EXECUTION OF DISTRIBUTED DATABASE QUERIES ON A HPC SYSTEM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İBRAHİM SEÇKİN ÖNDER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2010

Approval of the thesis:

EXECUTION OF DISTRIBUTED DATABASE QUERIES ON A HPC SYSTEM

submitted by **İBRAHİM SEÇKİN ÖNDER** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Müslim Bozyiğit
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Ahmet Coşar
Supervisor, **Computer Engineering Dept., METU** _____

Examining Committee Members:

Prof. Dr. Adnan Yazıcı
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Ahmet Coşar
Computer Engineering Dept., METU _____

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Dept., METU _____

Prof. Dr. Faruk Polat
Computer Engineering Dept., METU _____

Dr. Emrah Tomur
Information Management Dept., BDDK _____

Date: 27/01/2010

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: İBRAHİM SEÇKİN ÖNDER

Signature :

ABSTRACT

EXECUTION OF DISTRIBUTED DATABASE QUERIES
ON A HPC SYSTEM

Önder, İbrahim Seçkin
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Ahmet Coşar

January 2010, 81 pages

Increasing performance of computers and ability to connect computers with high speed communication networks make distributed databases systems an attractive research area. In this study, we evaluate communication and data processing capabilities of a HPC machine. We calculate accurate cost formulas for high volume data communication between processing nodes and experimentally measure sorting times. A left deep query plan executer has been implemented and experimentally used for executing plans generated by two different genetic algorithms for a distributed database environment using message passing paradigm to prove that a parallel system can provide scalable performance by increasing the number of nodes used for storing database relations and processing nodes. We compare the performance of plans generated by genetic algorithms with optimal plans generated by exhaustive search algorithm. Our results have verified that optimal plans are better than those of genetic algorithms, as expected.

Keywords: High performance computing, Computer clusters, Message passing interface, Distributed database, Query processing, Genetic algorithm.

ÖZ

HPC SİSTEMİ ÜZERİNDE DAĞITIK VERİ TABANI SORGULARININ ÇALIŞTIRILMASI

Önder, İbrahim Seçkin

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Doç. Dr. Ahmet Coşar

Ocak 2010, 81 sayfa

Bilgisayarların artan performansları ve bilgisayarları birbirine bağlayan yüksek hızlı ağlar, dağıtık veritabanları konusunda yapılan çalışmaları ilgi çekici kılmıştır. Bu tez çalışmasında, bir Yüksek Performanslı Hesaplama ortamının komünikasyon ve veri işleme yetenekleri ölçülmüştür. Program çalıştırılan düğümler arası yüksek hacimli veri iletimi için maliyet formülleri ile veri dizme süreleri deneysel olarak hesaplanmıştır. Paralel bir sistemin verilerin depolandığı ve işlendiği düğümlerdeki artışla beraber ölçeklenebilir bir performans artışı sağladığını teyit etmek amacıyla, bir tür sol derinlikli sorgu planı işleticisi geliştirilmiş ve geliştirilen bu sorgu işleticisi kullanılarak, iki farklı genetik algoritma tabanlı dağıtık veri tabanı sorgu eniyileyicisinin ürettiği planlar mesaj aktarma yöntemiyle çalıştırılmıştır. Genetik algoritma tabanlı eniyileyicilerin ürettiği planların performansı, tam kapsamlı arama algoritması ile çalışan eniyileyicinin ürettiği optimal planlarınki ile karşılaştırılmıştır. Elde ettiğimiz sonuçlar, beklendiği üzere, optimal planların genetik algoritmaların ürettiği planlardan daha iyi olduğunu göstermiştir.

Anahtar Kelimeler: Yüksek performanslı bilgi işleme, Bilgisayar kümeleri, MPI, Dağıtık veritabanı, Sorgu işleme, Genetik algoritma.

To My Family

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor Assoc.Prof. Dr. Ahmet Coşar for their guidance, advice, criticism, encouragements and insight throughout the research.

I would also like to thank my wife Gökçen for her encouragements and moral support.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Research on distributed databases has been going on since 70s to meet the information needs of business organizations typically with sites that have geographically separated computer systems connected via some communication network. Increasing performance of computers and ability to connect computers with high speed LAN/WAN networks, at very affordable costs, make distributed database systems an attractive research area. The change in computing hardware and communication networks has changed the parameters to be used making optimization decisions in designing and distributed databases and executing distributed queries. These new factors result in a more complex distributed database query optimization problem which has a great potential making it an important research problem in 2000s.

A distributed query optimization algorithm determines how and in what order the relations referenced in a SQL query are processed. In other words, a query optimization algorithm locates and selects the input relations, determines evaluation order of query expressions and where (at which node) the processing should be done. For this optimization task, the number and locations of relation fragments and replicas, network transfer costs and whether semijoin operators can be employed to reduce network communication costs, must be calculated and considered. While

doing this, query optimizer enumerates alternative plans, estimates the cost of every plan using a cost model and chooses the query plan with the lowest estimated cost.

Most of the modern commercial query optimizers are based on dynamic-programming algorithm as given in [Selinger (1979)]. While this algorithm produces good optimization results, its high computational complexity makes it inappropriate for complex queries. Query optimization of distributed database systems differ from the centralized ones because of increased problem size with a larger number of input parameters such as horizontal and vertical fragmentation of relations, replication of relations to nodes over a LAN/WAN, and network communication path selection, and network transmission and forwarding delays. So, centralized database query optimization techniques like dynamic programming become much more expensive for distributed database systems and we need to use more advanced join tree generation algorithms for dynamic programming as defined in [Moerkotte (2006)].

The core component of a query optimizer is its enumeration algorithm that determines which plans to consider. In fact, there is an inverse relation between the complexity of an enumeration algorithm and the quality of the plans generated by the algorithm. Dynamic programming runs in exponential time but generates optimal plans. Other algorithms have lover complexity than dynamic programming but they are not as successful as dynamic programming in generating the lowest cost optimal plans. Kossmann and Stocker evaluate the most important algorithms for distributed database systems in [Kossmann (2000)].

Genetic algorithm based optimization techniques have been developed since 90s and present a promising low cost alternative optimization method. Genetic algorithms have been successfully used for many difficult optimization problems. In this thesis, plans generated by a genetic algorithm based distributed database query optimizer are executed on a HPC system using message passing paradigm. The aim is to prove that a parallel system will have scalable performance by increasing the number of nodes used for storing database relations and processing nodes, and the system

performance will scale as close to linear as possible with a good GA based query optimizer.

The organization of the thesis is as follows. In the first chapter, definition of the problem addressed by this thesis is given and explained briefly. In Chapter 2, background information is provided about database management systems, relational data model, parallel computing, cluster machines, message passing interface (MPI), distributed databases, query processing, query optimization and genetic algorithms. Since message passing libraries are used, program development using MPI libraries and running a sample program on a HPC system is explained in detail in this section. In Chapter 3, design of the developed system executing distributed database queries on a HPC system is explained. In this part, algorithm of the programs developed and structure of the execution plan and relation files are explained. In Chapter 4, detailed clarification of experimental setup and the results of performed experiments are given. In this part, input relations used for our experiments are explained first. After that, plans are executed on our HPC machine and the obtained performance results are interpreted with charts and summarized. Some very accurate cost formulas have also been determined for transfer of relations between HPC nodes. In Chapter 5, results found by executing plans, influences of changing join orders and changing number of nodes on the results are discussed. Experimental results are also evaluated in terms of the consistency with the formulas given in the previous section. All scientific work and results of our experiments are compiled and obtained results are interpreted and whether the goals defined at the beginning are achieved is discussed in the conclusions.

CHAPTER 2

BACKGROUND

2.1. DBMS

A *database* is a structured collection of data records that are stored and processed in a computer system. A database management system (DBMS) is a software that is designed to control the organization, storage, management, and retrieval of data stored in a database [Ramakrishnan (2002)].

Without using a database management software, by directly working on standard operating system files and applications developed to work with the file structure implementations directly, it is difficult to meet the needs for shared and efficient access to data. These needs can be high performance when retrieving data, use of the same data by several users concurrently, protecting consistency of data which is changed by several different users simultaneously, restriction for specific user(s) to access only a certain part of data and restoring the database to a consistent state if the system (or user program) crashes while changes are being made. Using a DBMS instead of storing data in a collection of operating system files brings many additional advantages [Ramakrishnan (2002)]:

- Data independence: Application programs should be free from the details of data representation and storage. The DBMS can relieve application code

from such details by providing an abstract view of data and a standard based applications interface.

- Efficient data access: DBMS provides different kinds of complicated techniques (e.g. hash files and B-trees) to provide efficient read and write access to stored data.

- Query ability: Querying is the process of requesting stored information from various perspectives and combinations of constraints. A database query language allows users to interactively interrogate the database, analyze its data and update it according to the users' access privileges on data (also maintained by the DBMS itself).

- Data integrity: Often one wants to apply rules to information inserted or updated through the DBMS engine so that the information stored in a database are free from errors and reliable, satisfying all of the constraints defined by the database creator. DBMS can do this by enforcing integrity constraints on the data.

- Security: Often it is desirable to limit who can see or change which tables or groups of attributes. This may be managed directly by assigning rights to individual users themselves, or by the assignment of individuals and privileges to groups (who are assigned access rights), or through the assignment of individuals and groups to roles which are then granted priviliges.

- Data administration: When several users share the data, centralizing the administration of data can provide significant improvements by tunings made by experienced professionals who understand the nature and use of data.

- Concurrent access and crash recovery: A DBMS schedules concurrent access to the data seamlessly to the users that need to use the same data. A DBMS also restores the data to a consistent state if the system crashes while changes are being made.

- Reduced application development time: DBMS supports common functions to applications that need to access data stored in DBMS. This facilitates quick development of applications.

- Computation: There are common computations requested on attributes such as counting, summing, averaging, sorting, grouping, etc. Rather than having each computer application implement these facilities, they can rely on the DBMS to supply such calculations

"*A data model is a collection of high-level data description constructs that hide many low level storage details. A DBMS allows a user to define the data to be stored in terms of data model."* [Ramakrishnan (2002)]

In this thesis, we focused on the relational data model because of its wide use and importance. Other important data models include the earlier hierarchical and network models, and newer object-oriented and object-relational models.

A Relational Database Management System (RDBMS) implements the features of the relational model outlined in the next section. Most popular commercial and open source databases currently in use are based on the relational model. A short definition of an RDBMS may be a DBMS in which data is stored in the form of tables and the relationship among the data is also stored in the form of tables.

2.2 Relational Model

The *relational model* for databases is a data model, first formulated and proposed in 1969 by E.F. Codd [Codd (1969)].

The purpose of relational model is to describe data with its natural structure only, in other words, without superimposing any additional structure for machine representation purposes.

The central data description construct in this model is a relation, which can be thought of as a set of records [Ramakrishnan (2002)]. The fundamental assumption of the relational model is that all data are represented as mathematical *n-ary relations*, an n-ary relation being a subset of the Cartesian product of n domains.

"*The term relation is used here in its accepted mathematical sense. Given sets S1, S2,…, Sn (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples, each of which has its first elements from S1, its second element from S2, and so on. We shall refer to Sj as the jth domain of R and R is said to have degree n. Relation of degree n is called n-ary.*" [Codd (1970)]

In this definition, a *relation* is defined as a set of *n-tuples*. In both mathematics and the relational database model, a set is an unordered collection of items, although some DBMSs impose an order to their data. In mathematics, a tuple has an order, and allows for duplication [wikipedia (2009)].

E.F. Codd originally defined tuples using this mathematical definition, but he also uses the domain and roles to label columns instead of using their orders [Codd (1970)]. Today, attribute names are used instead of an ordering which is much more convenient in a computer language based on relations. Though the concept has changed, the name "tuple" is still in use.

A relation consists of a relational schema and a relational instance. A description of data in terms of data model is called a *schema*. In the relational model, the schema for a relation specifies relation's name, the name of each column (or attribute) and the domain of each field [Ramakrishnan (2002)].

Example (a sample relation schema): student(name: string, department: string, year: integer, gpa: real)

Data are operated upon by means of a relational calculus or relational algebra which are formal query languages specialized for querying relational data [Ramakrishnan (2002)].

The relational model of data permits the database designer to create a consistent, logical representation of information. Consistency is achieved by including declared *constraints* in the database design. The theory includes a process of database normalization by which a design with certain desirable properties can be selected from a set of logically equivalent alternatives.

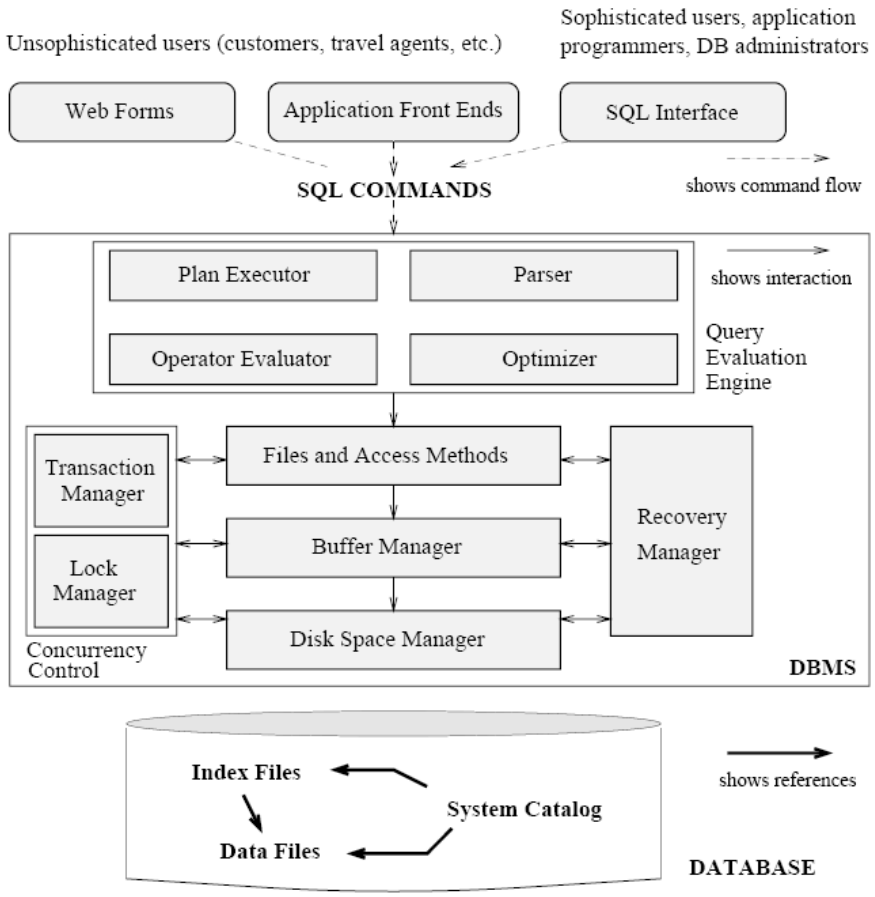


Figure 2.1: Architecture of a DBMS [Ramakrishnan (2002)]

In Figure 2.1, structure of a typical DBMS based on Relational Model is given. As shown in the figure, Issued queries are parsed by *Parser* and processed by *Query Optimizer* to produce an efficient *Execution Plan*. Query execution plans, other implementation and operation details, such as a sort-merge or nested-loops join method will be used, are handled by the *Query Evaluation Engine*. Concurrency control and recovery are provided by *Transaction Manager*, *Lock Manager* and *Recovery Manager*. Transaction manager schedules the execution transactions, lock manager handles the lock requests and recovery manager restores the system to a consistent state after a crash [Ramakrishnan (2002)].

2.3 Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously (preferably on more than one CPUs/Computers to increase performance), operating on the principle that large problems can often be divided into smaller ones, which are then solved on multiple CPUs and/or computers in parallel [Almasi (1989)].

There are several different forms of parallel computing: bit-level parallelism, instruction-level parallelism, data level parallelism and task level parallelism. Bit-level parallelism is done by increasing the word size which is the amount of information the processor can execute per cycle. Instruction-level parallelism is done by re-ordering of the program so that the program can be combined into groups which are then executed in parallel without changing the result of the program. Data-level parallelism means distributing data across different computing nodes to be processed in parallel. Task parallelism is the characteristic of a parallel program that entirely different calculations can be performed on either the same or different sets of data [Culler (1999)].

Parallel programming and distributed programming are two basic approaches for achieving concurrency with a piece of software. Parallel programming techniques assign the work a program has to do to two or more processors within a single physical or a single virtual computer. Distributed programming techniques assign the work a program has to do to two or more processes where the processes may or may not exist on the same computer. That is, the parts of a distributed program often run on different computers connected by a network or at least in different processes.

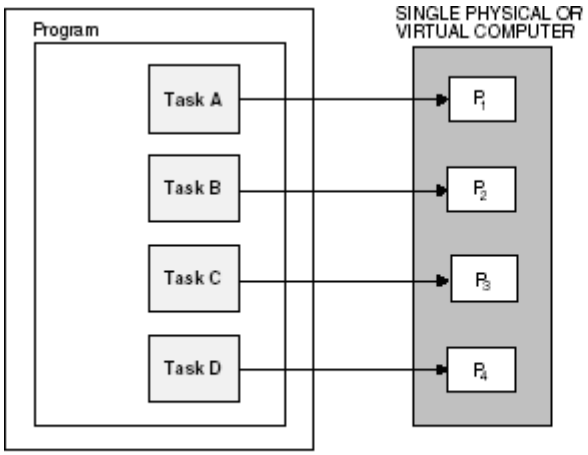


Figure 2.2: Typical architecture for a parallel program [Hughes (2003)]

The PRAM (Parallel Random Access Machine) is a simplified theoretical model where there are n processors labeled as P1, P2, P3, ... Pn and each processor shares one global memory. This basic shared memory system is given in Figure 2.3. All the processors have read and write access to a shared global memory.

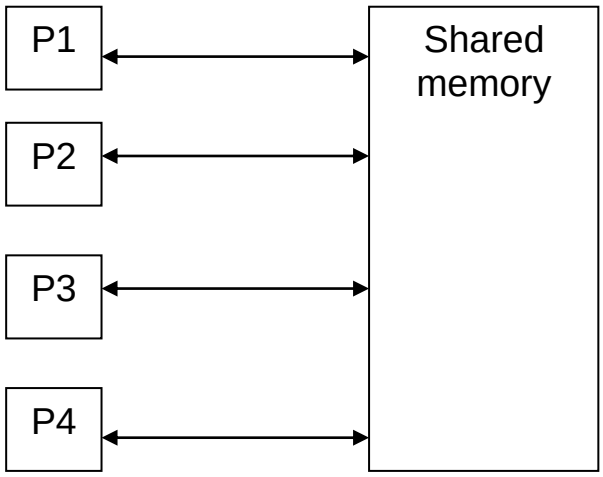


Figure 2.3: Shared memory

The read/write conflicts in accessing the same shared memory location simultaneously are resolved by one of the following strategies:

1. Exclusive Read Exclusive Write (EREW): Every memory cell can be read or written to by only one processor at a time.

2. Concurrent Read Exclusive Write (CREW): Multiple processors can read a memory cell but only one can write at a time.

3. Exclusive Read Concurrent Write (ERCW): Multiple processors can write into the same memory cell but read access remains exclusive.

4. Concurrent Read Concurrent Write (CRCW): Both multiple read and multiple write privileges are allowed.

A simplified scheme for classifying the parallel computers was introduced by M.J. Flynn [Flynn (1966)]. These were SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) which are later extended to SPMD (Single Program Multiple Data) and MPMD (Multiple Program Multiple Data). The SPMD (SIMD) scheme allows multiple processors to execute the same instruction or program with each processor accessing different data. The MPMD (MIMD) scheme allows for multiple processors with each executing different programs or instructions and each with its own data.

Parallel computers can also be roughly classified according to the level at which the hardware supports parallelism. While multi-core and multi processor computers are single machine that have multiple processing elements, grids and clusters use multiple computers to do the same task.

The most common environments for parallel and distributed programming are clusters, MPPs, and SMP computers. *Clusters* are collections of two or more computers that are networked together to provide a single, logical system which appears to the application as a single virtual computer. *MPP* (Massively Parallel Processors) is a single computer that has hundreds of processors. *SMP* (Symmetric Multiprocessing) is a single system that has processors that are tightly coupled

where the processors share memory and the data path. SMP processors share the resources and are all controlled by a single operating system [Hughes (2003)].

Since our work is performed on a cluster computer, the next section is focused on cluster systems.

2.3.1 Cluster

A *cluster* is a group of loosely coupled computers (each with its own RAM, CPU, hard disk, and O/S) that work together closely, so that in many aspects they can be thought as a single computer [Bader (2001)]. In another definition, a cluster is a type of parallel or distributed computer system, which consists of a collection of inter-connected (with a high-bandwidth network switch) stand-alone computers working together as a single integrated computing resource [Pfister (1998)].

Clusters are typically used for "high availability" (HA) to achieve greater reliability or High Performance Computing (HPC) to provide greater computational power than a single computer can provide.

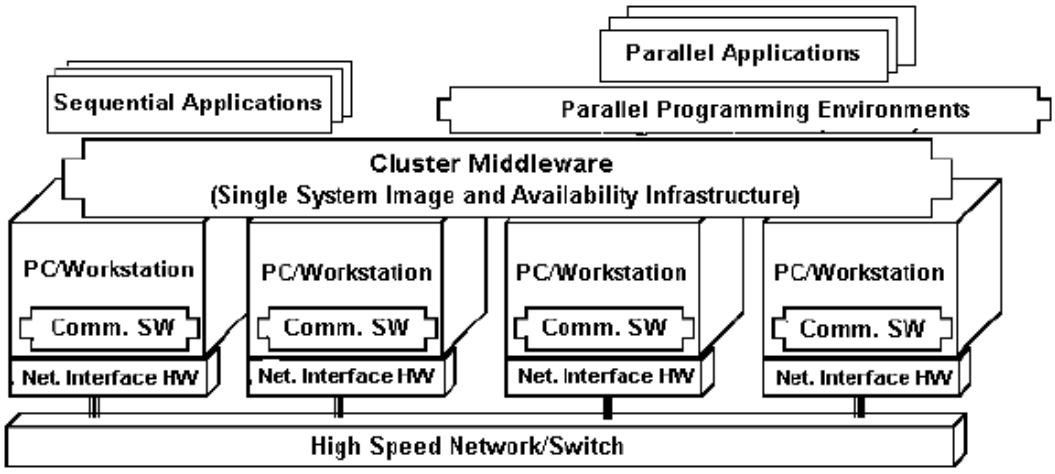


Figure 2.4 : Sample cluster architecture [Buyya (1999)]

The typical architecture of a cluster computer is shown in Figure 2.4. The key components of a cluster include, multiple standalone computers (PCs, Workstations), an operating system, a high performance interconnect, communication software, middleware and applications.

Interconnection technologies may be classified into four categories, depending on whether the internal connection is from the I/O bus or memory bus, and depending on whether the communication between the computers is performed primarily using messages or using shared storage. Of the four categories, I/O attached message-based systems are by far the most common [Baker (2000)].

Concurrent programming languages, libraries, APIs, and parallel programming models have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architectures: shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing between computers. POSIX Threads and OpenMP are two of the most widely used shared memory APIs, whereas Message Passing Interface (MPI) is the most widely used message-passing system API.

2.3.2 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a specification for message passing libraries, designed to be a standard for distributed memory, message passing parallel computing [mhpcc (2009)]. The aim of the MPI is to provide a widely used standard for writing message-passing programs. MPI established a practical, portable, efficient, and flexible standard for message passing. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one processor to that of another process through cooperative send() and receive() operations on each processor [mpi-forum (2008)].

To understand MPI, we have to mention about some basic concepts: [mhpcc (2009)]

- **Distributed Memory:** Every processor has its own local memory which can be accessed directly only by its own CPU. Transfer of data from one processor to another is performed over a network. Therefore, distributed memory systems differ from shared memory systems which permit multiple processors to directly access the same memory resource via a memory bus.

- **Message Passing:** By message passing, data from one processor's memory is copied to the memory of another processor. In distributed memory systems, data is generally sent as packets of information over a network from one processor to another.

- **Process:** A process is a program running on a processor. In a message passing system, all processes, even if they are running on the same processor, communicate with each other by sending messages. For reasons of efficiency, although more than one process may execute on a processor, message passing systems generally associate only one process per processor.

- **Message Passing Library:** Refers to a collection of routines which provide send(), receive() and other message passing operations in application code.

- **Send / Receive:** Transfer of data from one process to another is done by Send and Receive operations. Message passing requires the cooperation of both the sending and receiving process. In Send operation, the sending process specifies the data's location, size, type and the destination. Each Receive operation should match a corresponding send operation.

- **Synchronous / Asynchronous:** A synchronous send operation will complete only after a short acknowledgement message arrives at the sender side that the message was safely received by the receiving process. Asynchronous

send operations may complete even though the receiving process has not (yet) actually received the message.

- **Application Buffer:** The address space that holds the data which is to be sent or received. For example, the application buffer for a variable is the program memory location where the value of that variable resides.

- **System Buffer:** System space for storing messages allowing communication to be asynchronous. Depending upon the type of send/ receive operation, data in the application buffer may be required to be copied to/from system buffer space.

- **Blocking Communication:** A communication routine is blocking if the completion of the call is dependent on certain "events". For sends, the data must be successfully sent or safely copied to system buffer space so that the application buffer that contained the data is available for reuse. For receives, the data must be safely stored in the receive buffer so that it is ready for use.

- **Non-blocking Communication:** A communication routine is non-blocking if the call returns without waiting for any communications events to complete (such as copying of message from user memory to system memory or arrival of message). Non-blocking communications are primarily used to overlap computation with communication to effect performance gains.

- **Communicators and Groups:** Communicators and groups are objects used to define which collection of processes may communicate with each other. Most of the MPI routines require a communicator as an argument. We use “MPI_COMM_WORLD”, which is the predefined communicator including all of our MPI processes, whenever a communicator is required.

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

Figure 2.5: Sample C program using MPI

A sample C program using MPI library as message passing system API is given in Figure 2.5. When we run the sample program in two nodes, both nodes execute the same program but each one will have different behavior depending on their rank which is obtained by “MPI_comm_rank()” command. In the sample program, if the rank is equal to zero, which means the program is running on the node with rank zero, it sends one character message to the node with rank one and waits to receive a one character reply message from the same node. If the rank is equal to one, which means program is running on the node with rank one, it waits to receive a one character message from the node with rank zero and then it sends a one character message to node with rank zero. Send and receive operations are done by “MPI_Send()” and “MPI_Receive()” commands.

Before we can discuss how we can compile and run this sample program on HPC system, first of all we have to compile our program to generate an executable:

“mpicc sample.c –o sample”

Next step is preparing our Portable Batch System (PBS) [sissa (2009)] script file which specifies a list of resources required for its execution and sending our job to queue by command:

“qsub scriptfile”

The Portable Batch System (PBS) is the software that performs batch job and computer system resource management. It accepts batch jobs (shell scripts with control attributes), preserves and protects the job until it is run, runs the job, and delivers output back to the submitter [sissa (2009)]. METU HPC System uses the *Torque* scheduler and *Maui* to handle job submissions to the cluster as its PBS.

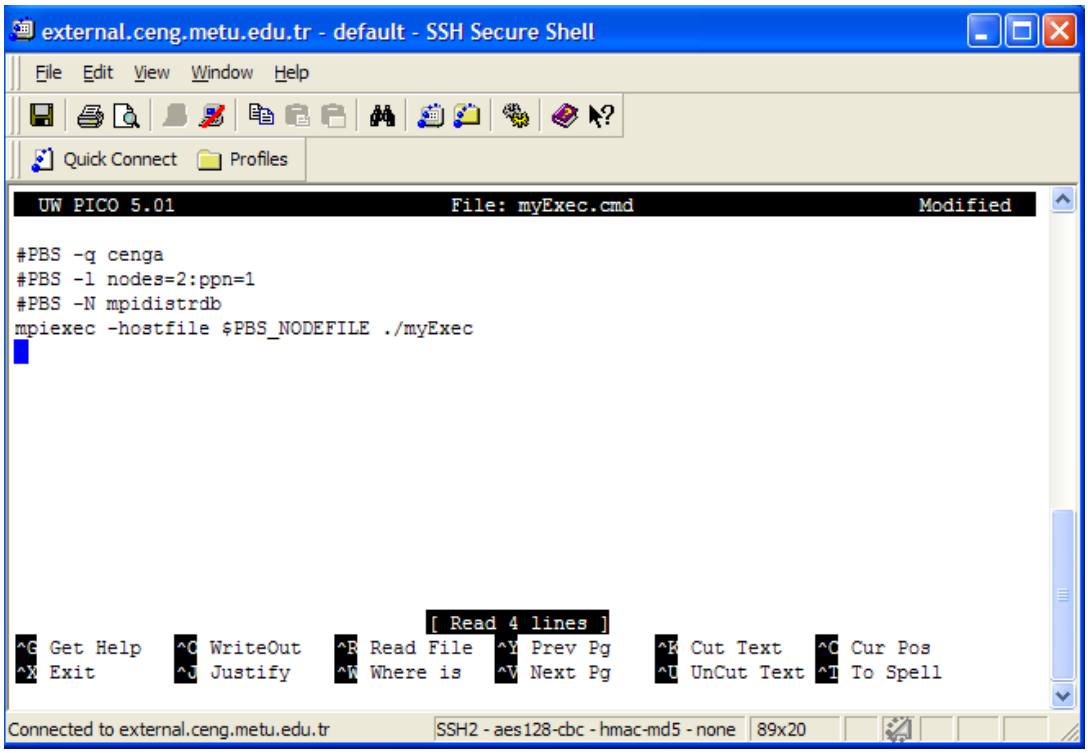


Figure 2.6: Sample PBS script file

Sample shell script for running our sample program is shown in Figure 2.6. The first line specifies the name of the queue that will be used. Second line indicates that our code will run on two nodes with one CPU each. Third line gives the name of our job which is “mpidistrdb” in our sample [ufl (2009)]. Finally, configuration is executed by “mpiexec” [die (2009)] command, which executes serial and parallel jobs in Open MPI [open-mpi (2009)]. METU HPC System has OpenMPI library, which is open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners.

We can submit our job by command “qsub scriptfilename” [sissa (2009)]. After submitting job to queue, we can check the status of out job by command “qstat” [ufl (2009)]. Sample execution of “qsub” and “qstat” commands are shown in Figure 2.7.

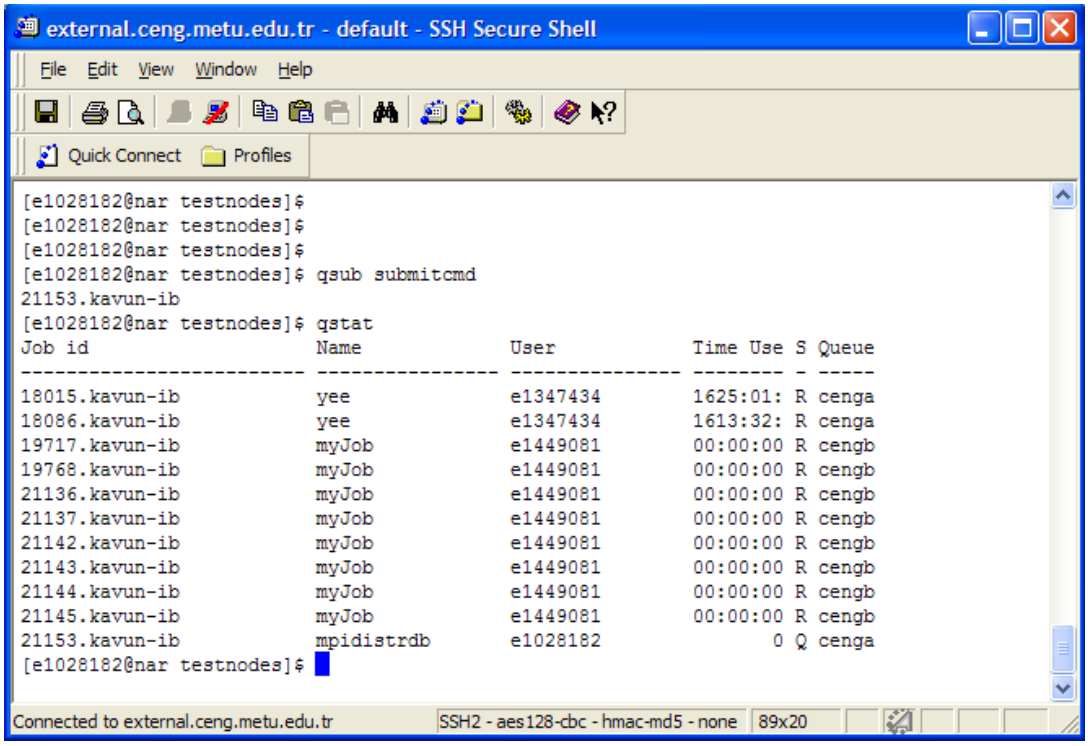


Figure 2.7: Submitting a job to a queue.

When the execution completes, system produces a file for error log and another file for output. Since our simple example doesn't produce any output, this simple run produces only two empty files as output.

Complete software and hardware configurations of METU Computer Engineering Department HPC System, which is our experiment environment, are given in Appendix-A and Appendix-B [ceng (2009)].

2.4 Distributed Databases

2.4.1 Distributed Database System

A *distributed database* (DDB) [Özsu (1999)] is a data collection which resides on more than one machine with computational power connected via a communication network. *Distributed database management system* (distributed DBMS) is the software system that permits the management of the DDB and makes the distribution transparent to the users. It lets a relational database to be distributed among the sites of a computer network while providing a relational database to be accessed as if it were stored in a single site. Users of the distributed database management system feel they work on the entire database and they have the opportunity to declare what they want.

In a distributed database management system, data is stored at a number of sites in a computer network. Each site is assumed to logically consist of a single processor, with all resources included in a single system, it's not a consideration of distributed DBMS to manage the parallelism in these sites. The processors of these sites are interconnected with a computer network rather than a multiprocessor network. Sites have their own operating systems and operate independently so the processors are loosely interconnected.

Distributed DBMS is a fully functional DBMS. It's not only a distributed file system or a transaction processing system.

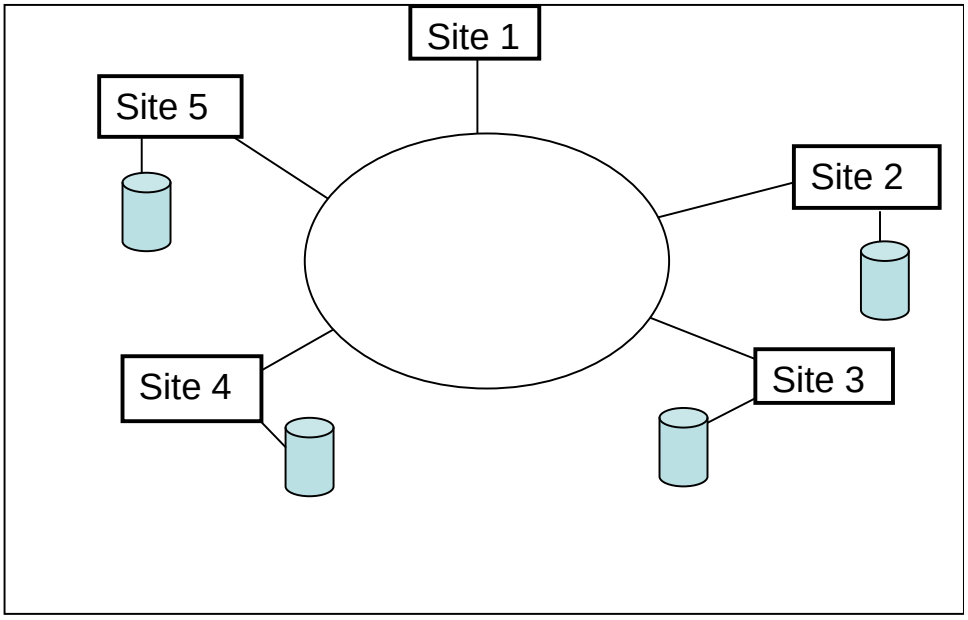


Figure 2.8: Distributed database environment [Özsu (1993)]

In Figure 2.8, we can see the architecture of a Distributed DBMS environment in which a database is distributed across the sites such that each site typically manages a single local database. Today, majority of the existing distributed systems are built on top of local area networks in which each site is usually a single computer.

Distributed DBMS supports data fragmentation, where a relation in the database can be divided into pieces called *fragments* for physical storage. There are various ways to splitting a relation to fragments. In *horizontal splitting*, grouping is done on complete tuple. In *vertical splitting*, grouping is done on attribute values of all tuples. The database is called *partitioned*, if the resulting fragments of these horizontal and/or vertical splits are placed at different sites. The database is called *replicated*, if copies of relations and/or fragments are placed at different sites [Apers (1988)].

2.4.2 Query Processing in a Distributed Database System

The distribution of data in a network or distributed computer system suggests several advantages over the centralizing of the data at a single computer. These advantages include increased data reliability (one of the copies being available if a

relation is replicated), faster access of data and upward scaling of data capacity. The problem is the efficient processing of queries in a distributed system.

In general, satisfying a user request in a distributed database involves five major steps [March (1995)]:

1. Determining where the needed data is stored,
2. Determining an access strategy that specifies which copy of the data to access, where the data will be processed, and how it will be routed,
3. Sending request messages to the appropriate nodes and establishing locks if necessary,
4. Accessing and processing of data at each related node and,
5. Routing the response to the requesting node for final processing.

While a distributed system benefits by storing, processing and transferring data in parallel at separate sites in the network, the necessary transmission of data between these sites results in considerable time delays. As a consequence, the DBMS must derive effective solutions to arrange local data processing and data transmission while processing distributed queries. This arrangement of data processing and data transmission is known as distribution strategy for a query [Hevner (1979)].

2.5 Query Optimization

Problem of determining good evaluation strategies for join expressions has been addressed as early as 1979 with the development of System R [Selinger (1979)]. The work in this area can be divided into two major streams: First, the development of efficient algorithms for performing the join itself, and second, the algorithms that determine the nesting order in which the joins are to be performed.

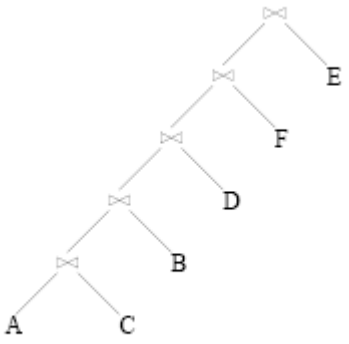
The input of the optimization problem is given as the *query graph*. The query graph consists of all the relations that are to be joined as its nodes. Edges indicate the joins

and can be labeled with the *join selectivity* which denotes the ratio of the included tuples to the total tuples [Steinbrunn (1993)].

The *search space* or *solution space* is the set of all possible evaluation plans that generates the same result. A *point* in the solution space is a particular solution for the problem. A solution is described by the *processing tree* for evaluating the join expression. The processing tree, which we also will use to visualize in the next section to show the join orders of the execution plans, is a binary tree. Its leaves correspond to base relations as and inner nodes correspond to join operations. Edges indicate the flow of partial results from the leaves to the root of the tree. Each evaluation plan (point of the solution space) has a *cost*. The aim of the optimization is to find the evaluation plan with the lowest possible cost [Steinbrunn (1993)].

Generally, the solution space is defined as the set of all processing trees that compute the result of the join expression and that contain each base relation exactly once. While leaves of the processing trees consist of the base relations, inner nodes match joined results of their child nodes. If inner relation of each join is a base relation, this type of processing tree is called *left-deep tree*. There are n! ways to allocate n base relations to the tree's leaves for these type of processing trees. If there is no restriction about the processing tree, it is called *general* or *bushy tree*. Examples of a left-deep tree and a bushy tree for a query with six relations is shown in Figure 2.9.

(a)



(b)

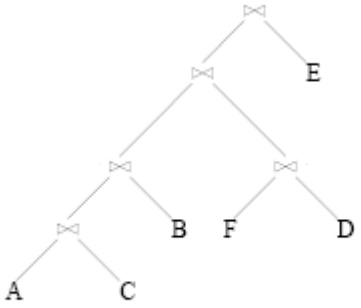


Figure 2.9: (a) Left deep tree; (b) Bushy tree [Bennett (1991)]

2.6 Join Algorithms

In this section, fundamental join algorithms will be explained with their cost formulas. In the formulas to use, we assume that we have two relations namely R and S to join and there are M pages in R and N pages in S. Since Sort merge Join algorithm is used in our implementation, this algorithm is given more detailed below.

We can categorize the join algorithms in three main headings:

1. **Nested Loop Join:** In this algorithm, every tuple of the outer relation is checked against the every tuple of the inner relation. Algorithm can be given as in Figure 2.10.

```
foreach tuple r ∈ R do
    foreach tuple s ∈ S do
        if r_i==s_j then add ⟨r, s⟩ to result
```

Figure 2.10: Simple nested loop join [Ramakrishnan (2002)]

By page at a time refinement, in which for each page of R we retrieve each page of S, cost can be calculated as M + M*N.

2. **Sort Merge Join:** In sort merge join algorithm, both relations are sorted on the join attribute and qualifying tuples are merged. In merge phase, two scans start at first tuple in each relation. Scan of R is advanced while the join attribute in the current R tuple is less than the one in the current S tuple. Similarly, scan of the S tuple is advanced as long as the join attribute in the current S tuple is less than the one in the current R tuple. When tuples are found that Tri = Tsi, the join is outputted. There may be several R and S tuples with the same value in join attributes, which are called as R and S partition. For each tupple r in current R partition, all tuples s in the current S partition is scanned and joined tuple <s,j> is outputted. In figure 2.11, algorithm is given.

```
proc smjoin(R, S, `R_i = S'_j)

if R not sorted on attribute i, sort it;
if S not sorted on attribute j, sort it;

Tr = first tuple in R;                                  // ranges over R
Ts = first tuple in S;                                  // ranges over S
Gs = first tuple in S;                                  // start of current S-partition

while Tr ≠ eof and Gs ≠ eof do {

    while Tr_i < Gs_j do
        Tr = next tuple in R after Tr;                  // continue scan of R

    while Tr_i > Gs_j do
        Gs = next tuple in S after Gs                   // continue scan of S

    Ts = Gs;                                            // Needed in case Tr_i ≠ Gs_j
    while Tr_i == Gs_j do {                             // process current R partition
        Ts = Gs;                                        // reset S partition scan
        while Ts_j == Tr_i do {                         // process current R tuple
            add ⟨Tr, Ts⟩ to result;                     // output joined tuples
            Ts = next tuple in S after Ts;}             // advance S partition scan
        Tr = next tuple in R after Tr;                  // advance scan of R
        }                                               // done with current R partition

    Gs = Ts;                                            // initialize search for next S partition

    }
```

Figure 2.11: Sort merge join [Ramakrishnan (2002)]

Cost of sort merge join is O(M + N + M logM + N logN).

3. **Hash Join:** In hash join algorithm, hashing is used to identify partitions. Both relations are hashed on the join attribute using the same hash function. Let assume that we hash each relations into k partitions, R tuples in partition i will join only with S tuples in partition i. In Figure 2.12, algorithm is given.

```
// Partition R into k partitions
foreach tuple r ∈ R do
    read r and add it to buffer page h(r_i);          // flushed as page fills

// Partition S into k partitions
foreach tuple s ∈ S do
    read s and add it to buffer page h(s_j);          // flushed as page fills

// Probing Phase
for l = 1, ..., k do {

    // Build in-memory hash table for R_l, using h2
    foreach tuple r ∈ partition R_l do
        read r and insert into hash table using h2(r_i) ;

    // Scan S_l and probe for matching R_l tuples
    foreach tuple s ∈ partition S_l do {
        read s and probe table using h2(s_j);
        for matching R tuples r, output ⟨r, s⟩ };

    clear hash table to prepare for next partition;
    }
```

Figure 2.12: Hash join [Ramakrishnan (2002)]

2.7 Join Ordering

2.7.1 Optimization Algorithms

So as to join ordering strategies, there are many algorithms developed for query optimization in database systems. Optimization algorithms generating an optimal execution plan can be classified in three main group and all algorithms fall into one of these classes or are combination of these fundamental algorithms [Kossmann (2000)]:

1. **Exhaustive search:** Algorithms that fall into this group are guaranteed to find the best plan, according to the optimizer's cost model. These algorithms have exponential time and space complexity. The most important example of this class of algorithms is *(bottom up) dynamic programming* [Selinger (1979)] which is currently used by most database systems.

2. **Heuristics:** A heuristic algorithm is an algorithm that is able to produce an acceptable solution to a problem in many practical scenarios, but for which there is no formal proof of its optimality. Algorithms that fall into this group have polynomial time and space complexity, but may produce more expensive plans than plans generated by exhaustive search. Greedy algorithms are good examples for this group.

3. **Randomized algorithms:** Randomized algorithms usually perform *random walks* in the state space via a series of moves. A state is a *global minimum* if it has the lowest cost among all states. *Simulated Annealing* [Kirkpatrick (1983)] and *Iterative Improvement* [Nahar (1986)] are the best known randomized algorithms. In simulated annealing, the aim is to find an acceptably good solution in a fixed amount of time, rather than the best possible solution. In Iterative Improvement local optimizations are repeated until a *stopping condition* is met. The best known randomized algorithm is called *Two Phase Optimization* which is a combination of these two algorithms [Ioannidis (1990)]. Since these algorithms are indeterministic, execution time for most of these Randomized algorithms can not be predicted. For simple queries, randomized algorithms are slower than heuristics and dynamic programming, however faster than both algorithms in very large queries [Kossmann (2000)].

The development of genetic algorithm (GA) based optimization techniques in 1990s presents a promising alternative methodology.

Genetic algorithms (GA) are powerful stochastic search and optimization methods based on the concept of adaptation in natural organisms.

In general, a genetic algorithm has five components [Gen (2000)]:

1. A genetic representation of solutions to the problem.
2. A way to create an initial population of solutions.
3. An evaluation function rating solutions in terms of their fitness.

4. Genetic operators that alter the genetic composition of children during reproduction.

5. Values for the parameters of genetic algorithms.

GA applies principles of natural selection to a randomly generated pool of genetic populations consisting of chromosomes each representing a complete solution to the problem and using these initial solutions tries to evolve better solutions to the problem. Each chromosome in the population is calculated an associated fitness value to choose competitive chromosomes that will form the next generation. There are two types of transformation: *mutation*, which create new individuals by making changes in a single individual, and *crossover*, which creates new individuals by combining pairs from two individuals.

Genetic algorithms for join query optimization have been studied in [Bennett (1991)] and [Steinbrunn (1993)]. They transform QEPs into chromosomes, on which crossover, selection and mutation are applied.

2.7.2 A New Genetic Algorithm [Sevinç (2009)]

2.7.2.1 Formulation

The goal in this work is to develop a genetic algorithm based heuristic for the optimization of distributed queries. A *New Genetic Algorithm (**NGA**)* is presented by evaluating its performance compared to an existing GA algorithm. A total of three algorithms will be discussed in order to show that NGA has a better performance when compared to others.

In order to see how close are the GA generated solutions to the optimum solutions, an Exhaustive Search Algorithm (ESA) is implemented which takes very long time to return a plan but makes it possible to evaluate performance of the GA algorithms. Another technique to decide whether a given GA algorithm is good a second algorithm is implemented that randomly generates an equal number of random

solutions. If a given GA algorithm shows no (or very little) improvement compared to the completely random algorithm, then we can that the proposed mutation and crossover operators for the GA make no positive contribution to the search process. This algorithm is called as "***Random***".

As mentioned before there is already a GA based algorithm proposed in [Sangkyu (1997)]. We will call it *Rho's Genetic Algorithm* (**GA**) throughout this study. GA has a comprehensive query optimization model that, integrates copy identification, join order, join site selection, and reduction by semijoins into a single model. It exploits the concepts of gainful semijoins and pure join attributes. It considers both network communication and local processing costs. Sites and communication links can be heterogeneous in terms of unit costs and capacities.

The last algorithm is our GA based algorithm with new mutation and crossover operators (**NGA**). We also use a greedy algorithm that improves a given plan by selecting copies of replicated relations at the nearest site.

2.7.2.2 Chromosome Structure

All possible query execution plans will be represented using a chromosome structure. This representation is the same as the one used in GA. The chromosome has *n* genes each one for a join condition given in the query. The gene order says in which order joins are evaluated and at which node. Execution starts with G_1 on the left-hand side and finishes with the last Gene, G_n seen on the right-hand side.

N shows the number of irreducible sub-queries in the query. In all our examples, the queries are assumed to contain such joins. In other words, queries will not be tried to be optimized.

The chromosome structure of a query is shown in Figure 2.13.

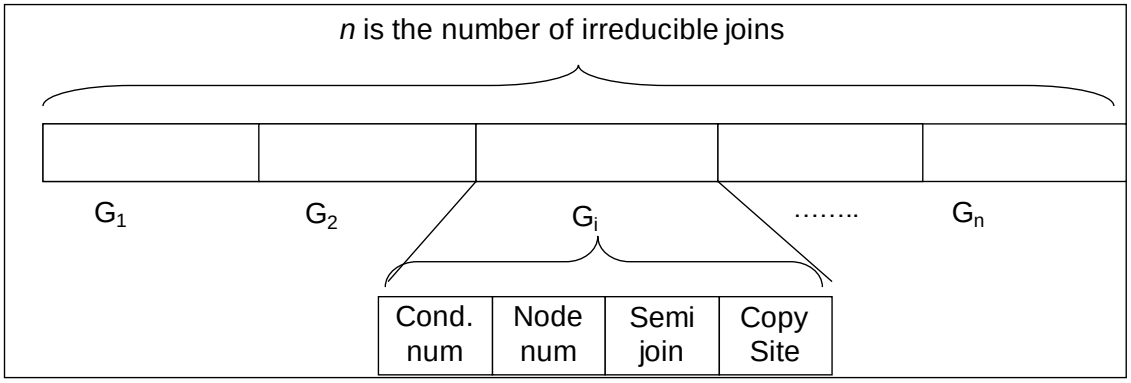


Figure 2.13: Chromosome structure

The chromosome structure of a query is shown in Figure 2.13. Each gene, G_i, has the following information;

- Condition number
- Node number
- Semijoin bits (2 bits) and
- Copy Site

Below, the crossover and mutation operators in NGA will be explained. In this paper, our proposed crossover is named as *New-Crossover* and mutation as *New-Mutation*. In our work we use two-point crossover with 50% truncation technique since it is shown to be better than other alternatives in a set of distributed database design experiments [Bayir (2007)]. Rest of the parameters for our GA is listed in Table 2.1.

Table 2.1: Parameter values for genetic algorithm

Initial Pool Size	100
Mating Population	50
Convergence Ratio	95%
Crossover type	Truncate, 2-point
Truncate ratio	50%
Crossover Ratio	0.7 (70%)
Mutation ratio	0.005 (0.5%)

2.7.2.3 Optimization Model

The model is given as graph *G* containing a set of conditions, nodes and input relations residing at various sites.

G = (C, N, S), where C is the set of conditions in the query graph, N is the set of nodes and S denotes set of source sites/nodes.

The model used in this work is explained in Figure 2.14.

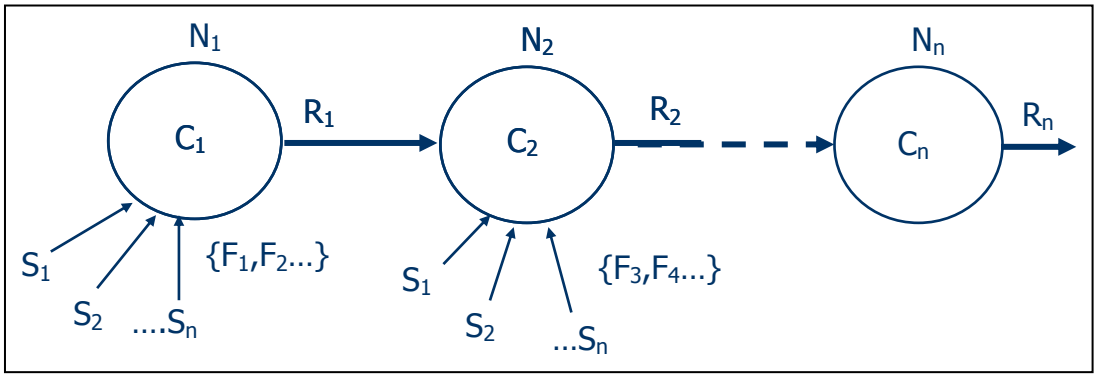


Figure 2.14: Optimization model

Each condition, $C_i \epsilon C$, has input fragments (F_n) of relations at various sites, S_n. Then each condition is evaluated at $N_i \epsilon N$, then the result (R_i) is sent to the next node which might also be the same as N_i. Since we're working with distributed queries, horizontal fragments or replicas must be taken into consideration for the condition to be evaluated. Each of the fragments or replicas (F_n) are fetched from (S_n) sites, optionally performing a semijoin operation. These operations are all done in parallel; maximum of these operations is the communication time to get the needed files from the residing sites (S_n).

After deciding the best QEP, the Master Node which the query is issued by will order the related nodes to execute the sub queries that they are responsible for.

Semi join technique has also been implemented for D-QOA if feasible, which is different from the execution strategy. This is also another ongoing study for D-QOA which was presented shortly [Ozsu (2007)].

2.7.2.4 Query Execution Model

The model is given as a graph *G = (C, S, F)* containing a set of join conditions(*C*), sites(*S*) and input relations/fragments residing at various sites(*F*).

Each join condition, C_i, has input fragments/replicas (F_j) of relations stored at sites, S_k. Each condition is evaluated at site S_k, after which the result (R_j) is sent to the next site which might also be the same as S_k. Since we're working with distributed queries, horizontal fragments or replicas of a relation must be taken into consideration for a join operation to be evaluated. Optionally, a semijoin operation can be performed on each F_j. These operations are all done in parallel, and the longest of these operations is the communication time to transfer the input relations/fragments from their sites.

Query Execution Plan (QEP) which is prepared using Query Execution Model is given in Figure 2.15. Dashed lines denote semijoin operations.

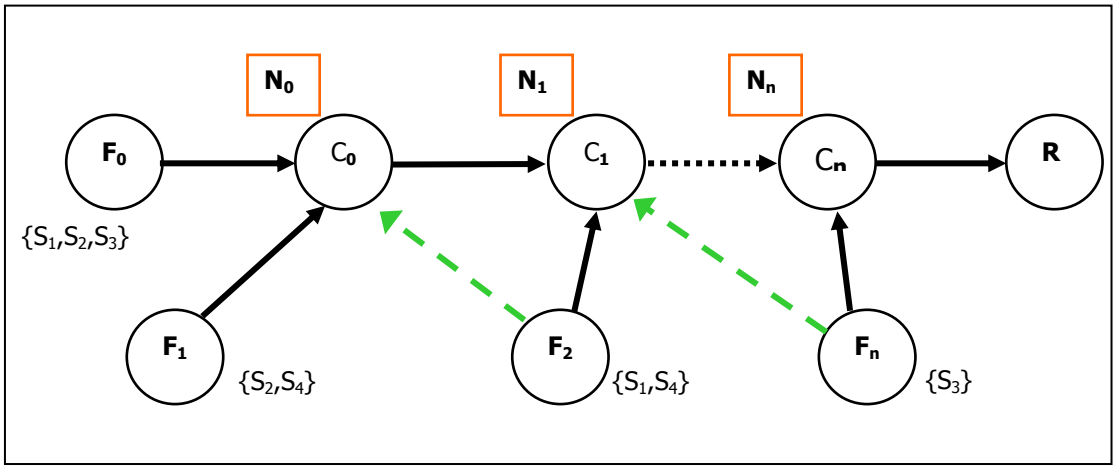


Figure 2.15 : Query execution plan

The cost of an execution plan, denoted by *Cost(P)* is calculated by using Formula 2.1 and 2.2 below.

$$\textbf{Cost (P)} = \sum_{i=0..n} \text{comm.\_cost}(Rel_i, S_{ki}) + \sum_{j=0..m} \text{Proc\_cost}\ (C_j) + \sum_{k=0..m} \text{comm\_cost}(R_k) \quad \textbf{(2.1)}$$

$$\textbf{Comm\_cost } (\textbf{Rel}_\textbf{i},\textbf{S}_\textbf{k}) = \max_{j=0..NF_i} | (\text{comm\_cost}(F_{ij},S_k), \text{ where } Rel_i \text{ has } NF_i \text{ fragments} \quad \textbf{(2.2)}$$

Our formula contains three different areas. First we begin with the communication costs of the related relations. In order to execute a sub query, firstly the fragments/replicas (F_i) of those relations must be fetched to the sites, S_k. This is done in parallel in our model, thus the cost will not be the total of the whole time but the maximum of them. For example, if R001 and R002 are to be fetched for a sub query then max communication time of the decided fragments/replicas will be taken as the communication time of the related files.

Then secondly we see *Proc_cost(Cj)* which denotes the local processing cost of the i^{th} sub query. All the calculations are done due to related formulas. Test bed has been explained in Table 2.1, 2.2 and 2.3.

Table 2.2: Relation schema

Relation ID	Attributes
Rel_1000	(attr1, attr2, attr3, attr4, attr5)
Rel_1001	(attr1, attr6, attr7, attr8, attr9, attr10)
Rel_1002	(attr6, attr11 attr12, attr13,attr14,attr15)
Rel_1003	(attr11, attr16, attr17, attr18, attr19, attr20)
Rel_1004	(attr16, attr21, attr22, attr23, attr24, attr25)
Rel_1005	(attr21, attr26, attr27, attr28, attr29, attr30)

- All key fields are 4-byte, rest of the fields are all assumed 6-byte long.
- Rel_1000 has 120000, Rel_1001 has 100000, Rel_1002 has 80000, Rel_1003 has 60000, Rel_1004 has 40000 and Rel_1005 has 30000 tuples.
- Any relation is vertically fragmented.
- If horizontally fragmented, then the total number of tuples for that relation is randomly separated among the fragments.

Table 2.3 : Selectivity factors among relations

Percentage (%)	Rel_ 1000	Rel_ 1001	Rel_ 1002	Rel_ 1003	Rel_ 1004	Rel_ 1005
Rel_1000	---	21	16	34	60	12
Rel_1001	21	---	28	45	36	34
Rel_1002	16	28	---	43	5	30
Rel_1003	34	45	43	---	39	33
Rel_1004	60	36	5	39	---	29
Rel_1005	12	34	30	33	29	---

For local processing times, only Block Nested Loop (BNL) has been used. In this type of calculations, BNL is commonly used for the sake of simplicity and gives results realistic enough. Other types of indexing (B+ tree, hash index, sort merge joins etc.) are out of vision throughout this study, since BNL works regardless of indices. According to Formula 2.3, BNL is evaluated;

$$\text{Local Processing Cost (Proc\_cost(Cj))} = N + M * \left\lceil \frac{N}{B-2} \right\rceil \quad \textbf{(2.3)}$$

where *M* is the number of pages of bigger relation, *N* is that of smaller relation and B is the number of Buffer Pages

If the number of Buffer Pages (B) are big enough to hold the smaller relation, namely B>N+2, and the smaller relation fits in the memory then Formula 2.4 is used;

$$\text{Local Processing Cost (Proc\_cost(Cj))} = M + N \quad \textbf{(2.4)}$$

One of two more pages is used for reading the larger relation page-by-page and the other page will serve as an output buffer.

All network wide communications are calculated due to bandwidths listed in the same section. All data have been first thought as packets and then time is assessed due to those packets to take time through the WAN/LAN environment.

Another important parameter for executing the queries is their selectivity. Selectivity Factor (SF) has been taken due to database statistics. The selectivity factors for input relations are given in Table 2.3, and they are used for calculating the expected size

of join results that will greatly affect the communication costs in a distributed database environments. All formulations use the same value any time for the same process. Experiments are done in order to find out which strategy is better than the others under the same conditions.

There are three parameters of NGA that will greatly affect the performance of a GA based optimization algorithm. These parameters are (1) mutation percentage, (2) crossover percentage and (3) initial population size. In order to decide best values for these we performed three experiments plotting performance graphics for varying values of them.

The results in Figure 2.16, Figure 2.17, and Figure 2.18 show that a crossover percentage of 0.6, mutation rate of 0.015, and initial population size of 100 gives the best results. In fact larger population sizes will slightly improve the solutions but only at the cost of an exponential increase in the GA runtime.

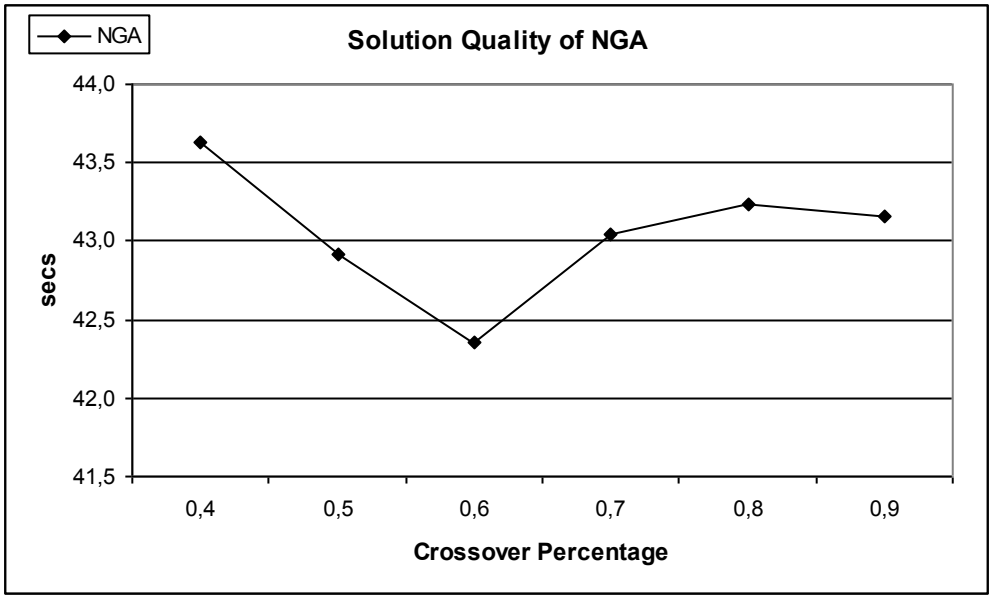


Figure 2.16 : The performance of NGA for increasing crossover percentages

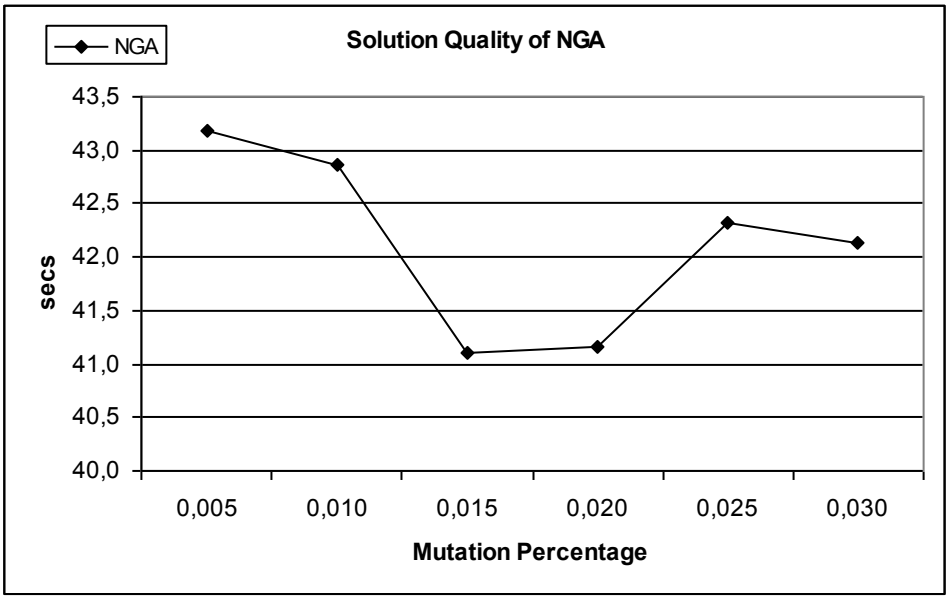


Figure 2.17 : The performance of NGA for increasing mutation rates

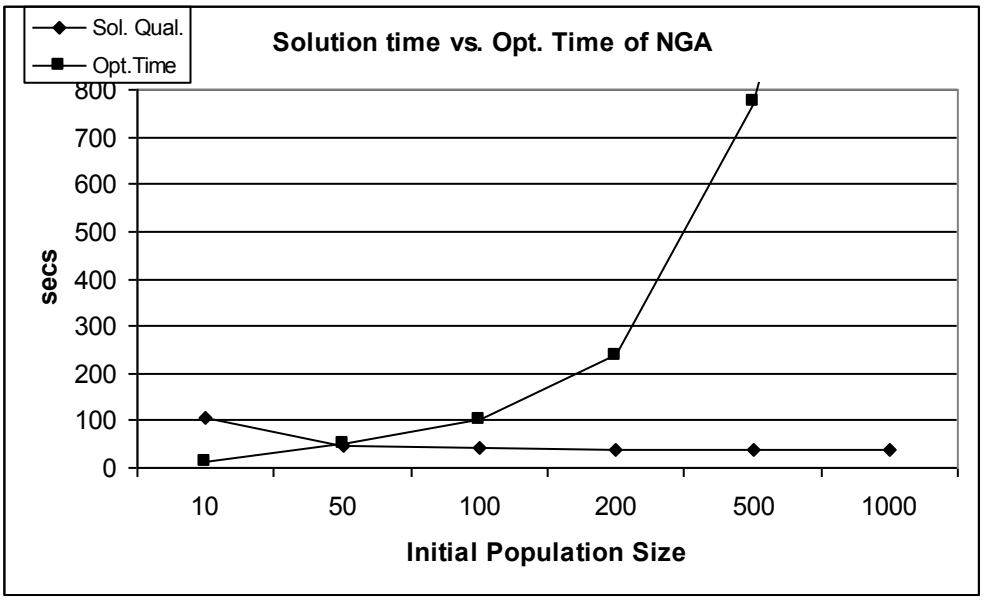


Figure 2.18 : The performance of NGA for increasing initial population size

The crossover operation also has two widely used methods, one-point and two-point. In one-point a random position is selected on the chromosome and genes up to this point are copied from the first (second) parent and remaining genes are copied from the corresponding positions of the second (first) parent. In two-point crossover two

random points are selected on the chromosome and the genes between these two points are swapped. Both one-point and two-point crossover will generate two new individuals.

Table 2.4: Types of genetic algorithms

Genetic Algorithm	Selection Type	Crossover Type
GA1	Tournament	One-point
GA2	Tournament	Two-point
GA3	Roulette Wheel	One-point
GA4	Roulette Wheel	Two-point
GA5	Truncate	One-point
GA6	Truncate	Two-point

In order to decide what combination of one-point/two-point crossover and tournament/roulette-wheel/truncate methods will give the best GA method, we have implemented 6 combinations as defined in Table 2.4, and compared them experimentally. The results are shown in Figure 2.19;

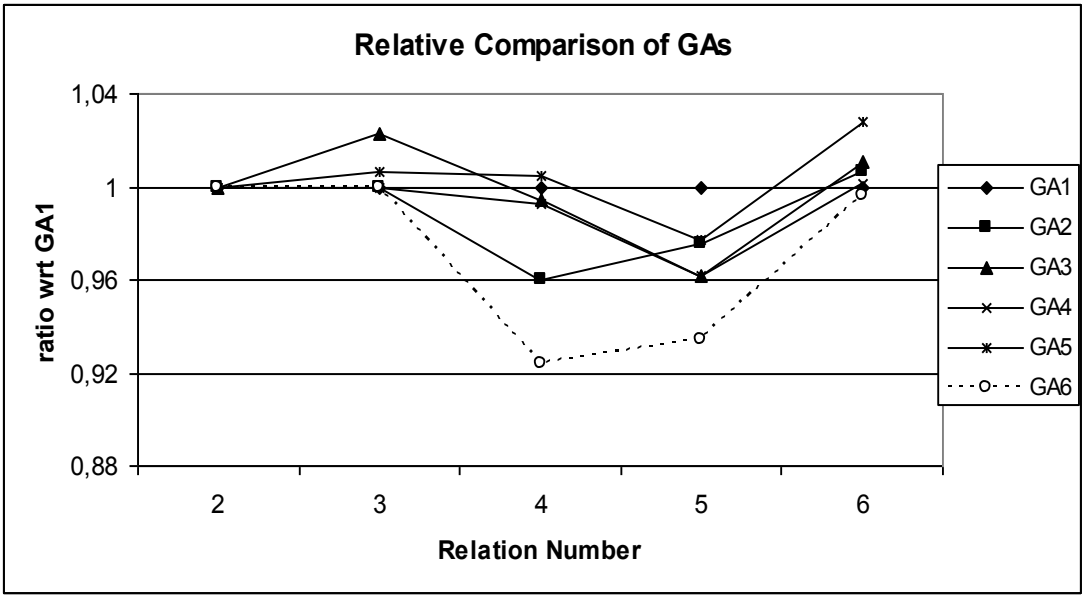


Figure 2.19 : Solution quality based comparison of selection and crossover type combinations

CHAPTER 3

SYSTEM DESIGN

In our study, we can divide our work into three parts. First one is preparing a program which simulates a distributed database by reading data from nodes and processing execution plans in HPC environment. Experiments comparing the optimization algorithms are performed by using this program.

The second one is preparing a program which generates test data for the first program to use.

The third part of the work includes the program testing the communication between the nodes (processors) and the program testing the sort times of the relations.

3.1. Program Processing Execution Plans

Program processing execution plan is written in C programming language by using MPI libraries. Program execution is as the following:

1. Program starting to execute at any node reads all the relations defined in the file **"**RelationList**"**, all the foreign key information from the file **"**JoinList**"** and then wait for completion of other nodes' reading. In the file **"**RelationList**"**, file name, number of columns and range values for the selection is specified for each relation. While reading, selection is done on the rows of the relation by using the maximum and minimum values given in the file **"**RelationList**"**.

2. Program running at node with number zero is run in server mode, programs running on other nodes run as client.

3. Server process reads the execution plan from file "RelationPath". After that, it parses the execution plan by creating a join processing tree. The tree created by this process is a left deep tree, which is a linear join processing tree whose inner relation of all joins are base relations. [Bennett (1991)]

4. While traversing the tree in-order [Dale (1992)], the server process sends commands to clients:

   i. It sends the name of relations to be read to the nodes defined in the leaves of the binary tree.

   ii. It sends the source and destination information to the nodes which will receive and join relations.

   iii. It doesn't wait for a completion of the messages to be received by the client processes. This provides possible parallelism in the case of reading and joining operations are performed at different nodes concurrently.

   iv. When the execution plan is complete, it sends a finish command to the all clients.

   v. It waits for completion of other nodes to be ready for termination.

vi. When all nodes are ready to finish, it gets the time of job finishing and then terminates itself.

5. Client processes wait command from server process. When client gets a command:

i. If the command is read relation:

a. It gets the relation name from server and takes related relation from the relation set which is red previously.

b. It waits for a delay which is simulating the disk I/O time for reading the whole relation from the disk. Calculation of this delay is given at next section.

c. It receives the target node for relation data to send from the server process.

d. If target node is the node itself, it saves a pointer to the relation data. If not, it sends number of columns and number of rows to the target node and then it sends the data by messages consisting of specified number of rows. Different message sizes are examined at experiments and message size is fixed to 200 rows.

e. It waits for a new command.

ii. If the command is join:

a. Join process has two relations, the firs one is the result of the previous joins and the second one is a relation to join to the result set.

b. The node performing the join process gets the sources of two relations to join from the server process.

c. If one or both of the relations are at the same node, it simply gets the pointer to the related relations' data.

d. If a source node is another node, it gets the number of columns and number of rows from the source node and gets

the data from the source node by predefined message length in rows.

e. Join is performed by sort merge join [Ramakrishnan (2002)] algorithm.

- ❖ A relation can be associated with at most two relations.
- ❖ If there is no association between two relations, relations are joined by taking cross product of the relations.
- ❖ If two relations have only one column to join, equijoin is performed on this column.
- ❖ If two relations have two columns to join, equijoin is performed on both of these columns.

f. It receives the target node for join result data from the server process:

- ❖ If it is the final node that having the result set of the query executed, it sends the result set to the root node.
- ❖ If target node is the node itself, it saves a pointer to the relation data.
- ❖ If target node is another node, it sends number of columns and number of rows to the target node and then it sends the data by predefined message length in rows.

g. It waits for a new command.

iii. If the command is finish:

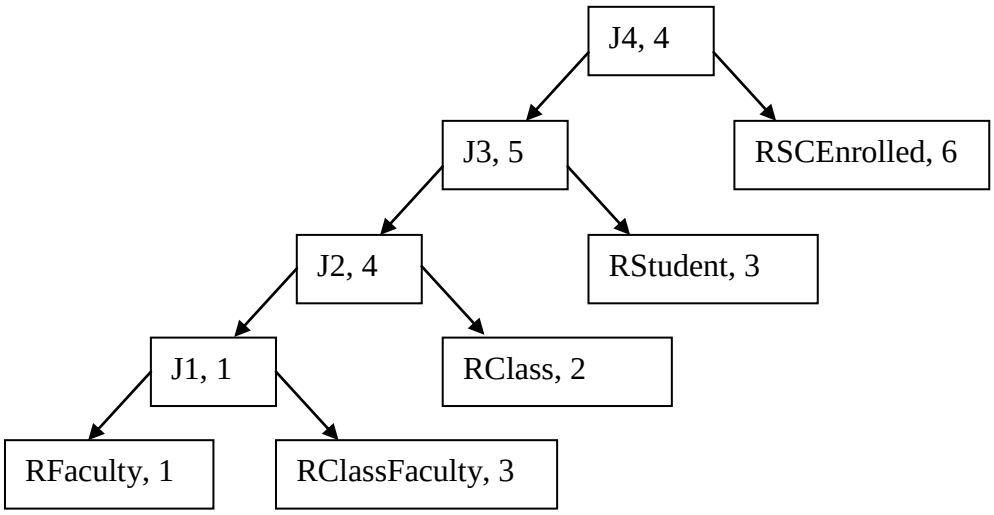
a. Program running at a node waits for completion of other nodes to be ready for termination.

b. Program running at node finishes execution.

An important experience about message passing parallel computing is that, synchronization must be considered in all phases of the program. If one of the nodes which has sent its data to another node finishes before its receiving partner node to complete receive operation, execution crashes at the node which hasn't completed the receive operation yet.

Another important issue is that, when a program running on multiple nodes gives an error, it completes its execution with runtime error message in its error message file. It means, if program is crashing at some nodes with zombie process, it's probably related with the problem about synchronization.

In our programs, we synchronized programs executing on all nodes with the command "MPI_Barrier()".



(((((RFaculty,1),(RClassFaculty,3),1),(RClass,2),4),(RStudent,3),5),(RSCEnrolled,6),4)

Figure 3.1: A sample query plan and its corresponding join tree

3.1.1 Calculating Disk I/O Time for Reading the Whole Relation from the Disk

Properties of a sample disk similar to the disks used in METU HPC system is given in [baracuda (2009)]. In the sample disk average seek time is 8.5 ms, drive transfer rate is 300 Mbps and average latency is 4.16 ms.

There are tree main parameters used to measure the time of transferring data from disk to memory. The firs one is *seek time* which is time it takes to move the disk arm to the correct cylinder. Seek time is the larger in cost relative to the others. When the arm is placed on the correct cylinder with the head on the correct track switched on, it takes some time for disk to resolve that correct place on the track is reaching the head. This time is called as *rotational latency*. *Block* is a unit or amount of data which is read from the disk to memory at a time. The time for the head to pass over a block is called as *block transfer time*. [Salzberg (1988)]

When retrieving the relation information from the disk, data is read block by block. If size of the relation data is less then a block size, a whole block is read from the disk even if the data needed is a single row. In our study, we assumed that a single block size is 1KB which is equal to 8Kb. So, we evaluate the data transfer times in terms of blocks. Transfer time for a block is calculated as:

Block Transfer Time = 8Kb/300Mb second =0,026 ms = 26 μs **(3.1)**

For reading a file from disk with the assumption that blocks are placed on disk sequentially, time which is spent for I/O operation can be calculated as follow [Salzberg (1988)]:

Transfer Time = *Seek Time* + number of blocks* *Block Transfer Time* + *Average Latency* **(3.2)**

By the formula given in Formula 3.2, we can calculate transfer time as:

8.5 + number of blocks*0,026 + 4.16 ms = 8500+ number of blocks* 26+ 4160 μs

Since an integer size is 4 bytes, we can calculate number of blocks to read when reading a relation as:

Number of blocks to read = floor((4 * number of integers to read)/1000) +1 **(3.3)**

3.1.2 File Formats

3.1.2.1 RelationList

This file keeps the names of the files keeping the relation data. It also has the column and value range information for select operation for each relation. All relation data is assumed to be in type integer. Each row consists of:

- *Relation Name* (string): Name of the relation file to be read.
- *Number of Columns* (integer): Number of column in the relation.
- *Selection Column* (integer): Column number starting from 0 to apply select statement.
- *Minimum Value* (integer): Minimum value for selection.
- *Maximum Value* (integer): Maximum value for selection.

Let we have four relations namely R1, R2, R1R2 and R3. R1 has 4 columns integer data, R2 has 5 column integer data, R1R2 has 2 column integer data and finally R3 has 3 column integer data. All integer values are between 0 and 100.000 in the relations and we want all rows from all tables in our query, except R3 which is required to have only the values less than or equal to 1000 in column with number 2. “RelationList” file for this configuration will be as in Figure 3.2.

R1 4 0 0 100000

R2 5 0 0 100000

R1R2 2 0 0 100000

R3 3 2 0 1000

Figure 3.2: A sample "RelationList" file

3.1.2.2 JoinList

This file keeps the foreign key constraints to use in join operations. A foreign key constraint is defined as the following:

- *Table 1* (string): Name of the first relation.
- *Table 2* (string): Name of the second relation.
- *Column of the first relation (integer)*: Column number of the first relation.

Let R1 and R1R2 has a foreign key constraint such that R1[0]=R1R2[0], R2 and R1R2 has a foreign key constraint such that R2[0]=R1R2[1]. The "JoinList" file for this configuration will be as in Figure 3.3.

R1 R1R2 0

R2 R1R2 0

R1R2 R1 0

R1R2 R2 1

Figure 3.3: A sample "JoinList" file

3.1.2.3 RelationPath

File holding execution plan. Execution plan denotes from which nodes the relations will be read and in which nodes to join.

Let *G* is the grammar of execution plan and *Start* is the start symbol.

Node: {1, 2, 3, …, k}, k is the number of nodes.

Join: Operator.

Relation: {R1, R2, R3, …., Rn}, n is the number of relations.

Rules of the grammar G can be given as:

Start->Join

Join-> (Join, (Relation, Node), Node)

Join-> (Relation, Node)

Let relation R1 will be read at node 1, relation R1R2 will be read at node 3 and both relations will be joined at node 1. The result at the node 1 will be joined at node 4 with the relation R2 coming from node 2. The result at the node 4 will be joined at node 5 with the relation R2R3 coming from node 3. The result at node 5 will be joined at node 4 with the relation R3 coming from node 6. "RelationPath" file for this execution plan will be as in Figure 3.4.

(((((R1,1),(R1R2,3),1),(R2,2),4),(R2R3,3),5),(R3,6),4)

Figure 3.4: A sample "RelationPath" file

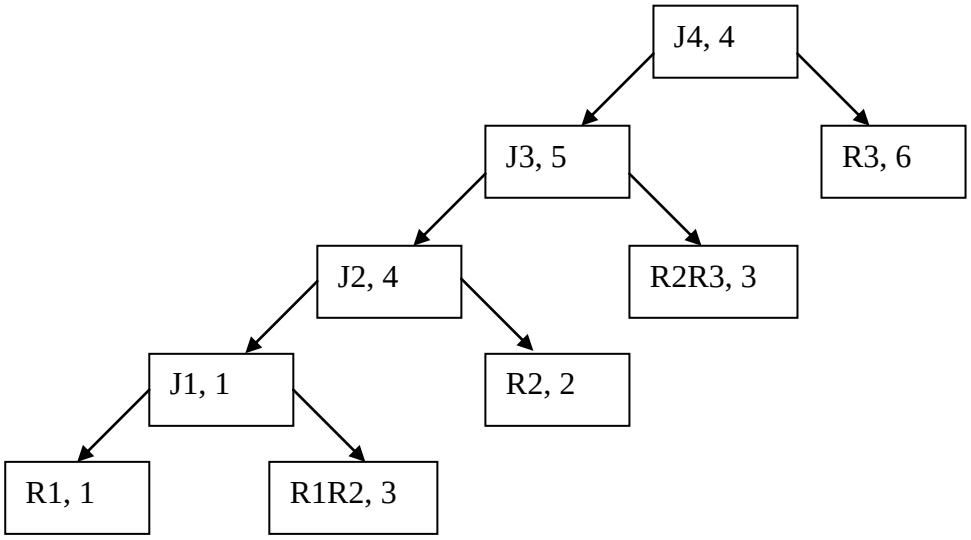


Figure 3.5: A sample join processing tree

So as to relation files, names of the relation files must be the same as the relations. For example, if “R1” is a relation containing 4 integer values at each row, a sample relation file with the name “R1” can be as the following:

1, 23, 45, 34

2, 34, 12, 444

3, 56, 23, 5

A sample R2 file containing 4 integer values at each row can be as the following:

1, 15, 44, 16

2, 98, 28, 2

A sample R1R file containing 2 integer values at each row can be as the following:

1, 1

1, 2

2, 1

3, 2

3.2 Program Generating Sample Relation Files

Program generating sample Relation files is written in C programming language. This program takes list of relations and relationships including row, column and value range information and generate relation files in a format which is appropriate to use in the program processing execution plans.

There are two parameter files namely “Relations” and “Joins”. The file “Relations” include the relation name, number of columns and number of rows that the relation will have. A sample “Relations” file is given in Figure 3.6.

RStudent 4 100000

RClass 5 25

RFaculty 3 5

Figure 3.6: A sample “Relations” file

In Figure 3.6, we have three base relations, which are “RStudent” with 4 columns and 100000 rows, “RClass” with 5 columns and 25 rows and “RFaculty” with 3 columns and 5 rows.

The file “Joins” include the name of the relationships (as a relation) which joins two relations, name of the relations to join, percentage of the rows to include, how many

times of the fully included relation with higher cardinality will be used in the result set of the join, and finally number of columns of the resulting relation.

RSCEnrolled RStudent RClass 100 100 5 4

RCF RClass RFaculty 100 100 1 3

Figure 3.7: A sample "Joins" file

In the first row of Figure 3.7, RSCEnrolled is a relationship joining RStudent and RClass, both relations will be fully included at the relation RSCEnrolled. Since RStudent has 100000 rows, RSCEnrolled will have 500000 rows and 4 columns. Selectivity of table RStudent will be %20.

3.3 Program Transferring Data between Two Nodes by Given Message Size

Program sending data consisting of 10.000.000 integer values from a source and receive the data at the destination node. Source node, destination node and size of the packages which denotes number of integer values in each package are parameters of the program. This program is prepared to test the effects of changing message sizes and changing nodes on transmission cost of fixed amount of data between two nodes. This program is also used for generating some basic metrics like message transfer times.

3.4 Program Sorting Relation Data

Program reading the column and row numbers of relations from a file, generating the relation data and sorting the relation to record time elapsed while sorting these relations with different number of columns and different number of rows. By using this program, the effect of increase in the size of the relation to the time cost of sort operation is measured.

CHAPTER 4

EXPERIMENTAL SETUP AND RESULTS

4.1 Experimental Setup

In the experimental study, different tests are applied on both a simple program transferring data between two single nodes and another program processing the execution plans.

In the firs experiment, effects of changing the nodes of a single data transfer between two nodes and changing the nodes of relations and joins in an execution plan are examined.

In the second experiment, by changing the message sizes, effects of message response times and message receive times on data transmission are examined in a single data transfer between two nodes. The same tests are applied to the program processing the execution plans so as to get optimal message sizes to use in next experiments.

In the third part, different execution plans are executed using the program processing the execution plans.

In the last part, to measure the effects of the data sorting cost to overall join cost are examined by sorting tables with different number of columns and different number of rows.

4.2 Effect of Changing Nodes on Transmission Time Cost in a Single Data Transfer

In this experiment, data consisting of 10,000,000 integer is transferred between two nodes by sending the integer values one by one. The same data is transferred between two different node pairs. In the first execution, data is transferred from node 3 to node 2. In the second execution, data is transferred from node 2 to node 1.

Table 4.1 : Transfer time of 10,000,000 integer values between different nodes

	Duration(sec) From node #3 to #2	**Duration(sec) From node #2 to #1**
Run #1	8.88	9.08
Run #2	8.75	8.63
Run #3	8.88	8.78
Run #4	8.88	7.89
Run #5	9.08	8.79
Run #6	8.80	9.11
Run #7	7.78	9.10
Run #8	8.84	9.02
Run #9	8.87	9.25
Run #10	9.08	8.72
Average	**8.87**	**8.90**

As shown in Table 4.1, transferring the data between the two nodes cost the same in time, independent of the nodes used. In other words, changing the nodes used in data transmission doesn't affect the time cost.

4.3 Effect of Changing Nodes on Transmission Time Cost While Processing an Execution Plan

In this experiment, the following execution plans which return 500.000 rows with 17 columns at last node are executed by using the program processing execution plans:

Execution Plan #1: (((((RFaculty,1),(RClassFacultyMapped,2),3),(RClass,4),5),(RSCEnrolled,6),7),(RStudent,8),9)

Execution Plan #2: (((((RFaculty,4),(RClassFacultyMapped,7),9),(RClass,3),2),(RSCEnrolled,8),5),(RStudent,6),1)

Table 4.2 : Transfer time of processing the same execution plan with different nodes

	Execution Plan #1 (Seconds)	**Execution Plan #2 (Seconds)**
Run #1	10.2999	10.2797
Run #2	10.2186	10.0230
Run #3	10.2790	10.2484
Run #4	10.2415	10.1953
Run #5	10.3123	10.2732
Run #6	10.1937	10.2822
Run #7	10.2403	10.2991
Run #8	10.2036	10.2475
Run #9	10.2346	10.3116
Run #10	10.2666	10.2701
Average	**10.2480**	**10.2619**

As shown in Table 4.2, executing the same execution plan by only changing the nodes used cost the same in time. In other words, changing the nodes used in an execution plan doesn't affect the time cost.

4.4 Effect of Changing the Message Size on Transmission Time Cost in a Single Data Transfer

In this experiment, effects of message response times and message transfer times on data transmission are examined. These parameters are also calculated approximately by using the time results we get at the end of experiments.

The correlation between the message sizes and the time cost of the data transfer between two nodes is another aspect.

In Table 4.3, transfer time values of data consisting of 10.000.000 integer values between two nodes with respect to the message sizes are shown. Message sizes applied in sample runs are as the following:

a. In the first column, 1 integer value is sent in each message which is 4 byte long,

b. In the second column, 10 integer values are sent in each message which is 40 byte long,

c. In the third column, 25 integer values are sent in each message which is 100 byte long,

d. In the fourth column, 100 integer values are sent in each message which is 400 byte long,

e. In the fifth column, 250 integer values are sent in each message which is 1 Kbyte long,

Table 4.3: Transfer time of 10.000.000 integer values between two nodes with respect to the message sizes

	Duration (sec) 1 Row (4 Byte)	**Duration (sec) 10 Rows (40 Byte)**	**Duration (sec) 25 Rows (100 Byte)**	**Duration (sec) 100 Rows (400 Byte)**	**Duration (sec) 250 Rows (1 KByte)**
Run #1	9.56	0.95	0.55	0.26	0.21
Run #2	9.06	1.14	0.51	0.28	0.23
Run #3	8.27	1.11	0.61	0.26	0.21
Run #4	9.48	1.12	0.51	0.28	0.22
Run #5	9.28	1.09	0.56	0.27	0.19
Run #6	9.15	1.06	0.51	0.29	0.21
Run #7	9.06	1.13	0.57	0.25	0.22
Run #8	9.48	1.08	0.55	0.28	0.22
Run #9	9.02	0.94	0.55	0.28	0.20
Run #10	9.11	1.12	0.55	0.28	0.21
Average	**9.20**	**1.08**	**0.54**	**0.27**	**0.21**

According to the Table 4.3, variation of the average time cost with the message size is visualized with the graphic in Figure 4.1.

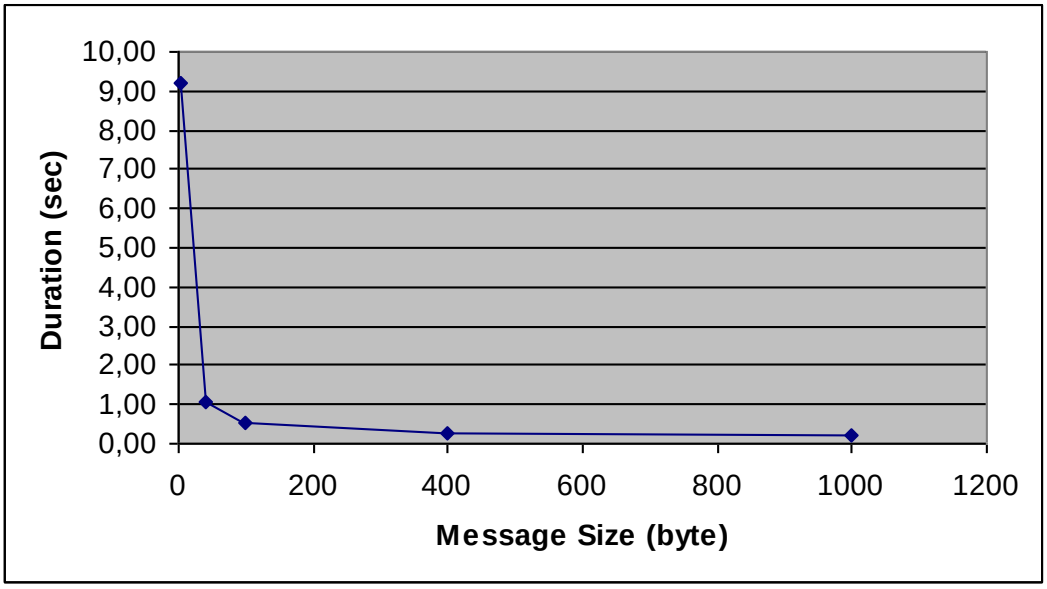


Figure 4.1: Transmission cost variation with message size

As shown in Table 4.3 and graphic in Figure 4.1, time cost of a single data transmission between two nodes is decreased sharply when we increase the message size from 4 byte to 40 byte. When message size reaches 0.4 Kbyte, since the message response time looses its importance relative to the transfer time of a single message, effect of increase in message size to the time cost of data transmission is decreased dramatically. This experiment proves the formula:

Transmission Time = *Number of Messages* * (*Response Time* + *Message Transfer Time*) **(4.1)**

By using the values in the last column of Table 4.3 in which we can ignore the response time, with the assumption of *Transmission Time* will converge to 0.2 second when we send the whole 40.000.000 Byte data in a single message, we can

simply calculate the *Message Transfer Time* values for a single message consisting of a single Byte value by:

Message Transfer Time of 1Byte = 0.2 sec/40,000,000 Byte= **0.005μs /Byte**.

By using the values in the first column of Table 4.3, in which each integer value is sent in a single message, average *Response Time* for a single message transferred can be calculated as:

9.2 = 10,000,000 * (Response Time + 4 * 0.005)

Response Time = **0.9μ sec/message**.

Transmission Time = ***Number of Messages*** * (**0.9μs + 0.005μs * BytesPerMsg**) **(4.2)**

4.5 Effect of Changing the Message Size on Transmission Time Cost While Processing an Execution Plan

In this experiment, the correlation between the message sizes and the time cost of the data transfer is examined on the program processing the execution plans to determine the optimal message size to use in next steps.

In the experiment, the following execution plan, which returns 500.000 rows with 17 columns at last node, is executed by using the program processing execution plans:

Execution Plan : (((((RFaculty,1),(RClassFacultyMapped,2),3),(RClass,4),5),(RSCEnrolled,6),7),(RStudent,8),9)

In Table 4.4, total execution time values for the preceding execution plan with respect to message sizes are shown. Message sizes applied in sample runs are as the following:

a. In the first column, 1 row of data is sent in each message which is 68 byte long at the last node,

b. In the second column, 10 rows of data is sent in each message which is 680 byte long at the last node,

c. In the third column, 50 rows of data is sent in each message which is 3.4 Kbyte long at the last node,

d. In the fourth column, 100 rows of data is sent in each message which is 6.8 Kbyte long at the last node,

e. In the fifth column, 250 rows of data are sent in each message which is 13.6 Kbyte long at the last node.

Table 4.4: Execution time of a sample execution plan with respect to the message sizes

	Duration (sec) 1 Row (68 Byte)	**Duration (sec) 10 Row (680 Byte)**	**Duration (sec) 50 Row (3.4 KByte)**	**Duration (sec) 100 Row (6.8 KByte)**	**Duration (sec) 200 Row (13.6 KByte)**
Run #1	10.2999	5.8977	5.5236	5.5268	5.5571
Run #2	10.2186	5.9018	5.5340	5.5292	5.5159
Run #3	10.2790	5.7749	5.5271	5.4962	5.5276
Run #4	10.2415	5.8690	5.5308	5.5507	5.5228
Run #5	10.3123	5.9077	5.5260	5.5036	5.5305
Run #6	10.1937	5.9474	5.5362	5.5741	5.5228
Run #7	10.2403	5.9160	5.5451	5.5528	5.5495
Run #8	10.2036	5.8955	5.5642	5.5183	5.5050
Run #9	10.2346	5.9018	5.5193	5.5157	5.5105
Run #10	10.2666	5.9248	5.5208	5.5113	5.5280
Average	**10.2480**	**5.9018**	**5.5304**	**5.5261**	**5.5246**

According to the Table 4.4, variation of the average time cost with the message size is visualized with the graphic in Figure 4.2.

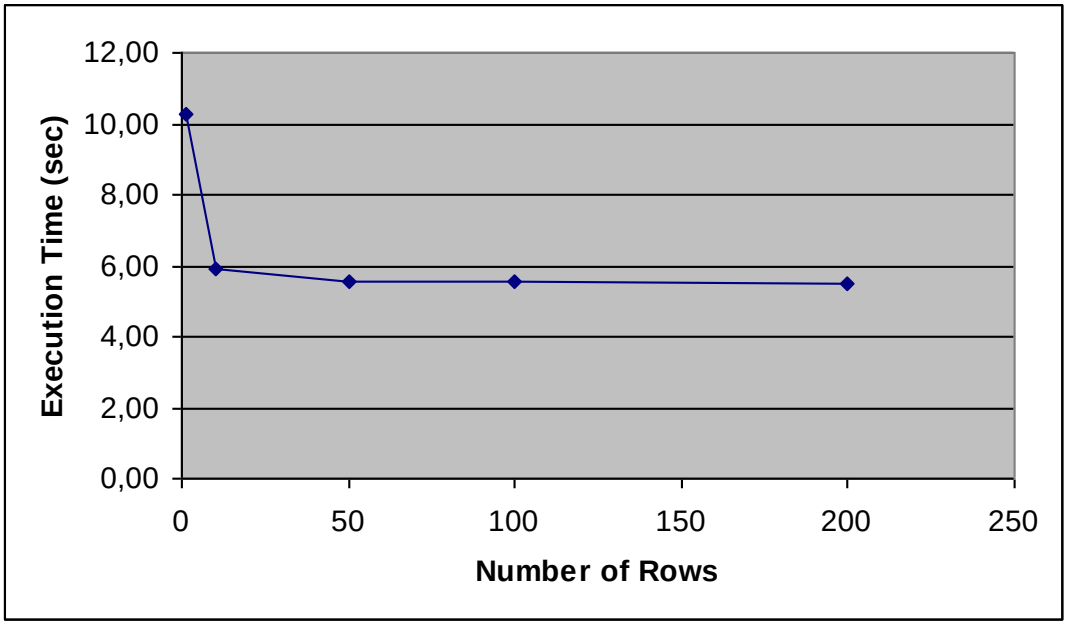


Figure 4.2: Execution time variation with message size

As shown in Table 4.4 and graphic in Figure 4.2, time cost of processing a whole execution plan is decreased to half when we increase the message size from 1 row to 10 rows. After 10 rows, effect of increases in message size to the execution time is decreased dramatically. After the message size of 50 rows which is 3.4 Kbyte long, effect of the increasing message size to the total execution time is ignorable.

In next experiments, message size which corresponds to 1 Kbyte or more will be optimal for efficiency in messaging.

4.6 Experiment with Data Generated by Three Query Optimization Algorithms

In this experiment, execution plans generated by three of the query optimization algorithms, namely Exhaustive Search, Genetic Algorithm and New genetic

algorithm, are generated by the program prepared in the thesis study of Ender Sevinç [Sevinç (2009)].

4.6.1 Execution Plan Generation

The program needs 4 input and 2 output files:

1. **"Environment.txt":** In this file, we give program the parameters denoting number of nodes to use, initiating node, which SQL file to execute and finally the file containing names and the row numbers of the relations in the query.

"Environment.txt" file used in our experiment is as the following:

```
//number of the nodes in the WAN
5
//query_initiated_node
0
//sql file to use; e.g. "4"= sql4.sql
5
//deployment = "4r_1r" means 4 relation and 1 is replicated
5r
```

Figure 4.3: Environment.txt file for 5 relations

2. **"Sql5.sql"**: This file has the query to be executed. Attribute names are not important, it is enough to give an equation to denote join. The selection at last row removes %30 of the data from the result set. Sample file used in our experiment is as the following:

```
SELECT Rel_1000.attr_0 , Rel_1000.attr_1
FROM Rel_1000 ,Rel_1001 ,Rel_1002 ,Rel_1003,Rel_1004
WHERE Rel_1000.attr_2 = Rel_1001.attr_2
AND Rel_1001.attr_3 = Rel_1002.attr_3
AND Rel_1002.attr_4 = Rel_1003.attr_4
AND Rel_1003.attr_1 = Rel_1004.attr_1
AND Rel_1001.attr_2 =  some_val
```

Figure 4.4: Sql5.sql file

3. **"rels_5r.txt"**: This file contains the relation number, its fragmentation and replication information with row numbers. In our case, neither fragmentation nor replication is used. Usage and impact of fragmentation and replication will be given in next experiments.

Sample file used in our experiment is as the following:

```
// Relation ID,Fragment#,Replica# Node# Cardinal#
1000,00,00 -1 1200000
1001,00,00 -1 600000
1002,00,00 -1 300000
1003,00,00 -1 100000
1004,00,00 -1 50000
```

Figure 4.5: rels_5r.txt file

4. **"Selectivity"**: Selectivity file gives the selectivity matrix of the relation to be used. It's calculated by dividing number of the rows in small relation to the number of the rows in larger one.

Sample file used in our experiment is as the following:

0.0, 0.5, 0.25, 0.08, 0.04, 0.02, 0.01
0.5, 0.0, 0.5, 0.17, 0.08, 0.04, 0.02
0.25, 0.5, 0.0, 0.33, 0.16, 0.08, 0.04
0.08, 0.17, 0.33, 0.0, 0.5, 0.24, 0.12
0.04, 0.08, 0.16, 0.5, 0.0, 0.48, 0.24
0.02, 0.04, 0.08, 0.24, 0.48, 0.0, 0.5
0.01, 0.02, 0.04, 0.12, 0.24, 0.5, 0.0

Figure 4.6: Selectivity file

5. **"result.doc"**: This is the output of the program generating the execution plans according to the three of the query optimization algorithms, which are Exhaustive Search, Genetic Algorithm and New genetic algorithm.

ESA (((((Rel_1003, 3) ,(Rel_1004, 4), 3) ,(Rel_1002, 2) , 2) ,(Rel_1001, 1) , 1) ,(Rel_1000, 0) , 0)
NGA (((((Rel_1003, 3) ,(Rel_1004, 4), 2) ,(Rel_1002, 2) , 2) ,(Rel_1001, 1) , 1) ,(Rel_1000, 0) , 0)
GA (((((Rel_1002, 2) ,(Rel_1003, 3), 3) ,(Rel_1004, 4) , 3) ,(Rel_1001, 1) , 1) ,(Rel_1000, 0) , 0)

Figure 4.7: result.doc file

6. **"Result 5.csv"**: This file gives the calculated average query execution times of the best plans generated by each algorithm in seconds. It also shows time spent to reach the best plan in each of these algorithms.

Table 4.5: Result 5.csv

	Average execution time of the best plan	**Time elapsed by optimizer**
ESA	63.7700	24360
NGA	97.1446	73.4
GA	121.7832	110.9

As seen in Table 4.5, ESA gives the best results but optimization time is dramatically greater than the other algorithms. GA and NGA give worse results than ESA but they generate these results in a very short time compared to ESA. NGA generate better result from GA in a shorter time period.

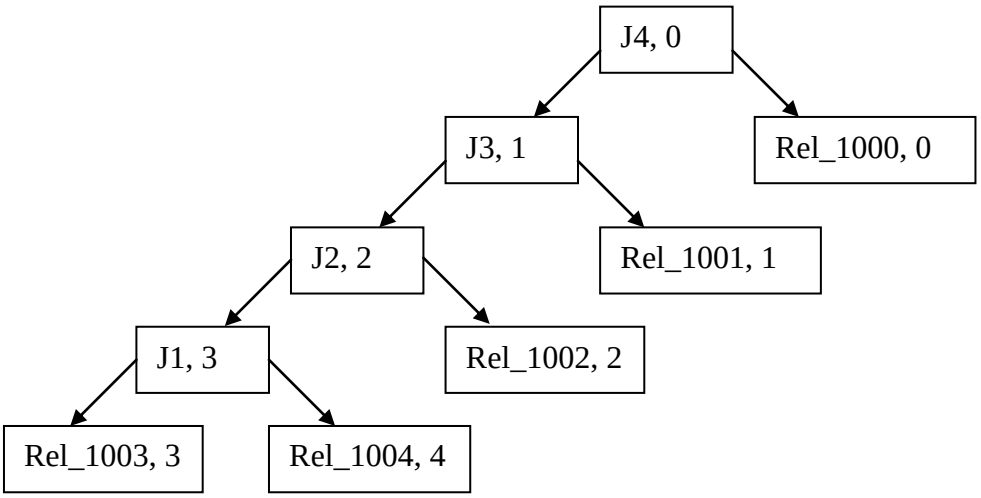


Figure 4.8: Join process tree of ESA for 5 relations

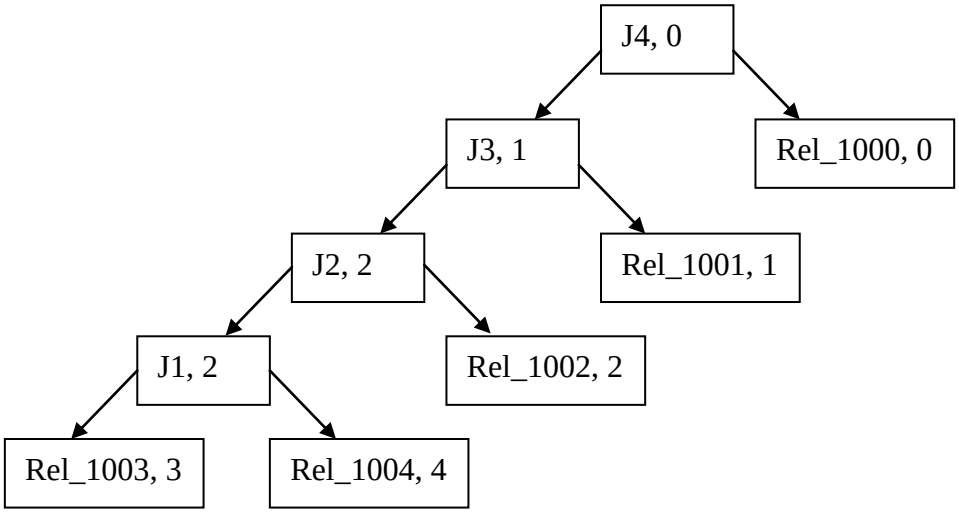


Figure 4.9: Join process tree of NGA for 5 relations

4.6.2 Results

We executed each of these three execution plans by our program processing execution plans in 20 different trials for each.

Table 4.6: Time result of 5 relations execution plans

Run #	Duration(sec)	Duration(sec)	Duration(sec)
1	0.9652	1.0748	1.1254
2	0.9888	1.0515	1.1371
3	0.9559	1.0531	1.1209
4	0.9533	1.0623	1.1156
5	0.9501	1.0548	1.1393
6	0.9569	1.0770	1.1110
7	0.9514	1.0792	1.1073
8	0.9515	1.0572	1.1211
9	0.9671	1.0798	1.1091
10	0.9720	1.0553	1.1087
11	0.9527	1.0614	1.1097
12	0.9498	1.0587	1.1144
13	0.9535	1.0598	1.1179
14	0.9699	1.0568	1.1270
15	0.9740	1.0708	1.1311
16	0.9532	1.0621	1.1184
17	0.9492	1.0586	1.1365
18	0.9554	1.0629	1.1236
19	0.9541	1.0498	1.1383
20	0.9610	1.0626	1.1232
Average	**0.9582**	**1.0622**	**1.1216**
Variance	**0.0001**	**0.0001**	**0.0001**

In the Table 4.6, the execution times of the plans generated by the three different algorithms are shown. Execution times of plan generated by Exhaustive Search Algorithm (ESA) are placed at the first column. As expected, ESA gave the best result since it searches the best plan by trying all possible plans in advance. In the second column, the execution times of plan generated by NGA [Sevinç (2009)] algorithm are given. As expected, NGA which has enhancements on GA have better results from GA given in last column.

An important point we have to mention about is the parallelism supported by our database implementation. In our program, root process sends commands to the nodes without waiting for the completion of each node. So, nodes in the same level of join processing tree process in parallel unless the same node is used successively.

4.7 Experiment with Larger Number of Tables

In this experiment, execution plans generated for 7 relations by optimization algorithms, Genetic Algorithm and New Genetic Algorithm by the program prepared in the same thesis [Sevinç (2009)] study are executed. Because of the increasing number of relations, Exhaustive Search algorithm couldn't produce a result in timely manner, so it is excluded in this experiment.

Table 4.7: Relations prepared for 7 relations execution plans

	Relation	**Number of Columns**	**Number of Rows**	**Selected Rows**
1	Rel_1000	4	12000	%100
2	Rel_1001	4	6000	%100
3	Rel_1002	5	3000	%100
4	Rel_1003	3	1000	%100
5	Rel_1004	3	500	%20
6	Rel_1005	4	5000	%100
7	Rel_1006	5	9000	%100

```
SELECT *
FROM Rel_1000, Rel_1001, Rel_1002, Rel_1003, Rel_1004, Rel_1005, Rel_1006
WHERE Rel_1000.attr_0 = Rel_1001.attr_0
AND Rel_1001.attr_1 = Rel_1002.attr_0
AND Rel_1002.attr_0 = Rel_1003.attr_0
AND Rel_1003.attr_1 = Rel_1004.attr_0
AND Rel_1004.attr_0 = Rel_1005.attr_0
AND Rel_1005.attr_1 = Rel_1006.attr_0
AND Rel_1004.attr_0 <  100
```

Figure 4.10: SQL statement of join with 7 relations

Relations and number of rows used in our experiment is as in the Table 4.7. SQL statement which is the query to be executed is in Figure 4.10. Execution plans generated by algorithms are as the following:

Execution plan of NGA: (((((((Rel_1003, 3) ,(Rel_1004, 0), 3) ,(Rel_1001, 0) , 2) ,(Rel_1005, 1), 2) ,(Rel_1002, 1), 2) ,(Rel_1000, 3) , 3) ,(Rel_1006, 2), 2)

Execution plan by GA: (((((((Rel_1003, 3) ,(Rel_1004, 0), 1) ,(Rel_1002, 1) , 1) ,(Rel_1001, 0), 2) ,(Rel_1005, 1), 2) ,(Rel_1000, 3) , 3) ,(Rel_1006, 2), 2)

Table 4.8: Time result of 7 relations execution plans

Run#	**GA Execution Time(sec)**	**NGA Execution Time(sec)**
1	0.0814	1241.4557
2	0.0907	1240.6300
3	0.0846	2863.1545
4	0.0772	2611.4929
5	0.0819	3468.8578
6	0.0903	1246.7934
7	0.0776	1241.1550
8	0.0863	1248.6002
9	0.0852	1242.7521
10	0.0984	1245.7114
11	0.0861	1247.6044
12	0.0936	1240.5195
13	0.0772	1241.4973
14	0.0776	1247.3641
15	0.0739	1247.4823
16	0.0898	1247.7713
17	0.0901	1244.3565
18	0.0753	1248.7758
19	0.0885	1247.0859
20	0.0973	1248.7422
Average	**0.0850**	**1411.2458**
Variance	4.11301E-05	234611.0904

While executing, since cross product process takes place in plan produced by NGA, it generates big overhead in joining all relations without any selection. In a sample run, it took 10 hours to join whole data in NGA algorithm. Because of this situation, in order to get faster and more accurate results, %20 of the data in relation 1004(Rel_1004.attr_0 < 100) is selected in each execution. Execution times of the plans are as in the Table 4.8.

Each execution plan is executed 20 times by the random data generated by test data generator program mentioned earlier. In the plan generated by GA, the order of joins allows equijoins in all steps. Execution times are close each other and variance is low.

As seen in Table 4.8, since the plan produced by NGA has order of relations, which cannot be joined directly in each step by equijoins, causes a cross product and execution times increases dramatically.

It is also seen that execution times has a big variance when the load of the system is high. It can be concluded that when the execution time is high, it is sensitive to the system load.

In NGA, %20 of data is joined approximately 25 minutes in average. However, %100 of data, which means 4 times increase in data processed, is joined in approximately 10 hours. It means, execution time doesn't increase linearly when the amount of data to be processed and transferred increase. Process having long execution times may loose their priorities while processing.

In the NGA experiments, a sample sun which takes 1233 seconds in total has spent 1218 seconds in a single join operation, between (Rel_1003 Joined Rel_1004 Joined Rel_1001) with 1,200,000 rows and Rel_1005 with 5,000 rows, resulting 12,000,000 rows as output. These data denote that most of the time cost belongs to join

operation and data transmission cost is very low for these type of join operations with high number of rows.

This high time cost may be caused by sort operation in join operations or bad implementation of join algorithms. In the next experiment, time cost of sort operation is calculated with increased number of rows and columns.

4.8 Evaluating the Time Cost of Sort Operation

In this experiment, in order to evaluate the effect of increase in the size of the relation to the time cost of sort operation, two experiments have prepared.

In the first experiment, a relation with 100,000 rows is sorted for increased number of columns. Time cost of the sort operations are given in Table 4.9.

Table 4.9: Cost of sort operation on relation with 100,000 rows

Run#	Time Cost of 10 Columns	Time Cost of 100 Columns	Time Cost of 1000 Columns	Time Cost of 10.000 Columns
Run#1	0.032617	0.145594	0.052031	0.072830
Run#2	0.032289	0.046263	0.046263	0.071970
Run#3	0.032298	0.046339	0.151610	0.172845
Run#4	0.042534	0.045962	0.051631	0.072819
Run#5	0.042664	0.045766	0.051673	0.071926
Run#6	0.042612	0.045976	0.052146	0.072425
Run#7	0.032456	0.045949	0.051974	0.072792
Run#8	0.042566	0.045719	0.051775	0.072442
Run#9	0.042891	0.045801	0.052008	0.072437
Run#10	0.042565	0.045644	0.051281	0.072296
Average (second)	**0.038789**	**0.045963**	**0.051815**	**0.072501**

As seen in the Table 4.9, number of columns in the relation has a little effect to the time cost of sort operation.

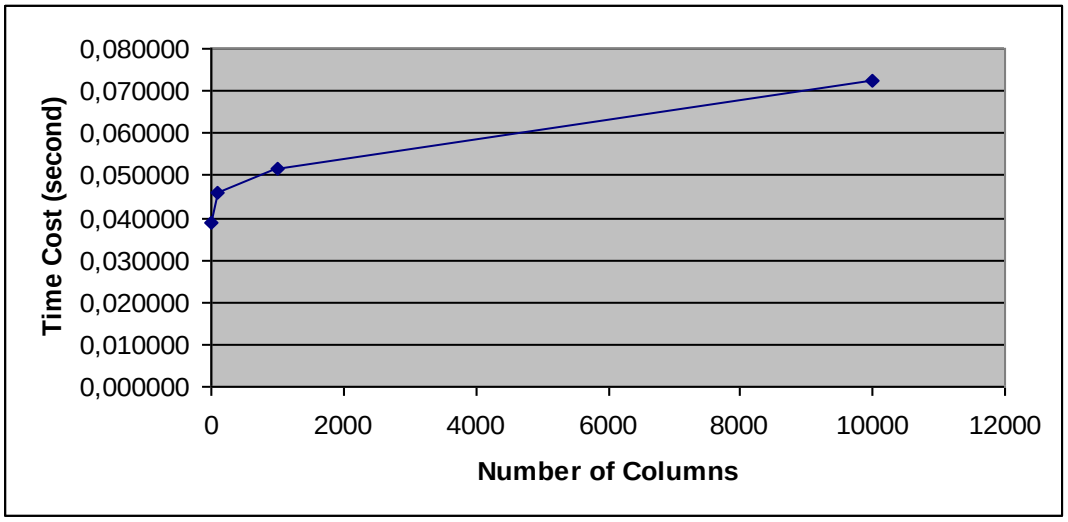


Figure 4.11: Relation sorting time cost variation with increasing columns

In Figure 4.11, the graphic of the data in the previous table is shown. The graphic shows that number of columns in the relation have logarithmic effect on the time cost of sort operation.

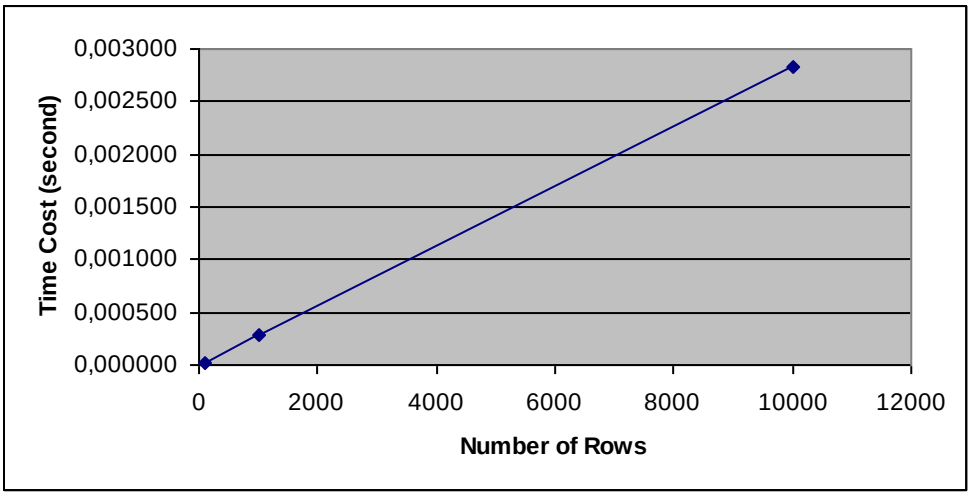
In the next experiment, a relation with 10 columns is sorted for increased number of rows. Time cost of the sort operations are given in table 4.10.

Table 4.10: Cost of sort operation on relation with 10 columns

Run #	Time Cost of 100 Rows	Time Cost of 1000 Rows	Time Cost of 10.000 Rows	Time Cost of 100.000 Rows	Time Cost of 1000.000 Rows	Time Cost of 10.000.000 Rows
Run#1	0,000014	0,000248	0,002414	0,032617	1,179475	17,246742
Run#2	0,000014	0,000249	0,002395	0,032289	1,079227	17,334322
Run#3	0,000014	0,000265	0,002415	0,032298	1,183822	17,184029
Run#4	0,000020	0,000322	0,003193	0,042534	0,580762	8,529009
Run#5	0,000019	0,000326	0,003204	0,042664	0,579851	8,537733
Run#6	0,000019	0,000325	0,003191	0,042612	0,579474	8,530770
Run#7	0,000014	0,000250	0,002398	0,032456	0,579637	8,557478
Run#8	0,000020	0,000321	0,003593	0,042566	0,580147	8,554834
Run#9	0,000023	0,000255	0,002610	0,042891	0,598005	8,520175
Run#10	0,000019	0,000330	0,003206	0,042565	0,580169	8,533297
Average (second)	**0,000017**	**0,000289**	**0,002829**	**0,038789**	**0,719659**	**10,708132**

As seen in the Table 4.10, effect of number of rows in the relation to the time cost of sort operation is close to linear. In fact, results this experiment proved that complexity of ***Quick Sort*** algorithm is **O(n log n)**.

(a)



(b)

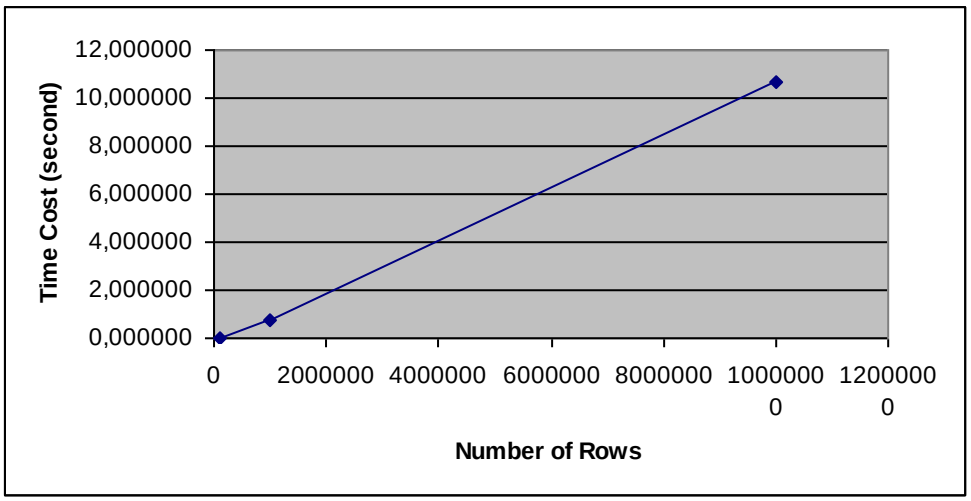


Figure 4.12: Relation sorting time cost variation with increasing rows

In Figure 4.12, results of the experiment are demonstrated in graphic form. It is easily seen that graphic is close to linear.

In this experiment, we learned that cost of sort operation increases linearly when the size of the relation increase. In the NGA example with 12.000.000 rows below, time

cost of sort operation must be approximately 10 seconds while join operation costs 1218 seconds.

As a result we can conclude that high increase in the time cost of join operation is caused by bad implementation of sort merge join. In the future work, this implementation may be improved.

CHAPTER 5

CONCLUSIONS AND THE FUTURE WORK

In this work we have developed some tools to run query execution plans on HPC cluster system using message passing paradigm. In this context, we also developed a program to generate sample relation files to use in our experiments.

In our experiments, we compared the performance of plans generated by genetic algorithms with optimal plans generated by exhaustive search algorithm. Our results have verified that optimal plans are better than those of genetic algorithms.

In the program processing execution plans, only the execution plans having left deep processing tree are executed. In the future work, program can be extended to handle bushy trees.

In the program processing execution plans, since all nodes are reading the whole relation at the beginning, all tables can be considered as fully replicated between nodes. In fact, since the plan must consider the existence of the data in the nodes of a plan, it is not important for the program processing the execution plans whether the data is replicated or not. As a result, replication is consideration of the query optimizer.

In the future work, data can be read from the disk allocated to the node instead of reading from the shared disk. In this case, by removing the reading whole data from the program, replication will be still supported by only copying the same data to the nodes planned to have replicas.

Horizontal fragmentation is not fully supported since there is no mechanism simulating union operation in our program. Horizontal fragmentation can be partially supported by dividing a table to n sub tables horizontally, preparing n! different tables containing all possible combination of fragments and defining these fragments as a separate table in configuration files. In the future work, union operation can be added to the program. In this case program can fully support horizontal fragmentation. Vertical fragmentation can be simulated by splitting the table to sub tables vertically and defining these tables in parameter files as if they are new table.

We evaluate the effect of join operation to the overall execution time and the share of the sort operation in join operation. We saw that most of the time cost belongs to join operations, although the sort operation is not so effected when the size of the relations and result set increase. Dramatic increase of the execution time values in bad execution plans are caused by the bad implementation of the selected join algorithm. In future work, join algorithm implementation can be improved.

We have found out that, when execution time of a process increases, variance of the time values of the same job also increases. This situation can be caused by process having long execution times may loose their priorities while processing.

By using our message transferring programs, we evaluated the message transfer time parameters and we calculated accurate cost formulas for data communication of the HPC environment of METU.

REFERENCES

[Almasi (1989)] G.S. Almasi, and A. Gottlieb, *Highly Parallel Computing*, Benjamin-Cummings publishers, Redwood City, CA, 1989.

[Apers (1988)] Peter M. G. APERS, *"Data Allocation in Distributed Database Systems"*, ACM Transactions on Database Systems, Vol.13, No.3, September 1988, pp. 263-304

[Bader (2001)] A. Bader and Robert Pennington, *"Cluster Computing: Applications"*, The International Journal of High Performance Computing, Vol. 15(2), May 2001, pp.181-185.

[Baker (2000)] Mark Baker, Amy Apon, Rajkumar Buyya, Hai Jin, *Cluster Computing and Applications*, 18.09.2000, http://www.gridbus.org/papers/-encyclopedia.pdf, last visited on 10 June 2009.

[baracuda (2009)] Seagate Barracuda 7200.11 series hard drive specifications, http://hornet.microdirect.co.uk/(40104)Seagate-1TB-hard-disk-drive-1000GB-Barracuda.aspx, last visited on 04 July 2009.

[Bayir (2007)] M.A. Bayir, I.H. Toroslu and A. Cosar, *"Genetic Algorithm for the Multiple-Query Optimization Problem,"* IEEE Transactions On Systems, Man, And Cybernetics-Part C: Applications and Reviews, Vol. 37, no. 1, January 2007, pp. 146-153.

[Bennett (1991)] Kristin Bennett, Michael C. Ferris, and Yannis Ioannidis, "*A Genetic Algorithm for Database Query Optimization*". Technical Report TR1004, University of Wisconsin, Madison, WI, 1991.

[Buyya (1999)] R. Buyya, *High Performance Cluster Computing: Architectures and Systems,* Vol.1, Prentice Hall PTR, NJ, USA, 1999.

[ceng (2009)] METU HPC System Specifications, http://hpc.ceng.metu.edu.tr/-sistem, last visited on 11 June 2009.

[Codd (1969)] E.F. Codd, *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*, IBM Research Report, 1969.

[Codd (1970)] E.F. Codd, *"A Relational Model of Data for Large Shared Data Banks",* Communications of the ACM, Vol. 13, June 1970, pp. 377-387.

[Culler (1999)] David E. Culler, Jaswinder Pal Singh, Anoop Gupta, *Parallel Computer Architecture - A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1999.

[Dale (1992)] Nell Dale and Chip Weems, *Introduction to Pascal and Structured Design* (*3rd Edition*), 1992.

[die (2009)] mpiexec - Linux man page, http://linux.die.net/man/1/mpiexec, last visited on 14 June 2009.

[Flynn (1966)] M.J. Flynn, "*Very high-speed computers",* In Proceedings of the IEEE 54, December 1966, pp. 1901-1909.

[Gen (2000)] Mitsuo Gen and Runwei Cheng, *Genetic Algorithms and Engineering Optimization*, John Wiley & Sons Inc, 2000.

[Hevner (1979)] Alan R. Hevner and S. Bing Yao, "*Query Processing in Distributed Database Systems*", IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 177-187

[Hughes (2003)] Cameron Hughes, Tracey Hughes, *Parallel and Distributed Programming Using C++*, Addison Wesley, August 25, 2003.

[Ioannidis (1990)] Y.E. Ioannidis and Y.C. Kang, *"Randomized Algorithms for optimizing large join queries,"* In Proc. ACM SIGMOD Int. Management of Data, Atlantic City, New Jersey, USA, May 1990, pp. 312-321.

[Kirkpatrick (1983)] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "*Optimization by Simulated Annealing"*, Science 220, 4598 (May 1983), pp 671-680.

[Kossmann (2000)] D. Kossmann, K. Stocker. "*Iterative dynamic programming: a new class of query optimization algorithms"*. ACM Transactions on Database Systems (TODS), Vol. 25, Issue 1, March 2000, pp. 43 - 82.

[March (1995)] Salvatore T. March and Sangkyu Rho, "*Allocating Data and Operations to Nodes in Distributed Database Design*", IEEE Transactions on Knowledge and Data Engineering, Vol.7 No.2, April 1995, pp. 305-317

[mhpcc (2009)] SP Parallel Programming Workshop, Message Passing Interface (MPI) http://www.mhpcc.edu/training/workshop/mpi/MAIN.html, last visited on 11 June 2009.

[Moerkotte (2006)] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products", In VLDB, September 2006, pp 930-941.

[mpi-forum (2008)] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 2.1, June 2008, http://www.mpi-forum.org/docs/docs.html, last visited on 11 June 2009.

[Nahar (1986)] S. Nahar, S. Sahni and E. Shragowitz, "*Simulated Annealing and Combinatorial Optimization",* in Proceedings of the 23rd Design Automation Conference, 1986, pp 293-299.

[open-mpi (2009)] Open MPI, http://www.open-mpi.org, last visited on 14 June 2009.

[Ozsu (1993)] M.T. Özsu and P. Valduriez, *"Distributed Data Management: Unsolved problems and new issues*", Distributed and Parallel Databases 1, 1993, pp. 137-165

[Ozsu (1999)] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems (2nd edition)*, Prentice-Hall, 1999.

[Ozsu (2007)] M.T. Ozsu and P. Valduriez, *Principles of Distributed DB Systems*, Prentice Hall, 01 July 2007, Chapter 8.

[Pfister (1998)] G.F Pfister, *In Search of Clusters (2nd Edition)*, Prentice Hall PTR, 1998.

[Ramakrishnan (2002)] Ragnu Ramakrishnan and Johannes Gehrke, *Database Management Systems(2nd Edition)*, 2002.

[Salzberg (1988)] Betty Salzberg, *File Structures: An Analytic Approach*, Prentice Hall, 1988.

[Sangkyu (1997)] R. Sangkyu, S.T. March, *"Optimizing distributed join queries: A genetic algorithm approach,"* Annals of Operations Research 71(1997), pp.199 – 228.

[Selinger (1979)] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "*Access path Selection in a Relational Database Management System"*. In Proc. of the ACM SIGMOD Conf. on Management of Data, Boston, USA, May 1979, pp. 23–34.

[Sevinç (2009)] Ender Sevinç, *Genetic Algorithms for Distributed Database Design and Distributed Database Query Optimization*, METU, October 2009.

[sissa (2009)] An Introduction to Portable Batch System (PBS), http://hpc.sissa.it/pbs/pbs.html, last visited on 12 June 2009.

[Steinbrunn (1993)] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper, "*Optimizing Join Orders*", Technical Report MIP9307, Faculty of Mathematic, University of Passau, Germany, 1993.

[ufl (2009)] Job Submission Queues, http://wiki.hpc.ufl.edu/index.php/-Job_Submission_Queues, 14 June 2009.

[wikipedia (2009)] Tuple, http://en.wikipedia.org/wiki/Tuple, last visited on 12 November 2009.

APPENDIX A

METU CENG HPC SYSTEM - HARDWARE PROPERTIES

1. Capacity:
Computational and Storage capacity of our system can be summarized as following:
46 x 2 = 92 CPUs
46 x 2 x 4 = 368 Cores
46 x 16 GB = 736 GB Memory
46 x 146 GB = 6.5 TB Local Disk (halved by RAID)
2 x 3 TB = 6 TB Common Storage Area (halved by RAID)

2. Servers:
HP ProLiant DL360 G5 (Used for system management)
- Intel Xeon 5110 Dual-Core CPU (1.60 GHz, 4 MB L2 Cache, 1066 MHz FSB)
- 2 GB Memory
- 2 x 72 GB = 144 GB Local Disk (RAID)

2 x HP ProLiant DL380 G5 (Used for storage management)
- 2 x Intel Xeon E5430 Quad-Core CPU (2.66 GHz, 12 MG L2 Cache, 1333 MHz FSB)
- 16 GB Memory
- 2 x 146 GB = 292 GB Local Disk (RAID)

3. Computation Nodes:
46 x HP ProLiant BL460c (One of them is used as a user interface)
- 2 x Intel Xeon E5430 Quad-Core CPU (2.66 GHz, 12 MP L2 Cache, 1333 MHz FSB)
- 16 GB Memory
- 2 x 146 GB = 292 GB Local Disk (RAID)

4. Storage:
4 x HP MSA60
- 12 x 250 GB = 3 TB
- 7200 rpm SATA disks

5. Network Switches:
2 x 3Com 4200G 24-port Gigabit Ethernet Switch
1 x Voltaire 9240D 24-port Infiniband Switch

APPENDIX B

METU CENG HPC SYSTEM - SOFTWARE INSTALLED ON SYSTEM

1. Scientific Linux v4.5 64-bit (Operating System):
Scientific Linux v4.5 is n open source free Linux distribution developed by Fermilab (Fermi National Accelerator Laboratory) and CERN (European Organization for Nuclear Research). Most of the cluster systems use Scientific Linux as operating system.

2. Lustre v1.6.4.2 (Parallel File System):
A file system developed for large clusters with Linux operating system.

3. Torque v2.1.9 (Resource Manager):
An open source spin-off PBS (Portable Batch System) from Cluster Resources. Its primary task is to allocate computational tasks, i.e., batch jobs, among the available computing resources.

4. Maui v3.2.6 (Job Scheduler):
Maui is an open source job scheduler for use on clusters and supercomputers. Maui is capable of supporting multiple scheduling policies, dynamic priorities, reservations, and fair share capabilities.

5. LDAP (User Authentication)
An application protocol for querying and modifying directory services running over TCP/IP.

6. Compilers
For C and C++

- gcc, g++ (C/C++) - v3.4.6
- icc - v10.1

For Fortran

- g77 (Fortran 77) - v3.4.6
- ifort - v10.1

For Java

- gcj (Java) - v3.4.6
- javac - v1.5

Debuggers

- gdb - v6.3
- idb - v10.1

7. Libraries:

MPI

- MVAPICH v0.9.9
- MVAPICH2 v0.9.8
- OpenMPI v1.2.4

Intel MKL v10.0 (Math Kernel Library)
Intel IPP v5.3 (Integrated Performance Primitives)
FFTW v3.1.2 (Fast Fourier Transform in the West)