Allocation of Join and Semi Join Operations based on Dynamic Selectivity Factor in a Distributed Database Query

Richa Arora
Student
DCSE
GNDU, Amritsar

Ankita Bhalla
Student
DCSE
GNDU, Amritsar

R. S. Virk, Ph.D
Asociate Professor
DCSE
GNDU, Amritsar

ABSTRACT

Distributed databases are gaining popularity due to advancement in technology of computer networks and due to need of the business. In distributed databases as data is located at different sites, so to get access to a particular type of data requires a query to be subdivided into subqueries and executing those subqueries at different sites. To accomplish this join operator is used. But using join sometimes incurs extra communication cost when complete relation is not required for join. In such cases to reduce the communication cost involved between two sites semi join is used. But semi join is not always useful. In this paper join operator allocation has been done dynamically by dynamically calculating selectivity factor for join and semi join for the dynamic distributed database simulated in MATLAB. This dynamic selectivity factor is given as input to the simulator built in MATLAB based on which fragment size for join operation is calculated. The simulator by using the genetic algorithm computes the minimum communication cost involved in executing the query by using combination of join and semi join.

Keywords: Distributed Database, Join, Semi Join, Selectivity Factor.

1. INTRODUCTION

A collection of files or tables constitute a database. Database Management System (DBMS) is a set of programs that enable a user to interact with database like storing and retrieving information from the database. Now data sets are becoming enormous that they are almost impossible to store in centralized databases. So data is managed through Distributed Database Systems. In Distributed database systems data is stored at multiple locations. It may be stored in multiple computers located in the same physical location, or may be dispersed over a network of interconnected computers.

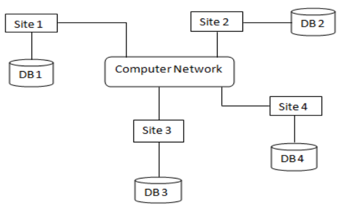


Figure 1: Distributed Database System.

Query processing is complex in distributed environment than in centralized environment because a large number of parameters affect the performance of distributed queries, like relations may be fragmented and/or replicated. In distributed databases as databases are located at geographically different locations, so a simple query that needs to access databases from various locations can be decomposed into sub queries. Those queries have to be executed in such a way that it provides an appearance of a centralized database system. For this query has to be optimized so that it is executed in minimum possible time.

1.1 Join and Semi Join

In distributed databases as data is distributed among different sites so in order to execute any query there is need to combine the data from two sites. This can be accomplished by using joins. For the join operation table from one site has to be transmitted to other site for join to take place. It is not necessary that whole of the relation is required for join. It may be possible that only few values of the relation are required. In such a case transmitting the whole relation to other site will create redundancy of data; it will also incur extra communication cost. In such a case only part of the relation which is required for joining two tables should be passed to the other site. This is accomplished by using semi join. The basic idea behind semi join is that it reduces the quantity of data and data communication among sites [1]. In case of semi join, the unique join attribute values from one relation are sent to the site containing second relation. Based on the attribute values of first relation, records from the second relation are selected by joining the two relations. Then this reduced file is transmitted to the site containing first relation. If all the records of the second relation are selected during semi join operation then there is no use of using semi join. In that case instead of decreasing, it will increase the communication cost [2].

2. RELATED WORK

Query processing in distributed databases is a complex task due to following two reasons:

- Data must be allocated to different sites.
- It must be efficiently accessed, processed and communicated to meet the desired retrieval and update requirements by user.

Genetic algorithm provides an efficient way to solve the above two problems.

In [3] Rho and March have designed a nested genetic algorithm that iteratively allocates data to nodes and to meet the efficient retrieval and update requirements where to process and access the data.

Optimizing join queries is a major problem in distributed database systems, particularly when files are replicated at different nodes in the network. Rho and March [2] in their paper developed a genetic algorithm based solution for efficient query execution plans.

The most important concern in query processing in distributed databases is minimizing the query execution time. So different allocation of sub queries to sites and their execution plans need to be optimized based on query type. This subquery allocation problem is NP-Hard. Therefore, Narasimhaiah Gorla and Suk-Kyu Song [4] had optimized the sub query allocation using genetic algorithm. It had been found that GA produced better results in much less time than exhaustive method.

In distributed databases data replication, join node selection, join order, and reduction by semi join all have significant impact on the efficiency of the distributed database system. Rho Sangkyu, T. March Salvatore [5] in their paper had compared the various distributed database design models. They have found that replication was most effective for retrieval intensive and high selectivity situations. Join node selection, join order, and reduction by semi join were most effective for balanced retrieval/update and low selectivity situations.

3. COST MODEL

The cost in distributed database systems is in terms of execution time. A general formula for determining the total time is as follows [6]:

$$\text{Total time} = T_{CPU} * \#insts + T_{I/O} * \#I/Os + T_{MSG} * \#msgs + T_{TR} * \#bytes$$

Where, T_{CPU} is the time of a CPU instruction, $T_{I/O}$ is the time of a disk I/O, T_{MSG} is the fixed time of initiating and receiving a message. Thus the communication time of transferring #bytes of data from one site to another is assumed to be a linear function of #bytes:

$$CT\ (\#bytes) = T_{MSG}+T_{TR} *\#bytes$$

The main concentration of this paper will be on CT (communication time) [6].

4. DATABASE STATISTICS

In distributed databases the main factor affecting the execution strategy is the size of intermediate relations that are produced at each level of query execution. When a subsequent operation is located at a different site, the intermediate relation must be transmitted over the network. Therefore, it is of prime interest to estimate the size of the intermediate results of relational algebra operations in order to minimize the size of data transfers [6]. In this paper fragment size for join operations are calculated dynamically at run time by connecting to the dynamic database available. Consider a relation R having attributes A = {A_1, A_2…A_n} and fragmented as R_1, R_2… R_r. The selectivity factor of an operation is the proportion of tuples of an operand relation that participate in the result of that operation [6].

4.1 Selectivity Factor for Join

The join selectivity factor, denoted SF_J, of relations R and S is given by [7]:

$$SF_J(R, S) = \frac{\text{card}(R \bowtie S)}{\text{card}(R)*\text{card}(S)}$$

Where, card (R ⋈ S) returns tuples after joining relations R and S, card(R) returns tuples in relation R, card(S) returns tuples in relation S. SF_J is a real value between 0 and 1 [6].

4.2 Selectivity Factor for Semi Join

An approximation for the semi join selectivity factor is given as [7]:

$$SF_{SJ} = \frac{\text{card }(\prod_A(S))}{card(dom[A])}$$

Where card ($\prod_A$(S)) returns the tuples of relation S that satisfy the joining criteria and card (dom[A]) returns the number of distinct values of attribute A. If R.A being a foreign key of S (S.A is a primary key). In this case the semi join selectivity factor is 1 since $\prod_A(S) = card(dom[A])$ [6]. Based on these selectivity factors fragment size of join and semi join operations are calculated dynamically through following formula:

- In case of using Join, frag_size= SF_J * size of relation.
- In case of Semi Join, frag_size= SF_{SJ} * size of relation

In this paper in place of #bytes, #blocks are used as here size of relation is considered to be static and it is considered in blocks. This frag_size determines the number of blocks (#blocks) which helps in computing CT, communication cost involved in transmitting the relations.

5. EXPERIMENTAL SETUP

The following bank database schema has been assumed for experimental analysis:

account = (account_number, branch_name, balance)

branch = (branch_name, branch_city, assets)

depositor = (customer_name, account_number)

borrower = (customer_name, loan_number)

loan = (loan_number, branch_name, amount)

customer = (customer_name, customer_street, customer_city)

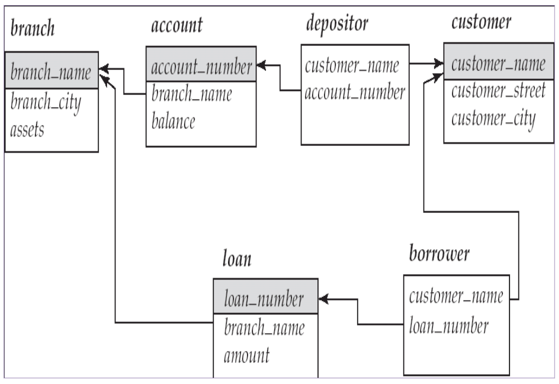


Figure 2: Schema Diagram for bank enterprise [8].

The Experimental query considered is as follows:

Select the details of those customers who are both depositor and borrower in the bank.

5.1 SQL Query:

Select * from account,branch, borrower, depositor, loan,customer where
customer.customer_name=depositor.customer_name AND
customer.customer_name=borrower.customer_name AND
borrower.loan_number=loan.loan_number AND
loan.branch_name=branch.branch_name AND
branch.branch_name=account.branch_name

5.2 Query Tree:

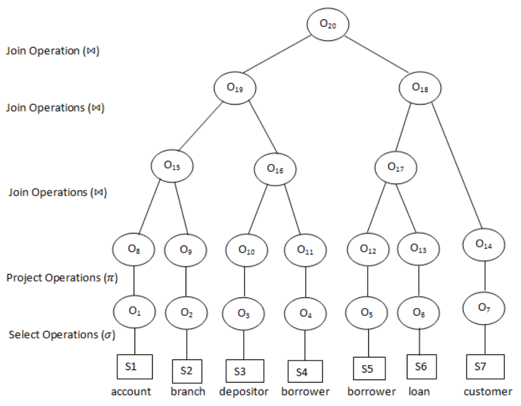
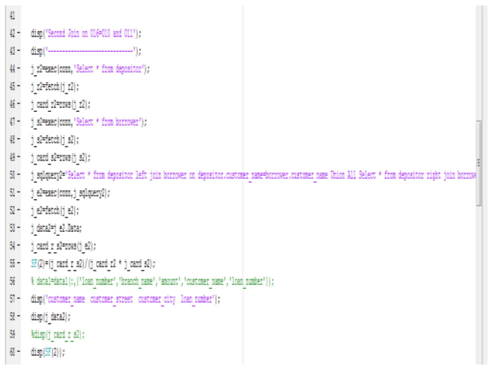
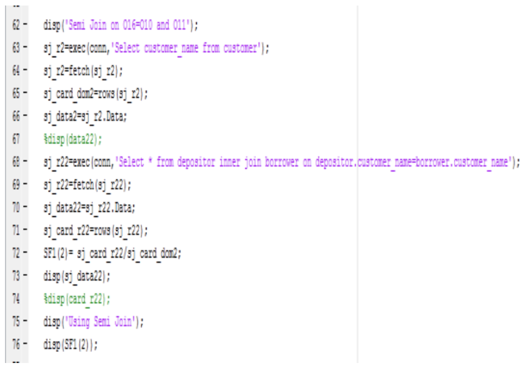


Figure 3: Query Tree for Distributed Database.



```
disp('Second Join on O16=O10 and O11');
disp('----------------------------');
j_r2=exec(conn,'Select * from depositor');
j_r2=fetch(j_r2);
j_card_r2=rows(j_r2);
j_s2=exec(conn,'Select * from borrower');
j_s2=fetch(j_s2);
j_card_s2=rows(j_s2);
j_sqlquery2="Select * from depositor left join borrower on depositor.customer_name=borrower.customer_name Union All Select * from depositor right join borrow
j_e2=exec(conn,j_sqlquery2);
j_e2=fetch(j_e2);
j_data2=j_e2.Data;
j_card_r_s2=rows(j_e2);
SF(2)=(j_card_r_s2)/(j_card_r2 * j_card_s2);
% data1=data1(:,{'loan_number','branch_name','amount','customer_name','loan_number'});
disp('customer_name  customer_street  customer_city  loan_number');
disp(j_data2);
%disp(j_card_r_s2);
disp(SF(2));
```

Figure 4: Screen shot for computing the selectivity factor for join operation.



```
disp('Semi Join on O16=O10 and O11');
sj_r2=exec(conn,'Select customer_name from customer');
sj_r2=fetch(sj_r2);
sj_card_dom2=rows(sj_r2);
sj_data2=sj_r2.Data;
%disp(data22);
sj_r22=exec(conn,'Select * from depositor inner join borrower on depositor.customer_name=borrower.customer_name');
sj_r22=fetch(sj_r22);
sj_data22=sj_r22.Data;
sj_card_r22=rows(sj_r22);
SF1(2)= sj_card_r22/sj_card_dom2;
disp(sj_data22);
%disp(card_r22);
disp('Using Semi Join');
disp(SF1(2));
```

Figure 5: Screen shot for computing the selectivity factor for semi join operation.

As database is dynamic in nature, it may be possible that certain tuples are deleted or new tuples are added. So if the cardinality of relation changes, the results will change. It may be possible that in more than one situations semi join will become beneficial. So the experiment was conducted for four different instances of database with varied cardinalities but for the same query. This dynamically calculated selectivity factor value is passed to the simulator which is built in MATLAB which dynamically computes the intermediate fragment size for join operations. Fragment size is then multiplied with communication coefficients to get the communication cost involved genetically. Lesser is selectivity factor lesser will be the communication cost involved.

6. RESULTS

The results shown here are taken at four different instances of dynamic database with varied cardinalities each time. In the graphs J1, J2, J3, J4 represent the following:

J1 = account⋈ branch

J2 = depositor⋈ borrower

J3 = borrower ⋈ loan

J4 = borrower ⋈ loan ⋈ customer

For the first join operation that is taking place between account and branch, branch name being the foreign key in account relation and primary key in branch relation so in this case SF_{SJ} (Selectivity factor for semi join) will be 1. Also changing the cardinalities of these two relations will not have any effect on SF_{SJ}. This is shown in figure 6 and figure 7. So for the next instances of database this join for relation account and branch has been omitted and in figure 8 and figure 9 SF_J and SF_{SJ} has been shown for join operations J2, J3 and J4. Table 1, Table 2, Table 3, Table 4 represent relations with different cardinalities at different instances of database.

Table 1: Cardinalities for 1st instance of the database.

Relation	**Cardinality**
account	100
branch	80
depositor	70
borrower	50
loan	55
customer	123

Table 2: Cardinalities for 2nd instance of the database.

Relation	**Cardinality**
account	100
branch	80
depositor	70
borrower	50
loan	55
customer	300

Table 3: Cardinalities for 3rd instance of the database.

Relation	Cardinality
account	100
branch	80
depositor	70
borrower	9
loan	80
customer	300

Table 4: Cardinalities for 4th instance of the database.

Relation	Cardinality
account	50
branch	80
depositors	70
borrower	6
loan	100
customer	200

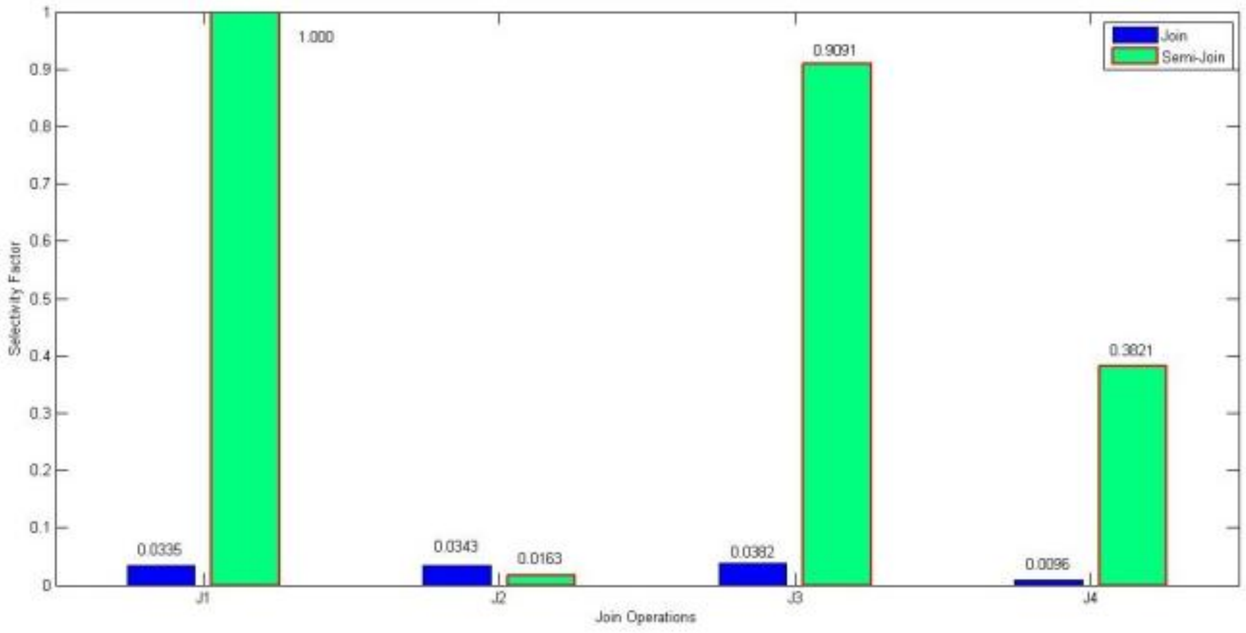


Figure 6: Selectivity Factor for 1st instance of the database.

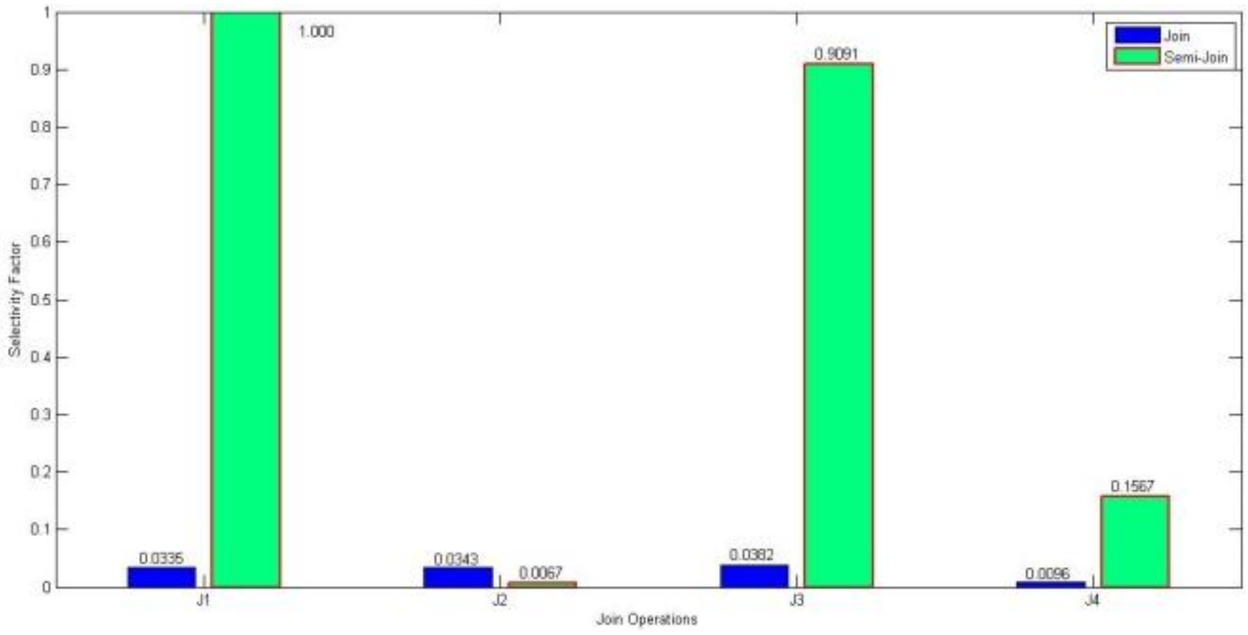


Figure 7: Selectivity Factor for 2nd instance of the database.

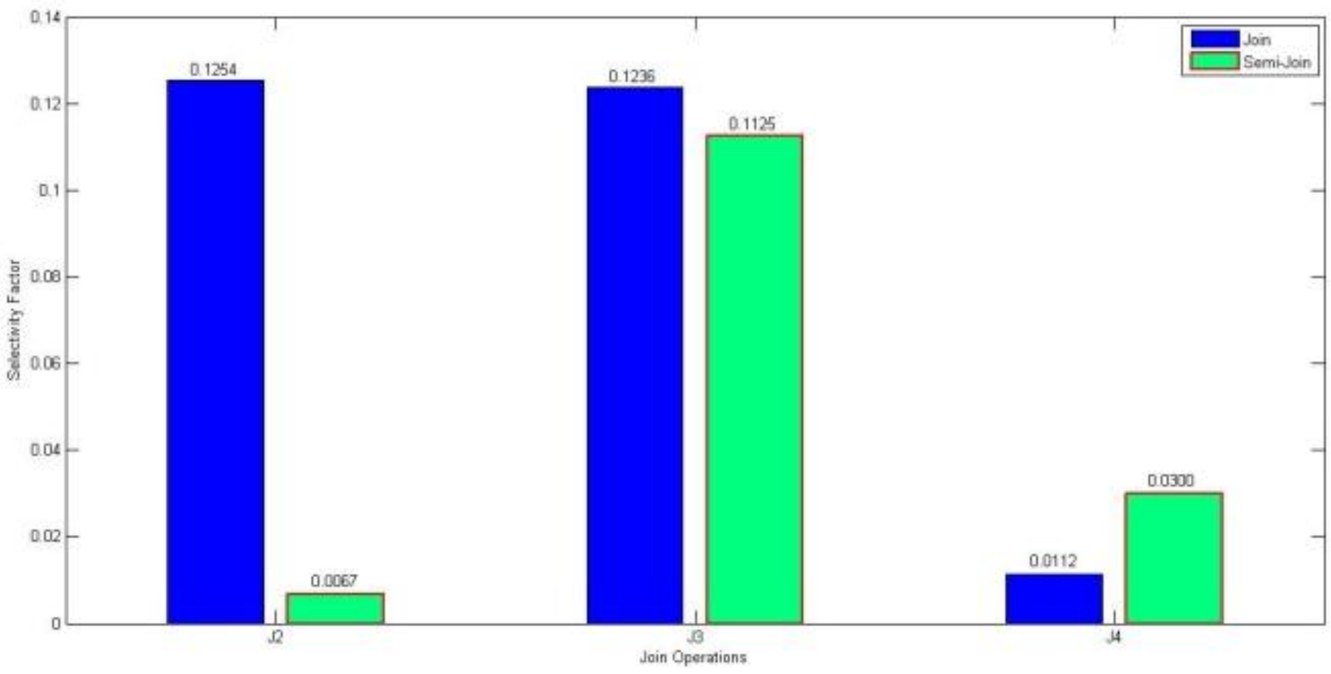


Figure 8: Selectivity Factor for 3rd instance of the database.

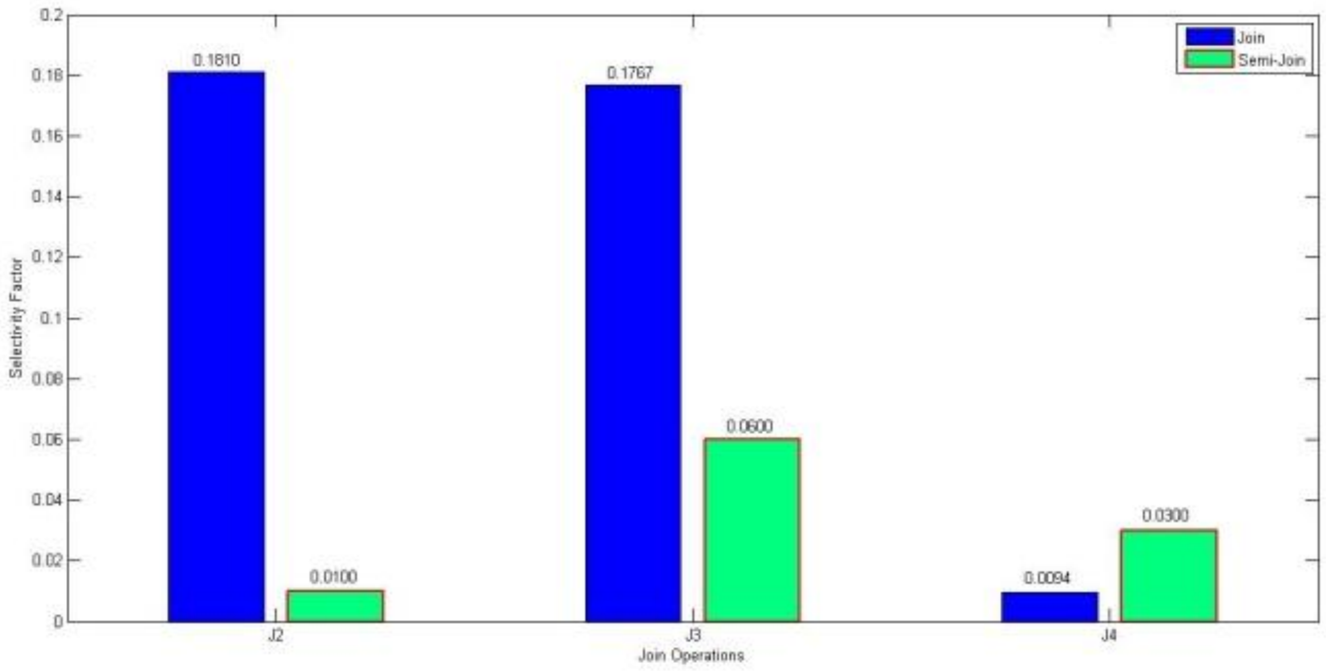


Figure 9: Selectivity Factor for 4th instance of the database.

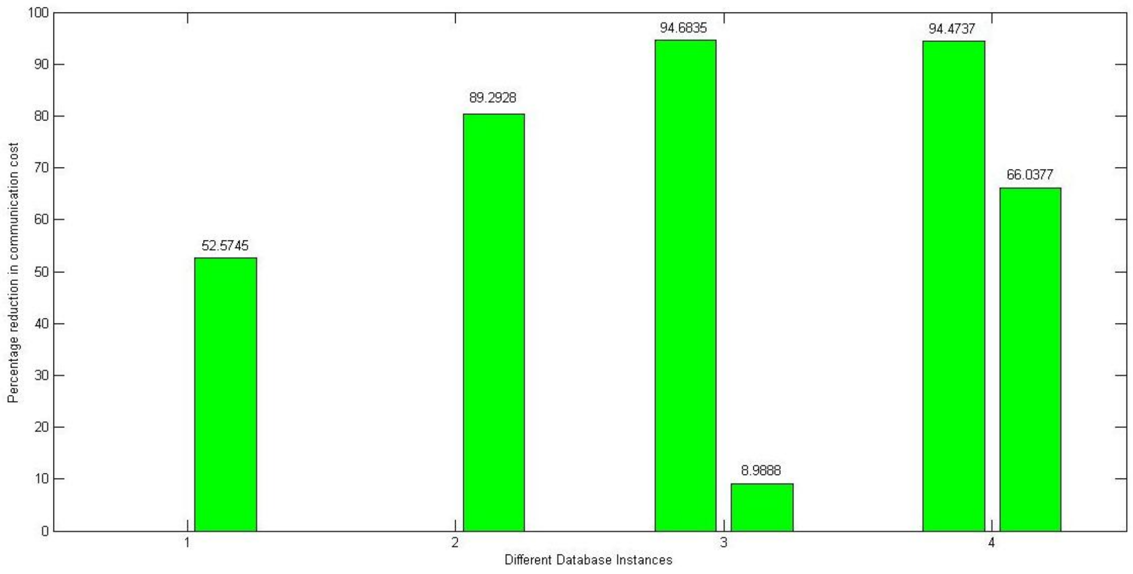


Figure 10: Percentage reduction in communication cost for semi join operation for various instances of database.

After experimenting with the actual dynamic database on calculating the selectivity factor dynamically it is seen that in very less cases $SF_{SJ}<SF_J$. Whenever selectivity factor for semi join comes less than join then only semi join should be used otherwise join should be used. Whenever very few tuples are required which is a very rare case for join operation only then semi join should be used otherwise normal join should be used.

Figure 10 denotes the percentage reduction in communication cost wherever semi join is beneficial. In this graph first bar represent that only at one place semi join was beneficial for 1st instance of database and percentage improvement is written above the bar. Similar is the case for 2nd instance of database. For the 3rd and 4th instance at two places it is proving to be beneficial than joins. Although in very less cases semi join is coming out to be beneficial but percentage reduction in communication cost is coming as high as 90% in those cases. So semi join greatly reduces the communication cost involved but in very few cases.

7. CONCLUSIONS

Distributed database system is a collection of databases that can be stored at different computer network sites. To combine data from multiple sites joins are used. In this paper a simulator has been designed in MATLAB which genetically computes the minimum cost involved for executing a query in distributed database. Main concentration is to reduce the communication cost involved by using combination of joins and semi joins. From the results it has been found that in very less cases semi join has proven to be beneficial. It has been observed that when very few tuples are required for joining the relation at other site only then semi join should be used and semi join is taking 90% less communication cost than join.

8. ACKNOWLEDGMENTS

Authors are highly indebted to Dr. R. S. Virk, Associate Professor, DCSE, GNDU, Amritsar for his precious guidance from time to time.

9. REFERENCES

[1] Lin Zhou, Yan Chen, Taoying Li, Yingying Yu, 2012, The Semi-join Query Optimization in Distributed Database System, CITCS

[2] Rho Sangkyu, T. March Salvatore, 1997, Optimizing distributed join queries: A genetic algorithm approach

[3] Rho Sangkyu, T. March Salvatore, 1994, A Nested Genetic Algorithm for Distributed Database Design, IEEE

[4] Narasimhaiah Gorla and Suk-Kyu Song, 2010, Subquery Allocations in Distributed Databases Using Genetic Algorithms, JCS&T Vol. 10 No.1

[5] Rho Sangkyu, T. March Salvatore, 2002, A Comparison of Distributed Database Design Models, Seoul Journal of Business Vol. 8 No. 1

[6] M. Tamer Ozsu, Patrick Valduriez, 2011, Principles of Distributed Database Systems, Third Edition, Springer

[7] A. R. Hevner, S. B. Yao, 1979, Query processing in distributed database systems IEEE

[8] Abraham Silberschatz, Henry F.Korth, S. Sudarshan, 2006, Database System Concepts, Fifth Edition