A C++ DISTRIBUTED DATABASE SELECT-PROJECT-JOIN QUERY PROCESSOR ON A HPC CLUSTER

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

BY

ERHAN CERAN

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

MAY 2012

Approval of the thesis:

A C++ DISTRIBUTED DATABASE SELECT-PROJECT-JOIN QUERY PROCESSOR ON A HPC CLUSTER

submitted by **ERHAN CERAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering, Middle East Technical University** by,

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

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

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

Prof. Dr. Özgür Ulusoy _____
Computer Engineering Dept., Bilkent University

Date: _____

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: ERHAN CERAN

Signature:

ABSTRACT

A C++ DISTRIBUTED DATABASE SELECT-PROJECT-JOIN QUERY PROCESSOR ON A HPC CLUSTER

Ceran, Erhan

M.Sc., Department of Computer Engineering

Supervisor : Assoc. Prof. Dr. Ahmet Coşar

May 2012, 75 pages

High performance computer clusters have become popular as they are more scalable, affordable and reliable than their centralized counterparts. Database management systems are particularly suitable for distributed architectures; however distributed DBMS are still not used widely because of the design difficulties. In this study, we aim to help overcome these difficulties by implementing a simulation testbed for a distributed query plan processor. This testbed works on our departmental HPC cluster machine and is able to perform select, project and join operations. A data generation module has also been implemented which preserves the foreign key and primary key constraints in the database schema. The testbed has capability to measure, simulate and estimate the response time of a given query execution plan using specified communication network parameters. Extensive experimental work is performed to show the correctness of the produced results. The estimated execution time costs are also compared with the actual run-times obtained from the testbed to verify the proposed estimation functions. Thus, we make sure that these estimation

functions can be used in distributed database query optimization and distributed database design tools.

Keywords: Distributed database, Query execution plan, Distributed database simulation, Query evaluation, Cost estimation.

ÖZ

BİR DAĞITIK VERİTABANI SELECT-PROJECT-JOIN SORGU İŞLEMCİSİNİN BİR HPC ÖBEĞİ ÜZERİNDE C++ İMPLEMENTASYONU

Ceran, Erhan

Yüksek Lisans, Bilgisayar Mühendisliği Ana Bilim Dalı

Tez Yöneticisi : Doç. Dr. Ahmet Coşar

Mayıs 2012, 75 sayfa

Yüksek performanslı bilgisayar öbekleri merkezi muadillerine göre daha ölçeklenebilir, ucuz ve güvenilir oldukları için günümüzde yaygınlaşmıştır. Veritabanı yönetim sistemleri dağıtık mimariler için oldukça uygun olmalarına rağmen, dağıtık veritabanları tasarımlarının zor olması nedeniyle yeterince geniş alanda kullanılmamaktadırlar. Bu çalışmada dağıtık veritabanı tasarımında karşılaşılan zorlukların üstesinden gelinmesine yardımcı olmak için simülasyona yönelik bir test yatağı geliştirilmesi amaçlanmıştır. Bunun için bölümümüzün HPC öbeği üzerinde select, project ve join işlemlerini dağıtık olarak çalıştıran bir sorgu planı işlemcisi implemente edilmiştir. Bunun yanında primary key ve foreign key kısıtlamalarını sağlayabilen bir veri üretim modülü hazırlanmıştır. Test yatağının bir diğer yeteneği ise verilen bir sorgu planın cevap zamanını ölçebilmesi, belirtilen iletişim ağı parametrelerine göre bu zamanı simüle ve tahmin edebilmesidir. Üretilen sonuçların doğruluğunu göstermek üzere deneyler yapılmıştır. Tahmin fonksiyonlarını doğrulamak için tahmin edilen çalışma zamanlarıyla test yatağından elde edilen gerçek zamanlar karşılaştırılmıştır. Bu şekilde tahmin fonksiyonlarının dağıtık veritabanı sorgu iyileştiricilerinde ve dağatık veritabanı tasarım araçlarında kullanılabileceği gösterilmiştir.

Anahtar Kelimeler: Dağıtık veritabanı, Sorgu çalıştırma planı, Dağıtık veritabanı simülasyonu, Sorgu değerlendirimi, Maliyet tahmini.

To My Family

ACKNOWLEDGEMENTS

I owe my deepest gratitude to my supervisor Assoc. Prof. Dr. Ahmet Coşar for his invaluable assistance, advice and guidance. Without him this thesis research would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Distributed databases have many advantages over traditional databases for organizations with non-centralized data structures. They utilize the locality of resources, resulting in a scalable architecture and better performance. Replication of relations makes distributed databases more reliable. Also, building a distributed database is cheaper than its centralized counterpart as creating a network of less powerful computers does not cost as much as one high-end computer.

Despite these advantages, distributed databases were not very popular because of data network limitations. However, with the advances in network technologies, interest in distributed databases is growing rapidly. The complexity of designing and creating efficient distributed database systems made it a popular field among researchers.

One of the hardest problems when building a distributed database system is the optimization of queries. For a given database query, there exists multiple ways of execution. These query execution plans differ in the order and the location of the operations that will be performed, as well as the cost of execution. The search space of query execution plans can become very large for a query with a lot of input relations on a large distributed database. To find good solutions with

reasonable costs, query optimizers run algorithms that evaluate the cost of candidate plans and select the lowest cost plan among considered plans.

In [Kossmann (2000)] query plan evaluation algorithms are examined extensively. Dynamic programming methods as proposed in [Selinger (1979)] have been used in traditional databases. This approach finds good solutions with low costs. However it is not very suitable for distributed databases, as there are many additional factors that need to be considered such as network costs, relation sites and replication, resulting in very costly search computations. Genetic algorithms are also used in query optimization. They are more suitable for distributed environments as they have significantly better search time and come up with acceptable plans.

Regardless of the search algorithm used, the approximate cost of a plan must be known by the algorithm to be able to compare plans without actually executing them. Cost estimators are used for this purpose. The cost estimator takes an execution plan and the state of the distributed database as input, and returns the approximate cost of executing that plan as output. In this study, we create a distributed query processor that can execute and estimate costs for distributed queries. While estimating the total cost, network communication costs and disk access costs are considered. We first implement a simple distributed database that is able to execute select, project and join operations which runs on a computer cluster. Another module is built to generate relation data according to given size and selectivity criteria. To evaluate the accuracy of the cost estimator, actual time, simulated time and the time estimated by the cost estimator is compared. Our results show that we can estimate the simulated costs accurately, while preserving a correlation between real and simulated costs.

In the first section we briefly explain the problem we study on this thesis. Following section gives detailed background information about distributed databases, the computer cluster we worked on, the database operators used and the optimization problem. Section 3 addresses the actual implementation of our system including design, capabilities, modules and tools used. The data formats used for input in our system is also explained in detail. In section 4 we depict the experimental results obtained from executing different queries on different database setups. In the last section we briefly discuss the overall result of this study and possible ways to make our system more realistic.

CHAPTER 2

BACKGROUND

2.1 Distributed Databases

A database is a collection of data organized in a way to describe related activities. Today, most of the databases are stored on digital environments. The difficulty of managing databases with large amount of data has raised the need for database managements systems (DBMS). A DBMS is a software system that aids the user to effectively manage data. [Ramakrishnan (2002)]

Users describe their data on a DBMS with high-level abstractions, called data models. There are various data models suitable for specific tasks such as hierarchical, network, relational and object-relational models. This study focuses on commonly used relational data model which was introduced in [Codd (1970)]. In relational model, data is represented with one or more relations. The schema of a relation describes the meta-data for that relation. The instance consists of tuples which are units of data with fields as described in the schema.

A distributed database (DDB) is a group of logically related databases residing at different sites on a network. A distributed database management system (distributed DBMS) software manages the databases on the network and hides the complexity of data distribution, making it transparent. The user of distributed

DBMS views the data as a centric database without concerns about relation locations. The data-storage sites on a DDB are independent. They can be physically close like nodes of a computer cluster, or far like two machines of an organization residing on different cities, connected with a WAN. Each site has its processor, disk, memory and operating system, allowing them to run applications on their own. Even if the site has multiple processors, Distributed DBMS is not concerned by the data parallelism inside one machine. Data is transferred between sites through computer networks, not interprocessor communication, resulting in a loosely coupled architecture. [Ozsu (1999)]

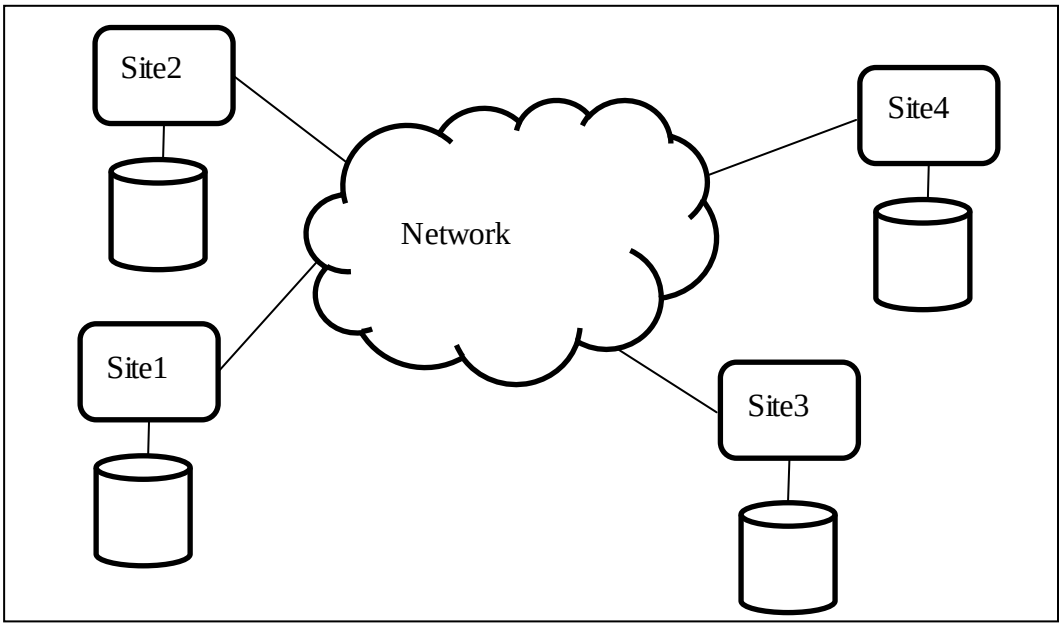

Figure 2.1: Overview of a Distributed DBMS

DDBs employ concepts called fragmentation and replication for data storage. Fragmentation is the separation of a relation into smaller relations. These smaller relations are usually kept on different sites. Breaking a relation into row groups while preserving the same columns in all the fragments is called horizontal fragmentation.

On the contrary, vertical fragmentation means separating a relation by columns, where each fragment has data about all the rows in the original relation. In both horizontal and vertical fragmentation, the original relation must be reconstructable from its own fragments with no data loss. Also it is mostly desired that fragments contain non-redundant data. Figure 2.2 depicts example fragmentations on a table.

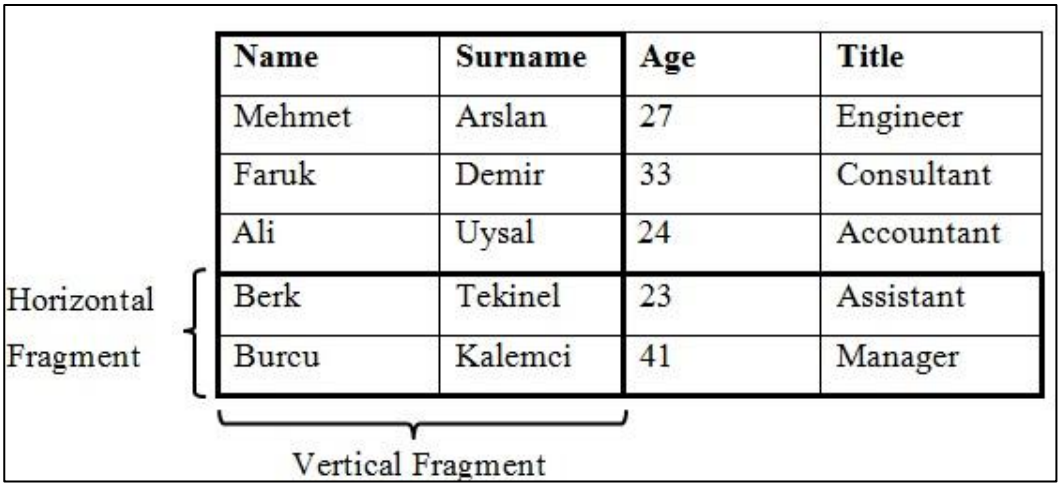

Figure 2.2: Horizontal and Vertical Fragments

Replication means storing multiple copies of a relation or some fragments of a relation in different sites. Storing these copies increase reliability of the system, as if a site crashes replicated relations can be used instead of the relations on the crashed site.

DDBs have the following main advantages over centralized databases:

- DDBs do not depend on the hardware and software capacity of one machine, making them very suitable for growth in data size. Today, computer processors have nearly reached their physical limit for clock frequency. This makes parallelism a must for systems requiring more computational power. DDBs naturally reflect parallelism.

- Ideally a DDB is independent of the hardware, operating system, network or the DBMS its sites run. This gives large organizations the opportunity to easily reflect their organization structure to DDB and make use of already existing databases.

- Fragmentation and replication increases reliability. In a centralized database, if the machine storing relations is down, the database is completely unusable. However on a DDB even if a site is down, replicas and fragments of the relations in the down site can still exist on running sites, making the DDB available.

- In a DDB relations can be stored on the sites where they are expected to be used most. This improves the performance by decreasing communication times. Another performance advantage is the utilization of multiple sites. Executing a query in parallel on different sites results in lower response times.

2.2 Database Operators

Retrieving data from a database is achieved through query languages based on Relational Algebra(RA). Every operator in the language takes one or two relations as input and produces a result relation, allowing operators to be chained. Complex queries can be created by composing these basic operators with a set of base relations as input. [Ramakrishnan (2002)].

For this study the basic select, project and join operators were implemented. Also two helper operators, "make distinct" and "reduce" were used in semi-join.

Select(σ) is a unary operator that retrieves the tuples of a relation matching a given condition. As an example, $\sigma_{c>N}(R)$ returns the rows in relation R where the value in column C is greater than N.

Project(π) is another unary operator that filters and outputs the selected columns of a relation as a new relation. Given a relation and a set of columns, it creates a sub-relation with all the tuples but only the desired columns of the original relation. For instance, $\pi_{C1,C2}(R)$ gets the values of C1 and C2 columns for all the rows, eliminating any other columns R has.

Join operation is used to combine tuples from two relations by matching tuples from both relations using a given condition (usually a common attribute having the same value). In other words, join is the equivalent of taking cross product of two relations and filtering out undesired rows. There are specialized names for join depending on the type of condition, such as condition join, natural join and equijoin. For the purpose of this study equijoin is used, which is depicted by the symbol '$\bowtie$'. In equijoin, a tuple from relation R1 is merged with a tuple from relation R2 if the specified column C1 of R1 has the same value with column C2 of R2. This join operation is expressed as: $R1 \bowtie_{R1.C1=R2.C2} R2$.

Join is an important binary operator whose evaluation is not as straightforward as unary operators. In fact, the hardest part in distributed query optimization problem is determining the order of join operations and which site to perform them. The algorithm used for the join operation itself is also a factor for performance. All of the algorithm types have advantages and disadvantages for different kinds of input relations. Join algorithms implemented for this study are explained below.

2.2.1 Nested Loop Join

Nested Loop Join (NLJ) is the most straightforward to implement join algorithm which can be preferred for joining small relations. For larger relations, the cost of I/O operations and memory consumption increases, making NLJ a bad choice. The algorithm scans every tuple of inner relation for each tuple in outer relation, adding them to result set if they match.

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

Figure 2.3: NLJ Pseudo Code [Ramakrishnan (2002)]

Our implementation uses the smaller input as outer relation, as the number of total I/O operations are decreased this way.[Ramakrishnan (2002)].

2.2.2 Sort-Merge Join

Sort-Merge Join (SMJ) is another join method that groups tuples of input relations according to their join attribute value by sorting them. Because input relations are partitioned on join attribute, it is possible to find matching tuples by iterating only matching partitions. This way join operation can be completed in a single iteration for both relations, eliminating the need for scanning the whole relation for every tuple in the other input relation. The pseudo-code for SMJ is as following:

```
proc smjoin(R, S, "R_i = S_j")

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

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

while Tr ≠ eof and Gs ≠ eof do {

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

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

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

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

    }
```

Figure 2.4: SMJ Pseudo Code [Ramakrishnan (2002)]

The most expensive part of the algorithm is the sorting of input relations. Therefore SMJ performance is good for relations that are already sorted or indexed on join attribute. Our implementation avoids sorting already sorted relations.

2.2.3 Hash Join

Hash Join (HJ) consists of two phases: building and probing. In building phase, the inner relation is hashed by the value of its join attribute. Using the hashed value as key, a hash table is created for tuples.

In probing phase, the outer relation is iterated. For each iterated tuple, a hash value is computed on the join attribute using the same hash function. If there are any values with the same key in the hash table created in the building phase,

actual tuples of the hash table are compared to see if they match. With this approach only the partitions with the same hash keys are scanned.

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

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

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

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

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

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

Figure 2.5: HJ Pseudo Code [Ramakrishnan (2002)]

HJ is particularly suited for joining large relations. One of the important factors to determine performance of HJ is how well the tuples are distributed on the hash table. We used the C++ standard library hash table for our implementation. As it performs best in average case scenarios, HJ is used as the join algorithm in this study unless otherwise stated.

2.2.4 Distributed Semi-Join

Distributed Semi-Joins [Kang(1987)] aim to reduce the response time of distributed join operations. During a distributed query execution, network communication costs between sites usually outweigh the local processing costs.

Semi-Joins aim to decrease the amount of total data communication by performing some extra operations locally.

For this study we used 2-way semi-join (2SJ) [Daniels(1982)]. Assume that there is a relation R residing in Site1, and relation S residing in Site2 and these relations need to be joined at Site3. To join R and S using 2SJ, first R and S are projected on the join attribute at their local sites resulting in relations P_R and P_S. Then P_R is sent to Site2, where it will be compared with the tuples of P_S for equality. Matching tuples are put in relation T1, non-matching tuples are put in relation T2. This operation is called "reduction". The smaller of T1 and T2 is sent back to Site1. In Site1, depending on the type of relation received from Site2 (T1 or T2), tuples of R matching with T1 or non-matching with T2 are sent to Site3. Likewise, Site2 sends tuples of S matching with P_R to Site3, where the join operation is performed locally. Note that, Site3 and Site2 can be the same site.

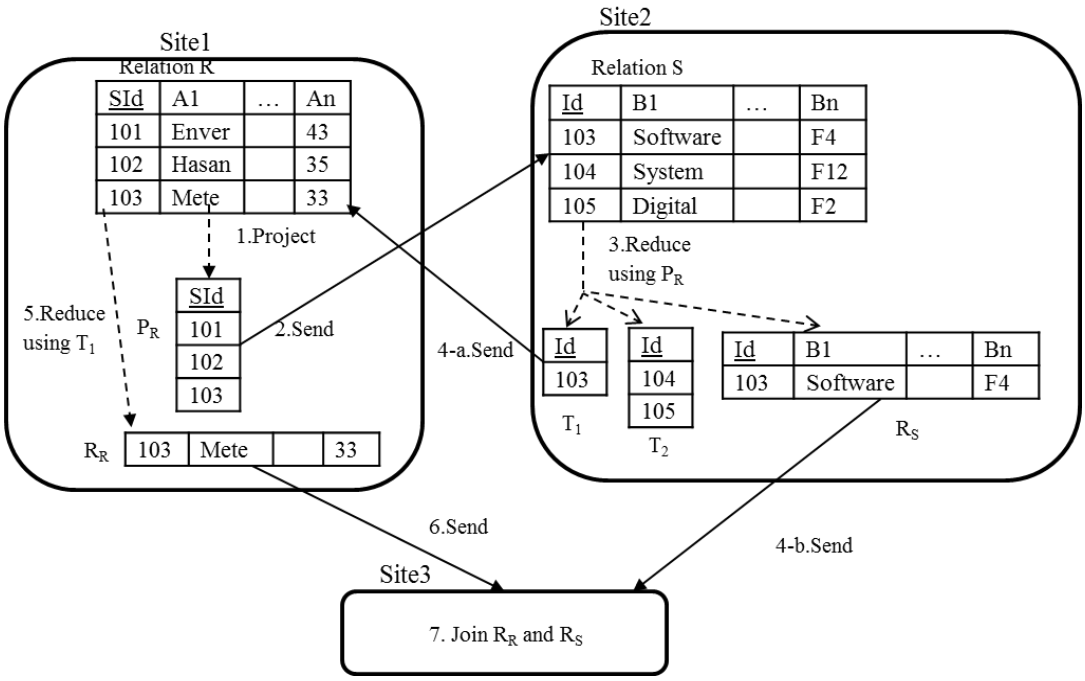

Figure 2.6: Distributed Semi Join

Briefly, instead of sending whole tuples on network, 2SJ sends the only necessary attributes. These attributes determine which tuples of both relations will be in the result relation, avoiding the transmission of non-matching tuples. However, there is an extra communication cost involved with 2SJ, which are the transfers of projected and reduced relations. If the whole size of the tuples of relations R and S are not much bigger than the join attribute, and the matching ratio between R and S is high, 2SJ can bring extra cost for join operation. Databases use statistics based approaches to determine if 2SJ is beneficial for a join operation.

We implemented 2 variations of reduce operations based on which site they will run. These reduce operations are not visible to the last user and employed only during 2SJ.

2.3 Computer Cluster

In a DDB, sites communicate with each other via a computer network. The network can vary in size such as a small LAN in an office building or a WAN connecting different countries. For this study, we used a high performance computer cluster (HPC) to execute distributed queries. The cluster consists of 46 compute nodes. Each node has its own primary and secondary storage, operating system and two quad-core processors, making them able to run individually. Note that we are not concerned with the parallelism inside a node in the scope of this study. Cluster communicates with the external world through a master node. User data is stored on a common hard disk that each node can access. Primary storages of nodes are not directly accessible and used for internal purposes such as caching. Computational communication between nodes is performed with InfiniBand network architecture.

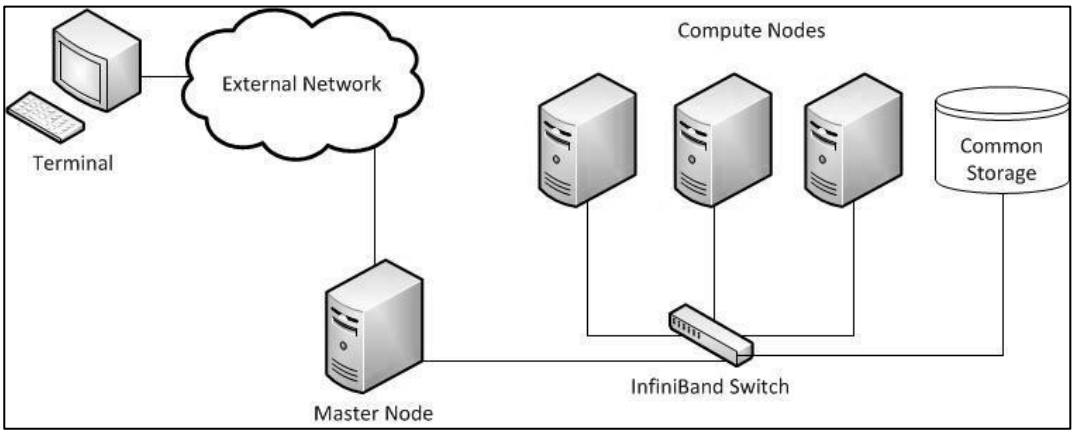

Figure 2.7: HPC Cluster Architecture

2.3.1 Message Passing Interface

Nodes of the cluster use a message passing specification for communication, called Message Passing Interface (MPI), which is widely used in parallel systems. MPI is a specification with multiple implementations. These implementations make use of the underlying hardware efficiently, while following the specification. This results in standardized, portable and practical message passing applications. [mpi-forum (2009)]

Basically, communication in MPI is achieved through send and receive operations. A node can send data to a group of nodes simultaneously or a single node at a time. Type of the data and a tag information to express metadata is also sent together with the payload. On the receiving side, a node declares the identities of source nodes that it expects data from. If data is received with an expected source and tag, it is accepted and put into a memory buffer. Allocation of this buffer is a user responsibility.

One of the main concerns for MPI in this study is the type of send/receive operations. These operations can be made in two ways: synchronous and

asynchronous. After a synchronous send/receive operation is performed, the flow of the program is suspended until the send/receive application buffer of MPI is available to use again. During blocking period no other instruction can be executed. On the contrary, asynchronous operations do not block and application can continue to run. Sent/received data is put temporarily in a memory area. The actual transmission status of data can be tested programmatically from the application.

Although asynchronous message passing decreases the total time spent on communication, in most cases it increases complexity of the application as data sent/receive status must be managed manually. In our implementation we preferred synchronous methods, as the system requires all the tuples of a relation before performing any operations.

2.3.2 Disk and Network Costs

The computer cluster we used does not only run our distributed queries, it serves other users with completely different applications through a job scheduling mechanism. Nodes assigned for our usage share common resources with other nodes such as network and shared storage. In other words, the load and state of the computer cluster at the moment query is executed critically affects response time. This is confirmed by the real response time differences between consecutive executions of the same query in our experiments.

In order to avoid these issues with real response times, we used simulated response time functionality. Simulated times provide more clear results that we can compare our estimated results to, enabling us to focus on network costs which are the most important factors that determine query response time[Banerjee(1993)].

We consider network and disk costs for simulation. How simulated costs are calculated is explained in Section 3. In this section, we give the simplified base formulas used for calculation of the time required to retrieve a given amount of data from the network or the disk.

To estimate the approximate time required to transfer an amount of data over the network, we use a modified version of the formula given in (2.1). In this formula, $T_{Latency}$ is the time required for one packet of data to be transferred between sites. $T_{Latency}$ is independent from the total data size and mainly dependent on round trip time (RTT) and the processing times at the endpoints of communication.

$$T_{Network} \approx T_{Latency} + \frac{Data\ Size}{Bandwidth} \quad (2.1)$$

Data size and bandwidth are other factors that affect network time. Bandwidth is the measure for expressing how much data a network can transfer per second. Usually bits per second (bps) or its multiples are used as bandwidth unit. In [Sevinc(2011)], the network time required to transfer a number of messages was measured as (2.2) for the cluster we work on. The experimental results this formula is based on are given in Figure 2.8. In our implementation we send relations in 1 message. Using (2.1), we can calculate $T_{Latency}$ is 0.9µs and $Bandwidth$ is approximately 1526 Mbps for our system.

Transfer Time = No. of messages * (0.9µs + 0.005µs * BytesPerMsg) (2.2)

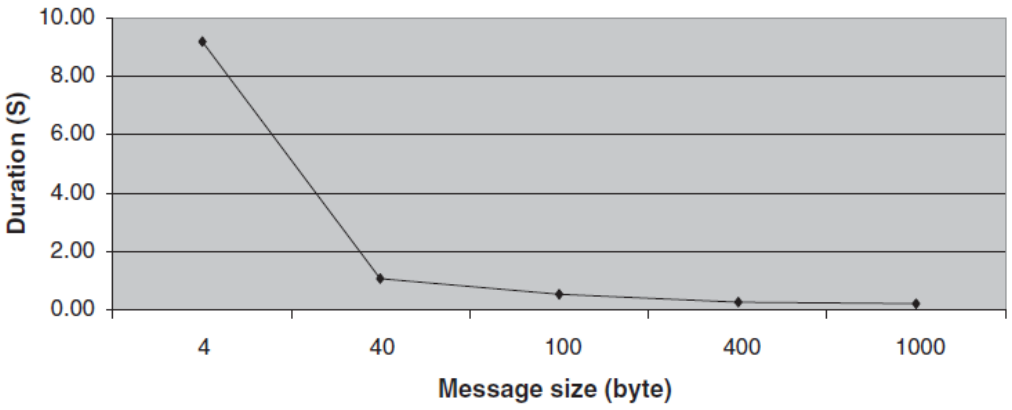

Figure 2.8: Transmission Cost Variation with Message Size [Onder(2010)]

Another time cost we consider is disk I/O time, as copying a page from disk to memory can considerably suspend the processor execution. Data is accessed block by block in hard disk. The smallest unit that can be read is a block. We assume that a relation exists on a single block on the hard disk. However, it is possible that the relation is fragmented and it resides on separate blocks on the hard disk, causing disk transfer time to increase.

Disk block transfer time calculation is done according three factors: seek time, rotational latency and transfer rate. Seek time is the time spent by the read/write head to position itself to the right track of the hard drive. With the read/write head on the track, the required sector takes some time to become available for access, which is rotational latency. Transfer rate means how much data per second is read from hard disk. We base disk cost calculation to the following formula with some modifications in our system:

$$T_{Disk} \approx T_{Seek} + T_{Rot.} + \frac{Data\ Size}{Transfer\ Rate} \qquad (2.3)$$

We used 9ms for T_{Seek}, 4.2ms from $T_{Rotational\ Latency}$ and 40MBps for Transfer Rate.

2.4 Query Optimization Problem

A database query can be executed in many different ways all with the same desired result. However these paths usually have huge differences at their time cost. Query optimizers try to find optimal or close to optimal query execution plans (QEP) in an acceptable time frame. Figure 2.9 shows a query optimizers place in query execution process.

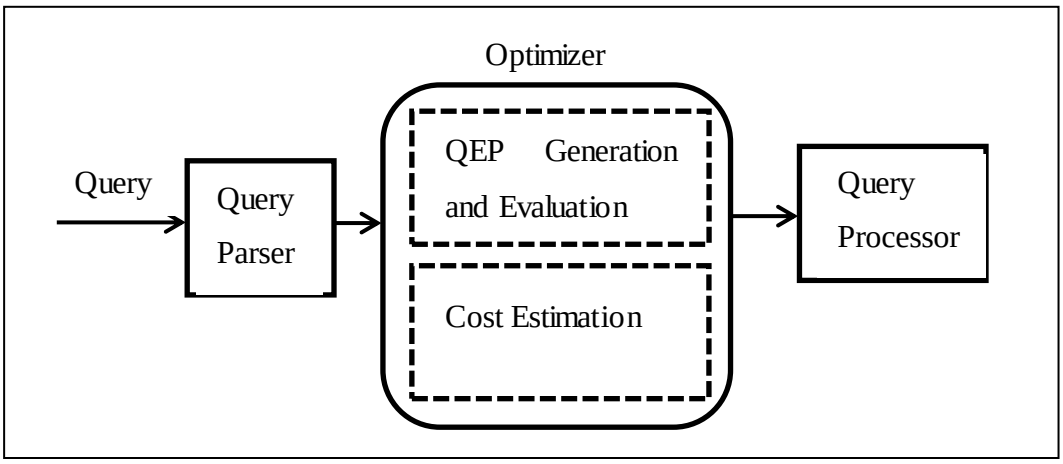

Figure 2.9: Query Optimizer Structure

In order to understand the possible cost difference of plans with the same result, consider we have the following relations:

Person(Name, Age, CityId)
City(CityId, Name, Population)

Two ways to execute the query "**select** Name **from** Person, City **where** Person.CityId = City.CityId **and** City.Population > 1000000 **and** Person.Age = 18" are:

- Build the cartesian product of the two relations by iterating all the tuples of Person for each tuple in City. From the cartesian product, select the tuples which satisfy all the three conditions given in the where statement of the query.

- Select the tuples with Age value of 18 from Person relation and put them in an intermediate relation. Likewise, select the tuples with population greater than 1000000 from the City relation. Scan the selected Person checking if any tuples with the same CityId exists in the selected City relation, adding the join row to the result relation if they match.

Although both execution paths give the same correct results, the second one is clearly more efficient in terms of memory and time. First one creates a cartesian product as intermediate relation, which is a very expensive operation especially for large relations. The cross product contains a lot of unnecessary tuples which will be filtered during selection phase. On the other hand, the second execution path performs single relation select operations first. This reduces the size of the relations that will be used for join, making it a better alternative.

Databases systems use indexing structures such as B+ trees to get better access times on certain attributes. A query optimizer will try to make use of existing indices on a relation in most cases. Because our implementation focuses on network costs, we do not consider the existence of indices for query plans.

The number of possible QEPs can be very large for a query, especially if it involves a lot of relations. For a query with N relations, the size possible join tree space is Ω(N!). In Figure 2.10, possible join trees are depicted that can be built to join three relations R⋈S⋈M. The node at the left side is assumed to be the outer relation of the join.

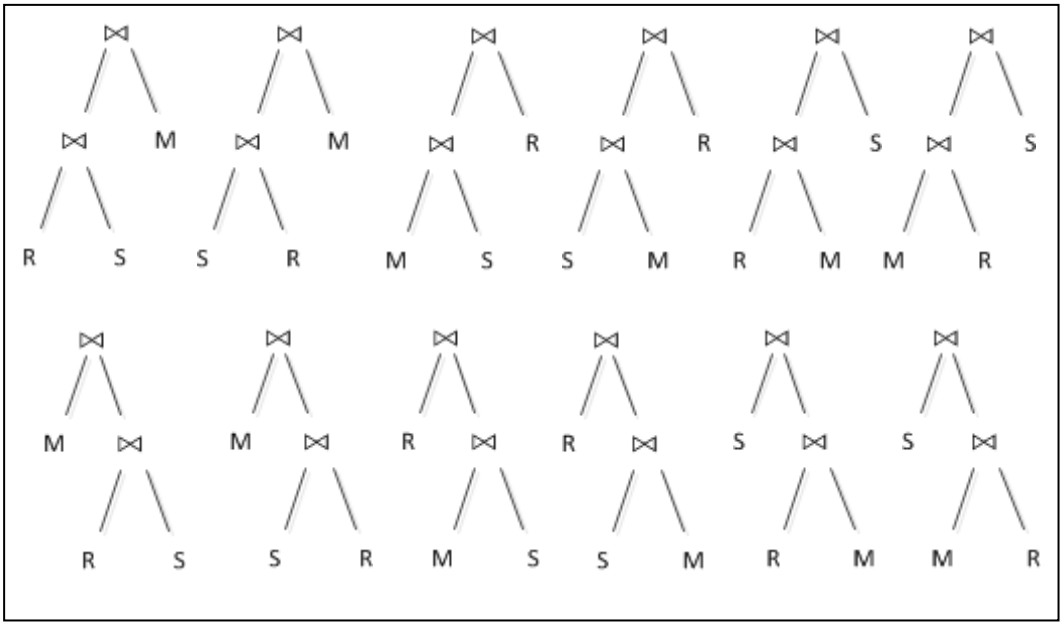

Figure 2.10: Possible Join Orders for a 3-way Join

It can be seen that possible join order number grows very quickly with increasing number of relations. Considering selects, projects, different types of join and which indexes to use further enlarges the search space. To limit the search space, most commercial optimizers follow three restrictions [Ioannidis(1996)]:

- If there is a selection on a relation, it is executed the first time the relation is accessed. On the contrary projections are made after other operations complete.

- Cartesian products of relations are avoided unless explicitly requested in the query.

- The inner operand of a join must be a relation, not an intermediate result obtained from a previous operation. Join trees obeying this restriction are called "left-deep" trees. If a join tree does not follow this restriction it is called a "bushy" tree. This is a heuristic restriction and may eliminate optimal plans. However, best left-deep trees do not have much higher cost than optimal plans in most cases.

For distributed queries, QEP search space is even larger. In addition to possibilities of a centralized query, a distributed query optimizer must also consider which site to perform an operation, how to transfer results and whether to use semi-join or not. Also, as operations can be done parallel, bushy trees cannot be eliminated. These choices make distributed query optimization a harder problem, and dynamic programming based algorithms [Selinger (1979)] used in centralized databases become too expensive. For distributed optimization, other approaches have been proposed such as iterative dynamic programming [Kossmann(2000)] and genetic algorithm based solutions [Sevinc(2011)]. These methods have less complexity than dynamic programming and they still find good enough QEPs.

2.4.1 Cost Estimation

Query optimization algorithms must know the approximate cost of given QEPs to be able to compare them and find a good plan. The aim may be to reduce total time or response time. Total time is the sum of the times spent by the sites involved in the execution of a query. Response time is the time from the start of query execution to the delivery of query result. One may choose to reduce

response time by increasing total time. In this thesis we focus on and use the term "cost" to express response time.

To estimate costs, optimizers employ a size estimation module to roughly predict the size of an operation's result without actually executing it. The accuracy of estimated size to real size greatly determines the success of optimizer. Query optimizers mainly focus on CPU and I/O waiting times, and network delays for distributed databases. CPU time's importance is debated compared to I/O time and in some research it is completely ignored. We also follow this approach in this study. Number of the disk page transfer operations required to read a relation from the hard drive should be estimated correctly to guess good I/O times. For communication costs, mostly number of the bytes sent through the network is used as measurement tool.

Current optimizers base their estimations on the data distributions of attribute values. To keep these frequencies equi-depth histograms are used on many systems [Mousavi(2011)]. Histograms aim to divide the values of an attribute into a number of equally sized buckets. Increasing the number of buckets improves estimation accuracy; however it is more expensive in terms of memory and processing time. When calculating result size of an operation, the size of input relations and attribute data distributions are used as cost formula parameters.

Two assumptions are made on the distribution of data by most optimizers which decrease estimation accuracy but needed to keep estimation cost acceptable [Mannio(1988)].

- Attribute values are uniform, meaning there are equal numbers of tuples for each attribute value. This assumption helps to guess the data distribution of attributes in intermediate results.

- Values of different attributes are independent and they do not correlate with each other. This assumption is wrong in many real life cases but keeping buckets for all combinations of attribute values is not practical. Bayesian network based methods have been proposed to reflect relations between attributes [Getoor(2001)].

In this thesis we generate our own synthetic data based on the rules defined by the user. The metadata for these rules are shared among data generation and query processing modules. Therefore our cost estimations are more exact, allowing us to focus on network costs and design of a specific distributed database.

CHAPTER 3

THE METHODS AND IMPLEMENTATION

This chapter explains how our system is built. First, we briefly describe our working environment and the tools we used for development. Then we give detailed information about system design and implementation.

3.1 Development Environment

Briefly mentioning the tools, libraries and process we used during development is beneficial as this is an implementation emphasized study. As previously stated, our system runs on HPC Cluster with Scientific Linux v5.2 64-bit operating system. The details of the cluster can be found in [metu-hpc(2012)].

We chose C++ as our development language for its object-oriented programming support and speed. To compile and link the source code we used "mpiCC", which is a wrapper for the local gcc compiler that adds constructs for MPI support. Following line is a simple command that builds the executable "runQuery" from all available *.cpp files:

```
mpiCC *.cpp –o runQuery
```

As the cluster serves multiple users, the prepared executable cannot be run directly. A "Portable Batch Script"(PBS) file must be prepared, which basically

states the location of the executable to run and its arguments, as well as the number of nodes to run the executable on. Then this script is submitted to job queue and handled by job-scheduling service of the cluster. An important point for our system is to make sure the number of nodes used in PBS file is equal to number of nodes involved in QEP file.

To access computer cluster from Windows operating system we used "SSH Secure Shell" and the commercial "ZOC Terminal" products. ZOC Terminal's support for scripts helped with recurring tasks. To be more practical, our code was mainly written on Windows and then transferred to the cluster machine. However, one issue encountered with this approach is end of line (EOL) characters. Windows and Linux handle EOL differently, which causes our processor to fail executing queries. If data is generated on Windows and uploaded to Linux, line endings must be converted. The standard "dos2unix" tool can be used for this purpose.

We included some open source external components in our system. During tests we saw that standard random number generator of C++ was not producing uniformly distributed results. Therefore we used a C++ implementation of Mersenne-Twister pseudo-random number generator as an external component [mtrand(2012)]. Another library included in the project was pugixml, which was used for parsing and building of our XML (eXtensive Markup Language) input files [pugixml(2012)]. To represent tree data structures the class published in [treehh(2012)] was used.

3.2 Design and Implementation

The system has an object-oriented architecture. In Figure 3.1, a simplified UML class diagram is given. Note that external libraries and C++ STL are not included

in the diagram. Classes that perform different tasks are loosely coupled. For example, "CDataGen" class, which is responsible for creation of test relations with dummy data, can be executed standalone.

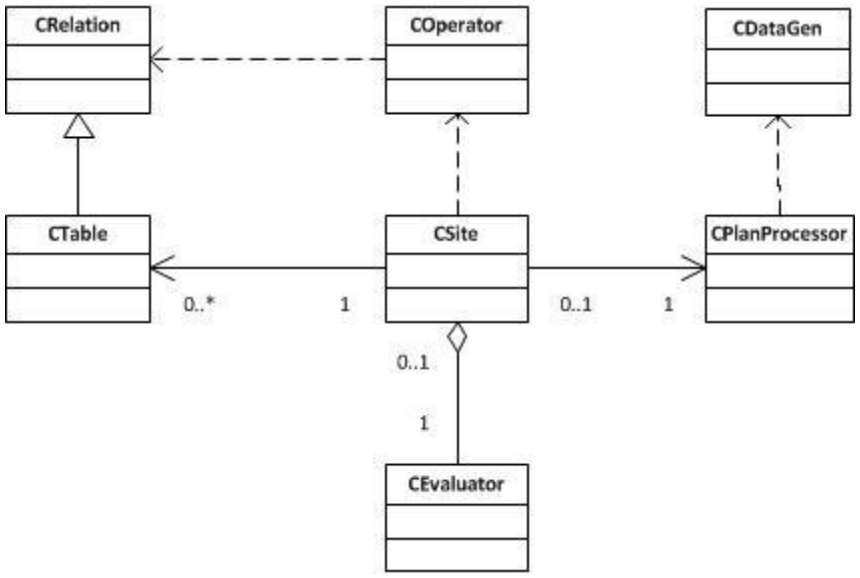

Figure 3.1: System Class Diagram

The responsibilities of the classes are briefly given below. Their working process is explained in more detail later in this section.

- **CRelation:** Represents a database relation. Stores a collection of attributes and tuples. Modification on these collections during runtime is allowed. Values are stored as null terminated character arrays. Metadata of relation such as size, attributes and relation name can be accessed. A relation can print its metadata and actual data to an output.

- **CTable:** Table is a specialized relation. They are physical relations that exist on files residing on the hard disk of a site. In addition to functionalities of a relation, tables can serialize themselves from files.

- **CDataGen:** The responsibility of this class is to generate relation data files according to definitions given in database schema input file. The application must be run in "generation mode" to create files.

- **COperator:** This is a utility class that contains the implementations of database operators. Select, project, nested loop join, sort merge join, hash join and "make distinct" operators are available to use directly from QEP definition files. Reduce operator is implicit and used for semi-join operations.

- **CPlanProcessor:** Plan processor parses the given tree structured QEP file and converts it into a sequential form of commands that is directly assignable to relevant sites. While converting, it infers required commands that are not directly stated in the plan and modifies/adds commands.

- **CSite:** CSite class includes the data and functionality to represent a distributed database site. Each site runs on its own node in the cluster and calls necessary functions required to execute a QEP. A site instance communicates with other sites via MPI infrastructure to exchange commands and data.

- **CEvaluator:** Evaluator class focuses on performance evaluation. It measures the real time, simulated time and estimated time required to execute a QEP. Also, parsing network and disk latency parameters from a file and calculating the cost of transferring/reading a given amount of data is this class' responsibility.

We have given a static overview of our architecture. In order to understand execution logic in more detail, the system can be examined under three sections according to the tasks performed.

3.2.1 Data Generation

Finding data suitable to the design of our system was not possible, so we implemented our own data generator. Our system uses series of numeric digits for attribute values and handles them as character arrays. Data generator produces relations according to this format.

The generator takes an XML input file describing the tables and Primary-Foreign key relations between the tables in the database. As output, it generates relation data files and a modified version of input database schema file.

```
<!--
This database schema input file is used to generate data files.
PK-FK sizes must be the equal.
-->
<DDBSchema>

        <Tables>
                <Table Name="Student" Records="1000">
                        <Column Name="SId" Size="10"/>
                        <Column Name="SName" Size="9"/>
                        <Column Name="C12" Size="2"/>
                </Table>
                <Table Name="Enroll" Records="3000">
                        <Column Name="FSId" Size="10"/>
                        <Column Name="FCId" Size="10"/>
                </Table>
                <Table Name="Course" Records="20">
                        <Column Name="CId" Size="10"/>
                        <Column Name="CName" Size="10"/>
                </Table>
                <Table Name="Staff" Records="30">
                        <Column Name="StaffId" Size="9"/>
                        <Column Name="StaffName" Size="31"/>
                </Table>
                <Table Name="Interest" Records="200">
                        <Column Name="InterestCode" Size="11"/>
                        <Column Name="FStaffId" Size="9"/>
                </Table>
        </Tables>

        <PFKeys>
                <PFKey TableP="Student" PK="SId" TableF="Enroll" FK="FSId"/>
                <PFKey TableP="Course" PK="CId" TableF="Enroll" FK="FCId"/>
        </PFKeys>

        <Selectivities>
                <Selectivity Table1="Staff" Column1="StaffId" Table2="Interest" Column2="FStaffId"
Value="0.003"/>
        </Selectivities>

</DDBSchema>
```

Figure 3.2: Example Database Schema File

Figure 3.2 shows an example database schema input file. Definitions follow standard XML notation and features like comments are supported. Every schema must have a "Tables" section, while "PFKeys" and "Selectivities" sections are optional.

- **Tables Section:** In this section database tables and their columns are defined. For each table, a table entity is created. "Name" attribute states the name of the table, and is used as the table file name. The number of

the tuples is given in "Records" attribute. Attributes are defined as separate "Column" tags. Here, "Name" specifies the name of a column and "Size" specifies the maximum number of characters the column has.

- **PFKeys Section:** Primary-Foreign key associations of relation attributes are defined here. By entering a "PFKey", the user states that two columns are connected and their data is generated accordingly. The column that will be used as primary key determined by setting "TableP" and "PK" fields. TableP is the name of the table that primary key is defined on, and PK is the name of the column that will be used as primary key. Foreign keys are defined in the same manner using "TableF" and "FK" fields.

- **Selectivity Section:** We added an alternative way to associate relations called "Selectivity". Although PFKey is more likely to be used, selectivity is also helpful for some simulations. To define a selectivity association, two columns are entered with their relations using "Table1" - "Column1" and "Table2" – "Column2" field pairs. Then a selectivity "Value" is entered. When the data generator finds a selectivity defined between two columns, it generates data so that if two tables are joined on their selectivity attributes, the result relation's tuple count will be approximately $|R1|*|R2|*Value_{Selectivity}$. Here |R1| and |R2| are the size of relations selectivity is defined for.

Data generator provides PFKey and selectivity by creating unique pseudo-random numbers as cell values. However, ensuring a generated random number is unique among in a set of attribute values is computationally expensive. Also, random number creation functions can give undesired results such returning the same value consecutively. Therefore the effects of PFKey and selectivity

definitions are not exact, but differences are negligible. In Figure 3.3 pseudo-code for data generation is given.

```
for each table
    for each column
        if a selectivity value is defined for this column then
            if connected column is not generated yet then
                generate independent values for this column
            else
                set matchProbability to the probability attribute value will exist on connected column
                for each tuple
                    set random to random number between 0 and 100
                    if random < matchProbability then
                        select a random value from connected column and add it to this column
                    else
                        generate independent value and add it to this column
                    endif
                endfor
        elseif a PFKey is defined for this column then
            set connectedColumn primary key column for this column
            if connected column is not generated yet then
                generate connected primary key
            endif
            for each tuple
                set a random value from connected column and add it to this column
            endfor
        else
            generate independent value and add it to this column
        endif
    endfor
endfor
```

Figure 3.3: Pseudo Code of Data Generation Algorithm

Figure 3.4 depicts a sample relation output file. The first non-comment row states how many records are in relation, number of the attributes and total size of a tuple. Names of the columns and their sizes are written in following lines. Finally, actual tuple values are listed. In the example figure only 5 of 1000 records are given. Comments starting with "//" are supported in the beginning of a relation data file. Note that even the column C12 size is given as 2, there are only 1 character long cell values. This is because the generator sets the range of possible attribute values so that a space character is reserved and cell value is

distinguishable by human eye. This behavior can be overridden and has no effect on how system handles values.

```
//Example relation data file
//First 4 rows are for metadata
1000 3 21
SId 10
SName 9
C12 2
    295889   283426 2
    151677    50359 6
    689169   433490 2
    216877   164116 5
      4602   958296 8
```

Figure 3.4: Example Relation Data File

Another output of data generation process is modified database schema file. Generator creates a file with the name "[InputSchemaName]_output.xml". The content of this file is the same as input schema file, except "Min" and "Max" attributes are added to "Column" tags. The range of values an attribute can have is determined automatically according to its length. This output schema file must be given as an argument when executing a query so that the estimator knows the metadata about relations.

When preparing a schema file, following constraints must be satisfied:

- Table column names must be unique among database. They must be at most 31 characters long.

- If a PFKey or selectivity association is defined between two columns, their size must be equal.

- A selectivity value can be between 0 and $\frac{1}{|R_L|}$, where R_L is the larger of the relations selectivity is defined between.

After preparing a database schema file, tables can be generated by running our executable in generation mode using the following command:

runQuery generate [dbschema_filename]

3.2.2 Query Plan Execution

The main part of our implementation is query plan execution. Given a QEP file in a predefined format, the system can execute that plan and find the result relation. The below command is used to run a QEP. Note that relations data files must be under the same directory with QEP definition file.

runQuery [QEP_filename] [networkparamaters_filename] [dbschema_filename]

Here, "runQuery" is the executable file. [QEP_filename] is the file containing QEP which will be given in this section. [networkparamaters _filename] is used for setting network latency and bandwidth parameters. [dbschema_filename] is the output of data generator.

3.2.2.1 QEP Definition File Format

QEP files are written in XML format, which is particularly good for representing QEPs as they have tree structure. Figure 3.5 depicts an example QEP file. Figure 3.6 is the tree representation for the same QEP.

```
<join type="hash" leftkey="CoursePId" rightkey="CourseFId" performsite="3" targetsite="0" display="t">
        <table site="3">Course</table>
        <join type="sortmerge" leftkey="StudentPId" rightkey="StudentFId" performsite="2" semijoin="t">
                <project attributes="StudentPId,Name">
                    <select criteria="Agelt20,Scholarshipgt300">
                      <table site="1">Student</table>
                    </select>
                </project>
                <table site="2">Enroll</table>
        </join>
</join>
```

Figure 3.5: Example QEP Definition File

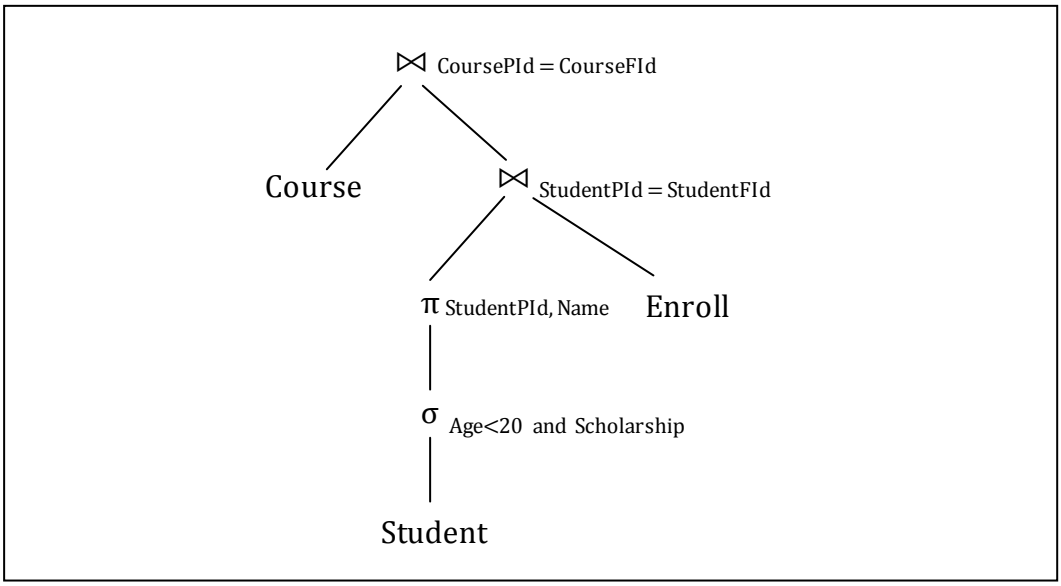

Figure 3.6: Example QEP Definition Tree

XML root of a QEP definition file is an operation. Select and project operations must have one child node, while joins must have two children. A child can be a table node or another operation node. Leafs of the tree can only be table nodes. "Table", "Select", "Project" and "Join" tags can be used in QEP definitions.

Table tag represents a table residing on a site. Table name is entered inside the tag. This tag has only one attribute:

- **Site**: Identifier number of the site the table resides on

The remaining tags are all operation tags. Following attributes are common to all operations and can be used with all of them:

- **Perform site**: Indicates on which site the operation will be executed. For unary operations, this attribute is optional. If no perform site is entered, the plan processor sets the site where the only operand resides as default perform site.

- **Target site**: Target site is required only in root operation. If not entered, it defaults to perform site. It determinates where the result relation will be sent after operation execution is complete. Although the system has the ability to send result any site, this must be set to 0 to be able to evaluate performance and terminate session.

- **Display**: This is also an optional parameter and assumed "f" (false) if not entered. If set to "t" (true), all the tuples of the operation result are printed to standard output. Note that setting this to true for intermediate result can degrade execution performance especially for large results. However displaying the result of the root operation has no side effects as performance measurement is already complete at that point.

The only attribute specific to project is:

- **Attributes**: Comma separated column names that will be projected. '*' character can be used as sole attribute, and it is equivalent to all attributes of a relation.

The only attribute specific to select is:

- **Criteria**: Comma separated condition strings. These strings are formatted as [ColumnName][Condition][NumericValue]. Condition can be "gt" (greater than), "et" (equal to) or "lt" (less than). For example, to find 20 years old students whose Id is greater 1000 we can use "Ageet20,StudentIdgt1000" as select criteria.

Following attributes can be used with join operations:

- **Type:** States the type of join operation. Possible values are "nestedloop", "sortmerge", "hash" and "noop". Noop is not a real join operation. It is included only to measure the time both operands of a join operation are available for experimental purposes.

- **Left Key and Right Key:** Join operation accepts the first child node as left operand and second child node as right operand. Left key and right key are the names of the columns join will executed on.

- **Semijoin:** This is an optional parameter and set to "f" (false) if not entered. If set to "t" (true) the join operation is performed as a semi-join.

3.2.2.2 QEP Processing

QEP definition files are handled by plan processor in our system. The output of plan processor is a list of sequential commands that will be assigned to worker sites from the initiator site, which is site 0.

The creation of command list is done in two phases. In the first phase, QEP definition tree is traversed in preorder and operation structures are created. The recursive algorithm given in Figure 3.7 is used in the first phase. An operation can have one or two operands depending on its type. Each operation result is

given an identifier. This way it is possible to know which operation expects which intermediate result even after tree structure is broken and its fragments are distributed to relevant sites. Before mentioned perform and target inferences are also made in this phase.

```
Operand function traverseTree(xmlnode node, integer targetSite, boolean isRoot)
begin
        Operand result
        if node is table then
                set result.residingSite = get node site attribute
                set result.name = get node child value
        else
                Operation oper
                set result.name = get next generated id
                if target site of oper is not supplied then
                        set oper.opTargetSite = oper.opPerformSite
                endif
                read operation specific fields and set oper
                if oper.Type = Project or oper.Type = Select then
                        set xmlnode singleChild = get only child of the node
                        if "performsite" attribute of tag is empty then
                                set oper.Operand1 = traverseTree(singleChild, -1)
                                set oper.PerformSite = oper.Operand1.residingSite
                        else
                                set oper.PerformSite = get "performsite" attribute of tag
                                oper.Operand1 = traverseTree(singleChild, oper->opPerformSite)
                        endif
                        set result.residingSite = oper.PerformSite
                elseif oper.Type = Join then
                        set result.residingSite = get "performsite" attribute of tag
                        set oper.PerformSite = result.residingSite
                        set xmlnode leftChild = get first child of the node
                        set oper.joinOperand1 = traverseTree(leftChild, oper.PerformSite)
                        set xmlnode rightChild = get second child of the node
                        set oper.joinOperand2 = traverseTree(rightChild, oper.PerformSite)
                endif
                if targetSite > -1 then
                        set oper.targetSite = targetSite
                else
                        set oper.targetSite = oper.PerformSite
                endif
                set oper.result = result
                add oper to the operations list
        endif
        return result
end
```

Figure 3.7: Pseudo Code of QEP Processing Algorithm at Master Site

Figure 3.8 depicts the execution of traversal algorithm on the QEP given in Figure 3.5. The algorithm starts on the root and follows the path shown with dashed arrows. The encircled numbers are the unique names of temporary result relations used as operands in other operations. Commands that will be assigned to the worker sites are created using these temporary names. For example, to represent the outer join a join command is created with two operands. These operands are "Course", which is a table and "2", which is a temporary relation. The result of outer join is given the name "1". Likewise, the inner join has operands "3" and "Enroll", while its result is named "2". The input-output relations between operations are achieved through relation names. When the outer join operation needs to be performed at its worker site, both its input relations must be available in that site.

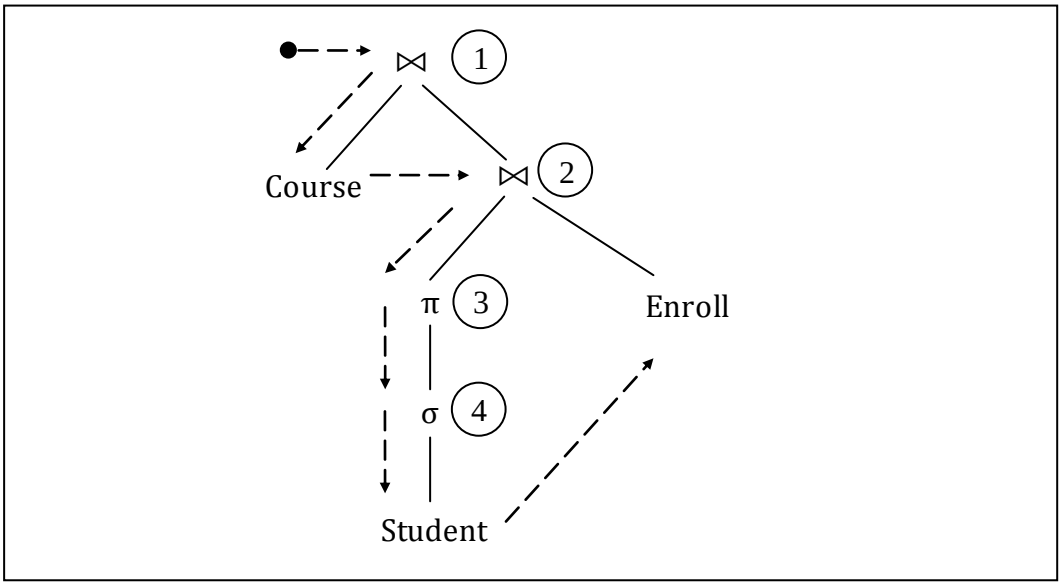

Figure 3.8: QEP Traversal Algorithm

In the second phase of QEP processing, we add implicit operations to the list obtained from first phase. Implicit operations are commands required for query executions but not directly stated in QEP definition such as "Move". For

example, consider a join operation that will join table “Student” from site 2 and table “Enroll” from site 3, and the operation itself will take place in site 2. Obviously, Enroll table must be available in site 2 for site 2 to execute this operation. To make this possible, plan processor creates a “Move Enroll to site 2” command for site 3. Implicit operations are not limited with move commands. For semi-joins, project, make distinct, reduce and move commands are created. In case of a sort-merge join, implicit sorts are generated. Also for every QEP, an implicit “Wait” command is used. This command is assigned to initiator site so that it does not terminate right after assigning commands to worker sites and waits to perform evaluation tasks.

3.2.2.3 System Architecture

The architecture of our system consists of one initiator site and several worker sites. Initiator site is responsible for running plan processor, assigning relevant commands to worker sites, receive the final QEP result and measure performance. On the other hand, worker sites await commands and execute them, sending results to required destinations.

Figure 3.9 gives an overview of our system. When the program is started, all sites start to run together. Initiator site parses QEP and sends each worker site the number of operations they need to execute. This is required for worker sites to terminate properly. Then the operations they need to execute are sent to each worker site. At this point initiator site starts to wait for QEP execution to complete.

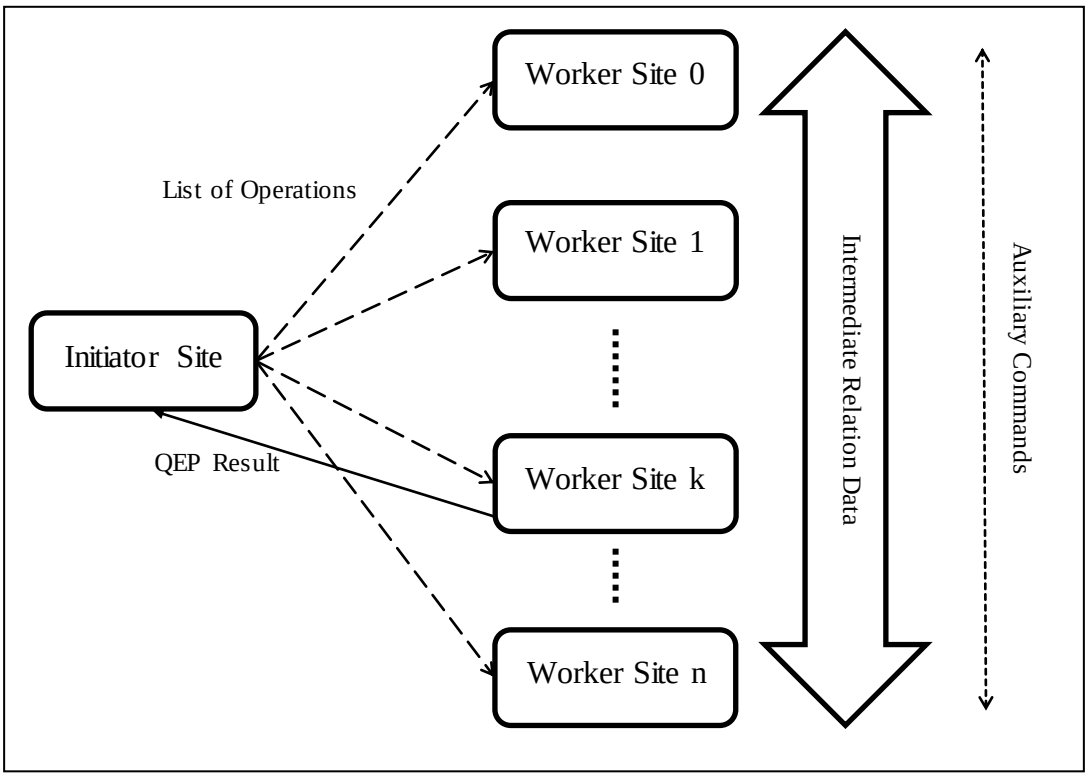

Figure 3.9: Inter-Site Message and Data Flow

Worker sites store the received operations they need to execute on a list. When they complete an operation or receive a relation from another worker site, they iterate the command list to see if an operation is performable, i.e., all its operands are locally available. If so, that operation is performed. The pseudo code for the algorithm running in worker sites is given in Figure 3.10.

```
while operation list is not empty do
        if data needs to be received then
                receive and store data in memory
        endif
        do
                set operation completed false
                for each operation in list
                        if all operands are available then
                                perform operation
                                remove operation from list
                                set operation completed true
                                break
                        end if
                endfor
        while operation completed
endwhile
```

Figure 3.10: Pseudo Code of Command Execution Algorithm at Worker Sites

It is possible that an intermediate relation needs to be sent to another site as a result of an operation. In this case, an auxiliary command "Receive" command is sent to destination site first. This way MPI receiver knows the metadata of the relation it expects and allocates required buffers. If the intermediate result will be consumed locally, it is stored in memory. This "wait-check availability-perform operation" cycle repeats until all the operations are executed. Then the worker site terminates.

The initiator site stops waiting after receiving QEP result, measures performance, logs them to a file named "results.txt" and terminates. A sample results file is given in Figure 3.11. Note that only execution summaries are stored in this file. More detailed error and debug level logs, as well as relation data are kept in output files created by job scheduling system. These files names are in [jobname].[ojobid] format. Flow of commands and result relation data can be found in these files.

Date	Query File	Real Time(ms)	Simulated Time(ms)	Estimated Time(ms)
26.02.2012 17:11	query_latmax.xml	13	13.3011	13.3011
08.04.2012 15:27	query_latmax.xml	415	113.384	113.384
14.04.2012 10:34	query_course.xml	92	14.9325	27.3

Figure 3.11: Example Result File

3.2.3 Performance Evaluation

The last capability we implemented in this study is the evaluation of QEPs. For this purpose we use three time measurements: real time, simulated time and estimated time.

3.2.3.1 Real Time

In our system real time measurement is made solely at initiator site, using OS provided clock function. We have to depend on one site to measure real time, as system clocks of different sites are not precisely synchronized. Initiator site starts its timer after assigning commands to the worker sites. The timer is stopped when QEP result is received. As mentioned before, HPC cluster's purpose and architecture is not directly fit to distributed query simulation. This causes differences in response times up to 3 times between consecutive executions of the same query. Also, there are a lot of details in real execution we cannot consider in the scope of this study.

3.2.3.2 Simulated Time

To overcome issues with real time, we implemented response time simulation, which take disk and network times into account. There is a basic idea behind simulation. The point in time for an operand to be available in a receiver site equals to the point in time for that operand is available in sender site plus the time interval required to transfer the operand through network. In other words, if a

relation R becomes available in site i at time Tav_R^i , and R needs to be transferred to site j, Tav_R^j is calculated as in (3.1), where $Network_R^{i,j}$ is the time spent on network to transfer R from site i to site j. Calculation of $Network_R^{i,j}$ is made according to (3.2), using parameters supplied in the parameters file.

$$Tav_R^j = Tav_R^i + Network_R^{i,j} \quad (3.1)$$

To calculate disk latency, (3.3) is used when a table is first read from hard disk. For intermediate relations disk latency is 0. As a site knows it needs to read a table from disk at the start of execution, we assume each site reads its first table at t_0, and continues to next tables from the time previous reading was completed.

In our trials we saw that impact of the tuple count was greater than the impact of the tuple size for both network and disk costs. For example, transferring or reading a relation with 10000 tuples where each tuple is 10 characters long took more time than a relation with 1000 tuples where each tuple is 100 characters long. In other words, the time it takes to create a relation from a network stream or disk is not a function of only data size for our system. Therefore, we modified formulas given in (2.1) and (2.3) by converting them to use tuple count and tuple size with some coefficients. Formula (3.2) is used for network costs, while (3.3) is used for disk costs. For relation R, |R| donates the tuple count and SZR donates tuple size. The constant numbers used in formulas are based on experimental results explained in next chapter.

$$T_{Network} = T_{Latency} + |R| * (\frac{1.984}{Bandwidth} + 0.000008 * SZR) \quad (3.2)$$

$$T_{Disk} = (T_{Seek} + T_{Rot.}) + |R| * (\frac{0.0046}{Transfer\ Rate} + SZR * 0.000002) \quad (3.3)$$

For unary operations, the result's availability time is equal to its only operand's availability time, because we ignore CPU costs. For binary operations, the result's availability time is the maximum of its operands' availability time.

Network parameters that are used for simulated and estimated times can be configured via a text file. Comments starting with "//" sequence are supported. This file represents an adjacency matrix where each cell keeps bandwidth and latency values. For example, the value in 3rd row, 4th column is used when data is being sent from site 2 to site 3. The adjacency matrix does not have to be symmetrical. Bandwidth unit is Mbps (megabits per second); latency unit is ms (millisecond). These values are separated with a "|" character. Latency can be omitted; in that case the default value 0.0009 ms is used. A negative value means very low bandwidth and very high latency. A sample network parameter file is given in Figure 3.12.

```
//Matrix values are separated either with tabs or spaces
//Edges are separated with pipes (|) -> Bandwidth|Latency
//Bandwidth unit is Mbps, Latency unit is ms
//Default value for latency is 0.0009

0       1       3       2       1
1       0       -1      2       1
3       -1      0       2       1
2       2       2       0       1
1       1       1       1       0
```

Figure 3.12: Example Network Parameters File

3.2.3.3 Estimated Time

Our system can estimate the approximate simulated response time of a QEP without executing it, which is useful for selecting a good plan in QEP search space. Select, project and non-semi joins are supported for estimation. Sort, move and make distinct operations can also be estimated but they are not directly available in QEP definition.

To estimate cost, the evaluator applies the same formulas used in latency simulation. First, the QEP tree structure is rebuilt from the operation list with additional implicit commands, which is the output of plan processor. Then, this tree is traversed in post order. This allows us to calculate each operations time as maximum of its children's availability time plus its own network transfer time.

The biggest challenge in cost estimation is guessing the size of intermediate relations, which is needed for network time calculation. To do this, we use the advantage of knowing characteristics of synthetically generated tables. Each operation type is handled specially during result size prediction.

- **Sort:** Sort does not change relation size.

- **Move:** Move does not change relation size.

- **Make Distinct:** As generated attribute values are mostly unique, we assume make distinct operation eliminates 1% of tuples.

- **Project:** The result size of a project operation can be calculated exactly, as size of each column and tuple count is known.

- **Select:** Estimation of select operations is limited to one select condition. Also it is assumed that numeric attribute values are uniformly distributed.

- **Join:** Join results sizes can be approximately calculated with help of PFKey and selectivity associations defined in database schema file. For PFKey, join result's tuple count is equal to the relation foreign key is defined for. If selectivity is defined, join result tuple count is multiplication of input relations' and selectivity value. Once tuple count is known, it is straightforward to calculate relation size as tuple size is sum of input relations' tuple size.

CHAPTER 4

EXPERIMENTAL RESULTS

In this section we perform experiments to see various aspects of our system and discuss obtained results. In [Onder (2010)], the nodes of the cluster we use were shown to be equivalent, therefore we do not perform those experiments. Our trials focus on query processing aspects of our system.

The relations that we use for experiments in this section are written in RelName(Att1Name:Att1Size, Att2Name:Att2Size,..):TupleCount format. Attributes used as primary key and foreign key will start with "P_" and "F_" prefixes respectively. For example to express a relation named "Student" with 10000 tuples and two attributes, a 10 characters long "Id" and 30 characters long "Name", we use Student(P_Id:10, Name:30):10000. The "P_" prefix in "P_Id" states it is a primary key.

4.1 Correctness

The first thing we need to verify to continue our experiments is the correctness of our results. We try data generation and perform operations to see if they work as expected.

4.1.1 Data Generation Correctness

For data generation we produce relations connected with PFKey and selectivity. Then we join them to see resulting tuple count.

PFKey: We test PFKey association with two relations: Player(PlayerId:10, F_CityId:10):1000 and City(P_CityId:10, PlateNo:3):80. When we join these relations on key attributes, we saw that the result had 1000 records. This is expected, as each record in Player matches with a record in City. The result size is equal to the relation with foreign key.

Selectivity: Selectivity is also tested with two relations: Restaurant(ResName:10, F_RCity:10):1000 and Cinema(CiName:10, F_CCity:10):1000. F_RCity and F_CCity attributes are connected with a selectivity value of 0.0003. The join between these relations resulted in 270 records. The expected value is 1000*1000*0.0003 = 300. The difference is caused by the probabilistic function data generator uses and acceptable.

4.1.2 Operation Correctness

To test operation correctness we use two small tables as input so that results can be shown and checked by eye: Employee(P_EmpId:10, EmpName:9, EmpAge:3):10 and Education(F_EmpId:10, EdCode:4):8. Select, project and join operations are executed one by one. Unless otherwise stated, only one worker site is used aside from the initiator site. The contents of two relations we use are given in Table 4.1 and Table 4.2.

Table 4.1: “Employee” Relation

Record#	P_EmpId	EmpName	EmpAge
1	155	3571	20
2	1808	2867	73
3	6625	8672	84
4	3397	5044	52
5	8558	7527	77
6	924	87	14
7	4113	7998	32
8	5609	9971	38
9	4272	9579	91
10	4450	1834	37

Table 4.2: "Education" Relation

Record#	F_EmpId	EdCode
1	8558	467
2	4450	425
3	6625	46
4	1808	794
5	4272	890
6	4113	856
7	8558	941
8	8558	158

Select: Select is performed with the following QEP definition and result is shown in Table 4.3:

```
<select criteria="EmpAgegt30" targetsite="0" display="t">
        <table site="1">Employee</table>
</select>
```

Table 4.3: Result of Select operation on Employee table

Record#	P_EmpId	EmpName	EmpAge
1	1808	2867	73
2	6625	8672	84
3	3397	5044	52
4	8558	7527	77
5	4113	7998	32
6	5609	9971	38
7	4272	9579	91
8	4450	1834	37

Select operation filters the records that do not satisfy EmpAge>30 condition as expected.

Project: Project is performed with the following QEP definition and its result is shown in Table 4.4:

```
<project attributes="P_EmpId,EmpAge" targetsite="0" display="t">
        <table site="1">Employee</table>
</project>
```

Table 4.4: Result of Project operation on Employee table

Record#	P_EmpId	EmpAge
1	155	20
2	1808	73
3	6625	84
4	3397	52
5	8558	77
6	924	14
7	4113	32
8	5609	38
9	4272	91
10	4450	37

Project operation only displays the attributes stated in QEP definition as expected.

Join: Join is performed with the below QEP definition and result is shown in Table 4.5. Even though only hash join is given below, nested loop and sort-merge joins were also tried. Their results were the same, only difference was sort-merge join's result was sorted on key attribute. Semi-join with hash algorithm was experimented with two worker nodes. Again, the result was the same.

```
<join type="hash" leftkey="P_EmpId" rightkey="F_EmpId" performsite="1" targetsite="0" display="t">
        <table site="1">Employee</table>
        <table site="1">Education</table>
</join>
```

Table 4.5: Result of Join operation on Employee and Education tables

Record#	P_EmpId	EmpName	EmpAge	F_EmpId	EdCode
1	1808	2867	73	1808	794
2	6625	8672	84	6625	46
3	8558	7527	77	8558	467
4	8558	7527	77	8558	941
5	8558	7527	77	8558	158
6	4113	7998	32	4113	856
7	4272	9579	91	4272	890
8	4450	1834	37	4450	425

Join result attributes are from both input relations and key values are the same. There are no missing records with equal keys in input relations, meaning join works correctly.

Composite Operation: In addition to simple operations, we executed a composite operation that consists of a select, project and join operation. Also, this query is executed on two worker sites. Our purpose here is to show that result of an operation can be used as input for another operation. The result is shown in Table 4.6:

```
<project attributes="EmpName,EdCode" targetsite="0" display="t">
        <join type="hash" leftkey="F_EmpId" rightkey="P_EmpId" performsite="1">
                <table site="2">Education</table>
                <select criteria="EmpAgelt50">
                        <table site="1">Employee</table>
                </select>
        </join>
</project>
```

Table 4.6: Result of Composite operation

Record#	EmpName	EdCode
1	1834	425
2	7998	856

Result relation is as expected. This is easier to see by checking Table 4.5 for tuples with EmpAge<50. 6^{th} and 8^{th} records satisfy this criterion.

4.2 Correlation between Real and Estimated Costs

When the execution times the system estimate using the raw disk cost (2.3) and raw network cost (2.4) formulas were compared real execution times, it was seen that the real times were not reflected correctly. In order to tune our formulas, we have run simple QEPs that isolate the time required to create a relation from a disk file as well as the time required to create a relation from network stream. Collected data suggests that network and disk costs are bound tuple count and tuple size, rather than just relation size. The cost formulas were revised with these two parameters. Also coefficients derived from experimental data were included, resulting in new formulas given in (3.2) and (3.3).

To measure the time required to create a relation from disk, we conducted two experiments. The experimental raw data used in this experiment is given in Table A.1**Error! Reference source not found.**. First, we increased tuple count of a relation while keeping tuple size constant. Figure 4.1 depicts how disk time changes according to tuple count for relation with tuple size 10 bytes. The real time measured, the raw disk cost formula's (2.3) estimation and the modified disk cost formula's (3.3) estimation are represented with different lines. It can be seen that modified disk cost formula reflects real time better. The increase in raw disk formula is not even visible on this scale. The reason for the difference between

raw cost formula's estimation and real time is that real time is affected by the relation construction.

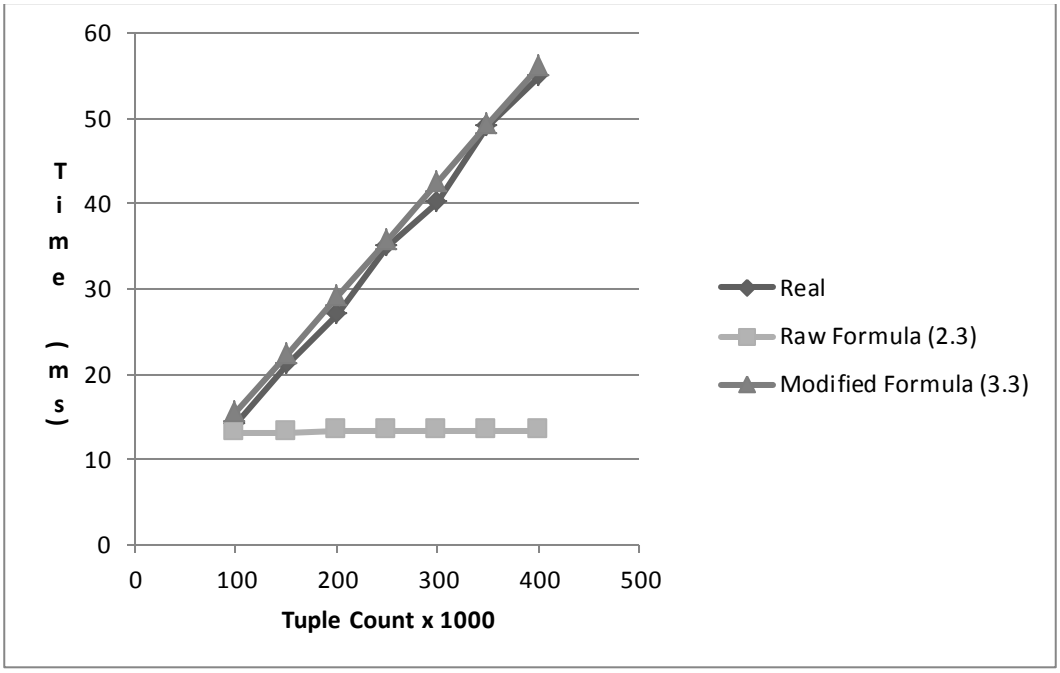

Figure 4.1: Change in Disk Read Time with Increasing Tuple Count

Next experiment is similar, however this time tuple count is set to 1000000 as constant and tuple size is changed. Figure 4.2 shows the results for this experiment. This chart has a similar pattern. However, we can see that tuple size does not impact disk cost as much as tuple size.

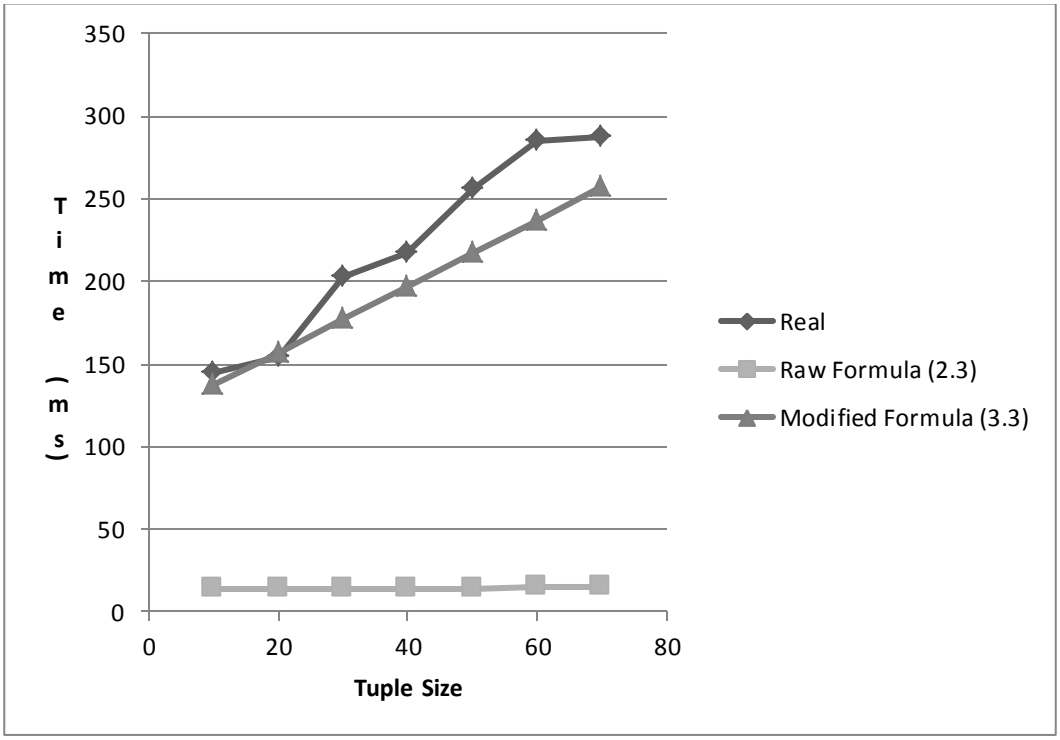

Figure 4.2: Change in Disk Read Time with Increasing Tuple Size.

We applied the same logic for network experiment. Figure 4.3 shows how the time required to create a relation from network changes with tuple count. Tuple size is 10 bytes. Again, we can see modified network formula (3.2) gives closer results to real time than raw network formula (2.1).

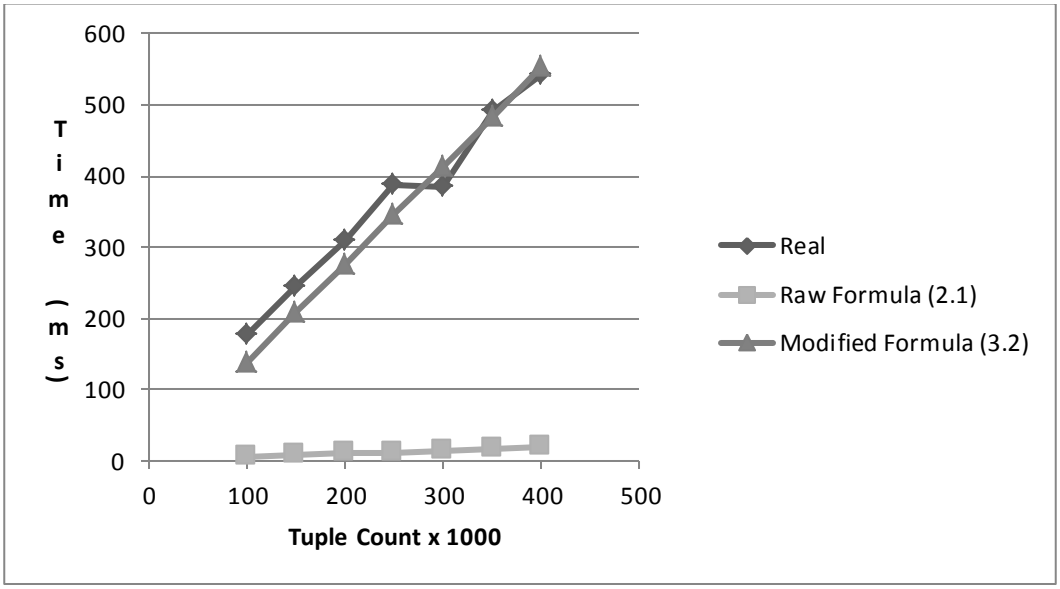

Figure 4.3: Change in Network Time with Tuple Count

The last experiment is changing the tuple size while keeping tuple count at 1000000. The results are given in Figure 4.4. It can be seen that tuple size is not as important as tuple count in network times.

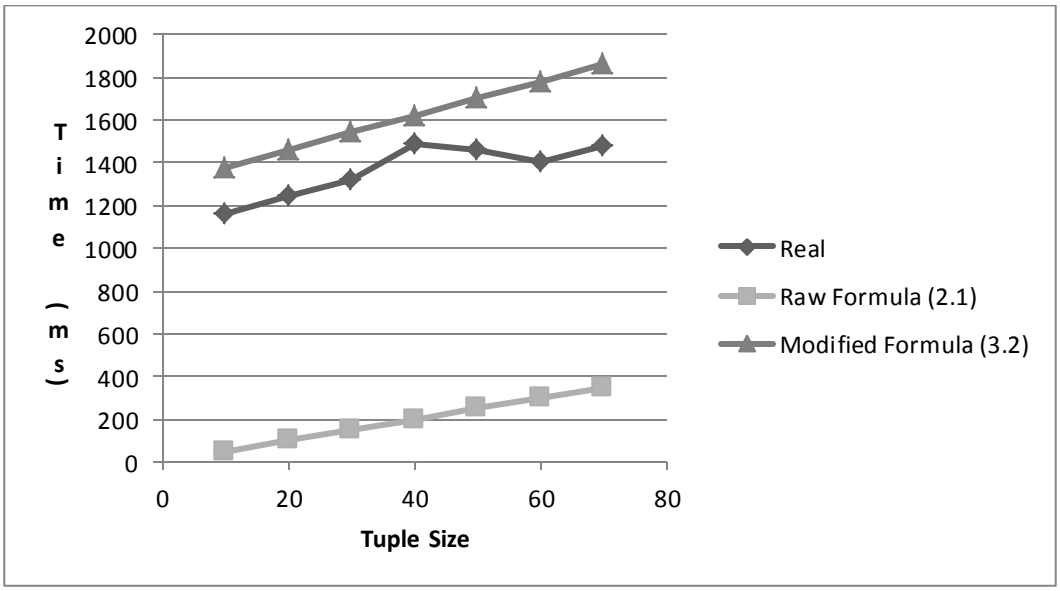

Figure 4.4: Change in Network Time with Tuple Size

Overall, it is clear that tuple counts and sizes must be handled separately for disk and network time estimation, and there is a need for coefficients for accuracy.

4.3 Comparison of Join Algorithms

In this experiment we compare the real response times of nested-loop, sort-merge and hash join algorithms. All the joins are performed using initiator site and one worker site. The same input relations were used for all algorithms. These relations were connected with PFKey association.

The Cartesian product size of input relations was changed in each run. Table 4.7 shows tested sizes, where A(P_A:10, AName:30) and B(F_A:10, BName:30) are input relations. The average measured times of 5 runs for join algorithms are given in Table 4.8. Figure 4.5 is the chart representation. Note that vertical axis in the chart uses logarithmic scale.

Table 4.7: Input Sizes

Input#	**|A|**	**|B|**	**|A| x |B|**
1	10	10	100
2	100	100	10,000
3	1,000	1,000	1,000,000
4	10,000	10,000	100,000,000
5	100,000	100,000	10,000,000,000

Table 4.8: Measured Real Times for Algorithms

Size/Time(ms)	Nested-Loop	Sort-Merge	Hash
100	7	1	5
10,000	11	18	7
1,000,000	641	80	30
100,000,000	58477	5025	203
10,000,000,000	5874204	489758	1524

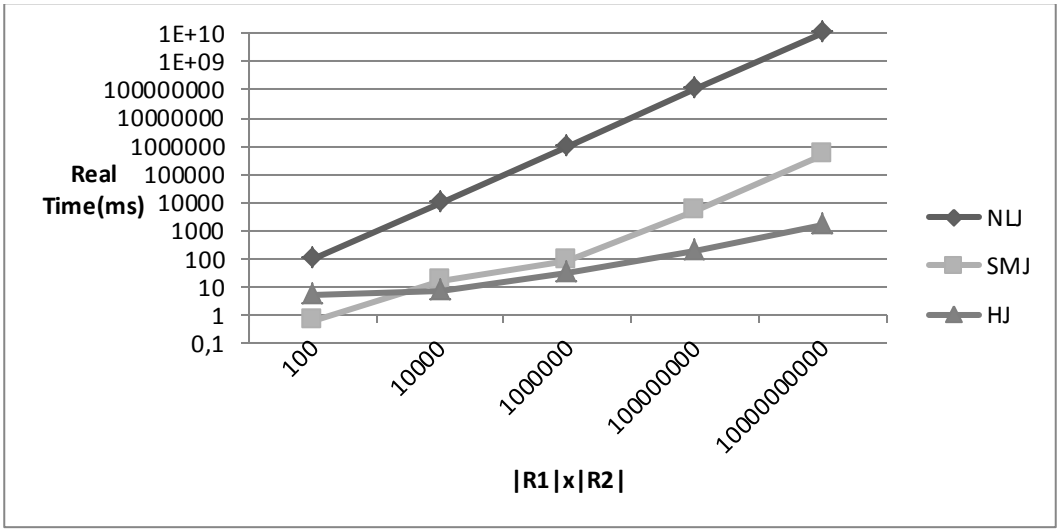

Figure 4.5: Measured Times for Algorithms

We can roughly say that hash join performs best and nested-loop performs worst overall. The difference in performance gets clearer with the increase in data size that needs to be processed. We must note that sort merge could perform better than hash join but our data is not sorted.

4.4 Effect of Network Bandwidth

We performed experiments to see how simulated response time is affected by network bandwidth. In this test setup, we run a join with two worker sites, which

requires a relation to be transferred to the site where it will be joined locally. Input relations are Applicant(AppName:10,F_HsId:10, F_UniId:10):1000, HighSchool(P_HsId:10, HsName:70):10000 and University(P_UniId:10, UniName:70):10000. The query is as following:

```
<join type="hash" leftkey="F_HSId" rightkey="P_HSId" performsite="1" targetsite="0">
        <table site="1">Applicant</table>
        <table site="2">HighSchool</table>
</join>
```

As operation will be performed at site 1, HighSchool relation is transferred from site 2 to site 1, and the final result is transferred from site 1 to initiator site. To minimize the impact of this transfer, we increased the bandwidth of the network line between site 1 and site 0 to a very high value (1000000Mbps). For network delays we use the default value, 0.009 ms, which is negligible during transfers of relations this large. We increased the bandwidth between sites 1 and 2 in every test run. We also run the same query with semi-join flag set to true. The results are given in Table 4.9. Figure 4.6 reflects these results on logarithmic scale.

Table 4.9: Effect of Bandwidth

Bandwidth (Mbps)	**Join Sim. Time (ms)**	**Semi-join Sim. Time (ms)**
1	6116.74	672.079
10	623.573	79.0917
100	74.2569	19.793
1,000	19.3252	13.8631
10,000	13.8321	13.2803

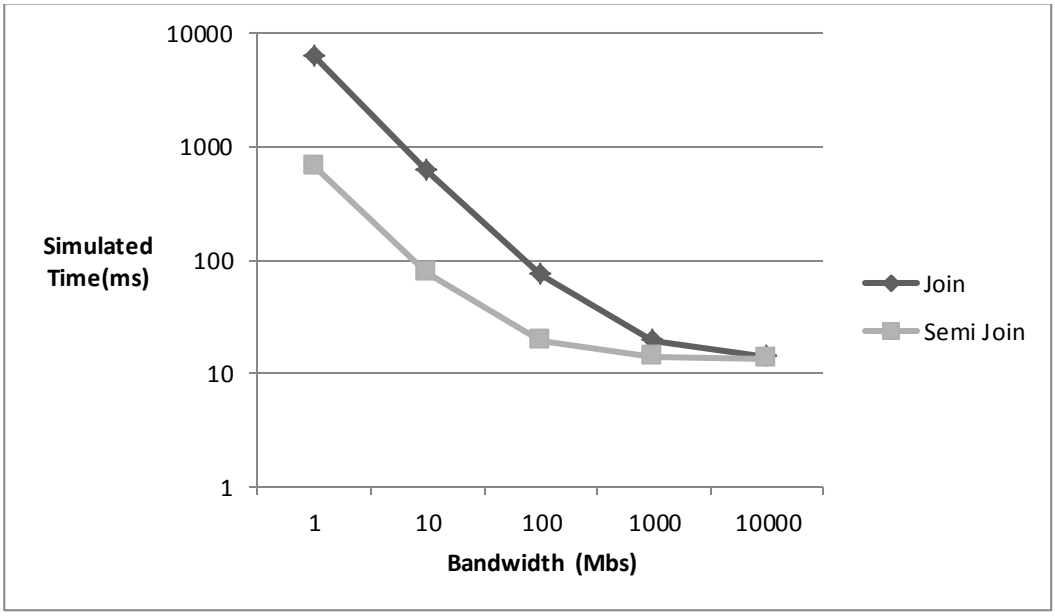

Figure 4.6: Effect of Communication Link Bandwidth

From these results we can see that response time gets better with wider bandwidth values. However, the decrease in response time is not linear as bandwidth gets bigger, it loses its position as a bottleneck and other factors such as disk times and latencies become more determining.

When comparing join with semi-join, we see that semi-join is advantageous over normal join for this query. However, as bandwidth loses its importance, the performance difference gets smaller. In this case semi-join can be actually harmful at some point as requires more processing locally.

4.4.1 Influence on QEP Selection

We made another experiment to observe how network infrastructure can affect selection of QEPs. For this purpose, we first executed the following QEP in two different network configurations. Then we changed the perform site of the inner

join and executed again with the same network configurations. Network parameters and simulated response times are given in Table 4.10.

```
<join type="hash" leftkey="F_UniId" rightkey="P_UniId" performsite="3" targetsite="0">
        <join type="hash" leftkey="F_HSId" rightkey="P_HSId" performsite="2">
                <table site="1">Applicant</table>
                <table site="2">HighSchool</table>
        </join>
        <table site="3">University</table>
</join>
```

Table 4.10: Network Parameters

Inner Join Site	**Bandwidth between Site1 and Site 3 (Mbps)**	**Bandwidth between Site2 and Site 3 (Mbps)**	**Simulated Response Time (ms)**
1	100	1	82.61
2	100	1	850.37
1	1	100	909.19
2	1	100	23.77

Here we can see that executing the inner join on site 2 can be a good or bad choice, depending on the network speed between sites. This is because the result of the inner join must be transferred to site 3 for further operations. We can conclude that, if the optimizer is comparing QEPs that can execute the same operation in different sites, it must consider network speeds.

4.5 Bushy QEP Execution

Another experiment we conducted was about bushy execution of joins in a query plan. In centralized databases, bushy plan trees are avoided as a heuristic. However in a distributed system, performing parallel operations can be beneficial in terms of performance.

4.5.1 Running The Same QEP Concurrently

In the first experiment, we prepared two join queries that will abstract our results from unwanted factors as much as possible. All bandwidths are equal between sites. We have two relations: A(AName:40, F_BId:10):10000 and B(P_BId:10, BName:40):10000. Our first query is a single-join as given below:

```
<join type="hash" leftkey="F_BId" rightkey="P_BId" performsite="1" targetsite="0">
        <table site="1">A</table>
        <table site="2">B</table>
</join>
```

In our second query, we replicate the same single-join in different sites. Then both the original and replicated queries' results are sent to initiator site. This is achieved through before mentioned "noop" type join, which only awaits its operands become available to complete. Our second query is given below. Table 4.11 depicts the average real times for 5 runs of these queries.

```
<join type="noop" leftkey="F_BId" rightkey="P_BId" performsite="0" targetsite="0">
    <join type="hash" leftkey="F_BId" rightkey="P_BId" performsite="1">
        <table site="1">A</table>
        <table site="2">B</table>
    </join>
    <join type="hash" leftkey="F_BId" rightkey="P_BId" performsite="3">
        <table site="3">A</table>
        <table site="4">B</table>
    </join>
</join>
```

Table 4.11: Concurrent QEP Join Time

QEP	Real time (ms)
Single-Join	158.4
Bushy-Join	189.4

Even though we perform equivalent single-join operations twice in the bushy join, we can see that real response time is not doubled. This was our expectation because the joins are processed in different sites. The small difference between times can be caused by the extra transfer of A⋈B from site 3 to initiator site.

4.5.2 Left-Deep and Bushy Tree Comparison

In the second experiment, we compare a left deep tree and a bushy tree with equivalent results. Four relations are used in this setup: A(F_BId:10):10000, B(P_BId:10,F_CId:10):10000, C(P_CId:10,F_DId:10):10000, D(P_DId:10):1000 All bandwidths are equal between sites. The left-deep QEP we used is as following:

```
<join type="hash" leftkey="F_DId" rightkey="P_DId" performsite="1" targetsite="0">
        <join type="sortmerge" leftkey="F_CId" rightkey="P_CId" performsite="1">
                <join type="sortmerge" leftkey="F_BId" rightkey="P_BId"
                performsite="1">
                        <table site="1">A</table>
                        <table site="2">B</table>
                </join>
                <table site="3">C</table>
        </join>
        <table site="4">D</table>
</join>
```

Following QEP was used as bushy plan:

```
<join type="hash" leftkey="F_CId" rightkey="P_CId" performsite="1" targetsite="0
        <join type="sortmerge" leftkey="F_BId" rightkey="P_BId" performsite="1">
                <table site="1">A</table>
                <table site="2">B</table>
        </join>
        <join type="sortmerge" leftkey="F_DId" rightkey="P_DId" performsite="3">
                <table site="3">C</table>
                <table site="4">D</table>
        </join>
</join>
```

Notice that we chose sort-merge join as join type in this experiment to emphasize CPU times. The average real times for 5 runs of these QEPs are given in Table 4.12.

Table 4.12: Left-Deep and Bushy Tree Comparison

QEP	**Real time (ms)**
Left-Deep	5330
Bushy	2768

We can see that bushy QEP performs better than left-deep QEP. This is because instead of site 1 performing all the sort-merge join processing task, the join between relations C and D is done in site 3 at parallel.

4.6 Performance Evaluation

In this experiment our purpose is to see the relation between real, simulated and estimated times. It is hard to design good experiments for this purpose, as there are a lot of parameters that can change the outcome of results. We used relations

A(AName:40, F_BId:10) and B(BName:40, P_BId:10):1000 for experiments. Bandwidth between all the sites was simulated to be 1526 Mbps.

In order to find how changes in data size affect times, we executed a simple project operation while incrementing the tuple count of relation A. The QEP is as follows:

```
<project attributes="*" performsite="1" targetsite="0">
        <table site="1">A</table>
</project>
```

One of the first things we noticed is that measured real time can vary a lot between consecutive runs. These variations cause noticeable differences, especially in operations with small response times. Table 4.13 shows real response times of 5 consecutive project operations where A has 10000 tuples. To compensate these variations, we run each test 5 times and use the average value as real time. On the other hand, simulated and estimated times do not differ between runs.

Table 4.13: Variations in Real Time

Run#	**Real Time(ms)**
1	89
2	40
3	27
4	25
5	22
Average	40.6

When we executed the operation for different tuple sizes of A, we got the results given in Table 4.14. It can be said that all the time values increase as A's tuple

count increases. Simulated and estimated times are the same, which is an expected outcome as project operation's output size can be estimated exactly.

Table 4.14: Measured Times for Different Tuple Counts

Tuple# of A	Real Time(ms)	Sim. Time(ms)	Est. Time(ms)
1,000	13.4	3.93	3.93
10,000	40.6	21.3022	21.3022
100,000	275.2	195.014	195.014
1,000,000	2058.2	1932.13	1932.13

In addition to this experiment, we run the following operations where A had 100000 tuples:

Select:

```
<select criteria="ANamegt50000000"  performsite="1"  targetsite="0">
        <table site="1">A</table>
</select>
```

Join:

```
<join type="hash" leftkey="F_BId" rightkey="P_BId" performsite="1"  targetsite="0">
        <table site="2">A</table>
        <table site="1">B</table>
</join>
```

Composite:

```
<project  attributes="AName,BName"  targetsite="0">
        <join type="hash" leftkey="P_BId" rightkey="F_BId" performsite="1">
                <table site="2">B</table>
                <select criteria="ANamegt30000000">
                        <table site="1">A</table>
                </select>
        </join>
</project>
```

The results for these operations are given in Table 4.15. We can see that real time measurement, time simulation and estimation are possible for different kinds of operations. Also, there is a small difference between simulated and estimated

times in select and composite operations. This happens because outcome of select operation cannot be known exactly. Because we used a PFKey between relations A and B, join estimation is exact. If they were connected with selectivity, estimated and simulated join times would also not be exactly the same.

Table 4.15: Measured Times for Different Operations

Operation	Real Time(ms)	Sim. Time(ms)	Est. Time(ms)
Select	222.4	109.938	110.007
Join	677	405.028	405.028
Composite	498.6	161.2	160.81

Overall, it can be seen that the simulated and real times are different. There are a lot of factors that cause these differences, such as the CPU costs we ignored. However, there is usually a correlation between simulated and real times. In most cases, this is enough from the perspective of a query optimizer because it only needs to know which QEP has lesser cost. Knowing the exact cost is not important when comparing QEPs as long as the rankings are correct.

4.7 String Comparison and Memory Copy Times

In this experiment we measure the times spent on string comparisons and memory copies. These operations are frequently performed during joins. String comparison is done to find if two attribute values match, and memory copy is done to copy matching tuples to result relation. Measuring these times gives an idea of CPU times spent on operations.

First, we measured the times for comparing strings with 10 characters with increasing comparison numbers. Table 4.16 gives average measured times of 3 runs. Only comparison is performed, no other action is taken. Figure 4.7 shows average times spent on comparisons.

Table 4.16: Measured Times for Increasing Comparison Numbers

Comparison #	Time(ms)
50,000,000	186
100,000,000	282
150,000,000	423
200,000,000	564
250,000,000	706

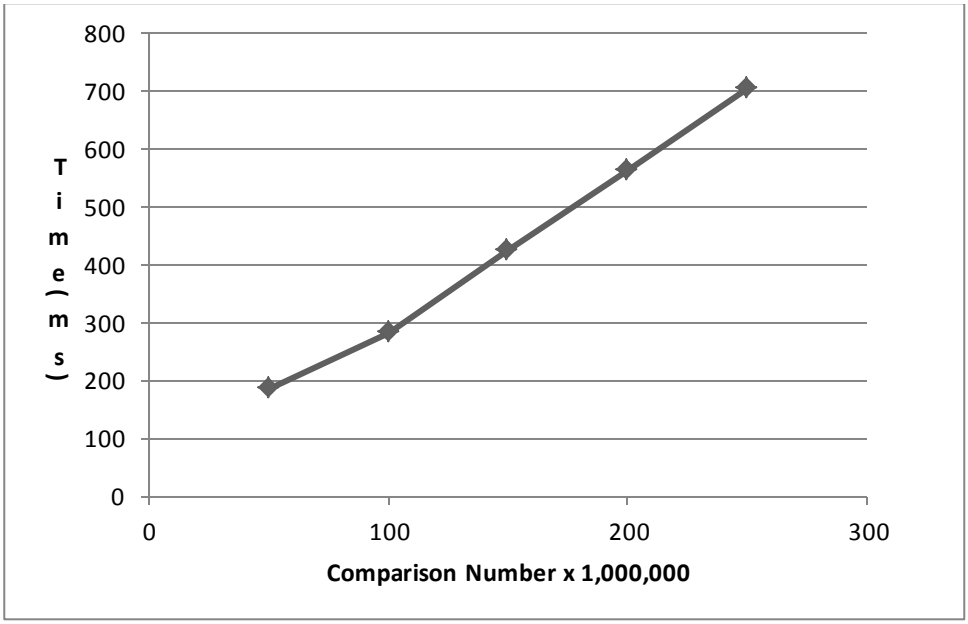

Figure 4.7: Time Spent on String Comparisons with Increasing Tuple Count

Next, we measured times to copy 500 byte long tuples for increasing tuple counts. Table 4.17 gives average memory copy times of 3 runs. In Figure 4.8 the linear increment in time cost can be seen. Note that given times include the allocation of destination memory area, as our system allocates the memory for result relation on runtime.

Table 4.17: Measured Times of Memory Copies with Increasing Tuple Counts

Tuple Count	Time(ms)
1,000	1
10,000	4.67
100,000	59.67
1,000,000	525.67

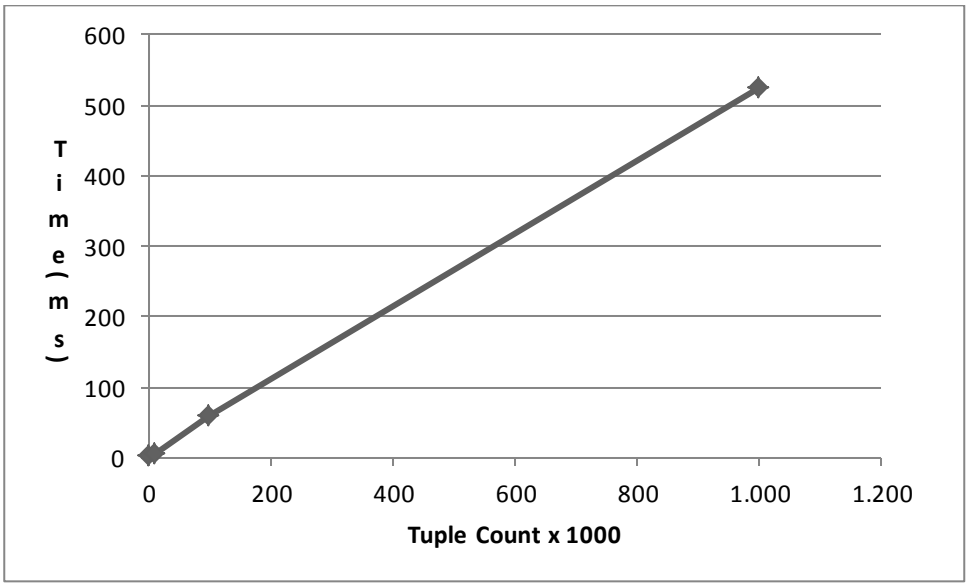

Figure 4.8: Time Spent on Memory Copies with Increasing Tuple Count

Both these experiments show that cost of comparisons and memory copies increase with tuple count. For large operations they become important factors. This explains why nested-loop join takes considerably more time than sort-merge and hash joins, as it performs a lot of comparisons.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this study we implemented a distributed database query processor. The processor can execute a given QEP. The generation of dummy relation data for test purposes is also done by the system. Another capability of the system is performance evaluation. We measure the real response time for QEP execution. However, to see the effects of database design clearly, real times are not dependable. To overcome this problem, we used simulated time which focuses on disk and network costs during QEP execution. We also estimated the approximate time a QEP will take without actually executing it. This is useful for comparing possible plans, which is a task performed by query optimizers.

We performed several experiments with our implemented system. First we proved data generator and query plan processor produce correct results. Then we compared join algorithms implemented and saw hash-join performs good overall in the scope of this study. Later we modified network parameters to see the effects of different configurations on QEP response times and saw the inverse ratio between bandwidth and cost. To see how the system handles parallel execution another experiment was done. The results were as expected. In the next experiment we tried how real, simulated and estimated times are measured for different query plans. We saw that that estimated times were very close to simulated times. Also, real times and simulated times had a correlation. In the last

experiment we measured times spent for frequently performed string comparison and memory copy operations to give an idea of CPU costs.

Our system can be extended in many ways. First of all, our data is synthetic numerical strings. Most real-world data cannot be represented in this form. Operation results and performance evaluation is affected by this artificiality. So, one of the main improvements to our system would be addition of different data types support.

Lack of indexes on tables is another important factor that damages realism. Existence of indexes enlarges the search space for query optimizers, and selecting different indexes with different join algorithms can dramatically change QEP cost. In the future indexes can be added to our system, which would be a valuable feature that reflects commercial database costs more accurately.

In this study we assume CPU costs and memory consumption are negligible. However, these are important factors to determine success of a QEP. Considering them is another possible future improvement.

Despite these short-comings, our system is a good environment for experimental query optimizers. QEP files serve as a well-defined interface for users or other software. Most settings used by the system are configurable. The architecture is stable and easily expandable.

REFERENCES

[Bernstein(1981)] P. A. Bernstein, D. Chiu. *"Using Semi-Joins to Solve Relational Queries"*. Journal of ACM, Vol. 28, Issue 1, January 1981, pp. 25-40.

[Banerjee(1993)] S. Banerjee , V. O. K. Li , C. Wang. *"Distributed Database Systems in High Speed Wide-Area Networks".*

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

[Getoor(2001)] L. Getoor, B. Taskar, D. Koller. *"Selectivity Estimation using Probabilistic Models"*. ACM SIGMOD Record, Vol. 30 No. 2, 2001.

[Ioannidis(1996)] Y. E. Ioannidis, *"Query Optimization", Citeseerx, doi=10.1.1.24.4154.*

[Kang(1987)] H. Kang, N. Roussopoulos. *"Using 2-way Semijoins in Distributed Query Processing"*. Proceedings of the Third International Conference on Data Engineering, Washington, DC, USA, 1987.

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

[Mannio(1988)] M. V. Mannio, P. Chu, T. Sager. *"Statistical Profile Estimation in Database Systems"*. ACM Computing Surveys, Vol. 20 No. 3, 1988.

[metu-hpc(2012)] High Performance Computing, http://www.ceng.metu.edu.tr/hpc/index, last visited on April 2012.

[Mousavi(2011)] H. Mousavi, C. Zaniolo. *"Fast and Accurate Computation of Equi-Depth Histograms over Data Streams"*. Proceedings of the 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 22-24, 2011.

[mpi-forum (2009)] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 2.2, September 2009, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, last visited on April 2012.

[mtrand(2012)] C++ Mersenne Twister Pseudo-Random Number Generator, http://www.bedaux.net/mtrand/, last visited on April 2012.

[Onder (2010)] I. S. Onder, "*Execution of Distributed Database Queries on A HPC System"*, METU, 2010.

[Ozsu (2011)] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems (3rd edition)*, Prentice-Hall, 2011.

[pugixml(2012)] pugixml, http://code.google.com/p/pugixml/, last visited on April 2012.

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

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

[Sevinc(2011)] E. Sevinc, A. Cosar. *"An Evolutionary Genetic Algorithm for Optimization of Distributed Database Queries".* The Computer Journal, Vol. 54 No. 5, 2011.

[treehh(2012)] tree.hh: an STL-like C++ tree class, http://tree.phi-sci.com/, last visited on April 2012.

APPENDIX A

MEASURED DISK AND NETWORK TIMES

Table A.1: Measured Disk and Network Times (ms)

Tuple Count	Size	Total Size	Disk (Real)	Network (Real)	Disk (2.3)	Network (2.1)	Disk (3.3)	Network (3.2)
100000	10	1000000	14	178	13.2	5.0	15.5	138.0
150000	10	1500000	21	245	13.2	7.5	22.2	207.0
200000	10	2000000	27	309	13.2	10.0	29.0	276.0
250000	10	2500000	35	388	13.3	12.5	35.7	345.0
300000	10	3000000	40	386	13.3	15.0	42.5	414.0
350000	10	3500000	49	492	13.3	17.5	49.2	483.1
400000	10	4000000	55	542	13.3	20.0	56.0	552.1
1000000	10	10000000	145	1164	13.4	50.0	137.0	1380.1
1000000	20	20000000	154	1247	13.7	100.0	157.0	1460.1
1000000	30	30000000	203	1323	13.9	150.0	177.0	1540.1
1000000	40	40000000	217	1491	14.2	200.0	197.0	1620.1
1000000	50	50000000	256	1456	14.4	250.0	217.0	1700.1
1000000	60	60000000	285	1403	14.6	300.0	237.0	1780.1
1000000	70	70000000	288	1482	14.9	350.0	257.0	1860.1