pipeline segments.
- Study of more adaptive or dynamic load-balancing algorithms. One example is to apply the dynamic load balancing techniques reported in [BFV96] to mitigate sudden load changes. Since those were applied in a hierarchical parallel database system, they may be more beneficial when combined with a hierarchical run-queue of query operators. Another load-balancing strategy uses estimated intermediate result distributions to load-balance parallel hybrid hash-joins [PI96]. An approach that looks at all potential resource bottlenecks (memory, disk and CPU) to load-balance parallel hash-join processing is given in [RM95]. It advocates an integrated consideration of the two sub-problems (inter-query and intra-query) in determining the degree of parallelism chosen to achieve static and dynamic load balancing.

- For low volatility of local PN load, combining the pipelining approach with an integrated (1PO) optimization strategy based on heuristics [SrE93, ZZB93], rather than the 2-Phase one assumed here, could be interesting.
- Query tree can be transformed suitably to improve response time by using tree-balancing techniques described in [DSD95]. This may be useful in BST and SRT executions to better synchronize the outputs produced by parallel segments.
- Using larger number of nodes, larger query complexity and larger data sizes than used here to confirm the prediction of this thesis that RDT should continue to show increasingly better performance as compared with BST, provided at least two or three PNs are available for every operation of the query.

Chapter 6 List of Abbreviations

ACM	Association for Computing Machinery
ANSI	American National Standards Institute
BST	Bushy Tree
CPU	Central Processing Unit
DBMS	Database management systems
I/O, i/o	Input/output
JN	Join node
KBMS	Knowledgebase management systems
LAN	Local area network
LDT	Left-Deep Tree
MAN	Metropolitan area network
Mbps	Megabits per second
MB	Megabytes
MIPS	Millions of instructions per second
MPI	Message Passing Interface
ODMS	Object oriented database management systems
PE	Processing element
PN	Processing node (could involve multiple PEs)
PST	Pipeline Setup (and wind-down) Time (also called T_{fi})
PVM	Parallel Virtual Machine
QET	Query execution time
QP	Query processor
QRT	Query response time
QWT	Query wait time
RDMS	Relational database management systems
RDT	Right Deep Tree
ROM	Read Only Memory
SD	Shared-disk
SN	Shared-everything (memory and disk)
SRT	Segmented Right-deep Tree
SN	Shared-nothing
SQL	Structured query language
VLDB	Very Large Databases
WAN	Wide area network

Chapter 7 References

- [AC95] T. E. Anderson, D. E. Culler, D. A. Patterson et al. A Case for NOW (Networks of Workstations). IEEE Micro. Vol. 15. No. 1. February 1995, pp. 54-64.
- [B80] Sheldon A Borkin. Data models: a semantic approach for database systems. Cambridge, Mass., M.I.T. Press. 1980. 269 pp.
- [B88] H. Boral. Parallelism and Data Management. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, Israel, June 1988.
- [Br84] K. Bratbersengen. Hashing Methods and Relational Algebra Operations. In Proc. 10th Intl. Conf. on VLDBs. Singapore, August 1984. pp.323-333.
- [BFV96] Luc Bouganim, Daniel Florescu and P. Valduriez. Dynamic Load Balancing in Hierarchical Parallel Database Systems. Proc. 22nd VLDB Conf., Mumbai (Bombay), 1996, pp.436-447.
- [C76] Donald D. Chamberlain. Relational Data-Base Management Systems. ACM Computing Surveys 8 no. 1, March 1976.
- [C90] Catherine Ricardo. Database Systems: Principles, Design and Implementation. MacMillan, 1990.

- [CAM62] Campbell (ed), Buchholz, Brooks, Bloch, et al, 1962, Planning a Computer System: Project STRETCH, IBM, 1962. Please see the URL http://www.acs.oakland.edu/~awesley/p_4_2.htm

- [CHY93] Ming-Syan Chen, Hui-I Hsiao and P. S. Yu. Applying Hash Filters to Improving the Execution of Bushy Trees. Proc. 19th Int. VLDB Conf., Dublin, Ireland, 1993, pp.505-516.

- [CL89] M. J. Carey and M. Livny. Parallelism and Concurrency Control Performance in Distributed Database Machines. Proc. ACM SIGMOD Conf., Seattle, Washington., June 1989, pp. 122-133.

- [CLYY95] M.-S. Chen, M. Lo, P. S. Yu, and H. C. Young. Applying Segmented Right-Deep Trees to Pipelining Multiple Hash Joins. IEEE Trans. Knowledge and Data Engineering, 7(4), Aug. 1995, pp. 656-667.

- [Codd72] E.F. Codd. Relational Completeness of Data Sublanguages. In Data Base Systems, R. Rustin (ed.), Englewood Cliffs, NJ, Prentice-Hall, 1972, pp. 65-98.

- [Codd70] E.F. Codd. A Relational Model for Large Shared Data Banks. Commun. ACM, Oct. 1970, 13(6):377-387.

- [CYW92] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries. Proc. 8th Intl. Conf. Data Engineering, pp.58-67, Feb. 1992.

- [D86] C. J. Date. Introduction to Database Systems, Vol. 1, 4th ed. Addison-Wesley, 1986.
- [D97] S.P. Dandamudi, Using Networks of Workstations for Database Query Operations. Proc. Computers and Their Applications, Tempe, Florida, March 1997.
- [DeW86] D. DeWitt et al. GAMMA – A High Performance Backend Database Machine. Proc. 12th VLDB Conf., Kyoto, Japan, Aug. 1986.
- [DeW88] D. DeWitt, S. Ghandeharizadeh and D. Schneider. A Performance Analysis of the Gamma Database Machine. Proc. ACM SIGMOD Conf., Chicago, Ill., June 1988.
- [DG92] D.J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. Comm. ACM, Vol. 35, No. 6, June 1992, pp.85-98.
- [DJ97] S.P. Dandamudi and G. Jain. Architectures for Parallel Query Processing on Networks of Workstations. Int. Conf. Parallel and Distr. Computing Systems, New Orleans, October 1997, pp.444-451.
- [DSD95] Weimin Du, M-C Shan and Umeshwar Dayal. Reducing Multidatabase Query Response Time By Tree Balancing. ACM-SIGMOD Int.Conf., 1995, pp.293-303.

- [DT97] S.P. Dandamudi and T. K. Thyagaraj. A Hierarchical Processor Scheduling for Distributed-Memory Multiprocessor Systems. Tech. Rep. TR-97-15, School of Computer Science, Carleton University, Canada, 1997.
- [DZ97] X. Du, X. Zhang. Co-ordinating Parallel Processes on Networks of Workstations. J. Parallel and Distr. Computing (to appear).
- [F81] T.-Y. Feng. A Survey of Interconnection Networks. Computer, Vol. 14. No. 12, December 1981. pp. 12-27.
- [FS76] Fry, James P., and Sibley, Edgar H. Evolution of Data-Base Management Systems. ACM Computing Surveys 8 (1), March 1976.
- [G93] PVM 3 User's Guide and Reference Manual, Oak Ridge National Laboratories, 1993. (Manual and source code available via anonymous ftp from netlib2.cs.utk.edu in directory /pvm3).
- [G94] PVM: parallel virtual machine: a user's guide and tutorial for networked parallel computing. Cambridge, Mass., MIT Press. 1994. 279 pp.
- [GLS94] W. Gropp. E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, Mass., 1994.

- [GP89] D. Gardy and C. Puech, On The Effects of Join Operations on Relation Sizes. ACM Trans. Database Systems, Vol 14/4, Dec. 1989, pp.574-603.
- [GS92] G. A. Geist and V. S. Sunderam. Network-Based Concurrent Computing on the PVM System. Concurrency: Practice and Experience, Vol. 4, No. 4, June 1992, pp. 293--311.
- [H96] W. Hasan. Optimization of SQL Queries for Parallel Machines. LNCS 1182. Springer-Verlag, 1996.
- [HCY94] Hui-I Hsiao, Ming-Syan Chen and P. S. Yu. On Parallel Execution of Multiple Pipelined Hash Joins. ACM-SIGMOD Int. Conf., Minneapolis, 1994, pp.185-196.
- [HoS91] W. Hong, M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. Proc. 1st PDIS Conf., Miami, Florida, Dec. 1991.
- [HL91] K.A. Hua and C. Lee. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. Proc. 17th VLDB Conf., Barcelona, Sep. 1991, pp.525-535.
- [HLY93] K.A. Hua, Y. Lo and H.C. Young. Including the Load Balancing Issue in Optimization of Multi-Way Join Queries for Shared-Nothing Database Computers. Proc. 2nd Intl. Conf. on Parallel and Distributed Information Systems, San Diego, Jan. 1993, pp.74-83.

- [IoKa91] Y. E. Ioannidis and Y. C. Kang. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. ACM SIGMOD Conf. Denver, 1991, pp. 168-177.
- [JTK89] B. C. Jenq, B. C. Twichell and T. Keller. Locking Performance in a Shared Nothing Parallel Database Machine. IEEE Trans. Knowledge and Data Eng., Vol. 1, No. 2. 1989, pp. 530-543.
- [KBZ86] R. Krishnamurthy, H. Borat, C. Zaniolo. Optimization of nonrecursive queries. Proc. 12th VLDB Conf., Kyoto, Japan, Aug. 1986. 128-137.
- [Kim80] Won Kim. A New way to compute the product and join of a relation. Proc. of SIGMOD, 1980. pp.179-187.
- [Kim89] Won Kim and Frederick H. Lochovsky. Object-Oriented Concepts. Databases and Applications. Addison-Wesley, 1989.
- [KNT89] M. Kitsuregawa, M. Nakayama and M. Takagi. The Effect of Bucket Size Tuning in Dynamic Hybrid GRACE Hash Join Method. Proc. 15th VLDB Conf., 1989, pp.257-266.
- [Kit90] M. Kitsuregawa, Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer. Proc. 16th VLDB Conf., Brisbane, 1990, pp.210-221.

- [LVZ93] R.S.G. Lancelotte, P. Valduriez, M. Zait. On the effectiveness of Optimization search strategies for parallel execution strategies. Proc. 19th VLDB Conf., Dublin, Ireland, Aug 1993, 493-504.
- [ME92] P. Mishra, M. H. Eich. Join Processing in Relational Databases. ACM Computing Surveys 24(1), March 1992.
- [OzVa91] M. T. Ozsü and P. Valduriez. Principles of Distributed Database Systems. Prentice-Hall, 1991.
- [PI96] V. Poosala and Y. Ioannidis. Estimation of Query-Result Distribution and its Application in Parallel-Join Load Balancing. Proc. 22nd VLDB Conference. Mumbai(Bombay), India, 1996.
- [Rich87] J.P. Richardson, H. Lu and K. Mikkilineni. Design and Evaluation of Parallel Pipelined Join Algorithms. ACM-SIGMOD Int. Conf., San Francisco, 1987, pp.399-409.
- [RM95] E. Rahm and R. Marek, Dynamic Multi-Resource Load Balancing in Parallel Database Systems. Proc. 21st VLDB Conf., Zurich, 1995; pages 395-406.
- [ScDe89] D. A. Schneider, D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. Proc. SIGMOD Conf., pp.110-121, 1989.

- [SD90] D. A. Schneider, D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. Intl. Conf. VLDB, Brisbane, 1990, pp.469-480.

- [Sel79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. Access Path Selection in a Relational Database Management System. Proc. ACM SIGMOD Conf. ACM, New York, 23. Reprinted in Readings in Database Systems, Morgan-Kaufman, San Mateo, Calif., 1988.

- [She93] E. J. Shekita, H. C. Young. Multi-Join Optimization for Symmetric Multiprocessor. Intl. Conf. VLDB, Dublin, Aug. 1993.

- [SOH96] M. Snir, S. W. Otto, S. Huss-Lederman, D. Walker and J. J. Dongarra, MPI: The Complete Reference, MIT Press, Cambridge, Mass., 1996.

- [STO75] Stone, Chen et al, 1975, 1980 Introduction to Computer Architecture Science Research Associates. 1975.

- [Stone86] M. Stonebraker. The case for Shared-Nothing. Database Eng. March 1986. Vol. 9, No. 1, pp. 4-9.

- [Stone88] M. Stonebraker. Readings in Database Systems. San Mateo, California. Morgan Kaufmann, 1988.

- [Swa89] A. Swami. Optimization of Large Join Queries: combining heuristics with combinatorial techniques. ACM-SIGMOD Int. Conf., 1989, pp.367-376.
- [Tera85] Teradata DBC/1012 Database Computer Systems Manual, Release 1.3. Teradata Corp., Document No. C10-0001-00, Feb. 1985.
- [TK78] D. C. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems. Inf. Syst. (1978), pp.173-191.
- [TPG88] The Tandem Performance Group. A Benchmark of Non-Stop SQL on the Debit Credit Transaction. Proc. ACM SIGMOD Conf., Chicago, Ill., June 1988.
- [U82] J. D. Ullman. Principles of Database Systems. Computer Science Press, Rockville, MD. 1982.
- [Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. Intl. J. Distributed and Parallel Databases, 1(2), 1993.
- [VG84] P. Valduriez and G. Gardarin. Join and SemiJoin Algorithms for a Multiprocessor Database Machine. ACM Trans. Database Syst. 9(1), Mar. 1984, pp.133-161.

- [WDJ91] C. B. Walton, A. G. Dale and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. Intl. Conf. VLDB, Barcelona, Sep 1991, pp.537-548.
- [WiA91] A. N. Wilshut, P. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. Proc. 1st Int. Conf. Parallel and Distributed Information Systems, Florida, Dec. 1991. pp.68-77.
- [Wil95] A. N. Wilshut, J. Flokstra and P. G. Apers. Parallel Evaluation of Multi-Join Queries. ACM-SIGMOD Int. Conf., San Jose, 1995, pp.115-126.
- [WYTD93] J.L. Wolf, P. S. Yu, J. Turek and D.M. Dias. A Parallel Hash Join Algorithm for Managing Data Skew. IEEE Trans. Parallel And Distrib. Sys. 4(12):1355-1371, Dec. 1993.
- [XD97] Y. Xu and S. P. Dandamudi. A Hierarchical Parallel Database System for Transaction Processing. Int. Conf. Computers and Their Applications, Tempe, Arizona. March 1997. (Expanded version available as technical report TR-97-2 from <http://scs.carleton.ca>.)
- [X88] B. Xie. Hierarchical Architecture for Parallel Query Processing on Networks of Workstations.
- [YCHL92] P. S. Yu, M.-S. Chen, H. Heiss and S. H. Lee. On Workload Characterization of Relational Database Environments. IEEE Trans. Software Engg., 18(4):347-355, April 1992.

- [Z98] S. Zeng. Centralized Architecture for Query Processing on Network of Workstations. M.C.S. Thesis, School of Computer Science, Carleton University, Canada, 1998.
- [ZZB93] M. Ziane, M. Zait and P. Borla-Salamat. Parallel Query Processing in DBS3. PDIS 1993.