Journal of Computer Sciences and Applications, 2015, Vol. 3, No. 1, 18-22
Available online at http://pubs.sciepub.com/jcsa/3/1/3
© Science and Education Publishing
DOI:10.12691/jcsa-3-1-3

Retrieving Queries by Applying Join Selectivity along with Various Join Mechanisms Using Soft Computing Approach

Sambit Kumar Mishra[1,*], Srikanta Pattnaik[2], Dulu Patnaik[3]

[1]Department of Computer Sc.&Engg, Ajay Binay Institute of Technology, Cuttack, Odisha, India
[2]S.O.A. University, Bhubaneswar, Odisha, India
[3]Government College of Engineering, Bhawanipatna, Odisha, India
*Corresponding author: sambit_pr@rediffmail.com

Received February 25, 2015; Revised March 11, 2015; Accepted March 17, 2015

Abstract In heterogeneous multiple query processing environments, usually the query processors depend upon estimated database cardinalities when evaluating the cost of the query plans. In this paper it is being projected to retrieve query plans along with their costs and fitness values by applying join selectivity techniques for relations used in query processing by applying genetic algorithm techniques. It has also been aimed to see that whether the evaluation of selectivity factor of sub query operation may be feasible and may reduce the total query cost.

Keywords: *query processing, query plans, cardinalities, join selectivity, join index, tuple, chromosome, primary key, foreign key*

Cite This Article: Sambit Kumar Mishra, Srikanta Pattnaik, and Dulu Patnaik, “Retrieving Queries by Applying Join Selectivity along with Various Join Mechanisms Using Soft Computing Approach.” *Journal of Computer Sciences and Applications*, vol. 3, no. 1 (2015): 18-22. doi: 10.12691/jcsa-3-1-3

1. Introduction

Join indexes are database indexes that facilitate the processing of join queries in large databases. To provide more efficient join operations, join index may be necessarily used. A join index is a relation of arity two. The tuple identifiers of the tuples of the relations participating in a join are usually concatenated with the tuples. These augmented relations are joined and the resulting relation is then projected on the tuple identifiers. Usually join selectivity may improve the performance of database queries by giving the desired optimizer more accurate information about the allocation of data in databases. In this situation, the role of the query processor is to decide how to execute a query in the most efficient manner attempting to minimize the I/O. To accomplish that, the processor considers different ways of joining results from various databases and different methods of retrieving the data. The result of the optimization process is known as query strategy. The choice of query strategy depends on its cost based on estimating the number of disk I/O operations. In general, usually the query processor performs better result in implementation of query strategies. It is well understood that the selectivity may be used to predict cardinality of the databases, and the predicted cardinality is used to estimate I/O cost. It is also known that the execution of a query processor may be divided into a static or dynamic phase. In static phase, the role of the query processor is to select the query strategy with optimality, whereas in dynamic phase, the query processor processes several competing indices.

2. Problem Analysis with Example

Consider two relation scheme, R and S. The join index for RXS (R.A=S.B) will only consist of tuples with the tuple identifier of relation R and S that participate in this natural join. A join index is useful for joins that have to be performed often. The number of tuples in the join index for RXS is equal to the cardinality of the join, |RXS|. The size of the tuples in a join index depends on the size of the tuple identifiers. The join selectivity of a relation R in a natural join with a relation S is the ratio of the distinct attribute values for the same attribute in the relation R. Usually in general processing strategies, selection operation is performed as early as possible. Selection reduces the subsequent processing time. After that number of unary operations if any are combined. Then the Cartesian product with a certain subsequent selection is converted into join. After computing common expressions, it is required to preprocess the relations.

While considering theta join, where two entities are joined not on the relationship exist between them but explicitly specifying some other field, it is seen that, no primary key, foreign key relationship in database level is there. But in natural join, it is almost similar to equi join. Here the join predicate arises implicitly by comparing all attributes in both relations that have the same attribute names in the joined relations. The resulting joined relation contains only one attribute for each pair of equally named attributes.

Usually sub query un nesting is always done for correlated sub queries with at most one relation in the FROM clause, which is used in ANY, ALL, and EXISTS predicates.

Assume there are two relations SalesorderItems and Products, and the sub query may be written as follows.
select S.* from SalesorderItems S
where EXISTS (select * from Products P where S.productID=P.ID AND P.ID=300 AND P.Quantity>20);

In the above example productID may be attribute of the relation SalesorderItems, ID and Quantity may be the attributes of the relation Products.

Following the conversion, this same statement may be expressed internally using JOIN syntax:
select S.* from Products P JOIN SalesorderItems S
on P.ID=S.productID where P.ID=300 AND P.Quantity>20;

Now consider another example of a sub query.
select P.* from Products P
where EXISTS (select * from SalesorderItems S where S.productID=P.ID AND S.ID=2004);

The above query contains EXISTS predicate in the sub query, which may match more than one row.

If the same query may be converted to an inner join, with a DISTINCT in the SELECT list, it may be rewritten as follows.
select DISTINCT P.* from Products P JOIN SalesorderItems S on P.ID=S.productID where S.ID=2004;

Usually predicate push-down is performed for a predicate if and only if the predicate refers exclusively to the columns of a single view or derived relation.

While optimizing the OR and IN-LIST predicates, it is seen that the optimizer supports a special optimization for exploiting IN predicates on indexed columns. This optimization also applies equally to multiple predicates on the same indexed column that are ORED together since the two are semantically equivalent.

For example suppose a query may be written as follows.
select * from salesorders where salesrepresentative=902 OR salesrepresenatative=199;

The query may be semantically equivalent to
select * from salesorders where salesrepresentative IN(199,902);

Usually while converting outer join to inner join, the optimizer generates a left deep processing tree for its access plans. The only exception to this rule is the existence of a right deep nested outer join expression. A left or right outer join is converted to an inner join if one of the following condition is true.

i. A null intolerant predicate referencing columns of the null supplying tables is present in the query WHERE clause.

ii. The null supplying side of an outer join returns exactly one row for each column from the preserved side. If this condition is true, there are no null supplied rows and the outer join may be equivalent to an inner join.

Consider another query, where for each row of the relation Salesorder items, there is exactly one row that matches the relation Products. Because the productID column may be declared not NULL and the relation Salesorderitems may have the foreign key:
"Foreignkey_productID"("productID") REFERNCING "product-ID".

The query may be rewritten as follows after a rewrite optimization.
select * from Salesorderitems S LeftouterJoin Products P ON(P.ID=S.ProductID); becomes
select * from Salesorderitems S Join Products P ON (P.ID=S.productID);
select * from Products P Key Left Outer Join Salesorderitems S where S.quantity>15;

The above query lists products and their corresponding orders for larger quantities; the Left outer Join ensures that all products are listed, even if they have no orders. The problem with this query is that the predicate in the WHERE clause eliminates any product with no orders from the result because the predicate S.Quantity>15 may be interpreted as false if S.Quantity is null. The query may be semantically equivalent to select * from Products P key Join Salesoredritems S where S.Quantity >15. The rewritten form is the query that the database server optimizes.

Table 2.1. Relation scheme with size

Sl.No.	Relations	Size (KB)
1	EMPLOYEE	100
2	ASG	100
3	PROJECT	100
4	F1	58

Sub-Query Operation A: Selectivity factor of selection operation on relation EMPLOYEE SFs (EMPLOYEE) = card (F1) Card (EMPLOYEE) SFs = 58/100 =0.58.

Sub-Query Operation B: Selectivity factor of selection operation on relation ASG SFs (ASG) = 56/100 =0.56.

Many of the query processing strategies in distributed databases are static in nature i.e., the strategy is completely determined on the basis of a priori estimates of the selectivity factor of sub query operations and it remains unchanged throughout its execution. Due to this, the cardinality of intermediate fragments is large.

Selectivity factor of various sub-query operations = [0.96, 0.92, 0.59, 0.48].

For each operation, the size of intermediate fragment is calculated by use of prefixed selectivity values for hose operations.

Sub-Query Operation 1:

$$(\sigma_{\text{Designation='Manager'}}(\text{EMPLOYEE}))$$
$$\rightarrow \text{F1, Tuples: } 100 \text{ x } 0.96(\text{Ps}) = 96$$

Sub-Query Operation 2:

$$(\sigma_{\text{Basic\_salary=16000}}(\text{ASG}))$$
$$\rightarrow \text{F2, Tuples: } 100 \text{ x } 0.92(\text{Ps}) = 92$$

Sub-Query Operation 3:

$$(\sigma_{\text{Designation='Manager AND Department'='Accounts'}}(\text{EMPLOYEE}))$$
$$\rightarrow \text{F3, Tuples: } 120 \text{ x } 0.59(\text{Ps}) = 71$$

Sub-Query Operation 4:

$$(\sigma_{\text{Basic\_Salary}>20000\ \text{AND Basic\_Salary}<27000}(\text{ASG}))$$
$$\rightarrow \text{F5, Tuples: } 120 \text{ x } 0.48(\text{Ps}) = 58$$

3. Review of Literature

Ridhi et.al [1] have elaborated and explained the selectivity and cost estimation in query optimization in large heterogeneous databases. They have also discussed different types of cost formulations to evaluate the cost of execution plans.

Carlo et.al [2] have proposed a method for estimating the size of relational query results. The approach was mainly based on the estimates of the attribute distinct values. They have also presented some experimental results on real databases showing the promising performance of analytic approach.

Fan and Mi Xifeng et.al [3] have designed a new algorithm based on heuristic optimization that can significantly reduce the amount of intermediate result data. The basic idea of this algorithm was based on relational algebra equivalence transformations to raise the connecting and merging operations in the query tree.

Gurvinder Singh et. al [4] have proposed a stochastic model simulating a Distributed Database environment and projected benefits of using innovative Genetic Algorithms (GA) for optimizing the sequence of sub-query operations allocation over the Network Sites. Also, they have analyzed the quality of the Genetic Parameters on Solutions.

Faiza et.al [5] have proposed a statistical method for estimating the cardinality of the resulting relation obtained by relational operator by using sample based estimation that execute the query to be optimized on small samples of real database and use the results of these trials to determine cost estimates.

Stratis D. Viglas et. al [6] have focused on shifting from a cardinality-based approach to a rate-based approach, and given an optimization framework which aimed at maximizing the output rate of query evaluation plans.

Areerat et.al [7] have proposed Exhaustive Greedy (EG) algorithm to optimize intermediate result sizes of join queries. Most intermediate result sizes of join queries estimated by the EG algorithm may be comparable to the results estimated by the Exhaustive Search algorithm (ESU)which may be modified to update join graphs.

Danh Le-Phuoc1et.al [8] have focused about query optimization in their paper which refers to the process of ensuring that either the total cost or the total response time for a query is minimized. Most modern cost-based optimizers decide between execution plans by minimizing the estimated cost of executing the query. A basic technique used in cost estimation is pre-estimation of Selectivity factor.

William I.et al [9] have used an adaptive selectivity estimation scheme for multidimensional queries where the distribution of the data is not known. Their innovative effort overcomes the disadvantages of previously formulated non-adaptive, static methods which may be relatively inaccurate in a dynamic database.

4. Problem Formulation

Individual plan is represented as chromosome and individual task in a plan is represented as gene. Since a gene in a chromosome represents the plan selected for the query corresponding to the gene position, in the mutation operation the plan number is only replaced with randomly selected valid plan's number for that query. Therefore a mutation operation always generates valid solutions. Different crossover operations can be applied to chromosomes. In our representation scheme, one point and multipoint crossover techniques produce valid solutions for the multiple query optimization problems. If two chromosomes are representing two valid solutions of the same multiple query optimization problem, then any crossover operation on these two chromosomes produces new chromosomes representing valid solutions for the same multiple query optimization problem. Since all chromosome segments that are going to be exchanged to produce a new chromosome represent valid plans for their corresponding queries, the new chromosome obtained by appending these segments represent a valid solution of the multiple query optimization problem.

4.1. Database Statistics

The estimation of size of intermediate results of relational algebra is based on statistical information about the base relations and formulae to predict the cardinalities of the result of relational sub operations. Sequence of operations is pre-fixed before computing cardinality of relations.

No of base relations = 10

No of operations = 7

No of sites =3

The size of each tuple of the relation is presumed to be 1KB.Size of the relation is calculated as:

Size of a relation = tuple size * number of tuples in a relation.

Size of base relations = 100 KB, 100 KB,120 KB, 120KB respectively.

Total cost of the query = local (I/O and CPU) cost + communication cost.

IO_cost is calculated in the basis of IO_speed which represents I/O speed coefficient of particular site where operation is performed and 'i' represents particular fragment generated after applying operation. This I/O cost is calculated for every fragment generated while executing query. Similarly, CPU_cost may be calculated on the basis of unary_IO which may be equal to unary_IO + IO_spd(s)* frag_size (i);

And IO_cost for Join operations may be calculated as follows.

Join_IO = Join_IO + IO_spd(s)* frag_size(i)+ IO_spd(s)* frag_size(i+1)+ IO_spd(s)* frag_size(i)* frag_size(i+1) ;

So the total_IO_cost= (unary_IO + Join_IO);

4.2. Experimental Analysis

Maximum generations, max_gen=20

Number of relations=7

Number of queries, (query_size)=20

Planquery(Size of Chromosome)=5

Population=round(rand(number of queries, planquery))

Pc(Probability for crossover operation)=0.06

Pm (Probability for mutation operation)=0.001

Cp(crossoverpoint)=round(1+rand*(planquery-1))

With the method described, although the number of genes of the chromosomes are kept for the whole population, it will vary according to the query that is being processed and the plans supplied in the feedback.

Genetic algorithm receives an initial population consisting of the chromosomes corresponding to the relevant plans, and to the query. Selection: The genetic algorithm uses simple random sampling as a selection mechanism. This is implemented by assigning to each individual a selection probability equal to its fitness value divided by the sum of the fitness values of all the individuals. If after generating the population, the best chromosome of the previous population is no longer present, the worst individual of the new population is withdrawn, and the missing best individual is put back.

4.3. Algorithm

```
for i=1: max_gen
planselect(i)= Queryplan(i)/(query_size *planquery);
est_cost(i)=planselect(i)/query_size + CPU cost
weight(i)=(Queryplan(i)*query_size)/(query_size-Queryplan(i));
fitness(i)=1+(query_size*weight(i))/((weight(i)^2)+(query_size)^2);
Queryplan(i) represents chromosomes.
Crossover point, cp=2
Size of chromosomes=5
```

Table 4.1. Query plans with cost and fitness values

Queryplan	Population	Est_Cost	Fitness
7	11100	0.00405	0.5099
9	10010	0.0040429	0.58257
11	11010	0.0040721	0.76471
14	01110	0.0041036	1.1622
19	01001	0.0041214	1.2831
23	11101	0.0041286	1.3152
24	00011	0.0041392	1.3472
25	10011	0.0041475	1.3727
30	01111	0.0041727	1.3927

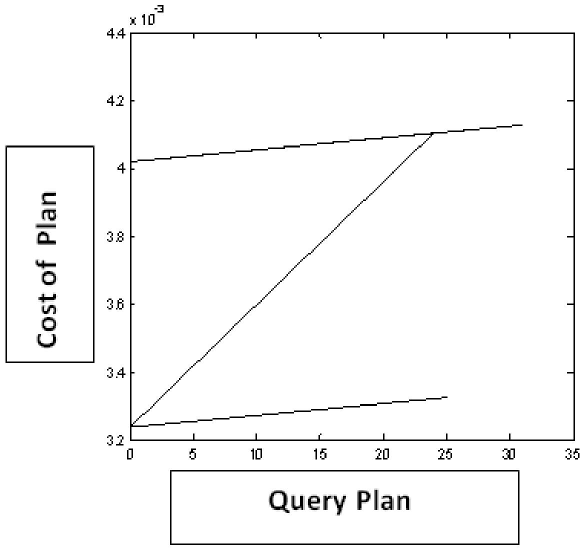

Figure-4.1. (Query Plan VS Est_Cost of Plan)

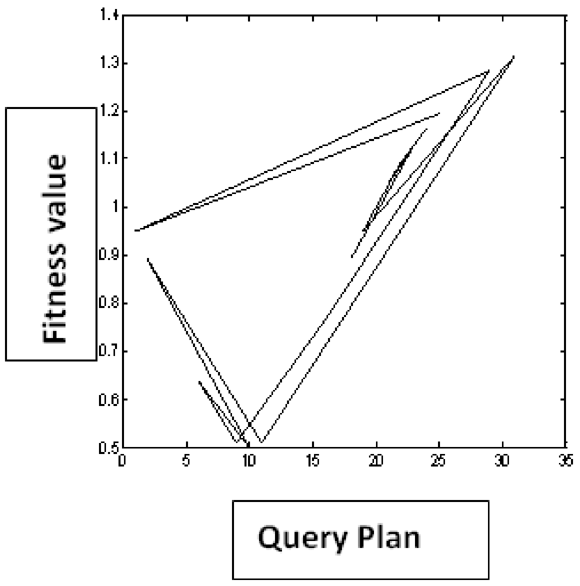

Figure 4.2. (Query Plan VS Fitness value)

5. Discussion and Future Direction

Selectivity estimation is the main part of query optimization. The selectivity factor of an operation is the number of tuples of an operand relation that participate in the result of that operation. It is denoted by SFOP, where OP represents the operation. The selection of the plan is usually based on the cost estimates of alternative plans, which in turn are based on the selectivity estimates of relational operators. Selectivity evaluation depends on cardinality of intermediate fragments generated in the query. The selectivity estimation is based on statistical information about the base relations and formulas to estimate the cardinalities of the results of the relational operations.

6. Conclusion

The main motivation in this paper is to analyze the effect of selectivity evaluation on the reduction of overall cost of the query. It has been observed that the size of query plans in the intermediate relations have been evaluated with close approximation using genetic algorithms. Therefore, it produced quite lesser cost of sub-query. But when cost of all sub-query operations on the various sites are added, the benefits achieved in the range of thirty to forty percent for various sub-operations like selection, projection and join.

References

[1] Ridhi Kapoor, Dr. R. S. Virk, “Selectivity & Cost Estimates in Query Optimization in Distributed Databases”, International Journal of Enhanced Research in Management & Computer Applications, June2013.

[2] Carlo Dell‟ Aquilla, Ezio Lefons, Filippo Tangorra, “Analytic-based Estimation of Query Result Sizes”, 2005.

[3] Fan Yuanyuan, Mi Xifeng. “Distributed database System Query Optimization Algorithm Research”, IEEE, 2010.

[4] Rajinder Singh, Gurvinder Singh, Varinder Pannu virk. “Optimized Access Strategies for a Distributed Database Design”, IJDE, 2011.

[5] Faiza Najjar and Yahya slimani. “Cardinality estimation of distributed join queries”. 2002.

[6] Stratis D. Viglas, Jeffrey F. Naughton. “Rate-Based Query Optimization for Streaming Information”, ACM, 2002.

[7] Areerat Trongratsameethong, Jarernsri L. Mitrpanont, “Exhaustive Greedy Algorithm for Optimizing Intermediate Result Sizes of JoinQueries”, IEEE, 2009.

[8] Danh Le-Phuoc1, Josiane Xavier Parreira, Michael Hausenblas, Manfred Hauswirth. “Continuous Query Optimization and Evaluation Over Unified Linked Stream Data and Linked Open Data”, DERI,2010.

[9] William I. Grosky, Junping Sun, Farshad Fotouhi. “Dynamic selectivity estimation for multidimensional queries”, springer, 1993.