GENETIC ALGORITHMS FOR DISTRIBUTED DATABASE DESIGN AND DISTRIBUTED DATABASE QUERY OPTIMIZATION

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

BY

ENDER SEVİNÇ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

OCTOBER 2009

Approval of the thesis:

GENETIC ALGORITHMS FOR DISTRIBUTED DATABASE DESIGN
AND DISTRIBUTED DATABASE QUERY OPTIMIZATION

submitted by **ENDER SEVİNÇ** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy 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 _____________________

Assoc.Prof. Dr. Halit OĞUZTÜZÜN
Computer Engineering Dept.,METU _____________________

Assoc. Prof. Dr. İbrahim Körpeoğlu
Computer Engineering Dept., Bilkent University _____________________

Date: 15 / 10 / 2009

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 : Ender Sevinç

Signature :

ABSTRACT

GENETIC ALGORITHMS FOR DISTRIBUTED DATABASE DESIGN AND DISTRIBUTED DATABASE QUERY OPTIMIZATION

Sevinç, Ender
Ph.D., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Ahmet Coşar

October 2009, 95 pages

The increasing performance of computers, reduced prices and ability to connect systems with low cost gigabit ethernet LAN and ATM WAN networks make distributed database systems an attractive research area. However, the complexity of distributed database query optimization is still a limiting factor. Optimal techniques, such as dynamic programming, used in centralized database query optimization are not feasible because of the increased problem size. The recently developed genetic algorithm (GA) based optimization techniques presents a promising alternative. We compared the best known GA with a random algorithm and showed that it achieves almost no improvement over the random search algorithm generating an equal number of random solutions. Then, we analyzed a set of possible GA parameters and determined that two-point truncate technique using GA gives the best results.

New mutation and crossover operators defined in our GA are experimentally analyzed within a synthetic distributed database having increasing the numbers of relations and nodes. The designed synthetic database replicated relations, but there was no horizontal/vertical fragmentation. We can translate a select-project-join query including a fragmented relation with N fragments into a corresponding query with N relations. Comparisons with optimal results found by exhaustive search are only 20% off the results produced by our new GA formulation showing a 50% improvement over the previously known GA based algorithm.

Keywords: Query optimization, Distributed database, Genetic algorithm, Mutation, Crossover.

ÖZ

DAĞINIK VERİTABANI IÇIN GENETİK ALGORİTMA
VE DAĞINIK VERİTABANI SORGU OPTİMİZASYONU

Sevinç, Ender
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ahmet Coşar

Ekim 2009, 95 sayfa

Bilgisayarların artan performansı, düşen fiyatlar, ucuz ATM geniş alan ağlarına ve gibabit Ethernet'li yerel alan ağlarına bağlanabilen sistemler dağınık veritabanı sistemlerini dikkat çekici kılmaktadır. Bununla birlikte, dağınık veritabanı sorgu optimizasyonu hala kısıtlayıcı bir faktördür. Merkezi veritabanı sorgu optimizasyonunda kullanılan dinamik programlama gibi en iyiyi bulan teknikler artan problem boyutu sebebiyle efektif değildir. Yeni geliştirilen genetik algoritma (GA) tabanlı optimizasyon teknikleri gelecek vaadeden bir alternatiftir. En iyi bilinen GA'yı rasgele çalışan bir teknikle kıyasladık ve bunun, neredeyse eşit sayıda üretilen rasgele çözümlerden daha iyiyi başaramadığının gösterdik. Sonrasında, GA'nın kullandığı parametre setini inceledik ve deneysel olarak, hangi parametrelerin bütün performansta etkili olduğunu gösterdik.

Bizim GA'da tanımlanan yeni mutasyon ve çaprazlama operatörleri deneysel olarak artan sayıda tabloların ve sitelerin olduğu suni dağınık veritabanında analiz edildi. Bu suni veritabanında tabloların kopyaları olmakla beraber, yatay/dikey bölümleme yoktu. N sayıda bölümlü bir tabloyu ihtive eden bir select-project-join sorgusu, N sayıda tabloyu ihtiva eden bir sorguya dönüştürülebilir. Tüm olasılıkların hesaplandığı en iyi sonuçlar, bizim yeni GA formülasyonumuzdan %20 daha iyiyken, önceden bilinen GA tabanlı çözümden %50 daha iyidir.

Anahtar Kelimeler: sorgu optimizasyonu, dağınık veritabani, genetic algoritma, mutasyon, çaprazlama

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 Prof. Dr. Adnan Yazıcı and Prof. Dr. İsmail Hakkı Toroslu for his suggestions and comments.

TABLE OF CONTENTS

ABSTRACT..iv
ÖZ...v
ACKNOWLEDGMENTS...vii
TABLE OF CONTENTS...viii
CHAPTER
1. INTRODUCTION 1
2. PREVIOUS WORKS…………………………..5
2.1 Distributed Database System 5
2.2 Heuristic-based Query Optimization..8
2.3 Genetic Algorithm Based Solutions……………………………………11
2.4 Exhaustive Search Methods……………………………………………18
2.4.1 IDP_1…………………………………………………………….21
2.5 Randomized Search Methods…………………………………………...24
2.5.1 Iterative Improvement (II) ………………………….................25
2.5.2 Simulated Annealing (SA) ………………………….................26
2.5.3 Two Phase Optimization (2PO) ……………………................27
3. DISTRIBUTED QUERY OPTIMIZATION …….....................................29
3.1 A New Genetic Algorithm Formulation...29
3.2 Chromosome Structure……………...30
3.3 Optimization model..32
3.4 Query Execution Model…………..33
3.5 New-Crossover…………………………………………………….40
3.6 New-Mutation……………………………………………………..45
4. EXPERIMENTAL SETUP AND RESULTS ……………...…………...51
4.1 Experimental Setup ..………..……………………………….......51
4.2 Experimental Results …………...53

5. DESIGN OF DISTRIBUTED DATABASE SCHEMA USING A GENETIC ALGORITHM……………………….……………………………………57
5.1 Distributed Database Schema Chromosome and Query Structure......58
5.2 Genetic algorithm for DDB Chromosome..60
5.2.1 Crossover ……………………….…………………………..60
5.2.2 Mutation ………….………..…………………………………62
5.3 System Structure ...………………...62
5.4 Distributed Database Schema Design 63
5.5 Experimental Setup and Results...68
5.5.1 Comparison of ESA,NGA and RGA ……………………........69
5.6 DDB Design Using Relation Clustering…..72
6. CONCLUSIONS...77
REFERENCES 79
APPENDICES
Appendix A: Test case 1 for DDB schema…………………….................82
Appendix B: Test case 2 for DDB schema …..83

LIST OF TABLES

TABLES

Table 1.1: Comparison of Query Optimization Algorithms ………….......................2
Table 2.1: Gene structures for sample query execution plans 15
Table 2.2: Implementation specific parameters for 2PO...28
Table 3.1: Parameter values for Genetic Algorithm..31
Table 3.2: Relation Schema……………………………….............................34
Table 3.3: Selection probability of a gene in New-mutation...................................35
Table 3.4: Types of Genetic Algorithms…………………….............................39
Table 5.1: Fragmentation of the relations…………………………….……………59
Table 5.2: Replication of the fragments/relations……………………….…………59
Table 5.3: Queries, frequencies and issuing nodes……………………………...62

LIST OF FIGURES

FIGURES

Figure 2.1: Distributed Database Environment ………..7
Figure 2.2: Dynamic Query Optimization Algorithm……......................................10
Figure 2.3: (Classic) Dynamic Programming Algorithm…......................................20
Figure 2.4: Iterative Dynamic Programming (IDP_1) with Block Size "k" 23
Figure 2.5: Iterative Improvement 26
Figure 2.6: Simulated Annealing 27
Figure 3.1: Chromosome Structure..31
Figure 3.2: Optimization model...32
Figure 3.3: Query Execution Plan..33
Figure 3.4: The performance of NGA for increasing crossover percentages ……..37
Figure 3.5: The performance of NGA for increasing mutation rates ……………...38
Figure 3.6: The performance of NGA for increasing initial population size ……...38
Figure 3.7: Solution quality based comparison of selection and crossover type combinations ……………………………………………………..39
Figure 3.8: Parent Chromosomes 40
Figure 3.9: Crossover Implementation (P1XP2) 42
Figure 3.10: Crossover Implementation (P2XP1) 43
Figure 3.11: Chromosome with condition numbers and costs of the genes……….46
Figure 4.1: File Descriptions………………..………………………..…………..52
Figure 4.2: The effect of increasing number of nodes……………………..……54
Figure 4.3: The effect of increasing number of relations…………………………..55
Figure 5.1: Chromosome Structure of a Distributed Database Schema ………….58
Figure 5.2: Crossover operation for a Distributed Database Sch. Chromosome…..61

Figure 5.3: Nested Genetic Algorithm for DDB Design …………..…………………65
Figure 5.4: The performance of DGA for increasing crossover percentages ……..66
Figure 5.5: The performance of DGA for increasing mutation rates ……………...67
Figure 5.6: The performance of DGA for increasing initial population size...……67
Figure 5.7: Optimization Times of DDB Design Algorithms ……………………..70
Figure 5.8: Query Execution Times of optimized DDB……………………….......71
Figure 5.9: CGA Pseudocode………………………………….……………...73
Figure 5.10: Query Execution Times of DGA and Clustered DGA……….……...74
Figure 5.11: Optimization Times of DGA and Clustered DGA…………………...75
Figure 5.12: Query Execution Times of DGA and Clustered DGA……………….76

CHAPTER 1

INTRODUCTION

Distributed database systems have been an active research area since mid 70s. The increasing performance, reduced workstation prices, ability to connect these systems with low cost gigabit ethernet networks makes distributed databases still very attractive for building modern high performance systems. However, the complexity of distributed database query optimization has been a limiting factor. Using centralized database query optimization techniques such as dynamic programming is not feasible because of the increased problem size due to a large number of input parameters (fragmentation, replication and network connections) in addition to the database query. The development of genetic algorithm (GA) based optimization techniques in 1990s presents a promising alternative methodology.

Optimizing queries is a major problem in distributed database systems, particularly when files are fragmented or replicated and copies stored at different nodes in the network. A distributed query optimization algorithm must select relations and determine how and where (at which node) those files will be processed, also deciding if a semijoin is also taken into consideration. Processing decisions must include both the files to be retrieved to the related site and the evaluation order of the conditions. We aim to extend the scope of distributed query optimization research by developing a model that, for the first time, includes heuristic algorithms

in a randomized approach. In this thesis, NGA which has been developed as a genetic algorithm based solution, quickly produces efficient query execution plans and reduces the optimization time of queries when compared to previously suggest genetic algorithms.

Table 1.1 : Comparison of Query Optimization Algorithms

Algorithms	Opt. Timing	Objective Function	Opt. Factors	Network Topology	Semi Joins	Stats*	Fragments
Dist. INGRES	Dynamic	Response Time or total cost	Msg. size, Proc. Cost	Point-to-point or LAN	No	1	Horizontal
R*	Static	Total cost	# msg., msg.size, IO, CPU	Point-to-point or LAN	No	1,2	No
SDD-1	Static	Total cost	msg.size	Point-to-point	Yes	1,3,4,5	No
GA	Static	Total cost	msg.size	Point-to-point	Yes	1,3,4,5	No
NGA	Static	Total cost	Msg.size, IO, CPU	Point-to-point	Yes	1,3,4,5	Horizontal

* 1=relation cardinality, 2=number of unique values per attribute, 3=join selectivity factor, 4= size of projection on each join attribute, 5= attribute size and tuple size.

One of the early distributed database management systems, SDD-1 [2], which was designed for slow wide area networks, made extensive use of semijoin operations. Later systems, such as R* [14, 23] and Distributed-INGRES [5], assumed faster networks and did not employ semijoins. Both R* and SDD-1 use static query optimization and they don't change the query execution plan during run-time, while Distributed-INGRES dynamically generates query execution plans at run-time using the available information (e.g. number of records returned in the intermediate results). R*, SDD-1 and Genetic algorithm (GA) [21] did not consider horizontal or vertical fragments, while Distributed-INGRES and our New Genetic Algorithm

(NGA) handles horizontal fragments. Except GA and NGA, none of the systems consider replication as seen in Table1.1.

In [21] a genetic algorithm based solution was given for the distributed database query optimization problem. Their model considered replication and semijoin operators, using the total cost of CPU processing, disk I/O and communication times for optimization. A comprehensive distributed database design approach using GA technique is presented in [15] which do not consider network latency or operation parallelism. In [10] this GA model was extended by including network latency and considering parallel processing in cost calculations. This extended model was used for designing efficient distributed databases that can make use of inherent parallelism in distributed databases.

Genetic algorithms may offer a powerful and domain-independent search method for a variety of tasks. But the applications for optimizing a distributed query have major drawbacks that are originating from strategy. Briefly in here, we shall try to solve this problem and make some adaptations for Genetic Algorithm with respect to the nature of the distributed query.

Since considering all possible alternatives for join sites, join order, replica selection, semijoins and join algorithm, causes distributed query optimization to take an exceptionally long time, genetic algorithm based solutions are very attractive. Using GA we can explore a very large search space considering all possible parameters while we can keep the search time low by maintaining and working on a relatively small set of alternative solutions and try to improve parts of a query execution plan where the execution costs are very high thus making it likely to find many good alternatives.

However, it is not a very good idea to expect even very simple optimization decisions to be randomly made by a GA. For example, if we know on which site a

join operation will be performed, it is very simple to find out which one of the replicas of an input relation would take the minimum time to be input to the join operation. Therefore, we need a mechanism to combine GA with other optimization techniques to perform a more effective search for finding better solutions in less time.

We show that a much more efficient GA search can be done by modifying the mutation operator in such a way that mutation of one part of a gene will also automatically cause another related part of the same gene to be modified accordingly such that these two parts of the same gene do not contain conflicting decisions made by each other. In fact, even in the formulation of GA given in [21] this approach is partially used since changing the join order of relations can generate invalid plans, where relations without a common join attribute can be placed next to each other. This problem has been taken care of by employing a so-called "inversion" operator instead of a random mutation operator. On the other hand, in our model we do not have such an additional artificial operator, but we handle this problem inside the mutation operator.

This thesis is organized as follows. In Section 2, we give previous work using heuristic algorithms and genetic algorithm based solutions for distributed database query optimization we explain previous works using heuristic and genetic algorithm based solutions for distributed query optimization. In Section 3, our genetic algorithm formulation is described. Section 4 presents the results of the experiments using a set of queries on synthetic distributed database schemas. Section 5, distributed database schema is designed by using our genetic algorithm and its performance is compared experimentally with that of exhaustive search algorithm. Finally, section 6 concludes this work and discusses possible future work.

CHAPTER 2

PREVIOUS WORKS

Earlier work on distributed database query optimization use several techniques which are listed below;

- sub-optimal greedy heuristics [19],
- genetic algorithm based solutions [6, 16],
- dynamic programming [7, 12, 22] and
- other randomized techniques [9].

These techniques will be discussed after the explanation of a distributed database system.

2.1. Distributed Database System

A distributed database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is defined as the software system that permits the management of the DDB and makes the distribution transparent to the users. We use the term distributed database system (DDBS) to refer to the combination of the DDB and the distributed DBMS. Assumptions regarding the system that underlie these definitions are:

Data is stored at a number of sites. Each site is assumed to logically consist of a single processor, resources included in a single system. Even if some sites are multiprocessor machines, the distributed DBMS is not concerned with the storage and management of data on this parallel machine.

- The processors at these sites are interconnected by a computer network rather than a multi-processor configuration. The important point here is the emphasis on loose interconnection between processors which have their own operating systems and operate independently. Even though shared-nothing multiprocessor architectures are quite similar to the loosely interconnected distributed systems, they have different issues to deal with (e.g., task allocation and migration, load balancing, etc.).

- The DDB is a database, not some "collection" of files that can be individually stored at each node of a computer network. This is also the same distinction between a DDB and a collection of files managed by a distributed file system. To form a DDB, distributed data should be logically related, where the relationship is defined according to some structural formalism, and access to data should be at a high level via a common interface. The typical formalism that is used for establishing the logical relationship is the relational model. In fact, most existing distributed database system research assumes a relational system.

- The system has the full functionality of a DBMS. It is neither a distributed file system nor a transaction processing system. Transaction processing is not only one type of distributed application, but it is also among the functions provided by a distributed DBMS. However, a distributed DBMS provides other functions such as query processing, structured organization of data, and so on that transaction processing systems do not necessarily deal with. [20]

Most of the existing distributed systems are built on top of local area networks in which each site is usually a single computer. The database is distributed across these sites such that each site typically manages a single local database in Figure 2.1. This is the type of system that we concentrate on for the most part of this study. However, next generation distributed DBMSs will be designed differently as a result of technological developments -especially the emergence of affordable multiprocessors and high-speed networks- the increasing use of database technology in application domains which are more complex than business data processing, and the wider adoption of client-server mode of computing accompanied by the standardization of the interface between the clients and the servers. Thus, the next generation distributed DBMS environment will include multiprocessor database servers connected to high speed networks which link them and other data repositories to client machines that run application code and participate in the execution of database requests.

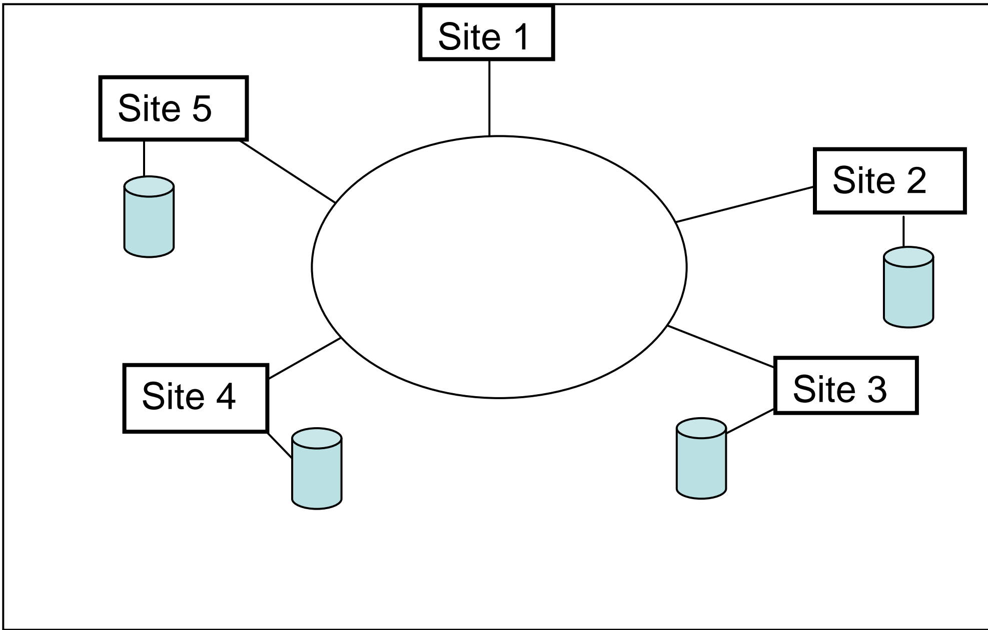


Figure 2.1: Distributed Database Environment [20]

A distributed DBMS as defined above is only one way of providing database management support for a distributed computing environment. A classification of

possible design alternatives along three dimensions are listed as autonomy, distribution, and heterogeneity.

- Autonomy refers to the distribution of control, and indicates the degree to which individual DBMSs can operate independently. Three types of autonomy are *tight integration*, *semi-autonomy* and *full autonomy* (or total isolation). In tightly integrated systems a single-image of the entire database is available to users who want to share the information which may reside in multiple databases. Partially autonomous systems consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data shareable. In totally isolated systems, the individual components are stand-alone DBMSs.

- Distribution dimension of the taxonomy deals with data. We consider two cases, namely, either data are physically distributed over multiple sites that communicate with each other over some form of communication medium or they are stored at only one site.

- Heterogeneity can occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of database systems relate to data models, query languages, interfaces, and transaction management protocols. The taxonomy classifies DBMSs as homogeneous or heterogeneous.[20]

2.2 Heuristic-based Query Optimization

The objective function of the algorithm is to minimize a combination of both the communication time and the response time. However, these two objectives may be conflicting. For instance, increasing communication time (by means of parallelism) may well decrease response time.

Thus, the function can give a greater weight to one or the other. This query optimization algorithm ignores the cost of transmitting the data to the result site. The algorithm also takes advantage of fragmentation, but only horizontal fragmentation is handled.
Since both general and broadcast networks are considered, the optimizer takes into account the network topology. In broadcast networks, the same data unit can be transmitted from one site to all the other sites in a single transfer, and the algorithm explicitly takes advantage of this capability. For example, broadcasting is used to replicate fragments and then to maximize the degree of parallelism.

The input to the algorithm is a query expressed in tuple relational calculus (in conjunctive normal form) and schema information (the network type, as well as the location and size of each fragment). This algorithm is executed by the site, called the master site, where the query is initiated.

One of the best known heuristic-based techniques used for distributed query optimization is the Distributed INGRES algorithm [5] which is derived from Centralized INGRES [18]. It uses a dynamic approach making optimization decisions at run-time in addition to pre-execution time. The *Dynamic Query Optimization Algorithm (D*-QOA)* [19], is given below:

In Figure 2.2, all monorelation operations (e.g., selection and projection) that can be detached (i.e. can be evaluated independently of other relations) are first processed locally [Step (1)]. Then, the reduction algorithm is applied to the original query [Step (2)]. Reduction is a technique that isolates all irreducible sub-queries and monorelation sub-queries by detachment. Monorelation sub-queries are ignored because they have already been processed in step (1). Thus, the REDUCE procedure produces a sequence of irreducible sub-queries $q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n$, with at most one join attribute (or join attributes for a composite key) in common between two consecutive sub-queries.[19]

based on the list of irreducible queries isolated in step (2) and the size of each fragment, the next sub-query, MRQ′, which has at least two variables, is chosen at step (3.1) and steps (3.2), (3.3), and (3.4) are applied to it. Steps (3.1) and (3.2) are discussed below. Step (3.2) selects the best strategy to process the query MRQ′. This strategy is described by a list of pairs (F, S), in which F is a fragment to transfer to the processing site, S. Step (3.3) transfers all the fragments to their processing sites.

```
Input: MRQ: multi-relation query
Output: result of the last multi-relation query
begin
   for each detachable OVQi in MRQ do
       run(OVQi){OVQ is a monorelation query}                    (1)
   endfor
   MRQ′ list ← REDUCE(MRQ)
   {MRQ replaced by n irreducible queries}                       (2)
   while (n<>0) do {n is the number of irreducible queries}     (3)
   {choose next irreducible query involving the smallest
   fragments}
       MRQ′ ← SELECT QUERY(MRQ′ list);                         (3.1)
   {determine fragments to transfer and processing site for MRQ′}
       Fragment-site-list← SELECT STRATEGY(MRQ′);              (3.2)
   {move the selected fragments to the selected sites}
       for each pair (F, S) in Fragment-site-list do
           move fragment F to site S                           (3.3)
       endfor
       execute MRQ′;                                           (3.4)
       n ← n - 1 {output is the result of the last MRQ′}
   endwhile
end. { Dynamic*-QOA }
```

Figure 2.2: Dynamic Query Optimization Algorithm [19]

Finally, step (3.4) executes the query MRQ′. If there are remaining sub-queries, the algorithm goes back to step (3) and performs the next iteration. Otherwise, it terminates. [19]

Optimization occurs in steps (3.1) and (3.2). The algorithm has produced sub-queries with several components and their dependency order (similar to the one given by a relational algebra tree). At step (3.1) a simple choice for the next sub-query is to take the next one having no predecessor and involving the smaller fragments. This minimizes the size of the intermediate result(s), hopefully generating a plan with minimal total query evaluation cost.

At step (3.2), the next optimization problem is to determine how to execute the sub-query by selecting the fragments that will be moved and the sites where the processing will take place. For an n-relation sub-query, fragments from n-1 relations must be moved to the site(s) of fragments of the remaining relation, R_p, and then replicated there. Also, the remaining relation may be further partitioned into k *"equalized"* fragments in order to increase parallelism. This method is called *fragment-and-replicate* and performs a substitution of fragments rather than of tuples. The selection of the remaining relation and of the number of processing sites k on which it should be partitioned is based on the objective function and the topology of the network. Replication is cheaper in broadcast networks than in point-to-point networks.

Furthermore, the choice of the number of processing sites involves a trade-off between response time and total time. A larger number of sites decreases response time (by parallel processing) but increases total time, in particular increasing communication costs [5].

2.3 Genetic Algorithm Based Solutions

A Genetic Algorithm (GA) is a general purpose search algorithm which applies principles of natural selection to a randomly generated pool of genetic populations consisting of chromosomes each representing a complete solution to the problem at hand, and using these initial solutions tries to evolve better solutions to the problem [6]. The basic idea is to maintain a population of chromosomes, which represent candidate solutions to the target problem that evolve over time through a process of

mating to merge two solution chromosomes to produce a new solution. Random mutations are also employed to ensure that a better (possibly optimal) solution not existing in the chromosome pool can also be randomly generated. Thus, finding an optimal solution will be guaranteed if the GA algorithm is run for a very long time. Each chromosome in the population is calculated an associated fitness value to choose competitive chromosomes that will form the next generation. Two operators used for this purpose are *crossover* and *mutation*.

Given a logical database (tables), a set of queries representing the update and retrieval requirements of a set of database users, and a network environment in which the system is to be implemented, the goal of a DDB design approach is to: (1) allocate data fragments to nodes in the network and (2) design query processing strategies for each query that most efficiently meet the identified needs. The first goal, termed data allocation, has been addressed by a number of researchers in a variety of network settings. All assume a fixed or extremely limited set of query processing strategies. The second goal, termed operation allocation or query optimization, has also been addressed by a number of researchers.

Each query has an origination node and a destination node at which the query results are required. Data may be accessed from and processed at different nodes within the network in an order determined by the database management system. If a retrieval query can be decomposed into independent sub queries, then judicious replication and placement of data can enable query-processing strategies that take advantage of parallelism [29] and data reduction by semi-join [3, 30] to reduce the response time for the query.

Of potential interest to parallelism in DDB design is query optimization in the context of multiprocessor computer architectures. Due to the proximity of processors and memories and the high-bandwidth bus architectures common in such systems, these models assume that communication time is insignificant compared to processor time and either ignore it completely or consider only the extra CPU

instructions stemming from communications. Hence, from the perspective of DDB in a high-speed wide area network where nodes are separated by hundreds of miles and latency is a significant component of response time, these models are of limited use.[10]

Genetic algorithms (GA) are a class of robust and efficient search methods based on the concept of adaptation in natural organisms [6, 8]. The basic concepts of GAs are:

- A representation of solutions, often in the form of bit strings, likened to genes in a living organism;
- A pool of solutions likened to a population or generation of living organisms, each having a genetic make-up;
- A notion of "fitness", which governs the selection of parents who will produce offspring in the next generation;
- Genetic operators, which derive the genetic make-up of an offspring from that of its parents (and possible random "mutation"); and
- A survival procedure that determines which parents and offspring are retained in the solution pool at each generation (often the survival procedure is "survival of the fittest").

A genetic algorithm begins by randomly generating an initial pool of solutions (i.e., the population). During each iteration (generation), the solutions in the pool are evaluated using some measure of fitness or performance. After evaluating the fitness of each solution in the pool, some of the solutions are selected to be parents. The probability of any solution being selected is typically proportional to its fitness.

Parents are paired and genetic operators applied to produce new solutions (offspring). A new generation is formed by selecting solutions (parents and offspring), typically based on their performance, so as to keep the pool size constant.

The genetic operators commonly used to produce offspring are crossover, mutation, and inversion. Crossover is the primary genetic operator. It operates on two solutions (parents) at a time and generates offspring by combining segments from each parent. A simple way to achieve crossover is to select a cut point at random and produce offspring by concatenating the segment of one parent to the left of the cut point with that of the other parent to the right of the cut point. Mutation generates a new solution by randomly modifying one or more gene values of an existing solution.

Mutation operator serves to guarantee that the probability of searching any subspace of the solution space is never zero. Inversion generates a new solution by reversing the gene order of an existing solution. Under inversion, two cut points are chosen at random and an offspring is produced by switching the end points of the middle segment.

As crossover produces new offspring, with solutions for parts of a problem, having good performance, begin to emerge in multiple solutions. Solutions with good performance typically contain a number of good DB schemas. Such solutions are more likely to be selected as parents than those with poor performance (which are expected not to contain as many good schemas). Thus, over successive iterations (generations), the number of good schemata represented in the pool tends to increase, the number of bad schemata tends to decrease and the average performance of the pool tends to improve.

A genetic algorithm stops when a given stopping condition is satisfied. Common stopping rules for genetic algorithms are maximum number of iterations and percent difference in the performance of the best and worst solutions. For real-time applications like distributed query optimization, a genetic algorithm can be stopped after a certain amount of time, or whenever the processor is ready to execute the query.

The gene structure for distributed database query optimization GA solutions consists of four parts, each corresponding to one of the four decisions in the distributed database query optimization model: [21]

- Selecting a replica of a relation
- Semijoin operations to reduce the communication cost
- Join site selection, and
- Join order.

Table 2.1 shows the gene structures for two sample execution plans for a distributed query having 3 join conditions in a 5-node distributed DBS having 4 relations. It also illustrates the effects of genetic operators on chromosomes.

Table 2.1: Gene structures for sample query execution plans [21]

Solution	Execution Plan	Copy Id.	Semijoin	Join Site	Join Order
1	Sample Plan 1	1 3 4 4	01 10 00	0 0 4	0 2 1
2	Sample Plan 2	**2 3 4 3**	**01 00 00**	**0 0 0**	**0 1 2**
3	Crossover 1,2	1 **3** 4 **3**	**01** 10 **00**	0 **0** 4	**0** 2 **1**
4	Mutation 3	1 3 4 4	11 10 00	1 0 4	0 2 1
5	Inversion 3	1 3 4 3	01 10 00	0 0 4	2 0 1

The third column, "***Copy Id***", represents the site number of the chosen replica for the input base files (relations). For example, the value "3" in "1 3 4 4" means that the second file (R2) will be taken from Site3. The "Semijoin" column identifies the type of semijoin operation to be employed on the inputs of three join operations. "00" means no semijoin operation will be performed on the input relations, while "10" and "01" represent that left and right join inputs, respectively, will be subjected to semijoin operations for reducing communication time, "11" is not an allowed value. The selection of the site where the join operation will be performed is given in the

"***Join Site***" column. For example the value "0 0 4" means the 1st and 2nd join operations will be performed at site S_0 and 3rd join operation at site S_4. The traditional problem of ordering the execution of joins is given by the last column where a permutation of the join values (0, 1 and 2) is given. The value "0 2 1" for join order means 1st join J_0 will be performed, then result of J_0 will be input to join J_2 and finally the result calculated so far will be input to J_1. The join attributes for individual join operations are given in the query input and is the same for all chromosomes.

This genetic algorithm uses uniform crossover [25] to combine file copy selections and a random mutation operator. In uniform crossover, the child inherits a value for each gene position from one or the other parent with probability 0.5 (i.e., randomly). Solution 3 illustrates a possible result of applying the uniform crossover operator to solutions 1 and 2. The first and third file sites were (randomly) taken from solution 1, the second and fourth from solution 2 (genes from solution 2 are shown in bold). Solution 4 shows a mutation of Solution 3 where R004 (4th file/relation) is randomly selected to be mutated. The mutation which is shown as underlined randomly changes its selected replica location from site S_3 to site S_4 (it must be mutated to a feasible site where a replica of the corresponding relation exists). A typical mutation probability (0.005) is used as suggested in the literature [6].

Semijoin operators are represented by a pair of bits, one pair for each join. If an elementary semijoin is to be performed, the value of the bit corresponding to the reducer file is set to 1, otherwise it is 0. As illustrated in the Semijoin column of Table 2.1, the semijoin strategy for solution 1 is "01 10 00" specifying the semijoin R2 → R1 and R2→ R3. A uniform crossover operator and a standard mutation operator are used to generate new semijoin solutions (again constrained to ensure feasibility). Again, solution 3 illustrates a possible result of applying the uniform crossover operator to solutions 1 and 2. In solution 3, values shown in bold come from solution 2 and the others come from solution 1.The semijoin strategy for join J_1 is taken from solution 1, those for joins J_0 and J_2 are taken from solution 2.

Join site decisions are represented by a vector with a value for each join in the query. Each value in the vector represents the site at which the join is performed. As illustrated in the Join Site column of Table 2.1, the join sites for solution 1 are given by 0 0 4, indicating that J_0 and J_1 are performed at site S_0, and J_2 is performed at site S_4. Again, a uniform crossover operator and a standard mutation operator are used to generate new join site solutions. Since join operations can be performed at any site, feasibility is not an issue.

Join order decisions are represented as a list of joins where the sequence indicates the order in which joins are performed. Alternatively, join order decisions can be represented as a list of files, where the sequence indicates the order in which files are joined. However, this type of representation cannot represent bushy query plans and plans for cyclic queries. As illustrated in the ***Join Order*** column of Table 2.1, the join order for solution 1 is given by 0 2 1, indicating that J_0 is performed first, J_2 next, and J_1 last. Standard crossover operators are not viable for this type of representation as they are likely to generate illegal solutions. There are several crossover operators that always produce legal solutions for this type of representation. They include edge recombination [28] and uniform order crossover [4]. This genetic algorithm employs uniform order crossover which outperformed edge recombination in our experiments. In a uniform order crossover operator, gene positions for which a child will inherit values from the first parent are randomly determined. Then values for the rest of the gene positions are determined based on the gene value order in the second parent. To illustrate how a uniform order crossover operator works, consider the following join orders:

2 1 3 0 (J_2 J_1 J_3 J_0),

1 3 0 2 (J_1 J_3 J_0 J_2).

Suppose that the second and fourth gene positions are inherited from the first parent. We then have the following partial solution: –1 – 0 (J_1 is performed second and J_0 is performed last). In the second parent, the order of the values not present in the

partial solution is 3 2 (J_3 is performed before J_2), thus we have 3 1 2 0. Solution 3 in Table 2.1 illustrates a possible result of applying the uniform order crossover operator to solutions 1 and 2. The second gene value is (randomly) inherited from solution 1 and the rest of the gene values are determined by the second parent.

Standard mutation operators frequently generate illegal solutions for this type of representation. Thus, an inversion operator is used instead of a mutation operator to Inversion generates a new solution by reversing the gene order of an existing solution. Under inversion, two cut points are chosen at random and an offspring is produced by switching the end points of the middle segment. Since standard mutation operators frequently generate illegal solutions for this type of representation, an inversion operator is used instead of a mutation operator to incorporate randomness. Solution 5 in Table 2.1 illustrates a possible result of applying the inversion operator to Solution 3. The order of the first two joins is reversed from $\langle J_0, J_2 \rangle$ to $\langle J_2, J_0 \rangle$.

Since GA's objective is to minimize the query processing cost, the cost function is mapped to the following fitness function to calculate fitness for each solution, S:

$$\text{Fitness (S) : 1- cost (S) / } k, \qquad \textbf{(2.1)}$$

where k is a normalizing constant [21].

2.4 Exhaustive Search Methods

Researchers and practitioners have been interested in distributed database systems since the 1970s. At that time, the main focus was on supporting distributed data management for large corporations and organizations that kept their data at different offices or subsidiaries. In some aspects, the early distributed database systems were ahead of their time. First, communication technology was not stable enough to ship megabytes of data as required for these systems. Second, large businesses somehow

managed to survive without sophisticated distributed database technology by sending tapes, diskettes, or just paper to exchange data between their offices.

A large number of alternative enumeration algorithms have been proposed in the literature; Steinbrunn et al. [24] contains a good overview, and Kossmann and Stocker [12] evaluate the most important algorithms for distributed database systems. In the following, dynamic programming is described. This algorithm is used in almost all commercial database products, and it was pioneered in IBM's System R project [22]. The advantage of dynamic programming is that it produces the best possible plans if the cost model is sufficiently accurate. The disadvantage of this algorithm is that it has exponential time and space complexity so that it is not viable for complex queries; in particular, in a distributed system, the complexity of dynamic programming is prohibitive for many queries. An extension of the dynamic programming algorithm is known as Iterative DP. This extended algorithm is adaptive and produces as good plans as basic dynamic programming for simple queries and "as good as possible plans" for complex queries for which dynamic programming isn't viable. [12]

We will first describe the classic dynamic programming algorithm [22], which is used in most commercial state-of-the-art optimizers today, then Iterative dynamic programming (IDP) [12] will be described. Figure 2.3 gives the classical dynamic programming algorithm. The algorithm works in a bottom-up way as follows;

First of all access-plans for all Tables R_i are generated (Lines 1 to 4). Such plans consist of operators like *table_scan*(R_i) or *index_scan*(R_i). They are inserted in a table-structure 'optPlan' which is set-indexed. This phase is called *access-root phase*. After that, in the following *join-root phase* (Lines 5 to 13) building-blocks of ascending size are produced. First 2-way joins by calling the joinPlans function on two access-plans, then 3-way join plans by combinations of all 2-way join plans and access-plans and so on up to n-way join plans.

```
Input:  Select-project-join (SPJ) query q on relations
        R1,…….. .,Rn
Output: A query plan for q
1:  for i = 1 to n do {
2:      optPlan({Ri}) = accessPlans(Ri)
3:      prunePlans(optPlan({Ri}))
4:  }
5:  for i = 2 to n do
6:      for all S ⊆ {R1,…….. .,Rn} such that |S| = i do {
7:          optPlan(S) = Ø;
8:          for all O ⊂ S do {
9:              optPlan(S) = optPlan(S) ∪
                      joinPlans(optPlan(O),optPlan(S -
                              O))
10:             prunePlans(optPlan(S))
11:         }
12:     }
13: return optPlan({R1,…….. .,Rn})
```

Figure 2.3: (Classic) Dynamic Programming Algorithm

The advantage of dynamic programming in contrast to full enumeration is that it discards inferior building blocks after every step. This approach is called *pruning*. A (sub-) plan *A* is inferior to Plan *B*, if it is in relevant plan parameters at most as good but in at least one property worse than *B*. Only the best (comparable) plans are retained in optPlan, such that only these plans will be considered as building-blocks in later steps. If two plans are incomparable, both are retained in optPlan. For example, *A sort-merge-join B* and *A hash-join B* are incomparable if the *sort-merge-join* is more expensive than the *hash-join* because the *sort-merge-join* produces ordered results which might help to reduce the cost of later operations. Pruning should be carried out as early as possible to avoid the unnecessary enumeration of inferior plans. In the algorithm of Figure 2.3 all *bushy* plans are considered as an extension to the originally proposed left-deep variant by Selinger [22]; most commercial query optimizers that are based on dynamic programming do the same thing. The complexity of this algorithm is $O(3^n)$ [17, 27].

It has been shown in [17, 27] that the time complexity of dynamic programming is $O(3^n)$ and the space complexity is $O(2^n)$ in a centralized system. In the following, in a distributed system the time complexity of dynamic programming is $O(s^3 * 3^n)$ and the space complexity is $O(s * 2^n + s^3)$, where *s* is the number of sites at which a copy of at least one of the tables involved in the query is stored plus the site at which the query results need to be returned. s, thus, is a variable whose value depends on the query and which might be smaller or larger than n, depending on the number of replicas of the tables used in the query.

The time complexity of dynamic programming is $O(s^3 * 3^n)$ in a distributed database system.

In [12] Iterative Dynamic Programming (IDP) was introduced with two versions. It's claimed to be a new class of query optimization algorithms that is based on iteratively applying dynamic programming and a combination of dynamic programming and the greedy algorithm. In all, eight different IDP variants have been shown to differ in three ways:

(1) when an iteration takes place (IDP_1 vs. IDP_2),

(2) the size of the building blocks generated in every iteration (standard vs. balanced), and

(3) the number of building blocks produced in every iteration (*bestPlan* vs. *bestRow*).

2.4.1 IDP_1

"IDP_1-standard-bestPlan" works essentially in the same way as dynamic programming with the only difference that IDP_1 respects that the resources (e.g., main memory) of a machine are limited or that a user or application program might want to limit the time spent for query optimization.

To see how IDP_1 does this it is assumed that a machine has enough memory to keep all access plans, *2-way, 3-way, . . . , k-way* join plans (after pruning) for a query with

exactly *n* tables., and also n > k. In such a situation, dynamic programming would crash or be the cause of severe paging of the operating system when it starts to consider *(k + 1)-way* join plans because at this point the machine's memory is exhausted. IDP1, on the other hand, would generate access plans and all 2-way, 3-way, . . . , k-way join plans like dynamic programming, but rather than starting to generate (k + 1)-way join plans, IDP_1 would break at this point, select one of the k-way join plans, discard all other access and join plans that involve one of the tables of the selected plan, and restart in order to build (k + 1)-way, (k + 2)-way, . . . join plans using the selected plan as a building block. That is, just like the greedy algorithm breaks after two-way join plans have been enumerated, IDP1 breaks after k-way join plans have been enumerate, the memory is full, or a time-out is hit.

For k = 2, IDP_1 behaves exactly like the greedy algorithm and for k = n, IDP_1 behaves like dynamic programming. For 2 < k < n, the complexity of IDP_1 is that the IDP_1 algorithm of Figure 2.4 has polynomial time and space complexity of the order of O ($s^3 * n^k$). In this analysis, k (the size of the building blocks) is considered to be constant, and *s (the number of sites)* and *n (the number of tables)* are the variables which depend on the query to optimize.

```
Input: SPJ query q on relations R1,…..,Rn, maximum block size k
Output: A query plan for q
1:   for i = 1 to n do {
2:        optPlan(fRig) = accessPlans(Ri)
3:        prunePlans(optPlan({Ri}))
4:   }
5:   toDo = { R1,…..,Rn}
6:   while |toDo| > 1 do f
7:        k = min {k, |toDo|}
8:        for i = 2 to k do {
9:             for all S ⊆ toDo such that |S| = i do {
10:                   optPlan(S) = Ø;
11:                   for all O ⊂ S do {
12:                        optPlan(S) = optPlan(S)∪
                           joinPlans(optPlan(O), optPlan(S - O))
13:                        prunePlans(optPlan(S))
14:            }
15:       }
16:  }
17:  find P, V with P ∈ optPlan(V), V ⊆ toDo, |V|=k such that
     eval(P) = min{eval(P’) | P’∈ optPlan(W), W ⊆ toDo, |W| = k }
18:  generate new symbol: T
19:  optPlan({T}) = {P}
20:  toDo = toDo - V ∪ {T}
21:  for all O ⊆ V do delete(optPlan(O))
22:  }
23: finalizePlans(optPlan(toDo))
24: prunePlans(optPlan(toDo))
25: return optPlan(toDo)
```

Figure 2.4: Iterative Dynamic Programming (IDP_1) with Block Size "k" [12]

In a centralized database system, the time complexity of the IDP_1 algorithm (Figure 2.4) is claimed to be the order of $O(n^k)$ for $2 < k < n$. Time Complexity of IDP_1 in a distributed database system, the time complexity of the IDP_1 algorithm is of the order of $O(s^3 * n^k)$ for $2 < k < n$, where the space complexity is the space complexity of the IDP_1 algorithm is of the order of $O(s * n^k + s^3)$ for $2 < k < n$.

In the same study also another variant IDP_2 is introduced. In fact, the figure shows the *"standard-bestPlan"* variant of this algorithm. This basic variant of the

algorithm is a similar idea to apply dynamic programming in order to re-optimize certain parts of a plan has also been proposed in form of the *bushhawk* algorithm. We'll not go in detail for this variant.

Comparing IDP_1 and IDP_2, it is observed that the mechanisms are essentially the same: both algorithms apply heuristics (i.e., plan evaluation functions) in order to select sub-plans, and both algorithms make use of dynamic programming. Also, both algorithms can (fairly) easily be integrated into an existing optimizer which is based on dynamic programming. The difference between the two algorithms is that IDP_2 makes heuristic decisions and applies dynamic programming after that; IDP_1, on the other hand, starts with dynamic programming and makes heuristic decisions only when it is necessary. In other words, IDP_1 is adaptive and *k* is an optional parameter of the algorithm which may or may not be set by a user in order to limit the optimization time. Another difference is that IDP_2 has lower asymptotic complexity than IDP_1.

In the study, eight different IDP variants are identified. The experiments showed that what they call as "*balanced*" IDP with "*bestRow*" should be used. No clear winner could be identified between the basic algorithm variants IDP_1 and IDP2. The overall picture is that IDP_2 is faster than IDP_1 and produces as good plans as IDP_1. On the negative side, however, IDP_2 requires a-priori tuning by a user or system administrator (i.e., setting of the *k* parameter) whereas IDP_1 is adaptive. The conclusion is that both IDP_1 and IDP_2 should be combined. That is, the optimizer should use IDP_2 with some default value of *k* in its main loop (e.g., $k = 15$), and the optimizer should employ IDP_1 (rather than dynamic programming) whenever it optimizes a building block. This way, the optimizer will always safely generate plans because IDP1 is adaptive, and users can overwrite the default value of *k* in order to use IDP_2 to speed-up the optimization process [12].

2.5 Randomized Search Methods

Since exhaustive search algorithms used commonly by current optimizers are inadequate for large queries, new query optimization algorithms are developed.

Randomized algorithms are successful samples in this area. Two such algorithms, Simulated Annealing [11] and Iterative Improvement [16] are the best known. Then Two Phase Optimization technique has been proposed for the optimization of large queries [9].

Randomized algorithms usually perform *random walks* in the state space via a series of moves. The states that can be reached m one move from a state '*S'* are called the neighbors of '*S'*. A move is called *uphill (downhill)*, if the cost of the source state '*S'* lower (higher) than the cost of the destination state. A state is a *local minimum* if in all paths starting at that state any downhill move comes after at least one uphill move. A state is a *global minimum* if it has the lowest cost among all states. A state is on a *plateau* if it has no lower cost neighbor and yet it can reach lower cost states without uphill moves.

2.5.1. Iterative Improvement (II)

The generic Iterative Improvement (II) algorithm is presented in Figure 2.5. The inner loop of II is called a local optimization. A local optimization starts at a random state and improves the solution by repeatedly accepting random downhill moves until it reaches a local minimum. II repeats these local optimizations until a *stopping condition* is met, at which point it returns the local minimum with the lowest cost found.

As time approaches infinity, the probability that II will visit the global minimum increases. However, given a finite amount of time, the algorithm's performance depends on the characteristics of the cost function over the state space and the connectivity of the latter as determined by the neighbors of each state.

```
procedure II() {
   minS = S∞;
   while not (stopping_condition)
      do {
       S = random state,
       while not (local_minimum(S))
       do {
          S' = random state in neighbors(S),
          if cost(S') < cost(S) then S = S',
          }
      if cost(S) < cost(minS) then minS = S,
      }
      return(minS),
}
```

Figure 2.5 : Iterative Improvement

2.5.2 Simulated Annealing (SA)

A local optimization in Iterative Improvement performs only downhill moves. In contrast Simulated Annealing (SA) does accept uphill moves with some probability, trying to avoid being caught in a high cost local minimum. The genetic algorithm, Simulated Annealing, is shown in Figure 2.6. The inner loop of SA is called a *stage*. Each stage is performed under a fixed value of a parameter T, called *temperature*, which controls the probability of accepting uphill moves. The probability is equal to $e^{-\Delta C/T}$, where ΔC is the difference between the cost of the new state and that of the original one. Thus, the probability of accepting an uphill move is a monotonically increasing function of the temperature and a monotonically decreasing function of the cost difference Each stage ends when the algorithm is considered to have reached an *equilibrium* Then, the temperature is reduced according to some function and another stage begins, i.e., the temperature is lowered as time passes The algorithm stops when it's considered to be *frozen*, i.e., when the temperature is equal to zero. It has been shown theoretically that, under certain conditions satisfied that by some parameters of the algorithm, as temperature approaches to zero, the algorithm converges to the *global minimum*.

A minimum state of another algorithm is selected as initial, S_0. Then SA is converged to this stage which is found to be as the minimum.

```
procedure SA() {
  S=S0,
  T=T0,
  minS = S;
  while not (frozen)
     do {
     while not (equilibrium)
      do {
        S' = random state neighbors(S),
        ΔC= cost(S') - cost(S),
        If (ΔC<=0) then S = S',
        If (ΔC>0) then S = S' with probability e^(-ΔC/T),
        if cost(S) < cost(minS) then minS = S,
      }
      T = reduce(T),
     }
return(minS),
}
```

Figure 2.6 : Simulated Annealing

2.5.3 Two Phase Optimization (2PO)

Two Phase Optimization (2PO) algorithm, a combination of II and SA will be introduced. As the name suggests, 2PO can be divided into two phases. In phase 1, II is run for a small period of time, i.e., a few local optimizations are performed. Then the output of that phase, which is found as the best local minimum found will be the initial state of the next phase. In phase 2, SA is run with a low initial temperature. Intuitively, the algorithm chooses a local minimum and then searches the area around it, still being able to move in and out of local minima, but practically unable to climb up very high hills. Thus, 2P0 is appropriate when such an ability is not necessary for proper optimization, which is the case for select-project-join query optimizations.

The neighbors of a state, which is a join-processing tree (e.g. a plan), are determined by a set of transformation rules. Each neighbor is the result of applying one of these rules to some internal nodes of the original plan once, replacing them by some new nodes, and usually leaving the rest of the nodes of the plan unchanged. There are known to be several sets of transformation rules.

For II, SA and 2PO, some specific parameters are listed in Table 2.2.

Table 2.2: Implementation specific parameters for 2PO [9]

Parameter	Value
stopping_condition(II phase)	10 local optimizations
Initial state S_0 (SA phase)	minS of II phase
Initial temperature T_0 (SA phase)	0.1*cost(S_0)

The parameters in Table 2.2 explain the definition of a local minimum for II. A state that satisfies the above operational definition is called *r-local minimum*. Every local minimum is an *r-local minimum*, but the converse is not true. *r-local minimum* as the stopping criterion for a local optimization implies that some downhill moves may be occasionally missed and a state may be falsely considered as a local minimum. But it is claimed that the saving in execution time by using this approximation outweighs the potential misses of real local minima. As the result, the performance of Two Phase Optimization algorithm is superior to those of the other algorithms.

CHAPTER 3

DISTRIBUTED QUERY OPTIMIZATION

3.1 A New Genetic Algorithm Formulation

Our goal in this work is to develop a genetic algorithm based heuristic for the optimization of distributed queries and we present a *New Genetic Algorithm (**NGA**)* and evaluate 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 we first implemented an Exhaustive Search Algorithm (ESA) which takes a very long 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 we have implemented a second algorithm 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***" and shown in the experiments in the next section.

As mentioned before there is already a GA based algorithm proposed in [21]. We will call it *Rho's Genetic Algorithm* (**GA**) throughout this study. As discussed in section 2.3, 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.

3.2 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 3.1.

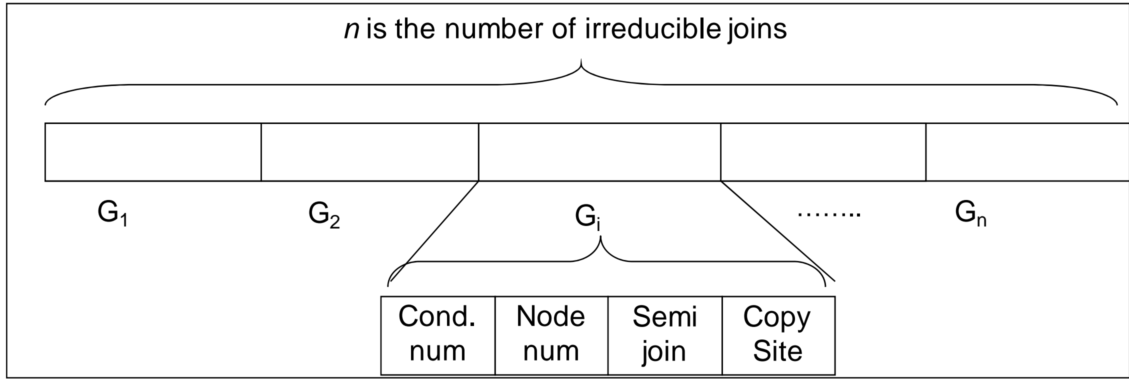


Figure 3.1: Chromosome Structure

The chromosome structure of a query is shown in Figure 3.1. 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 [1]. Rest of the parameters for our GA is listed in Table 3.1.

Table 3.1: 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%)

3.3 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 3.2.

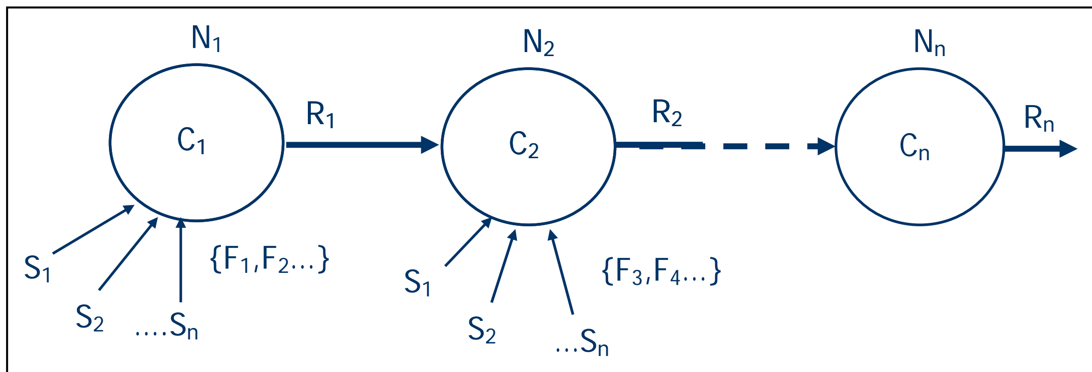


Figure 3.2: Optimization model

Each condition, $C_i \in C$, has input fragments (F_n) of relations at various sites, S_n. Then each condition is evaluated at $N_i \in N$, 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 [19].

3.4 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 3.3. Dashed lines denote semijoin operations.

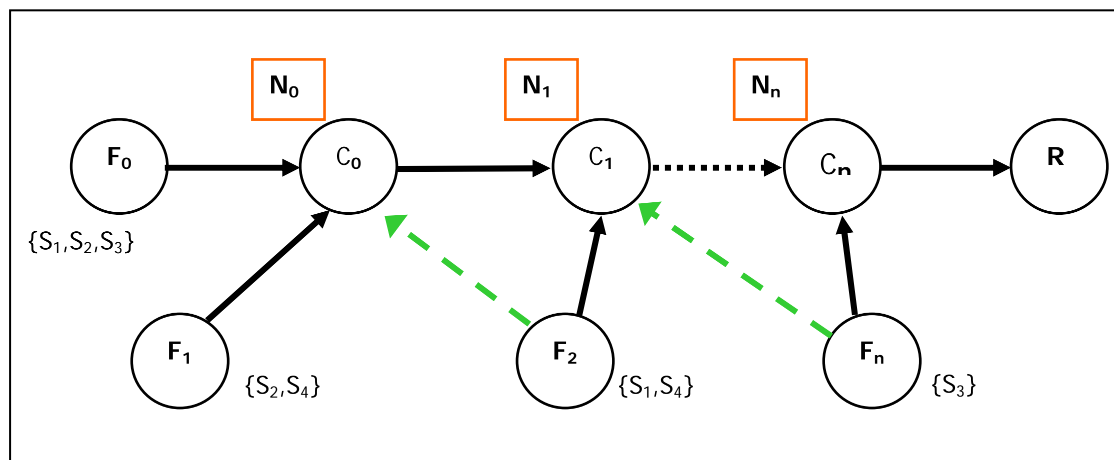


Figure 3.3 : Query Execution Plan

The cost of an execution plan, denoted by *Cost(P)* is calculated by using Formula 3.1 and 3.2 below.

$$\textbf{Cost (P)} = \sum_{i=0..n} comm_cost(Rel_i, S_{ki}) + \sum_{j=0..m} Proc_cost(C_j) + \sum_{k=0..m} comm_cost(R_k) \quad \textbf{(3.1)}$$

$$\textbf{Comm_cost (Rel}_\textbf{i}\textbf{,S}_\textbf{k}\textbf{)} = \max_{j=0..NF_i} | (comm_cost(F_{ij},S_k), \text{ where } Rel_i \text{ has } NF_i \text{ fragm.} \quad \textbf{(3.2)}$$

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 ith sub query. All the calculations are done due to related formulas. Test bed has been explained in Table 3.1, 3.2 and 3.3.

Table 3.2: Relation Schema

Relation ID	Attributes
Rel_1000	(<u>attr1</u>, attr2, attr3, attr4, attr5)
Rel_1001	(attr1, <u>attr6</u>, 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 3.3 : 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 3.3, BNL is evaluated;

$$\text{Local Processing Cost (Proc_cost(Cj))} = N + M * \left\lceil \frac{N}{B-2} \right\rceil \quad \textbf{(3.3)}$$

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 3.4 is used;

$$\text{Local Processing Cost (Proc_cost(Cj))} = M + N \quad \textbf{(3.4)}$$

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 3.3, 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 3.4, Figure 3.5, and Figure 3.6 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.

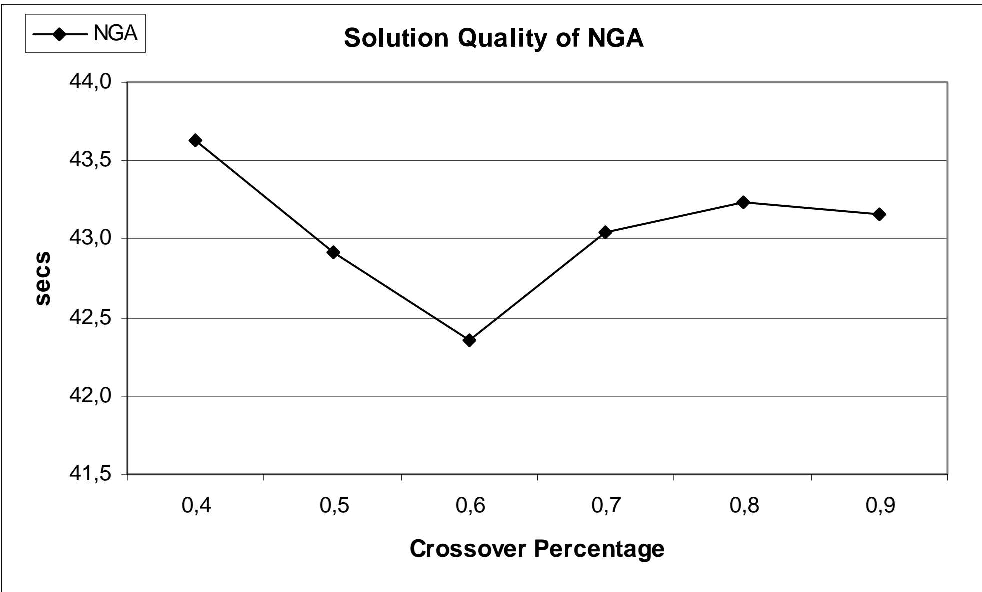


Figure 3.4 : The performance of NGA for increasing crossover percentages

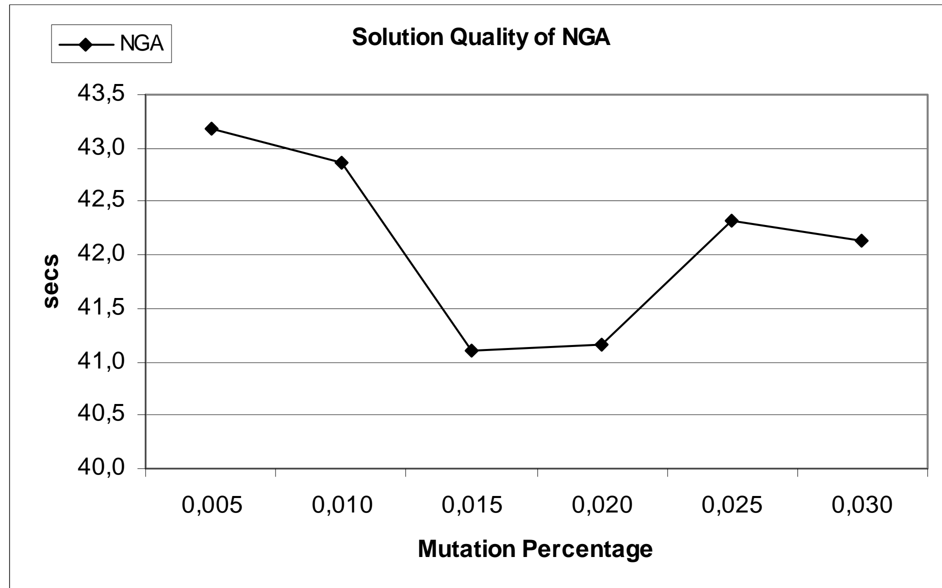


Figure 3.5 : The performance of NGA for increasing mutation rates

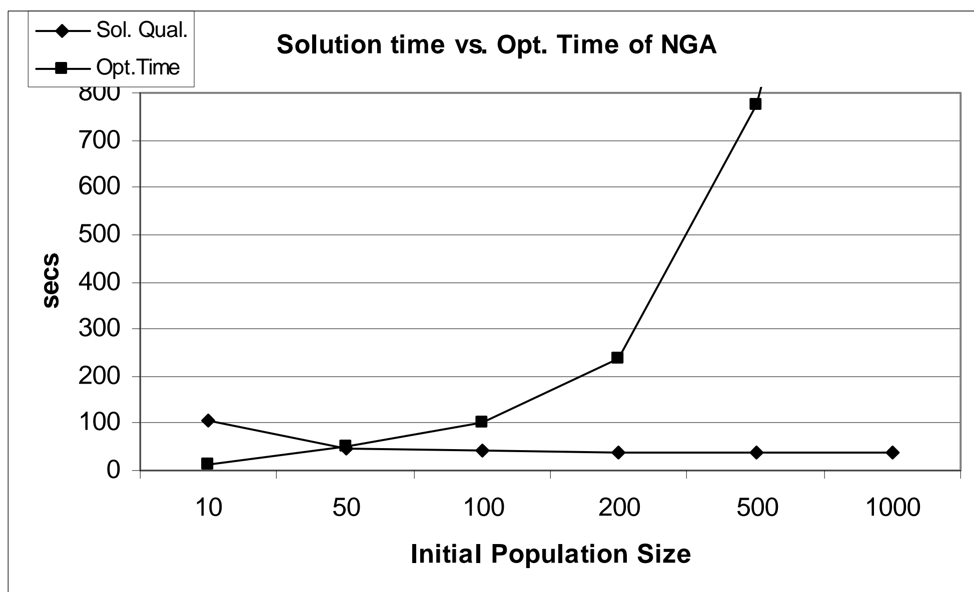


Figure 3.6 : The performance of NGA for increasing initial population size

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 3.4: 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 3.4, and compared them experimentally. The results are shown in Figure 3.7;

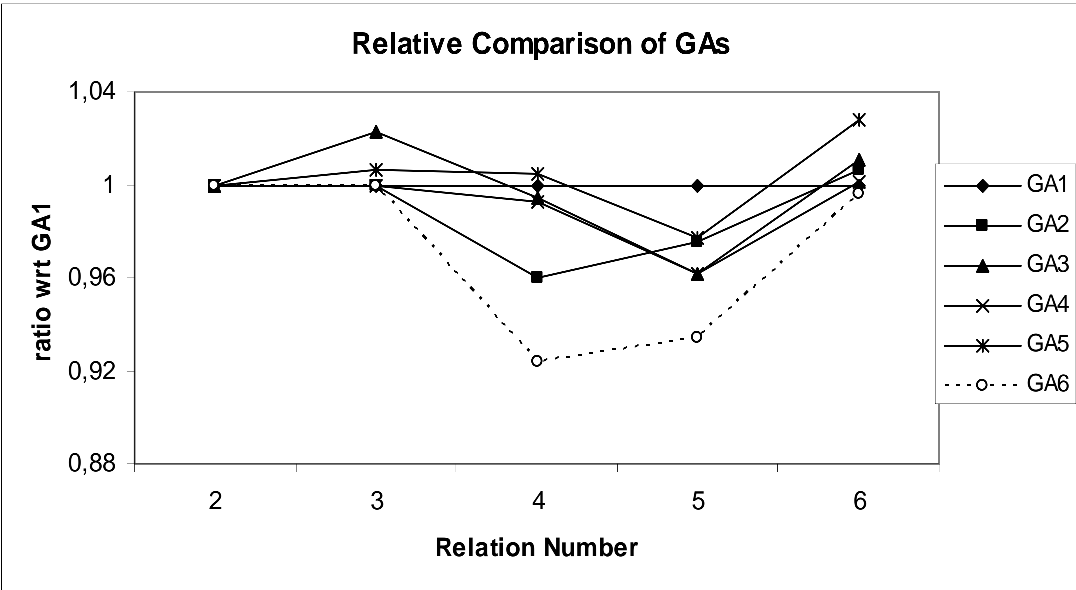


Figure 3.7 : Solution quality based comparison of selection and crossover type combinations

3.5 New-Crossover

The number of genes for crossover is determined by multiplying the crossover ratio with the total number of genes in the chromosome. Typically, 60%-70% is commonly used. We have taken the crossover ratio as 60% since it has proven to be the best as shown in Figure 3.4 for NGA. In GA usually the crossover point is decided randomly, but in NGA it is determined by a heuristic. This crossover heuristic uses costs of genes for this purpose. The minimal cost subsequence of genes is selected for crossing.

We will use chromosomes shown in Figure 3.8 to explain New-Crossover. The examples in this chapter are designed with respect to a query having eight irreducible sub-queries ($n=8$). Regard of being a randomized approach, rest of the values are used as in Table 3.1.

Parent 1	C1 1	C8 7	C3 17	C5 9	C7 3	C2 5	C4 6	C6 2
Parent 2	C5 9	C3 5	C7 1	C1 8	C6 14	C2 3	C4 1	C8 2

Figure 3.8: Parent Chromosomes (only condition numbers and cost of the genes are shown)

Definition(minimal k-length block): A minimum cost '*k*-length' subsequence of genes is called a ***minimal k-length block*** in a chromosome and it has the lowest cost compared to all other '*k*-length' subsequences of genes in that chromosome.

k-length subsequence is evaluated with Formula 3.5 below;

$$k = \text{Crossover Percentage} * \text{Chromosome Length} \quad \textbf{(3.5)}$$

For applying the New-Crossover operator, the first step is to find a minimum cost subsequence of genes. Our subsequence length, *k* will be evaluated as 5, since the sample chromosome length is 8 and the crossover percentage is 0.6. Consequently, we need to find a 5-gene sequence which has the minimum cost relatively. In a DDBS such a minimum cost subsequence of genes will tend to use a minimal number of nodes resulting in minimal communication cost and joins with smaller input relations resulting in smaller intermediate results.

In Parent 1, we have four alternative 5-length blocks. These are;

- "C1 C8 C3 C5 C7"
- "C8 C3 C5 C7 C2"
- "C3 C5 C7 C2 C4"
- "C5 C7 C2 C4 C6"

When we evaluate costs of all these blocks, the last one, "C5 C7 C2 C4 C6", is found to have the least cost when compared to others. The total cost (calculated by summing the gene costs under condition numbers in Figure 3.9) of this block is 25 seconds and is the smallest one in Parent1.

In the example in Figure 3.9, last 5 genes are taken from Parent 1 and then put into the same gene position in the generated offspring. Then, the first 3 absent genes are taken from Parent 2 preserving the order in which they appear in Parent 2.

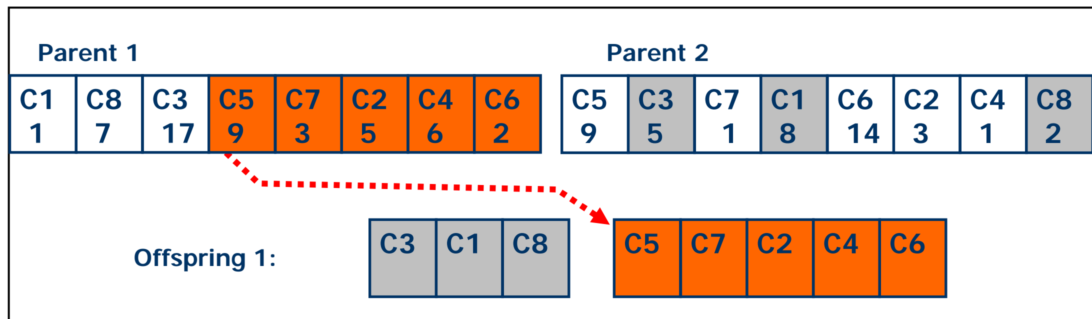


Figure 3.9 : Crossover Implementation (P1XP2)

Definition (New-Crossover)*:** New-crossover is an operator which takes a ***minimal k-length block from the 1st parent and preserves the positions and orders of these genes in the generated offspring. Then, the rest of the genes are copied from the 2nd parent in the order they appear in Parent 2.

When Parent 1 and Offspring 1, shown in Figure 3.9, are compared, it is seen that only the order of the first 3 genes of Parent1 are changed. This process saves time and decreases the "*Optimization Time"* of the query.

Here last 5 genes are taken from Parent 1 and then put to the same place in offspring. Then for the first 3 absent genes are taken from Parent 2 within the order that they take place in their original chromosome.

When the Parent 1 and Offspring 1 shown in Figure 3.9 are compared, in fact we've changed only the sequence of the first 3 genes of Parent1 and that is also quite appropriate for the evolution strategy of GA. Here, we check a different configuration of the first 3 genes over a known to be min cost 5-gene order. The trial is done over a known good sub tree, thus we prune the trials for the genes which are currently in the sub tree. Since we have a min cost order of genes selected from Parent 1 then rest is tried for a better solution. But now we're trying on a smaller set than original.

We believe that this strategy increment the possibility to reach a better sequence, if there is. It must always be kept in mind that despite of trying to find a better solution, this process might produce worse results as well because of randomness originating from its nature. Finally, this process is going to gain time and decrease the "*Optimization Time*" of the query. While gaining this time, there will be no loss in the other goal, namely *"Query Execution Time"*.

As the result, this is believed and proven to be a very suitable way of handling crossover operator of NGA for a distributed query, which we called *New-Crossover*. In our experiments, NGA produced better results than usual GA for almost every occasion.

To explain more clearly, now let's do vice versa and see how Parent 2 will be crossed with Parent 1(P2 X P1) in order to produce Offspring 2.

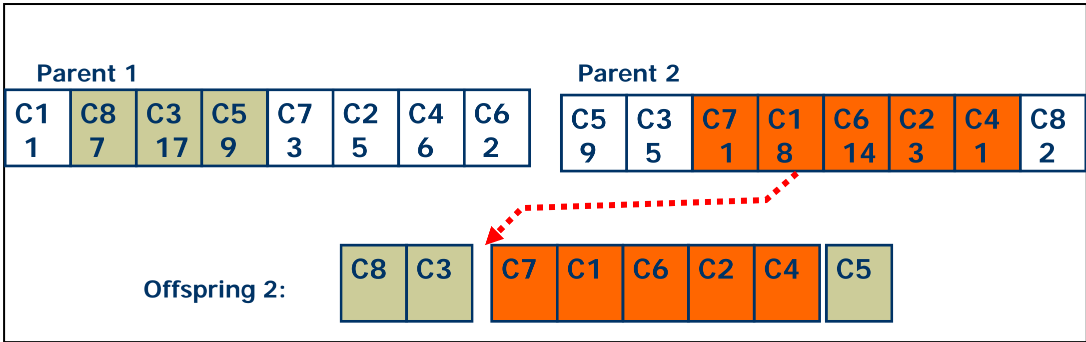


Figure 3.10: Crossover Implementation (P2XP1)

Parents are the same as presented in Figure 3.8. Similarly, we've chosen a 5-gene sequence which has the minimum cost order when compared to other gene sequences. In Figure 3.6, "C7 C1 C6 C2 C4" order is chosen from Parent 2. Then other places of the offspring are filled with the genes of Parent 1 in their original order. In this example, the genes with the condition numbers C8 and C3 is put to the first two spaces and C5 to the last place in the Offspring 2.

In this example, there is a probability of failure in producing a better result than original. This causes from the place of the sub tree. As you can see in the example, we've taken "C7 C1 C6 C2 C4" from Parent 2 (P2), because that sequence has the least cost relatively. But in front of those genes, there are other genes with the conditions C5 and C3 in P2. For the min cost 5-gene order "C7 C1 C6 C2 C4" in P2, C5 and C3 executed first, in other words the attributes for C5 and C3 have been fetched then our sub tree is executed. But in the offspring, we put C8 and C3 at the beginning which might possibly affect the cost of our min cost 5-gene order, and this election will affect the fetched attributes/relations in our *sub tree*. Unfortunately, the functional dependency for our sub tree might be possibly lost at that moment. This situation is the only drawback and inevitable occasion of this process. But you can meet with such a drawback more often in usual GA.

New-Crossover is found to be effective esp. if the sub tree is at the beginning or at the end of the parent chromosome. In such cases, we have the same environment as in the original one then we have a good opportunity to catch a better one, if there is of course. If the sub tree is located at the end or at the beginning, all functional dependencies will be preserved, and we have possibility to try another combination at the same time. The process is done on a smaller set of genes over a known good order of genes, *which increments the possibility to reach better.*

New-Crossover seems to be extracted or inferred from the idea that lies behind the Dynamic Programming which is one of the most important goals of this study. By taking a good known set of genes, in fact we prune rest of combinations for that set and do not need to look for better in that set. But we must not forget that this is just a possibility. It's believed that the way handling the problem improves the power of GA. It can be alleged that New-Crossover is an effective modified version of GA for distributed queries.

In both algorithms, the cost of each gene is known. Another important point is that decreasing or increasing the Crossover Rate is whether helpful or not for the

optimization of the query. That is a good future work in this study. But in briefly, it can be inferred that there must be an optimal point for this rate of a dist. query. If it is bigger or smaller than the optimal number then catching a sub tree will get harder, which shall degrade the overall performance of usual GA.

Now let's continue with our other operator, namely mutation operator which we call *New-Mutation.* This operator has an important effect for getting such a success in attaining the goals of this study as well. Let's explain how New-Mutation is done and achieved performance in the "optimization time" and "query execution time" of the query for NGA.

3.6 New-Mutation

New-mutation operator modifies one of the join node number, chosen replica, and semijoin option of a gene. This operator works similar to the one used in [21]. However, the gene selection criterion is different and uses the cost of individual genes for assigning mutation probabilities. This is a randomized algorithm and involves the following technique:

The costs of the genes are used as a selection probability which is obtained by dividing gene cost to the total chromosome cost. Then, a random number is generated and this number is used to select one of the genes for mutation where the probability of selection of a gene is proportional to its cost.

Mutation operator has two common uses in usual GA. Firstly, two bits/genes or whatever the single unit is, has been changed and secondly, which is commonly done in bitwise chromosomes, a bit is randomly chosen then flipped. These kinds of changes are in fact opposite to the nature of a query. Because, if a change is done in the execution sequence of the query totally regardless of the dependencies in QEP, then you cannot infer that the offspring has been derived from a good parent. This will most probably result in poor performance, since the result will be come out from an unknown execution sequence.

In NGA, New-Crossover is done among genes, it is a gene level operation, but New-Mutation is done one level below, within the units that a gene has. As mentioned in Section 3.2, Chromosome Structure, each gene has three main data. which are sequentially; *Condition Number, Node Number* and *Semi join* sections. The Node Number section holds the information of the node number for that gene. An example for New mutation is shown in Figure 3.11.

C1 1	C8 7	C3 17	C5 9	C7 3	C2 5	C4 6	C6 2	Cost 50

Figure 3.11: Chromosome with condition numbers and costs of the genes

Because of three main reasons listed below, *New-Mutation* is done at Node Number level represented in the chromosome structure. These are;

- Replacement at the gene level actually is done via New-Crossover, in other words we don't need to do same level operation with New-Mutation operator. The function of New-Crossover is to look for a better complete order over a smaller set of known order of genes. New-Mutation operator would rather operate on the sub sections of a gene.
- Secondly, we need a method to handle to correct the number of node. You can only reach the expected result unless there is a misplacement of the genes in the chromosome structure. Because hence that if the order of genes are correct but if the nodes are wrong, then there is no way to reach optimum. You must handle the Node Number of a distributed query in DDBMS for optimization, this is a must. On the other hand, after long trials this operator is found to be very powerful for decreasing the time for executing the distributed query. Even the experimental results showed that this method produces better results when compared to New-Crossover.

- Finally, elections of the most costly gene and changing the node number and semi join option of that gene have an important positive effect on the performance. This tendency is believed to originate from getting rid of network comm. cost. This does not always result as expected, but for generally speaking, executing the sub query at a wrong node result in worse performance according to our overall studies. Network comm. costs must be a great concern if the distributed environment has small and varying bandwidths among nodes [10]. This seems to be a very realistic situation on the other hand. This tendency can also be observed in Dynamic*-Query Optimization Algorithm.

There is always a possible danger for executing a query having many conditions in its “WHERE” clause. This danger originates from the nature of randomized algorithm. As known, these algorithms decide in condition number, node number and a semi join randomly. Though we try to do some operations deliberately, in fact at the beginning everything is handled randomly, and many other operations are done complying with the nature of GA. We choose the gene for New mutation in the same way as done in *Roulette Wheel Selection*.

The idea behind the roulette wheel selection technique is that each individual is given a chance to become a parent in proportion to its fitness. The chance of selecting a parent can be seen as spinning a roulette wheel with the size of the slot for each parent being proportional to its fitness. Obviously those with the largest fitness (slot sizes) have more chance of being chosen. Thus, it is possible for one member to dominate all the others and get selected a high proportion of the time.

Table 3.5: Selection probability of a gene in New-mutation

Number of gene	1	2	3	4	5	6	7	8
Cost of gene	1	7	17	9	3	5	6	2
Selection Probability	0.02	0.14	0.34	0.18	0.06	0.10	0.12	0.04

where N is the number of individuals in the population.

Table 3.5 shows the selection probabilities for an 8-gene chromosome given in Figure 3.11. The 3rd gene has the highest cost and, the 1st gene has the smallest cost, therefore they have the highest and lowest, respectively, probabilities for being mutated.

$$P_i = cost_i / \sum_{j=1}^{N} cost_j \qquad \textbf{(3.6)}$$

So, the bigger the cost of a gene is, the more probable for selection is. The probability of the Gene *i* is shown in Formula 3.6.

While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. With roulette wheel selection there is a chance some weaker solutions may survive the selection process; this is an advantage, as though a solution may be weak, it may include some component which could prove useful following the recombination process.

***Definition (New Mutation)*:** *New-mutation is an operator which does the selection according to the cost of each gene found in the chromosome rather than electing randomly. This process forces the cost of the genes to be in a reasonable range to an extent.*

The analogy to a roulette wheel can be summarized as each candidate solution represents a pocket on the wheel; the size of the pockets is proportionate to the probability of selection of the solution. Selecting N chromosomes from the population is equivalent to playing N games on the roulette wheel, as each candidate is drawn independently.

New-mutation has the following advantages and differences when compared to usual Mutation;

- New-mutation proves to produce very effective results with respect to usual Mutation. Since usual mutation handles the chromosome structure too clumsily, most of the time produces worse results. Because of that reason, *New-mutation* is faster and more likely to reach the result when compared to the *mutation* operator of usual GA
- New-mutation resembles usual mutation by means of changing one bit (Node Number) of the Chromosome or the semijoin option,
- New-mutation makes the change in harmony with the nature of QEP. It is not related with the condition of the gene, but that of node number.
- The most costly genes are more probable to be elected. This forces the cost of the genes to be in a reasonable range to an extent, and produces in very effective results.

The last two fields of a gene is Semijoin and Copy Site sections. Semijoin consists of 2 bits. It might have 4 different values. These values are explained below;

1. ‘00’, There is no semijoin
2. ‘01’, Related colons are taken from the 2nd relation and sent to the site that 1st relation resides.
3. ‘10’, Related colons are taken from the 1st relation and sent to the site that 2nd relation resides.

4. ‘11’, Both relations send their attribute colons to the site where that condition will be executed.

‘Copy site’ is a vector for each relation. Each value in this vector shows the site number from where that relation will be accessed. This vector is similar to ‘CopyId’ attribute of [21] which presented in Table 2.1.

CHAPTER 4

EXPERIMENTAL SETUP AND RESULTS

4.1 Experimental Setup

In order to compare NGA with ESA (Exhaustive Search Algorithm), we composed a synthetic distributed database schema. An interconnection network with 1Gbps Ethernet links is assumed. Each node is assumed to be able to communicate with any other node, without considering multi-hop transmissions and store-and-forward delays. Current technology with Gbps LAN/WAN links with very low delay protocols (e.g. MPLS and ATM) make our assumptions realistic and low cost.

A total of 6 distinct relations has been defined and presented in Table 3.2. Since we are not comparing join methods, we don't attempt to locally optimize a given query's access paths [13]. All queries are assumed as irreducible.

Each processing node is assumed to have the following specifications:

- Number of Buffers = 102 (each one page size)
- Page Size = 10240 bytes
- T_d (Disk I/O Time) = 0.01 sec (per page)
- RAM Size = 128 MB

All nodes are homogenous, in other words, all have the same RAM size, number of buffer pages and page sizes. The files used for the implementation are explained in Figure 4.1 below;

1. **environmental_values.txt** contains;

BufferSize, RAMSize, disk I/O time and PageSize = (In bytes)

2. **environmental.txt** contains the values of

number of the nodes in the WAN, query_initiated_node, sql file to use; e.g. "4"= sql4.sql and finally deployment = "4r_1r" means 4 relation and 1 is replicated

3. **GA values.txt** contains;

Initial_population, Crossover Percentage and Mutation Percentage.

4. **hops.txt** holds the info among the nodes in terms of hop numbers between each of the node in matrix style,

5. **rels_3r** : only 3 relation, node number and tuple num is given,

6. **rels_3r_1f** : (similar)3 rels , 1 is fragmented,

7. **rels_3r_1f_1r** : (similar)3 rels , 1 is fragmented, 1 is replicated,

8. **rels_3r_1r** : 3 rels, 1 is replicated,

9. **rels_4r** : only 4 relation, node number and tuple num is given,

10. **rels_4r_1f** : 4 rels , 1 is fragmented,

11. **rels_4r_1f_1r** : 4 rels , 1 is fragmented, 1 is replicated,

12. **rels_4r_1r** : 4 rels, 1 is replicated,

13. **schema_chr_vals.txt** : issuing_nodes, their frequencies and issued sqls will be executed in the given sequence.

14. **sql3-6** and **sql5__0-9** are different sqls having different number of rels given as the last number of the file.

15. **pool.txt:** Assignment table of relations which is created by clustering.

Figure 4.1 : File Descriptions

Since the GA model given in [21] ignores selection operations, we also consider only queries with join operations in our experiments. This way we can perform and give a fair comparison of our NGA with [21].

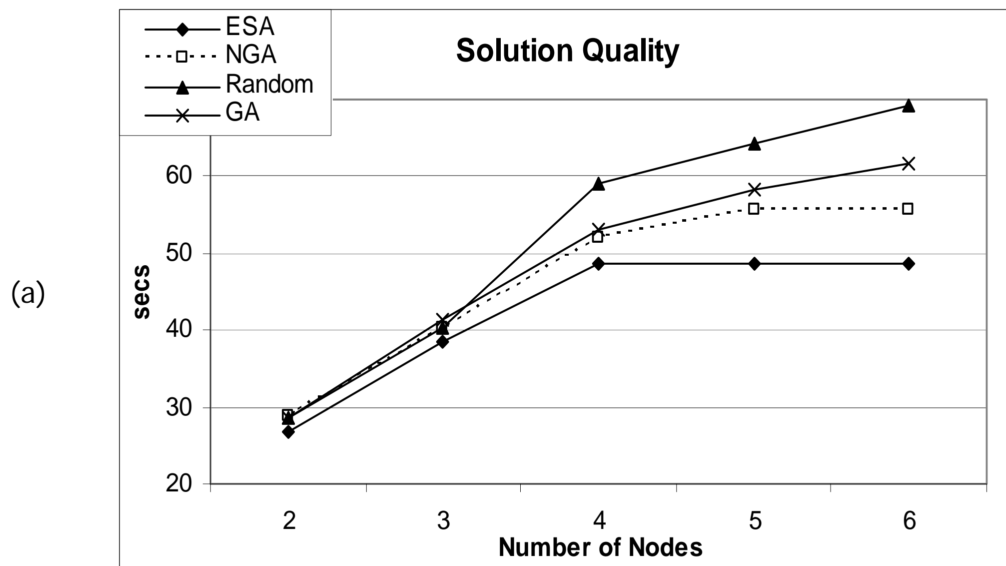
4.2 Experimental Results

5 different test distributed database schemas have been prepared for each of the experiments. These schemas form the X-axis of the graphs. Y-axis shows the solution quality times obtained for each schema.

The results will be mainly examined by using 2 different scenarios;

1) Number of nodes is increased from 2 to 6 as the other parameters remain the same. Same query is used for each case. This query is a linear type one which refers to 4 relations.
2) Number of relations is increased from 2 to 6 as the other parameters remain unchanged. For each query, Q_i the number of relations in it is i. All queries are linear type and total number of nodes is assumed as 4 for any case.

(a)



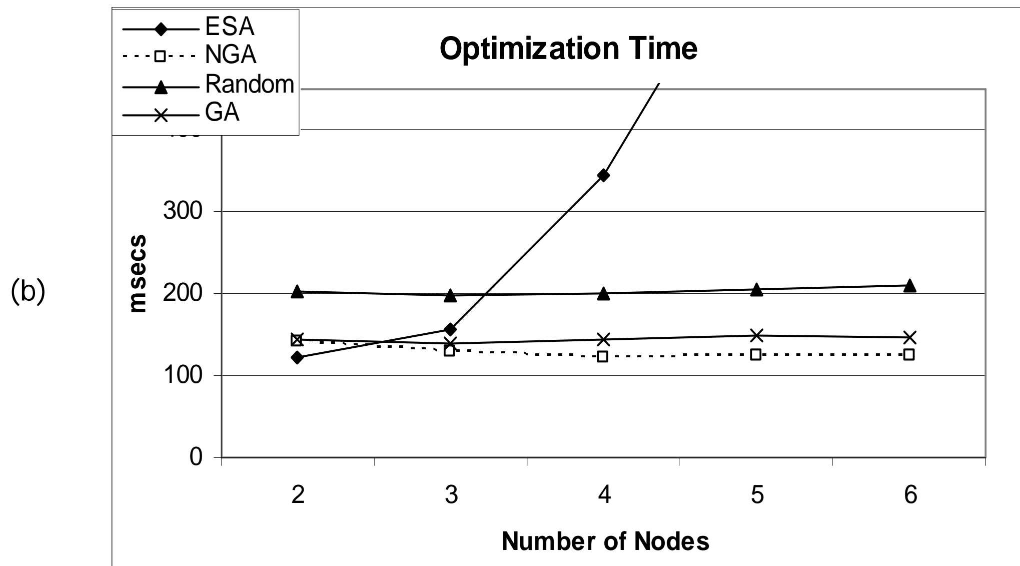


Figure 4.2: The effect of increasing number of nodes

Figure 4.2 shows the effect of increasing “Number of Nodes”. In Figure 4.2(a) ESA gives the optimum value and helps us to evaluate the performance of GA algorithms by comparing how close it is to the optimum value. We also give results for a “Random” algorithm which randomly generates the same number of solutions as the GA algorithms and we use it to determine if the GA contributes positively to finding and developing better solutions to the problem. If a GA performs worse than the “Random” algorithm, we can safely conclude that its crossover/mutation operators and selection strategy are ineffective. Our experiments show that NGA finds acceptable good solutions when compared to ESA and GA. Also, when the problem size increases, the time needed for ESA grows exponentially as shown in Figure 4.2(b), making ESA too costly and NGA a very competitive alternative.

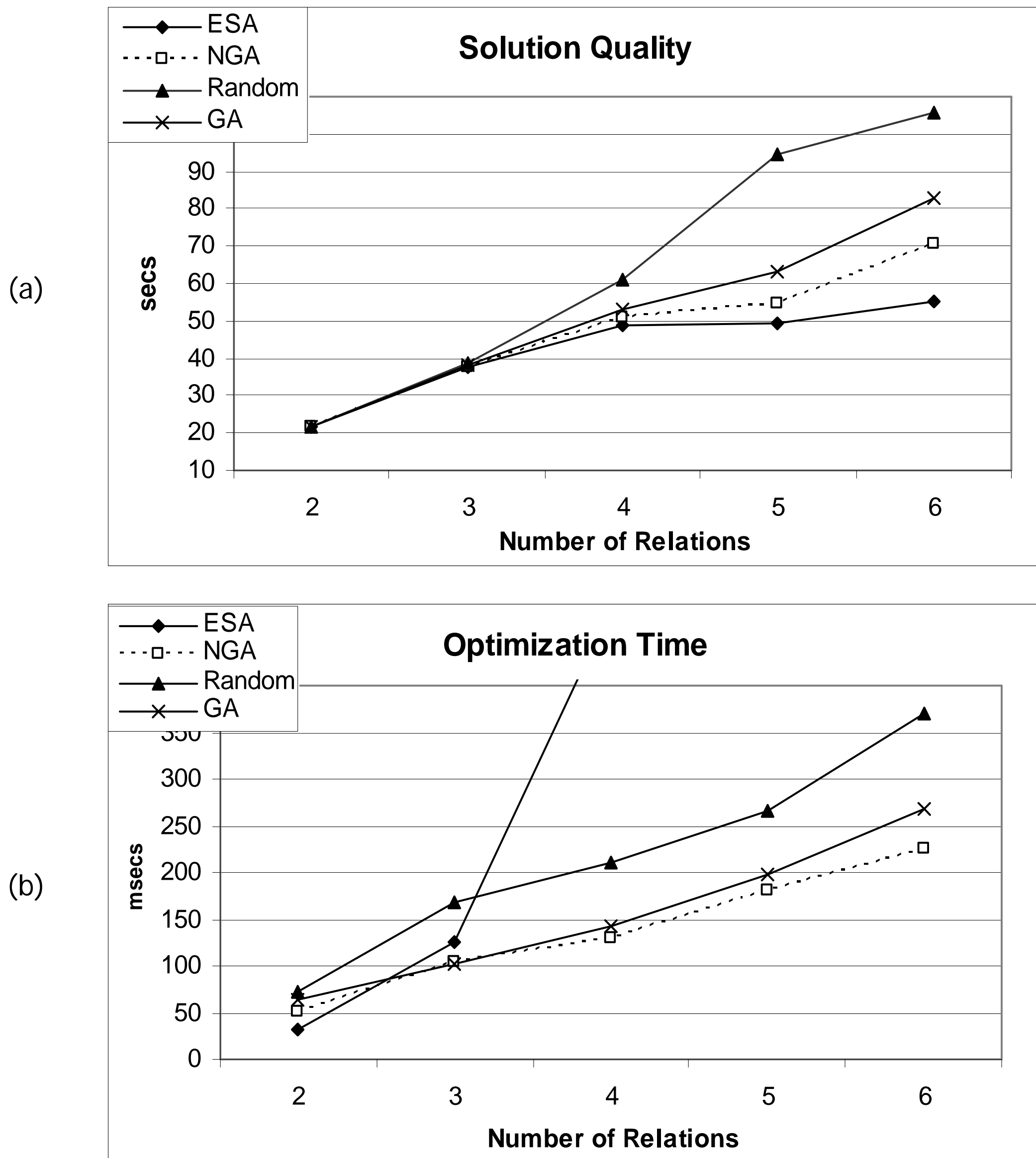


Figure 4.3: The effect of increasing number of relations

Figure 4.3 presents the effects of increasing "Number of Relations" on performance. Figure 4.3(a) shows the *"Solution Quality"* which is the execution time of the query plan found for answering the query. When the problem size is small all algorithms produce very close cost solutions, but as the problem size increases ESA will produce about 10% to 15% better solutions than NGA.

In Figure 4.3(b), it is seen that as problem size grows with the number of relations, the optimization (execution) time of the ESA algorithm increases exponentially. Thus, we can say that as problem size (esp. number of relations) grows, NGA becomes a much better alternative than ESA and GA.

CHAPTER 5

DESIGN OF DISTRIBUTED DATABASE SCHEMA USING A GENETIC ALGORITHM

One of the aims of this thesis is generation of a distributed database schema with a randomized approach for a given environment. An environment is composed of the following information;

- Queries,
- Query issuing nodes and frequencies of those queries,
- Relations,
- Nodes and network bandwidths (LAN/WAN),

Firstly, we have to decide the solution strategy of this problem. We again use usual genetic algorithm (GA). The values are as listed in Table 3.1. Our chromosome structure will be different than our NGA chromosome. Each chromosome is a solution for the problem and denotes a different Distributed Database Schema. Relation and nodes, all calculations with respect to queries will be done for each of the chromosomes. Although it seems to be cumbersome process, we'll control this evaluation process time by our standard operators.

5.1 Distributed Database Schema Chromosome and Query Structure

The chromosome structure of a Distributed Database Schema (DDB_Sch) is explained in Figure 5.1.

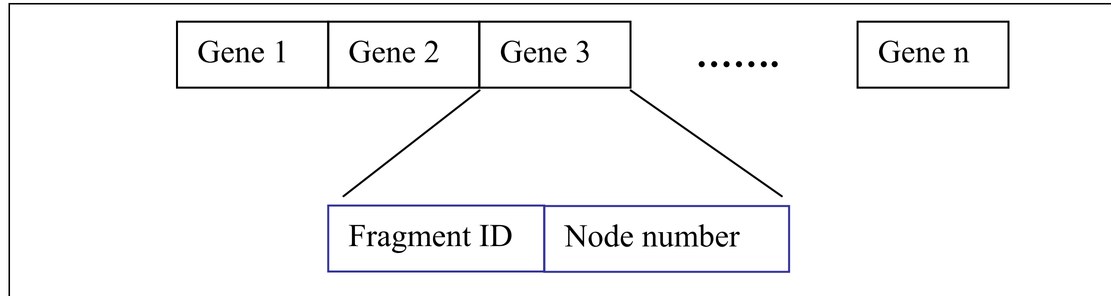


Figure 5.1: Chromosome Structure of a Distributed Database Schema

The chromosome has genes holding the information of the DDB Schema. It has the following fields;

- Fragment ID
- Node number

"Fragment ID" is the physical name of the table of a logical relation. Table 5.1 and 5.2 shows how those are assumed and handled in our model. In Table 5.1 and 5.2, fragmentation and replication is shown since our algorithm can handle both fragmentation and replication of those fragments/relations.

"Fragment ID" is composed of relation name, fragment name and replica name.

The id of ***Rel_xxxyyzz*** denotes that ***'xxx'*** logical relation number, where ***'yy'*** denotes the fragment number and ***'zz'*** denotes the replica number. For example, Rel_10020102 is used as the following;

Rel_1002 is the relation name, ‘01’ shows that this is the first fragment. This also means that there are at least two fragments. If there is no fragmentation, then that number must be ‘00’. Then, the final two digits, ‘02’, show that this entity is the second replica of that fragment. If there is no replication then replica number should be ‘00’.

Table 5.1 : Horizontal Fragmentation of the relations

Logical Relation Name	Fragment number	ID
Rel_1000	1	1000 **00** zz
Rel_1001	2	1001 **01** zz
		1001 **02** zz
Rel_1002	1	1002 **00** zz
Rel_1003	3	1003 **01** zz
		1003 **02** zz
		1003 **03** zz
Rel_1004	1	1004 **00** zz

In Table 5.2, how the replication is done will be described.

Table 5.2 : Replication of the fragments/relations

ID	Replica number	Fragment ID	Number of fragments
1000 **00** zz	1	1000 00 **00**	1
1001 **01** zz	1	1001 01 **00**	2
1001 **02** zz	3	1001 02 **01**	3
		1001 02 **02**	4

		1001 02 **03**	5
1002 00 zz	2	1002 00 **01**	6
		1002 00 **02**	7
1003 01 zz	2	1003 01 **01**	8
		1003 01 **02**	9
1003 02 zz	1	1003 02 **00**	10
1003 03 zz	1	1003 03 **00**	11
1004 00 zz	2	1004 00 **01**	12
		1004 00 **02**	13

Table 5.2 shows the physical entities derived from the logical relations. As mentioned in Table 3.2, all relations have changing number of cardinals. If any relation is fragmented then the total number of tuples for that relation is randomly separated among the fragments.

Since ESA response time has too big value, we have used a test bed consisting 5 nodes with 5 relations for the DDB Schema algorithm experiments. On the other hand, this type test bed is believed to be enough to show the performances of the algorithms.

5.2 Genetic algorithm for DDB Chromosome

5.2.1 Crossover

The crossover operator has the goal of combining good solutions of one generation producing offspring which will hopefully be superior to its parents. One of the characteristics of the DDB chromosome is that each *Fragment ID* is contained exactly once in each chromosome. It’s a unique number.

A possible crossover operator for a DDB chromosome is explained in Figure 5.2. Numbers of the genes denotes the Fragment IDs.

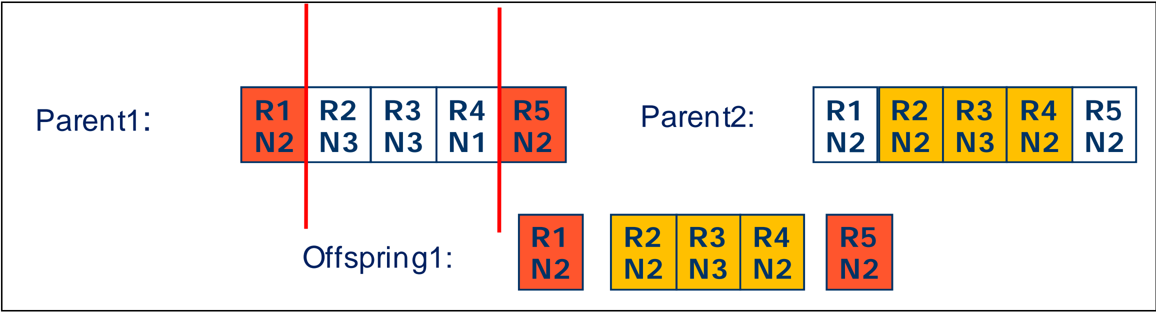


Figure 5.2: Crossover operation for a Distributed Database Schema Chromosome

Crossover operator for a DDB Chromosome works as follows;

1. Two parent chromosomes are chosen with respect to GA type, which is Truncate method with 2-point crossover type as discussed in Table 3.1 and 3.4.
2. The genes those will not be crossed are carried to the offspring in their order.
3. Finally, the genes which will be crossed are substituted with the genes of other parent (P2).

One of the problems that one might meet while deciding the design of a DDBMS is that each relation tends to be at the same node or two nodes at most. This tendency originates from the communication cost. Because if you place all the fragments at the same node, then it means that there will be no communication cost and this results as better performance for that DDB Chromosome, this behavior is centralized and out of the vision of this study.

We made some more assumptions in order to avoid from such a situation that all the fragments comes together at the same node. In the experiments, some assumptions are made to overcome such problems. These assumptions are helpful for also creating more realistic cases.

5.2.2 Mutation

Mutation operator has a similar process as usual GA. As it is not allowed that the same *Fragment ID* appears twice in a DDB Chromosome, the mutation operator simply changes the node numbers of two random genes. The order of the genes has no importance.

It is advisable to perform only a few mutations per generation. Otherwise, the evolution process might be severely disrupted. Commonly used value (1%) is used for DDB Mutation as well. On the other hand, for a randomized algorithm several tests have been held and for both crossover and mutation percentages, the optimum values are found. These will be explained in Section 5.4.

5.3 System Structure

Table 3.1 shows the logical structure of the relations used throughout this study. Here the "*Fragment ID"* denotes the physical name/fragment of that relation, and the "Node number" shows simply the node number where that physical entity is residing at.

Table 5.3: Queries, frequencies and issuing nodes

No	Queries	Frequency	Node #
1	SELECT <selection_list> FROM Rel_1000, Rel_1001, Rel_1002, Rel_1003 WHERE Rel_1000.attr_1 = Rel_1001.attr_1 AND Rel_1001.attr_6 = Rel_1002.attr_6 AND Rel_1002.attr_11 = Rel_1003.attr_11 AND Rel_1002.attr_12 < some_val	300	0
2	SELECT <selection_list> FROM Rel_1000, Rel_1001, Rel_1002, Rel_1003 WHERE Rel_1000.attr_1 = Rel_1001.attr_1 AND Rel_1001.attr_6 = Rel_1002.attr_6 AND Rel_1002.attr_11 = Rel_1003.attr_11 AND Rel_1000.attr_2 >= some_val	200	2

In Table 5.3, two sample queries referencing four relations are presented and used for DDB chromosome experiments as root. '<selection_list>' part of the sample SQLs denote a set of several attributes of the relations that are done on select-project-joins and 'some_val' is any value which the selection of records are done with regard to. These two values are assumed to be assigned with respect to database statistics in our experiments.

The queries with their frequencies and the issuing nodes are also given in the same table. These assumptions are similar to real life, e.g., in a DDBMS 500 queries are run totally in a selected period of time and in that period 1st query is run from the 2nd node 300 times in that DDBMS as shown in Table 5.3. This value is assumed and can be obtained from a DB expert in real life applications as statistical information.

5.4 Distributed Database Schema Design

Chromosome structure is the same as presented in Section 5.1. Distributed queries are executed for each instance of DDB Schemas. These queries are assumed as in Table 5.3. This set may have queries up to the number '*k*'. In this study, our query set composes of 2 queries, i.e., k=2. Each query is optimized due to NGA separately and DDB schema has a weighted cost for that set.

For the calculation of DDB chromosome, query frequencies are assumed to be found with respect to the following Formula 5.1 in that DDBMS.

$$\text{Frequency } (Fr_i) = \text{How many times query } i \text{ issued} \qquad \textbf{(5.1)}$$

where 'i' is the number of that query.

In our DDBMS model, it assumed to be 5 relations and the whole data is partitioned among 13 fragments. Now we have to decide the places of those entities. This will be modeled with our Distributed Database Schema (DDB_Sch) chromosome. Each of them represents a different schema. All queries will be executed by the issuing

node. Within that schema we will get a response time totally. Then with respect to our Fitness Function (Sol_{DDB}), Formula 5.2, we will have the DDB Schema.

$$\text{Fitness Function } (Sol_{DDB}) = \min(\text{Cost of DDB_Sch}) \qquad \textbf{(5.2)}$$

Cost of DDB_Sch ($Cost_{DDB_Sch}$) is the weighted cost of the response time of each query. The weighted cost of queries is obtained by the help of the frequencies of each query (Fr_i).

Frequency of a query is simply proportion of the execution number of that query to that of total. By the Formula 5.3 we get the "*$Cost_{DDB_Sch}$ of chromosome*";

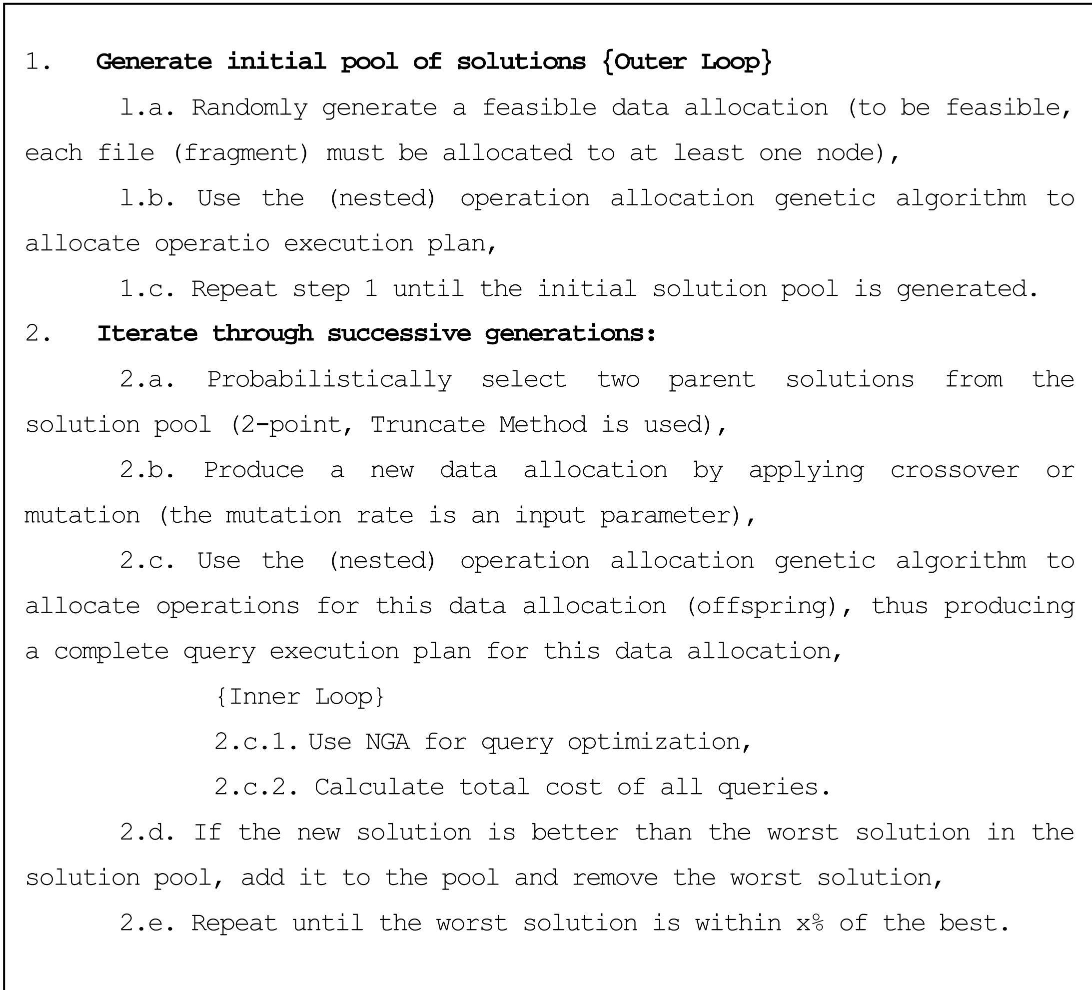
$$Cost_{DDB_Sch} = \sum_{i=0..k} Fr_i * \text{Response Time of Query}_i \qquad \textbf{(5.3)}$$

where k is the number of total queries.

The queries, their frequencies and the issuing nodes are given in Table 5.1. By the help of these input values, $Cost_{DDB_Sch}$ of all DDB chromosomes can be evaluated. The minimum one will be our DDB Schema, which is the goal of this study.

For solving the distributed database design problem we employ a nested genetic algorithm as defined in [10, 15, 21]. In the outer loop, the usual GA creates the DDB Schema and in the inner loop New GA (NGA) optimizes single distributed queries separately on that schema. Our design id presented in Figure 5.3. Each single query is evaluated with respect to the allocated DDB schema. In other words, each single query optimization is dependent on DDB schema is optimized by NGA in our

model. In the experiments, we compare our algorithm with Exhaustive Search Algorithm (**ESA**) and another nested genetic algorithm in [10, 15, 21]. For ESA, in the outer loop ESA is used for creation of DDB schema creation and in the inner loop ESA is used to optimize the distributed queries.



1. **Generate initial pool of solutions {Outer Loop}**

   1.a. Randomly generate a feasible data allocation (to be feasible, each file (fragment) must be allocated to at least one node),

   1.b. Use the (nested) operation allocation genetic algorithm to allocate operatio execution plan,

   1.c. Repeat step 1 until the initial solution pool is generated.

2. **Iterate through successive generations:**

   2.a. Probabilistically select two parent solutions from the solution pool (2-point, Truncate Method is used),

   2.b. Produce a new data allocation by applying crossover or mutation (the mutation rate is an input parameter),

   2.c. Use the (nested) operation allocation genetic algorithm to allocate operations for this data allocation (offspring), thus producing a complete query execution plan for this data allocation,

   {Inner Loop}

   2.c.1. Use NGA for query optimization,

   2.c.2. Calculate total cost of all queries.

   2.d. If the new solution is better than the worst solution in the solution pool, add it to the pool and remove the worst solution,

   2.e. Repeat until the worst solution is within x% of the best.

Figure 5.3 Nested Genetic Algorithm for DDB Design

For the other nested genetic algorithm, GA algorithm is used both for data allocation and for the operation allocation. GA is described as in [10, 15] for the outer loop and in the inner loop Rho's GA is used as described in [21]. This algorithm is named as **RGA** in the following experiments.

Our nested genetic algorithm, DDB Design Algorithm (named as **DGA**) and in the outer loop usual GA works but in the inner loop NGA is used for single distributed query optimization briefly. DGA operates as in Figure 5.3.

All DDB schemas are evaluated and the election is done according to GA rules, where the results will be presented in the following section. Each chromosome obtained by GA is a solution for us and all of them will be evaluated with respect to Formula 5.2 and 5.3. In fact, Formula 5.2 is our Fitness Function for GA.

For a DDB Schema chromosome several tests have been done in order to find relatively optimum values for crossover and mutation percentage and the initial solution size. The results are presented in Figure 5.4, Figure 5.5, and Figure 5.6. They show us that a crossover percentage of 0.7, mutation rate of 0.01, and initial population size of 12 gives the best results for a DDB Schema chromosome for GA. For the experiment presented in Figure 5.6, larger population sizes will slightly improve the solutions but only at the cost of an exponential increase in the GA runtime.

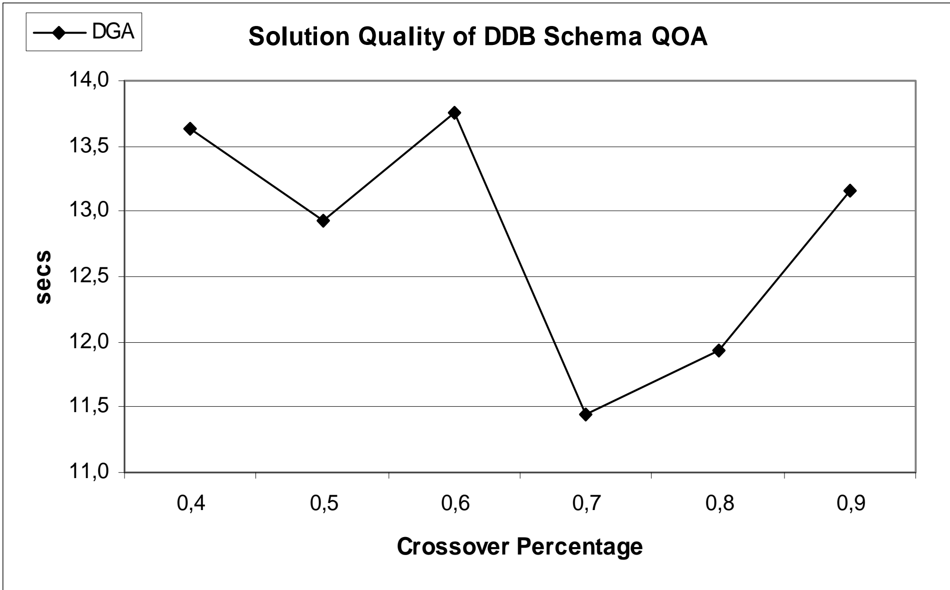


Figure 5.4 : The performance of DGA for increasing crossover percentages

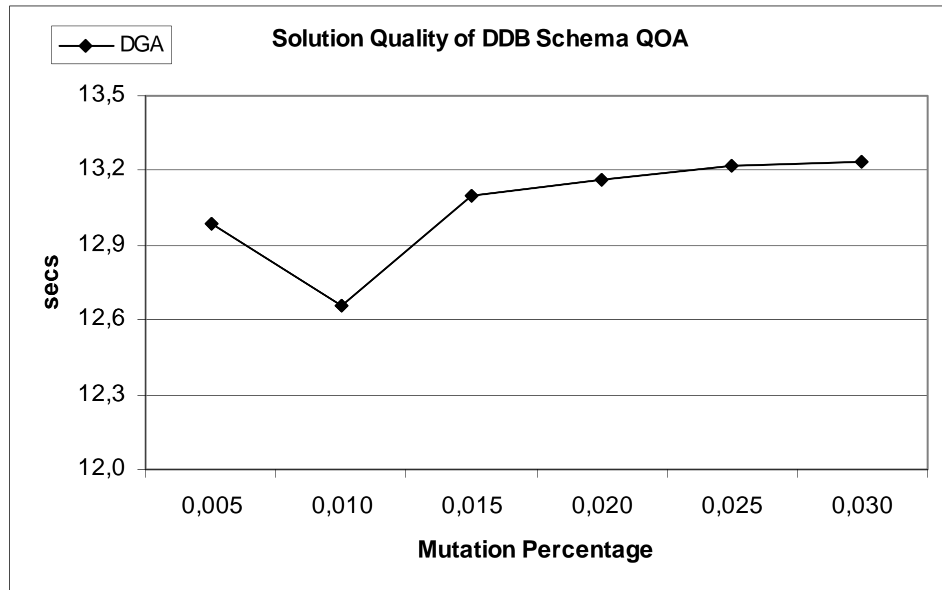


Figure 5.5 : The performance of DGA for increasing mutation rates

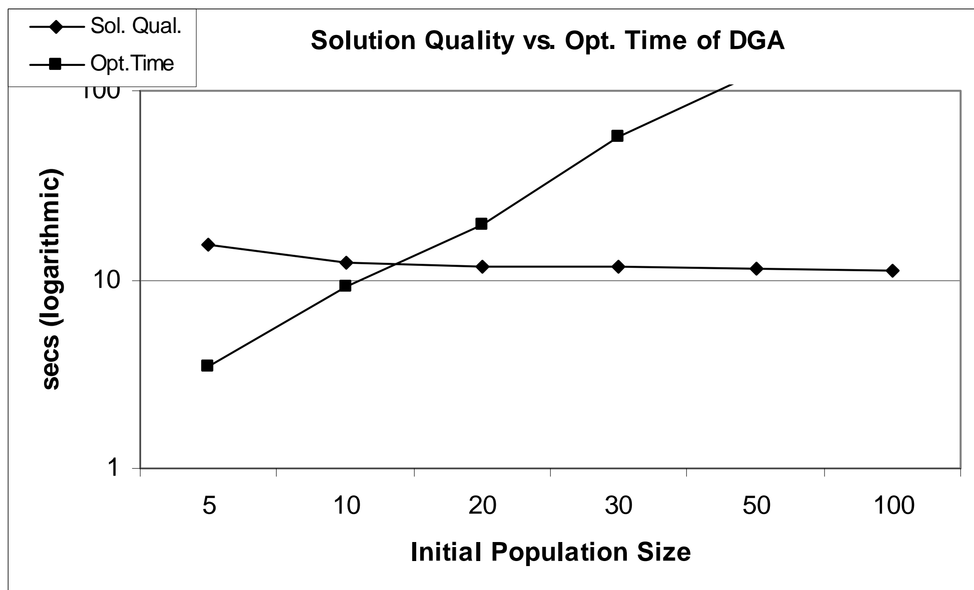


Figure 5.6 : The performance of DGA for increasing initial population size

5.5 Experimental Setup and Results

In our experiments we try to find a DDB Schema in such an environment which is designed as in the following;

- Relations are created, fragmented and replicated as in Table 5.1, 5.2,
- Queries, their frequencies and the issuing node are the same as in Table 5.3,
- Each node is assumed to be able to communicate with any other node, without considering multi-hop transmissions and store-and-forward delays.
- Relation Schema and the selectivities are as in Table 3.2 and 3.3,
- In order to avoid from the situation that all the fragments comes together at the same node, at least half of the nodes have been assigned at least one fragment.
- If any relation is fragmented, each fragment has equally divided number of total records presented in Table 3.2.

Our experiment parameters are read from text files and documented in the thesis for simplicity. All tests are done and discussed using 6 different cases listed below. Since ESA shall produce a hyperbolically execution time a small test-bed is used;

1. <u>3 Node-3 Relation:</u> There are only 3 nodes and all relations are not replicated,
2. <u>3 Node-3 Relation and 1 Relation replicated:</u> There are 3 nodes again, and one of the randomly selected relation is replicated once,
3. <u>4 Node-4 Relation:</u> There are now 4 nodes and 4 relations, and any of the relations is replicated,
4. <u>4 Node-4 Relation, 1 Relation replicated:</u> There are 4 nodes again, and one of the relations is replicated once,
5. <u>5 Node-5 Relation:</u> There are 5 nodes and 5 relations in the WAN, and any of the relations is replicated,
6. <u>5 Node-5 Relation, 1 Relation replicated:</u> There are 5 nodes again, and one randomly selected relation is replicated once.

Although DGA can evaluate DDB Schema cost for the fragmented relations, it is not used since RGA has not this capability. Because of that, all relations are assumed not to be fragmented or as only one fragment. Replicas are included for test cases.

In all cases, one of the relations is statically assigned to a node and then others are placed in order to get the minimum response time according to the algorithm, e.g., 2nd relation is assumed to be placed at Node 3, then the rest of the relations are decided according to the algorithm. This is done for the cases that take place as in real world problems. In some of the cases we might need to place a relation to a site compulsorily, or we might not have possibility to move a relation to any other place, and then the other relations must be placed with respect to that relation. For such cases in distributed database design issue, our application has solutions as well.

5.5.1 Comparison of ESA, DGA and RGA

The result of any case for DGA and ESA is the arithmetic mean of 20 independent runs for the outer loop, and any outer loop which is a DDB schema, is again the arithmetic mean of 20 independent runs of the inner loop. Inner loop denotes the algorithms, e.g. NGA and GA that are discussed and presented in Section 4 for a single distributed query optimization. Inner loop runs on the schema that comes from the outer loop for that instance.

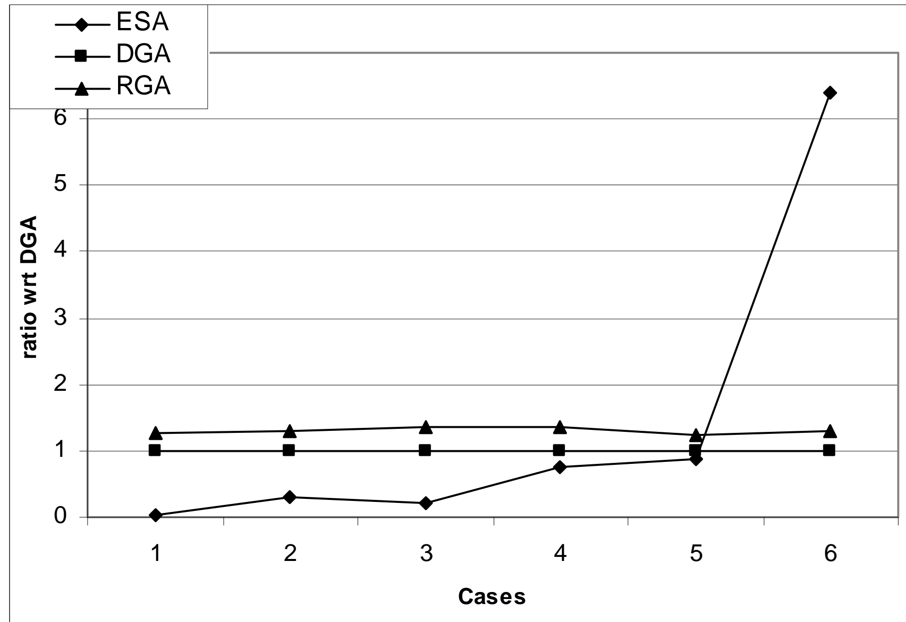


Figure 5.7 : Optimization Times of DDB Design Algorithms

Figure 5.7 is important since the total response time of DGA and DGA is presented for evaluating the each case. As the node number and relation numbers increases, its response time increases hyperbolically.

The result in Figure 5.7 is obtained with respect to DGA. DGA is taken as 1 and ESA and RGA results are relatively presented. If we closely examine Figure 5.7 we can see that though DGA works approx. 20% less in time than RGA and it finds better results than RGA as shown in Figure 5.8. It is believed that this difference originates from the power of NGA for solving single distributed query. Since ESA works combinatorial, as the node and relation number increases, its response time increases hyperbolically.

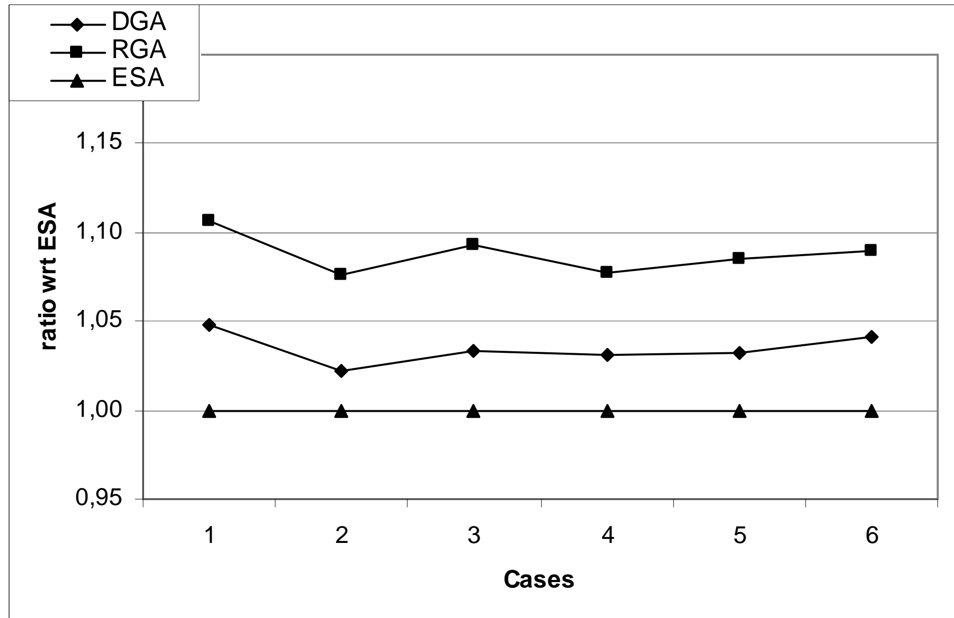


Figure 5.8 : Query Execution Times of optimized DDB.

In the inner loop, the results are evaluated for each of the single DDB queries and the outer loop evaluates the DDB schema cost with respect to these results within that schema by using the Formulas 5.2 and 5.3.

When we look at Figure 5.8, ESA, DGA and RGA is compared relatively. ESA value is taken as 1 and others find the execution time as ESA time fold. Execution time is the time obtained by the Formula 5.3. Since ESA finds the optimum value, the execution time of DGA and RGA are higher than that of ESA. If Figure 5.8 is closely examined, DGA finds approx. 3% higher than ESA while RGA finds approx. 9% higher than that. On the other hand, by the help of DGA, GA used for DDB chromosomes finds results close to optimal within its execution time. The response time is believed to be reasonable and DGA might be used for real-life problems. The experiments for a better performance of a DDB chromosome will be a good future work.

We prepared a test case for two input queries listed in Table 5.3 which will use a horizontally fragmented relation. The first case is shown for the frequencies in Appendix-A.

When we modify the frequencies of queries we observe that the storage location of a fragment will be changed. The algorithms obtain optimal execution times for both queries. If we use the 1st frequency for the 2nd query and similarly the 2nd frequency for the 1st query then the results are changed. This is the 2nd test case 2 and shown in Appendix-B.

The test cases show also that DGA results finds better results when compared to DDB results, and close to that of ESA.

5.6 DDB Design Using Relation Clustering

We defined a method, relation clustering, that gathers related relations into same or adjacent nodes in order to increase the performance of DDB Design Algorithm. The query plans are for these DDB designs are calculated using DGA and we achieve about 30% improvement in query execution results.

At the beginning, a pool has been created. This pool is used as the initial pool for the randomized algorithm rather than creating the initial pool randomly. That causes a remarkable performance increase as the search space size gets bigger. Figure 5.10 and 5.11 show that Clustered DGA, which will be abbreviated as CGA, reaches better results faster than usual DGA.

Relation clustering is done with respect to the following code, as presented in Figure 5.9;

```
Input: N number of queries in Query_list
Output: place M number of relations in a DDB referenced by
queries in Query_list.

begin

for i= 0..N {
    CALCULATE_COST(Qi)                                          (1)
}
QL ← SORT_DESCENDING{Q0..QN};                                   (2)
//sort the queries wrt their costs, the most costly is at the top

do {
  for k= 0..M { // for all Relations                            (3)
   for i= 0..N {  // for all queries                            (3.1)
     if (Relk is statically assigned to a Node) continue;       (3.2)
     if (Relk is not referenced by Qi) continue;                (3.3)
     if(RELk is not assigned to any Node) {                     (3.4)
      PLACE RELk TO ASSIGNED_NODE(RELk,Qi);
        RelCOSTk = Cost(Qi);
        continue;
     }
     Change Node of RELk (Qj, Probability);                     (3.5)
// Probability ={Cost (Qi) / (Cost (Qi)+ RelCOSTk)
   }
  }
  CREATE A INDIVIDUAL FOR INITIAL SOLUTION POOL;                (4)
}
while (a solution set list of relations are assessed)           (5)

end.
```

Figure 5.9: CGA Pseudocode

In Figure 5.10 (a), the query execution times found by DGA and Clustered DGA (CGA) are shown. In Figure 5.10(b), relative comparison of both algorithms is presented. It is noticeable that CGA proves to get a linear better performance while the search space increases. Cases are the same as listed in section 5.5.

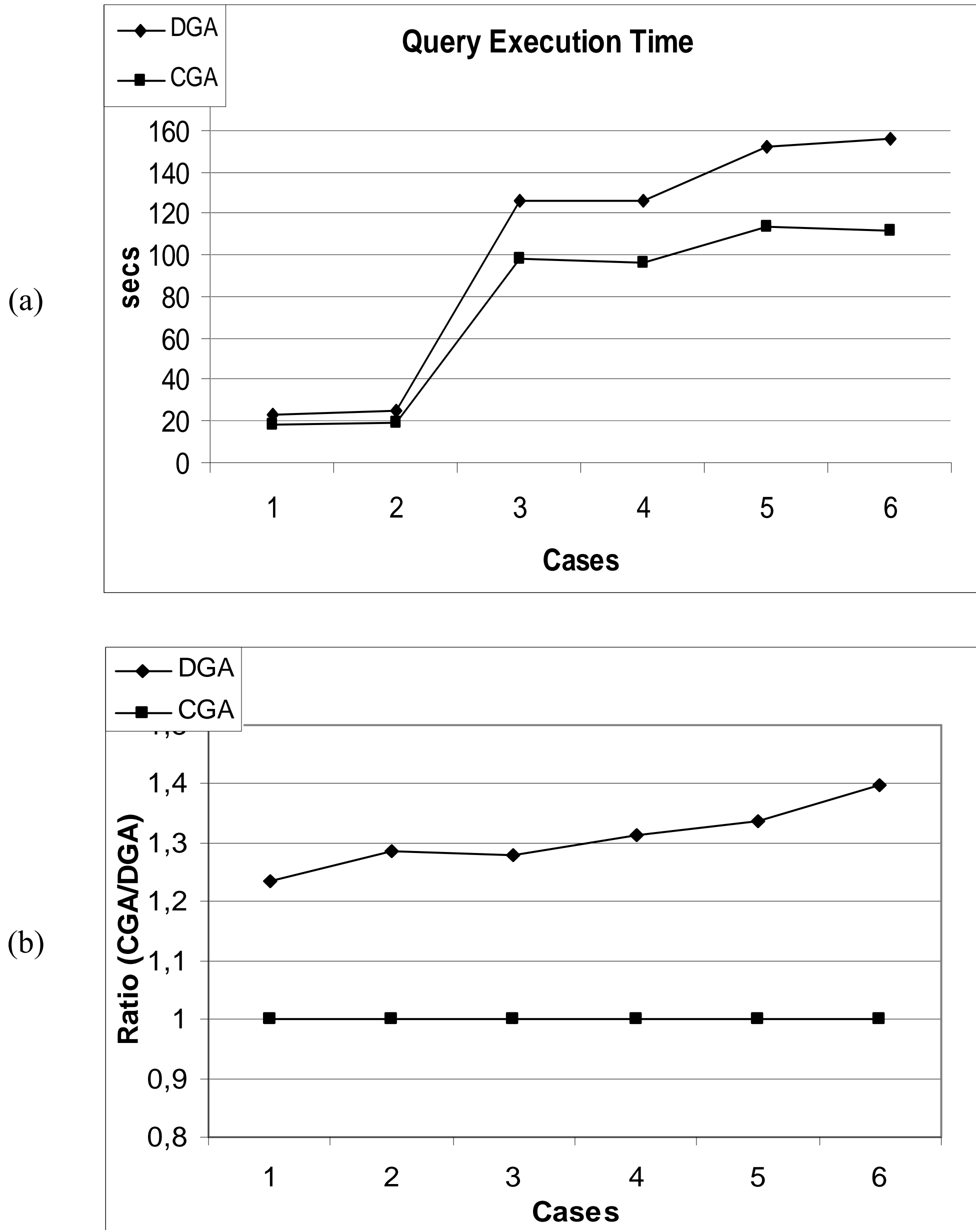


Figure 5.10 : Query Execution Times of DGA and Clustered DGA

Response time of both DGA and CGA is presented in Figure 5.11. Since CGA works on known good set, it reaches the result quicker than DGA.

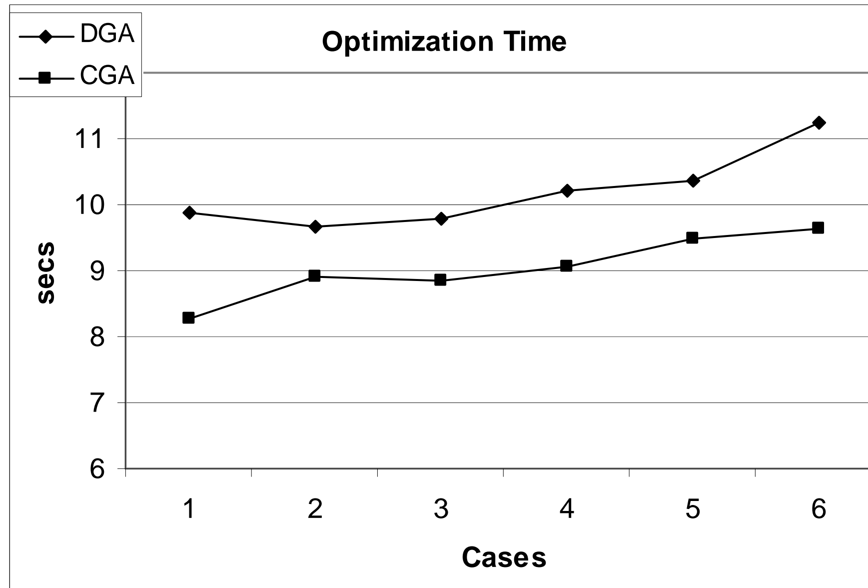


Figure 5.11 : Optimization Times of DGA and CGA

We also made similar experiments to make sure that relation clustering increases the performance. Figure 5.11 is prepared as the node number increases in WAN. Here we can observe a performance increase about 20%. Node number and relation number changes as shown in cases, might affect the result differently, since these parameters have different weights in the result. Optimization Time is not shown since it has a similar attitude as in Figure 5.11.

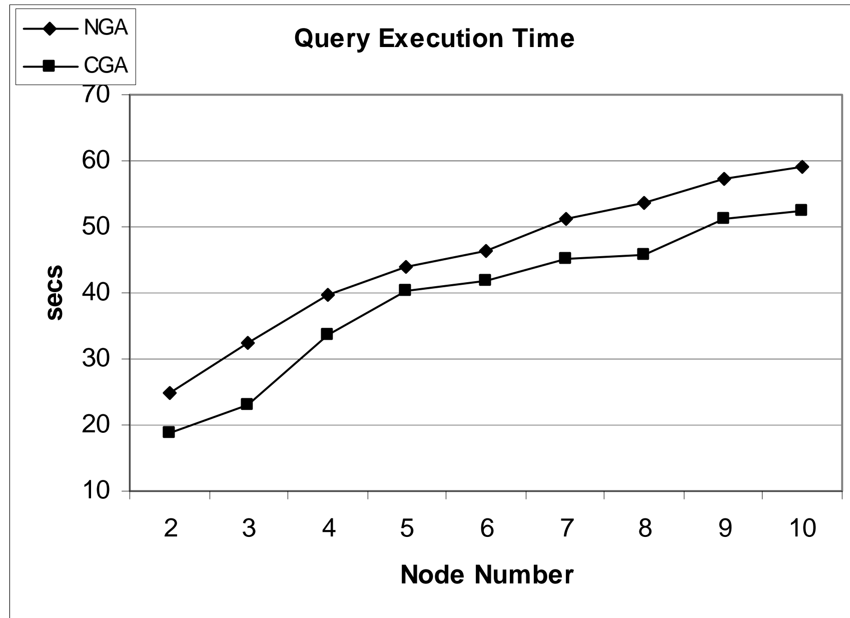


Figure 5.12 : Query Execution Times of DGA and Clustered DGA

Finally, we think that a relation/database deployment is the first point in the distributed query optimization area. This study is done to help database managers in order to decide their database schema. A distributed database schema inference with a minimum cost is the main goal of this thesis.

CHAPTER 6

CONCLUSIONS

In this work we have evaluated a thorough set of alternatives for a GA based solution to the distributed database query optimization problem. In order to be able to evaluate the performance of our NGA algorithm, we have compared it with a previously defined GA. We have also implemented exhaustive and random algorithms so that we will have upper (given by random algorithm) and lower(given by exhaustive algorithm) bounds for the solution costs determined by GA. If a genetic algorithm cannot produce better results than a random search algorithm, then it has no useful contribution to the problem solution. Our experimental results show that our proposed NGA is about 10% to 15% worse than the best achievable solution while its execution times are much lower than the exhaustive algorithm.

We have also investigated the effect of increasing number of nodes and number of relations in a distributed database. Our results show that the costs of the plans produced by NGA are very close to the optimal plans produced by the exhaustive algorithm. The execution time of NGA grows almost linearly (as expected from a genetic algorithm). NGA gives better plans than a previously derived genetic algorithm, GA.

As future work, we are planning to extend our exhaustive algorithm to work on a parallel processing machine such as Intel HPC grid and determine and compare results with our NGA for 10-20 node distributed database systems with a comparable number of relations. Development of algorithms for query optimization on distributed database systems with various LAN/WAN links and bandwidths especially in the domain of data warehouse applications [26] would also be a very interesting research direction as such systems are becoming more and more widespread with the huge cost reductions in high capacity hardware and high-speed WAN links.

We have started evaluating our NGA heuristic on other types of queries in addition to linear type ones. We expect to get better good performance for star and tree type distributed database queries when compared to other GAs.

Our genetic algorithm is also tried as DDB Design Algorithm (named as **DGA**) and our test results show that DGA finds better designs when compared to other known DDB Design Algorithms presented in Section 5.4 [10, 15, 21]. The results are close to that of an exhaustive algorithm, ESA, but as the search space gets bigger, DGA will become to be the most preferred algorithm as a DDB Design algorithm because of its low execution time compared to exponential increase in execution time of ESA.

We also defined a method, relation clustering that gathers related relations into same or adjacent nodes in order to increase the performance of DDB Design Algorithm. The query plans are for the DDB designs are calculated using DGA and we achieve about 30% improvement in query execution results.

REFERENCES

[1] 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.

[2] P.A. Bernstein, N.A.Goodman, E. Wong, C.L. Reeve and J. B. Jr. Rothnie, *"Query Processing in a system for distributed databases (SDD-1),"* ACM Transactions On Database Systems 6 (1981), pp.602-625.

[3] P. A. Bernstein & D. W. Chin, *Using semi-joins to solve relational queries*, 1981, Journal of the ACM, 28, 25-40

[4] L. Davis (ed.), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[5] R. Epstein, M. Stonebraker and E. Wong, *"Query Processing in a Distributed Relational Database System,"* In Proc. ACM SIGMOD Int.Conf. on Management of Data, May 1978, pp.169-180.

[6] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison- Wesley, (1989).

[7] J.M. Hellerstein, *"Optimization Techniques for Queries with Expensive Methods,"* ACM Transactions on Database Systems, Vol. 23, No. 2, June 1998, Pages 113–157.

[8] J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

[9] Y.E. Ionnidis 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.

[10] J.M. Johansson, S.T. March. and J.D. Naumann, *"Modeling Network Latency and Parallel Processing in DDB design,"* Decision Sciences, Volume 34 Number 4, Fall 2003, pp. 677-706

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

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

[13] D. Kossmann, *"The State of the Art in Distributed Query Processing,"* ACM Computing Surveys, Vol. 32, No. 4, December 2000, pp. 422–469.

[14] G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger and P. Wilms, *"Query Processing in R*,"* in [Kim et al.,1985], 1985, pp.31-47.

[15] S.T. March and S. Rho, *"Allocating data and operations to nodes in a distributed database design,"* IEEE Transactions on Knowledge and Data Engineering, vol. 7, no. 2, April 1995, pp. 305-317.

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

[17] K. Ono and G. Lohman. *Measuring the complexity of join enumeration in query optimization.* In Proc. of the Conf. on Very Large Data Bases (VLDB), pages 314–325, Brisbane, Australia, August 1990.

[18] M.T. Ozsu and P. Valduriez, **Principles of Distributed DB Systems,** Prentice Hall, 1999, Chapter 9, pp.239-243.

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

[20] M.T. Ozsu and P. Valduriez, Distributed Data Management: Unsolved problems and new issues,

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

[22] 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, pages 23–34, Boston, USA, May 1979.

[23] P.G. Selinger and M. Adiba, *"Access Path selection in Distributed Database Management Systems,"* in Proc. First Int. Conf. on Data Bases, 1980, pp.204-215

[24] M. Steinbrunn, G. Moerkotte, and A. Kemper. *Heuristic and randomized optimization for the join ordering problem.* The VLDB Journal, 6(3):191-208, 1997.

[25] G. Syswerda, *"Uniform crossover in genetic algorithm,"* Proceedings of the 3rd International Conference on Genetic Algorithms, 1989, pp. 2 – 9.

[26] Torlone, R.: Two approaches to the integration of heterogeneous data warehouses. Distributed and Parallel Databases, 23, 69-97 (2008)

[27] B. Vance and D. Maier. *Rapid bushy join-order optimization with cartesian product.* In Proc. of the ACM SIGMOD Conf. on Management of Data, pages 35–46, Montreal, Canada, June 1996.

[28] D. Whitley, T. Starkweather and D. Fuquay, *Problems and traveling salesmen: The genetic edge recombination operator,* Proceedings of the 3rd International Conference on Genetic Algorithms, Morgan Kaufmann, 1989.

[29] E. Wong. and R.H. Katz, *Distributing a Database for parallelism*, Proceedings of the 1983 ACM International Conference on Management of Data, New York, ACM Press, 1983, 23-29

[30] H. Yoo & S. Lafortune, *An intelligent search method for query optimization by semijoins*, 1989, IEEE Transactions on Knowledge and Data Engineering, 1, 226-237.

APPENDIX A

Test case 1 for DDB schema

1. ESA, Response Time: 3640 msecs, Query Execution Time: 9.366 secs.
Relation_ID : 10000000, Node_number : 2
Relation_ID : 10010000, Node_number : 2
Relation_ID : 10020000, Node_number : 2
Relation_ID : 10030000, Node_number : 0

2. NGA, Response Time: 1633 msecs, Query Execution Time: 10.263 secs.
Relation_ID : 10000000, Node_number : 2
Relation_ID : 10010000, Node_number : 2
Relation_ID : 10020000, Node_number : 2
Relation_ID : 10030000, Node_number : 1

3. DDB, Response Time: 1679 msecs, Query Execution Time: 10.955 secs.
Relation_ID : 10000000, Node_number : 2
Relation_ID : 10010000, Node_number : 2
Relation_ID : 10020000, Node_number : 2
Relation_ID : 10030000, Node_number : 2

APPENDIX B

Test case 2 for DDB schema

1. ESA, Response Time: 3359 msecs, Query Execution Time: 11.194 secs.
Relation_ID : 10000000, Node_number : 2
Relation_ID : 10010000, Node_number : 1
Relation_ID : 10020000, Node_number : 1
Relation_ID : 10030000, Node_number : 0

2. NGA, Response Time: 1663 msecs , Query Execution Time: 12.301 secs.
Relation_ID : 10000000, Node_number : 2
Relation_ID : 10010000, Node_number : 1
Relation_ID : 10020000, Node_number : 1
Relation_ID : 10030000, Node_number : 0

3. DDB, Response Time: 1719 msecs, Query Execution Time: 12.482 secs.
Relation_ID : 10000000, Node_number : 2
Relation_ID : 10010000, Node_number : 1
Relation_ID : 10020000, Node_number : 1
Relation_ID : 10030000, Node_number : 0

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name : Sevinç, Ender
Nationality : Turkish (TC)
Date and Place of Birth : 25 September 1969, İstanbul
Marital Status : Married
Phone : +90 312 384 28 93
Mobile : +90 535 439 76 55
Email : ender@ceng.metu.edu.tr

EDUCATION

Degree	Institution	Year of Graduation
MS	METU Computer Engineering	2000
BS	K.H.O Elec.and Eln. Engineering	1991

WORK EXPERIENCE

Year	Place	Enrollment
2007- Present	CC Software Section	DB. Project Off.
2005-2007	TGS Intelligence Dept.	IT Mngr.
1999-2005	TRADOC Simulation Center	Tech.Sys.Mngr.
1997-1999	TRADOC IT Center	Chief Asst.

FOREIGN LANGUAGES

Advanced English.

PUBLICATIONS

1. Cingil, İ., Doğaç, A., Sevinç, E. and Coşar, A., Dynamic Modification of XML Documents: External Application Invocation from XML. "ACM SIGEcom Exchanges", 1, (2000), s.1-6

2. Sevinç,E., Coşar, A. "An Evolutionary Genetic Algorithm for Optimization of Distributed Database Queries", ISCIS 2009 Conference

3. Sevinç,E., Coşar, A., Toroslu, İ.H., " A New Genetic Algorithm for Query Optimization in Distributed Databases", Cybernetics and Systems (Under Revision)

4. Sevinç,E., Coşar, A., "An Evolutionary Genetic Algorithm for Optimization of Distributed Database", Oxford Journals, The Computer Journal (Under Revision)

HOBBIES

Tennis, Music, Computer Technologies, Theatre, Motor Sports