A gene-base DBMS

Dr.Hadeel Noori, Kufa University, college of education for girls, Computer Department
Hanan A.and Wegdan A.
Technical Institute in Najaf, Computer Department

Abstract

The task of the query optimizer is to accept as input a tree of relational algebra operators and to produce a query execution plan. This plan specifies exactly which operations should be performed, in which order. Algebra space can express the solution space, whereas a good search algorithm can finding an optimal solution. In this work a genetic algorithm with two crossover strategies will be tested on a B+ tree sorted database example and nested loop join strategy. The results show that using M2S crossover with left-deep strategy will give better solutions.

Introduction

The success of a DBMS lies in the quality, functionality, and sophistication of its query optimizer[1], since that determines much of the system's performance. The optimizer considers the possible query plans for a given input query, and attempts to determine which of those plans will be the most efficient. Cost-based query optimizers assign an estimated "cost" to each possible query plan, and choose the plan with the smallest cost. Costs are used to estimate the runtime cost of evaluating the query, in terms of the number of I/O operations required, the CPU requirements, and other factors determined from the data dictionary.In This work, we have design of query optimization based on genetic algorithm, crossover, mutation , and reproduction of internal query representation are discussed. A theoretical computations show the effective gene-base optimization The set of query plans examined is formed by examining the possible **access paths** (e.g.

index scan, sequential scan) and **join algorithms** (e.g. *sort-merge join, hash join, nested loops*). The search space can become quite large depending on the complexity of the SQL query..

Database example

University (uname ,u-no, address)
College(c-no, u-no, cname)
Dept(d-no, c-no,dname)
Student(stname, d-no, st-no,ac-no)
Account(ac-no, balance)

Parameter description	Parameter value
Number of university tuples	100
Number of college pages Number of college tuples per page	10 100,000
Number of department pages	50
Number of department tuples per page	10,000
Number student pages	10000
Number of student tuples per page	1,000,000
Number account pages	10000
Number of account tuples per page	1,000,000
Cost of one page access	1.3ms
Indexing University College Department Student account	 B+ tree on x-no Where x represent any relation(eg. University u. ,department d.)

• Sorting Using A B+-Tree Index:

A **B+ tree** is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a *key*. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment (usually called a 'block' or 'node').

– If a B+-tree index is available on the attribute we want to sort by, we can use a completely different algorithm to produce a sorted output relation: just read the tuples in order off the leaf-level of the B+-tree index.
– If the index is clustered, this can be very fast: the heap is already basically sorted, so all we really need to do is read it in.
– If the index is unclustered, this actually turns out to be a pretty bad strategy: we may end up doing up to 1 random I/O for each tuple in the result set (not including the time it takes to read the leaf pages themselves).

The order of a B+ tree measures the capacity of nodes in the tree. It is defined as a number d such that $d <= m <= 2\ d$, where m is the number of entries in each node. For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 7 and 14 keys; the root may have between 1 and 14. For a b-order B+ tree with h levels of index, The maximum number of records stored is $n = b^h$ and Finding a record requires $O(\log_b n)$ operations in the worst case

Optimizer plan steps

The query optimizer is the core of the DBMS, which comprises plan generator and plan cost estimator. The query optimizer performs the following steps:

1. The optimizer generates a set of potential plans for the SQL statement based on available access paths and hints.
2. The optimizer estimates the cost of each plan based on statistics in the data dictionary for the data distribution and storage characteristics of the tables, indexes, and partitions accessed by the statement.

The **cost** is an estimated value proportional to the expected resource use needed to execute the statement with a particular plan. The optimizer calculates the cost of access paths and join orders based on the estimated computer resources, which includes I/O, CPU, and memory.
Serial plans with higher costs take more time to execute than those with smaller costs. When using a parallel plan, however, resource use is not directly related to elapsed time.
3. The optimizer compares the costs of the plans and chooses the one with the lowest cost.

Components of the Query Optimizer
The query optimizer operations include:
■ Transforming Queries
■ Estimating
■ Generating Plans
Query optimizer components are illustrated in Figure 1.

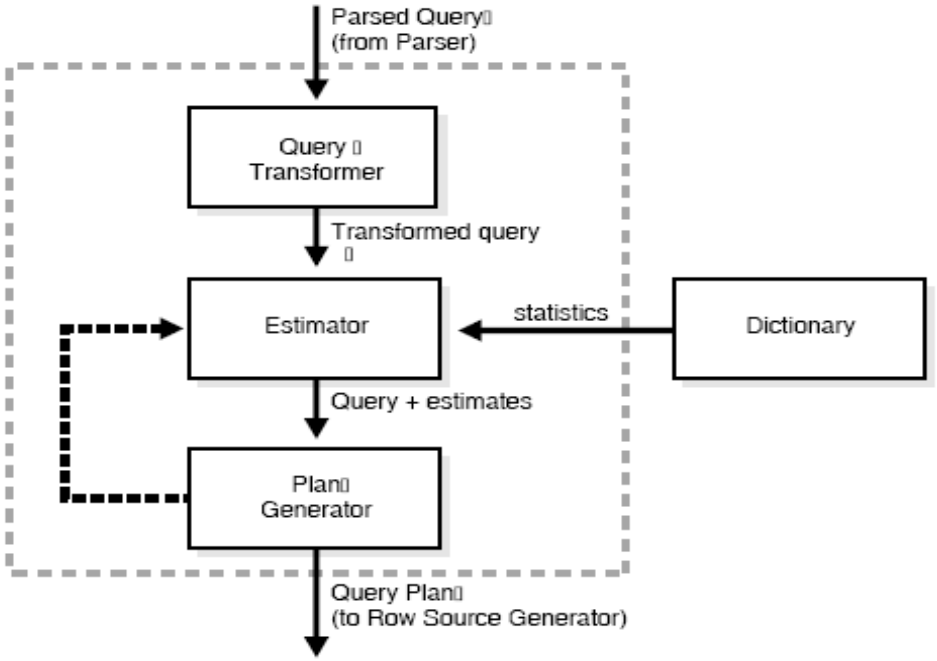


Figure 1 component of query optimizer

Plan spaces

The first step of optimization is parsing the query as mentioned in previous section, consider the following example:

Select (st-name, un-name, address, account)
From(account, student, department, college, university)
Where (acc.no=stu.ano and stu.dno=dep.dno and dep.cno=col.cno and col.uno=uni.no)

This resulting in rewriting the query in algebraic notation and by representing relations in our example by letters ordered alphabetically as follows:

(A ⋈ B) AND (B ⋈ C) AND (C ⋈ D) AND (D ⋈ E)

This algebraic notation expressed both join operation and its constraints, to select those tuples who satisfy the results. Thus the results can obtained from combining all tuples in relations that follow the query constraints and associated with join operations, but the optimizer should avoid combining tuples of relations didn't related by join operation because such combination produce large non-required results.

By the query graph, the optimizer define all the plans which guide the estimated solutions, where, the query can be represented in join processing tree, which its leafs represent the real DB's relations and its internal nods represents the join operation.

There are three structures (plans) that shape the join processing tree; left-deep tree, whose inner relations are base relations, right deep relation, whose outer relations are the base relation. The nodes of these two types have one leaf as a child and called linear tree. The other type called bushy tree. Figure 2 show such types for our example

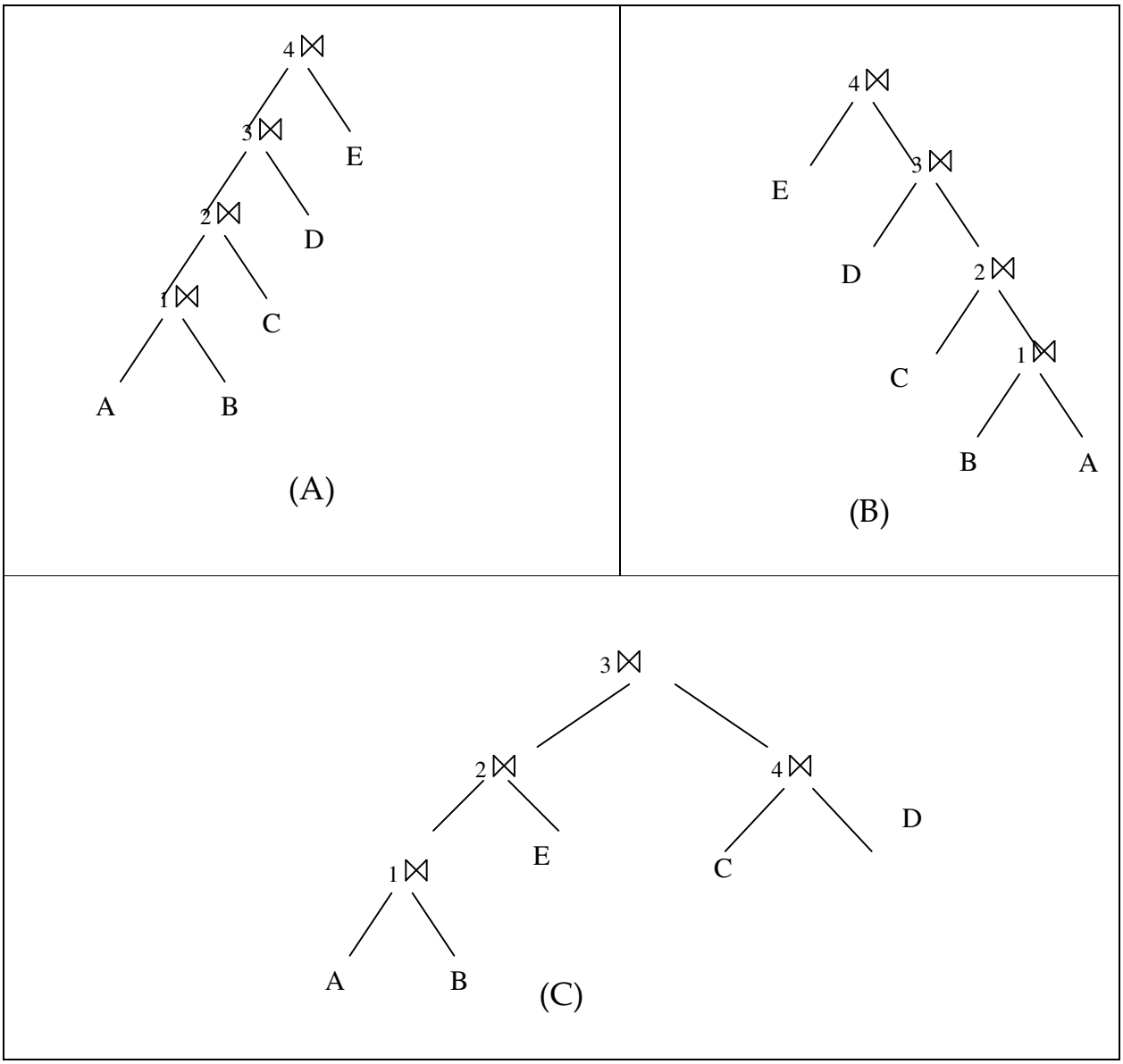


Figure2: join processing tree (A) left -deep tree (B) right -deep Tree (C) bushy Tree

The proposed gene-base optimization

In this section we will describe the genetic algorithm that used as search strategy in plan space of DBMS according to our database example. [2] give the detailed of the genetic operations crossover, mutation, and reproduction.

Repeat

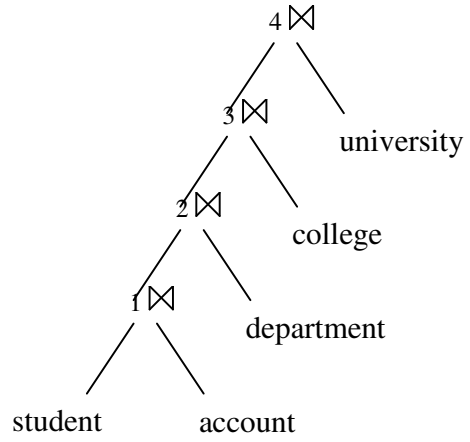
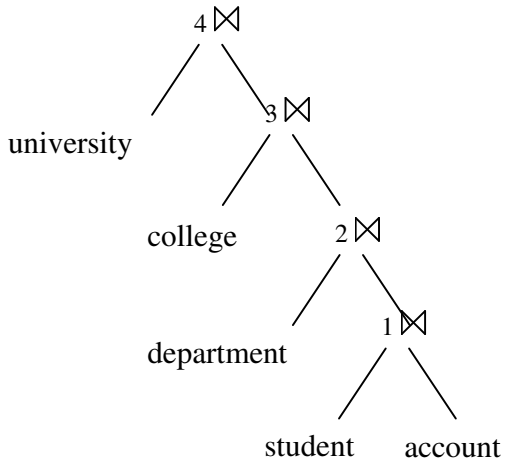
For each chromosome i do

Evaluate f(i)
select two parent chromosomes based on their fitness f(i)
mate chromosome I and chromosome j
crossover the parents
Mutate the resulting offspring with probability p(m)
Until the variance of f(i) is small

Coding of plan space

It is well known that each problem has its solution space that need for an efficient coding to be compatible with genetic procedures. Thus we need to describe the coding of the problem detailed in previous sections.

Left-deep strategy are a small subset of the plan space, each chromosome is an ordered list of genes, combined with join method, whereas right-deep strategy result in wide area of solutions (result from join relations that are not present explicitly in the query). [3] Use two coding methods one for linear trees and other for bushy tree. The former represent each chromosome as ordered list of genes combined with join method associated with each relation, so regenerating the processing tree result in plan with those join methods, whereas, the later represented with ordered list of genes associated with join method and orientation of relation (left or right). Figure 3 show the coding of these two strategies:

LEFT- DEEP TREE	RIGHT-DEEP TREE
	

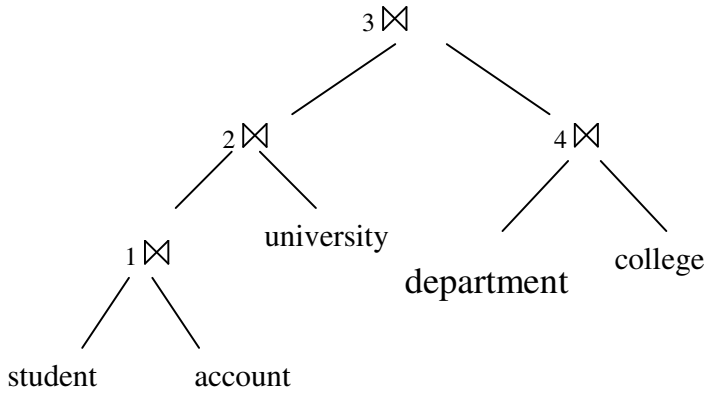
GENE REPRESENTATION **j super script represent the join method**	
^{j}**STU** ^{j}**ACC** ^{j}**DEP** ^{j}**COL** ^{j}**UNI**	^{j}**UNI** ^{j}**COL** ^{j}**DEP** ^{j}**STU** ^{j}**COL**
BUSHY TREE	
	
GENE REPRESENTATION where the subscript represent the orientation and the super script represent the join method	
$1_{L}^{J}2_{R}^{J}4_{L}^{J}3_{L}^{J}$	

Figure 3 coding of join processing trees a: left-deep b) right- deep c) bushy tree

CROSSOVER

As the order of genes in the chromosome must be kept, method of two swap(M2S) [4], was adopted to incorporated the information from the parents. Given parents x1,x2, randomly choose two genes from x1 and replace with the corresponding in x2.

Parents:
X1 : ^{j}**STU** ^{k}**ACC** ^{j}**COL** ^{j}**DEP** ^{j}**UNI**
X2: ^{j}**ACC** ^{j}**DEP** ^{j}**COL** ^{k}**STU** ^{j}**UNI**

Choosing STU , ACC for substitution result Offspring:

X1 : ^{j}**ACC** ^{k}**STU** ^{j}**COL** ^{j}**DEP** ^{j}**UNI**

X2: ^{j}**STU** ^{j}**DEP** ^{j}**COL** ^{k}**ACC** ^{j}**UNI**

The second method was adopted from [5], which refer to as CHUNK it was designed to be suitable for bushy case. CHUNK can be summarized in the following steps:

1- compute gene length l .
2- compute the start of the CHUNK as uniformly random number in [0,l/2]
3- generate the CHUNK length from [l/4, l/4]

These steps can be summarized in the following example

X1: $1^{j}a$ 2^{k}_{r} 4^{k}_{r} 3^{j}_{a}
X2: 1^{k}_{r} 3^{j}_{r} 2^{j}_{a} 4^{k}_{a}

And Generating the CHUNK [1 2] Result in genes 1 2 from x1 will copied into the same positions in the offspring then delete these genes from x2 and use the remainder to fill up the offspring. This process repeated by interchanging of x1 and x2.

Offspring 1: 1^{j}_{a} 2^{k}_{r} 3^{j}_{r} 4^{k}_{a}
Offspring 2: 1^{k}_{r} 4^{k}_{r} 3^{j}_{r} 2^{j}_{r}

Implementation

In this section we will show, implementing genetic algorithm with our database example. Using four quereies to provide two, three, four, and five relations consequently give a good comparison on the base of cost parameter.

Let P(N) and F(N) give the number of pages and the number of tuples per page, respectively, for some relation N. Using Simple Nested Loops Join algorithm shown bellow:

• For each tuple in the outer relation R, we scan the entire inner relation S:

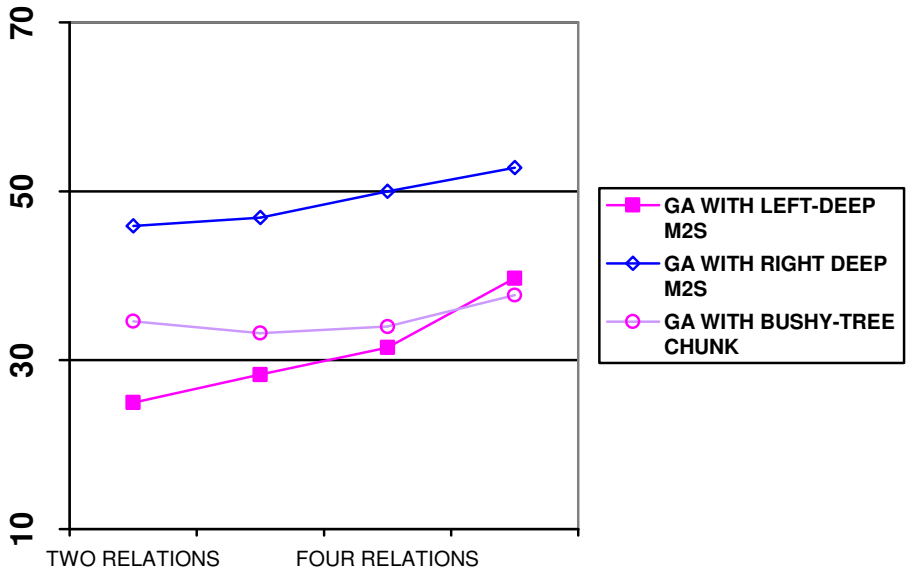
For each tuple r in R do

For each tuple s in S do

if r.id == s.id then add <r,s> to the result

•We get a cost about: P(R) + [F(R) _ P(R) _ P(S)]

Figure 4 show genetic algorithm runs for left-deep m2s coding, right- deep m2s coding, and bushy with chunk coding



Conclusion

Results shown in the implementation section exploit success of genetic algorithm in finding an optimal plan (ordered relational join strategy). Using left-deep representation result in further extra relations that can be discarded from the plan space. Also bushy tree with chunk coding result with better solution than right-deep with m2s coding

Also genetic algorithm can found solutions for larger queries in an acceptable time.

References

[1] M. Jarke and J. Koch. “query optimization in data base systems”, ACM computing surveys , 16(2): 111-152, June 1984.

[2] Goldberg, David E. "Genetic and Evolutionary Algorithms Come of Age." *Communications of the ACM 37*, 3 (March, 1994), pp. 113-119.

[3] K. Bennett, M. C. Ferris, and Y. Ioannidis. A genetic algorithm for database query optimization. In Proc. 4th Int. Conference on Genetic Algorithms, pages 400{407, San Diego, CA, July 1991.

[4]S. Lin and B. W. Kernighan. An Artificial heuristic algorithm for the traveling salesman problem. Operation Research, 21: 498-516, 1973

[5] H. Muhlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In Schaeffer [13], pages 416-421.