A Genetic Algorithm for Database Query Optimization

Jorng-Tzong Horng*, Cheng-Yan Kao**+ , and Baw-Jhiune Liu*
*Department of Computer Science & Information Engineering, National Taiwan University, Taipei, Taiwan
**Department of Computer Science & Information Engineering, National Central University, Chungli, Taiwan
cykao@csman.csie.ntu.edu.tw

+ All correspondences should be sent to the author Cheng-Yan Kao.

Abstract

Numerous decision support applications have been modeled as set covering and partitioning problems. We propose an extension to the database query language SQL to enable applications of these problems to be stated and solved directly by the database system. This will lead to the benefits of improved data independence, increased productivity and better performance. Six operators, namely cover, mincover, sumcover, partition, minpartition, and sumpartition are extended. In this paper, we presented genetic algorithms for the implementation of access routines for the proposed operators. We found that our genetic algorithm approach for extended operations and query optimization performed well both on the computational effort and the quality of the solutions through a variety of test problems. This approach makes it possible for a DBMS to respond to queries involving the proposed operators in a predicate restricted amount of time.

1 Introduction

Genetic algorithms [10, 11] are becoming a widely accepted method for several famous difficult optimization problems. In this paper, we describe a genetic algorithm for the implementation of the proposed operators in database query optimization. In order to give a detailed formulation of our genetic algorithm, we first briefly describe this particular application.

Relational database languages, due to their flexibility, power and simplicity, are playing an increasingly important role in the development of information systems. SQL [5, 24, 7, 4, 6] is the best known of these and has become the ANSI standard for relational DBMSs [25]. However, SQL is not adequate for certain important application domains like decision support systems. As an example, suppose suppliers, parts, and quantities are stored in the SP(s#, p#, qty) relation of a database. The SP relation gives a shipment of parts *p#* by the supplier *s#* and the shipment quantity is *qty*. We might want to determine a set of suppliers with minimum total cost who can supply all kinds of parts. In this example, we assume a cost is associated with a supplier for some subset of the parts he supplies. Moreover, we require not only a set of suppliers with minimum total cost who can supply all kinds of parts but also no two of them supply the same kind of part. Currently, to obtain these rather natural results, we need to use query statements of existing relational database systems embedded in a programming language. Thus, application programmers or users need to make some extra efforts to obtain the required information.

Investigations from the above examples, we found these situations can be modeled as set covering and partitioning problems. Set covering and partitioning problems are both theoretically and practically important in decision support systems. Numerous situations have been modeled as set covering and partitioning problems [9, 22]. We propose an extension to the database query language SQL to enable applications of these problems to be stated and solved directly by the database system.

The key to the success of a database management system (DBMS), especially of one based on the relational model [3], is the effectiveness of the query optimization module of the system. Query optimization has been an active area of research ever since the beginning of the development of relational DBMSs. Pertinent works on query optimization can be found elsewhere [15, 17]. Each DBMS typically has a number of general database access routines, which are written to implement operations or combinations of operations. We discuss the genetic algorithms which implement the access routines of the proposed operations for processing queries involving the proposed operators. The set covering and partitioning problems are NP-hard. Many algorithms [9, 1, 22, 19, 25, 16] were proposed to solve them. Recently, randomized algorithms are used in query optimization. Simulated Annealing, Iterative Improvement, Two-Phase Optimization, and Genetic Algorithm have already been successfully tried on query optimization [18, 12, 13, 2, 14]. This gives us ample reasons to believe that genetic algorithms will perform well in the proposed operations too.

This paper is organized as follows. The extended SQL operators are defined in Section 2. The semantics and syntax of the extended operators are described. The genetic algorithms used by access routines to implement the proposed operators are described in Section 3. A summary is given in Section 4.

2 Database Operators for Set Covering and Partitioning Problems

In this section, we described formulation of set covering and partitioning problems in SQL. We define special operators for the "WHERE" clause of SQL expressions to give a shorthand notation of such problems. Six constructs, namely, cover, mincover, sumcover, partition, minpartition, and sumpartition are defined. The semantics and syntax of the extended operators are described.

2.1 Definition of the Extended Operators in SQL

We sometimes want to obtain specific objects of an attribute in a relation that relate to the objects of an attribute in another relation. We can formulate this query in SQL by means of a special construct. The syntax of the construct is defined as follows:

```
SELECT   attribute-list
FROM     R1, R2
WHERE    COVER((R1.X, R1.Y1), R2.Y2)
```

Where attributes Y_1 in R_1 and Y_2 in R_2 are defined on the same domain. Relation R_1 describes the relationship between the instances of attributes X and Y_1. The relationship between them is that each instance of X is related to a set of instances of Y_1 and each instance of Y_1 is also related to a set of instances of X. Each distinct instance of Y_1 in R_1 has a corresponding instance of Y_2 in R_2. The semantics of this construct is formally defined as finding a cover of a set.

The COVER operation of relation R_1 on attributes X and Y and relation R_2 on attribute Z—

R_1, R_2 WHERE COVER((R_1.X, R_1.Y), R_2.Z)

—is a relation with the heading (X) and the body is a smallest subset of R_1 so that the tuples (X:x, Y:y) appears in R_1 for all tuples (Z:z) appearing in R_2.

The PARTITION operation is similar to the COVER operation.

The cover operator is an efficient way of determining a minimum cover (the smallest number of sets) of a set, if one exists. However, one may, in addition, wish to take into account the cost of a cover. In this case, one is interested in a cover that minimizes the total cost of the cover. The MINCOVER operator is to find a cover of minimum cost. Finding an optimum solution of this operation is an NP-hard problem. It may take a very long time to find such a solution even for a small size problem. Hence, we provide an alternative operator for users. The alternative operator is to find sub-optimal feasible solutions instead of optimal solutions. The syntax of the construct is defined as follows:

```
SELECT attribute-list
FROM   R1, R2, R3
WHERE MINCOVER((R1.X, R1.Y), R2.Y,
R3.Z), or SUMCOVERconst((R1.X, R1.Y), R2.Y,
R3.Z)
```

Where *const* in the SUMCOVER operator is a constant value. Every distinct instance of X in R_1 has a corresponding instance of Z in R_3 and is used as a cost of the former instance. The semantics of the MINCOVER operation is formally defined as finding a cover of minimum cost, and that of the SUMCOVER operation is formally defined as finding a cover whose cost is less than the given constant value.

The *MINPARTITION* and *SUMPARTITION* operations are similar to the MINCOVER and SUMCOVER operations.

3 Access Routines for Query Processing

Finding a solution of the COVER, MINCOVER, PARTITION, or MINPARTITION operation is NP-hard. To get a near-optimal solution instead of an optimal solution and reduce query evaluation time, we adopt genetic algorithms in our query processing algorithm.

3.1 Genetic Algorithms for the COVER and MINCOVER(SUMCOVER) Operators

A solution of the set covering problem can be easily encoded as a binary chromosome (0, 1, 1, 0, ...) which would be interpreted as the second and third columns are included in the solution, but the first and fourth are not. The fitness function *F(x)* is a large number MAX minus the sum of the cost of the columns used (i.e., the original objective function value) and a penalty for a failure to cover. Therefore, the fitness function is

$$F(x) = MAX - \sum_{i=1}^{n} w_i x_i - P(x)\eta(x)$$

where x_i and are binary with $\eta(x) = 0$ if the solution x is a cover and $\eta(x) = 1$ otherwise. *P(x)* is the penalty function procedure for the covering. The latter term

$$\sum_{i=1}^{n} w_i x_i + P(x)\eta(x)$$

will be referred to as "cost."

The *penalty function procedure* P3 of [19] is stated as follows:

P3: If S is a cover, cost = $\sum_{i \in S} Wi$

If S fails to be a cover, let R be the rows that remain uncovered. For each $i \in R$, let S_i be the columns that cover i. Let $w^*_i = \min \{w_j\}$ for $j \in S_i$. Then cost = $\sum_{i \in S} Wi + \sum_{i \in R} W^*_i$

In order to find a cover with lower cost, the above penalty function procedure P3 is simply modified to become P3' as follows:

P3': If S is a cover, cost = $\sum_{i \in S} Wi$

If S fails to be a cover, let R be the rows that remain uncovered. For each $j \in S$, strike column j and the rows covered by j. For some $i \in R$, let S_i be the unused columns that cover i. For each $j \in S_i$, calculate the cost-ratio(j)(w_j / number-of-1's-uncovered in j). Let j* = min { cost-ratio(j) } for $j \in S_i$. Append to S the column j*. The process is repeated until S becomes a cover.

Where cost-ratio(j) is the cost of column j over the number of 1's in the column j. The *greedy crossover* in [19] is used in our genetic algorithm.

3.2 Genetic Algorithms for the PARTITION and MINPARTITION(SUMPARTITION) Operations

The genetic algorithm starts by checking whether an individual violates the definition of partition. That is to say, no element is covered by more than one set. If an individual violates the definition of set partitioning problem, then a mutation operator is repeatedly applied until no individual violates the definition of set partitioning problem. Then the algorithm computes the fitness of all valid individuals by using the given penalty function procdure PP and uses the greedy crossover of [19] to generate the offspring. Finally, the algorithm examines whether the partition solution covers all elements. If the partition solution does not cover all elements, then it attempts to improve the partition solution obtained. The penalty function procedure PP is described below:

PP: if S is a partition, cost = $\sum_{i \in S} Wi$

If S fails to be a partition , let R be the rows that remain uncovered. For each $j \in S$, strike column j and the rows covered by j. In addition, for each $j \in S$, every column $k \neq j$ such that $a_{ij}=a_{ik}=1$ (i=1, ..., m) must be deleted. For some $i \in R$, let S_i be the unused column that cover i. If S_i is an empty set, let cost=∞. Otherwise, for each $j \in S_i$, calculate the cost-ratio(j)(w_j / number-of-1's-uncovered in j). Let j* = min { cost-ratio(j) } for $j \in S_i$. Append to S the column j*. The process is repeated until S is a partition or the cost is ∞.

The penalty function procedure PP is used for fitness evaluation. The *genetic algorithm for partition, sumpartition, and minpartition* is described below.

1. If any row *r* of *A* has all 0's , there is no solution since constraint *r* cannot be satisfied.
2. Choose a desired population size n and initialize the starting population *P* randomly.
3. Check whether a chromosome of the population *P* violates the definition of set partitioning problem. If any chromosome violates the definition, mutate a randomly chosen bit of 1's of this chromosome. The step is repeated until no chromosomes of population *P* violate the definition of set partitioning problem.
4. Evaluate fitness of each individual according to the penalty function procedure *PP*.
5. If the fitness of all individuals is equal to 0, there is no partition solution. The algorithm returns "No Partition.Solution" and terminates.
6. If termination-condition held, go to Step 7; else probabilistically select a pair of individuals (according to their fitness determined by the penalty function procedure PP) to generate the offspring using greedy crossover. The step is repeated until *n* offspring are generated. Return to step 3.
7. If the resultant solution, say S, contains a partition solution, the algorithm terminates. If S fails to be a partition, let R be the rows that remain uncovered. For each $j \in S$, strike column j and the rows covered by j. In addition, for each $j \in S$, every column $k \neq j$ such that $a_{ij}=a_{ik}=1$ (i=1, ..., m) must be deleted. For some $i \in R$, let S_i be the unused columns that cover i. If S_i contains only one column, append to S the column (say column c). Strike column c and the rows covered by c. Otherwise, for each $j \in S_i$, calculate the cost-ratio(j) (w_j / number-of-1's-uncovered in j). Let j* = min{cost-ratio(j)} for $j \in S_i$. Append to S the column j*. The step is repeated until S is a partition.

Step 7 in the algorithm attempts to improve the partition solution obtained in step 6 by applying the procedure like the penalty function procedure. After applying this local fix-up procedure of step 7, feasible solutions have been always produced.

3.3 Experiments and Results

Simulations have been performed for the set covering problem on 18 test problems, and for the set partitioning problem on 15 test problems. The problem size of set covering problems was obtained from the work [21]. The problem size of set partitioning problems 1-5 was obtained from [1]. The reamining problem sizes were designed by the authors. The examined instances have been generated as follows. The *A* matrix for all set covering and set partitioning problems was obtained from a matrix generator. All the test problems have coefficient matrices whose density varies from 1% to 20%. If a row or a column of *A* does not contain 1 in the row or the column then one 1 is added in any entry of the row or column. That is, each row or column must at least contain one 1. Following the work of [1], the coefficient of the objective function was equal to the number of ones in the corresponding column plus a random variable between 0 and 1. Information on these test problems, as well as on the computational results, is presented in Tables I and II. The genetic algorithms were programmed in C and run on a DEC station 5000/200.

TABLE I

Computational Results with Genetic Algorithm for Set Covering Problems

No.	m	n	cpu time*	cost	rate**(%)
1	15	32	0.060	22.132	47.5
2	30	30	0.156	48.089	60.2
3	30	40	0.218	43.152	43.8
4	30	50	0.316	40.230	34.1
5	30	60	0.339	46.789	55.9
6	30	70	0.476	44.121	47.0
7	30	80	0.546	44.265	47.5
8	30	90	0.437	40.984	36.6
9	200	300	17.281	375.320	87.6
10	200	413	24.580	367.645	83.8
11	50	450	4.402	66.269	32.5
12	36	455	3.706	41.410	15.0
13	104	498	10.731	161.914	55.6
14	200	500	23.350	396.976	98.4
15	46	683	7.921	54.867	19.2
16	26	777	4.890	28.910	11.1
17	50	905	12.041	58.920	17.8
18	134	1642	62.554	211.726	58.0

Note: m=Number of constraints, n=Number of variables

*DEC station 5000/200 (Seconds), ** rate= $\frac{|m - cost|}{m}$, the rate of worst case error estimate.

Also, we compared the genetic algorithms with the *interior point algorithm* of linear programming [16] on two set covering problems: A_{27}, A_{45} whose optimal solutions are known. Problems A_{27} and A_{45} are taken from [8]. These two problems are difficult set covering problems [16]. Table III shows the test problems, the size of the best known cover for each, and the size of the cover obtained from the proposed genetic algorithms.

For most examples, we can see that the rate of worst case error estimate is poor even when the solution is optimal. This is because the optimal solution is not known and the value of the number of constraints is used in the error estimate instead.

TABLE II

Computational Results with Genetic Algorithm for Set Partitioning Problems

No.	m	n	cpu time*	cost	rate** (%)
1	13	87	0.269	15.854	21.9
2	13	63	0.191	16.755	28.8
3	14	71	0.246	17.848	27.4
4	12	75	0.292	15.894	32.4
5	13	88	0.312	15.879	22.1
6	50	200	20.049	62.940	25.8
7	100	300	102.908	126.060	26.0
8	100	400	108.020	130.235	30.2
9	100	500	127.571	127.814	27.8
10	100	600	159.875	129.429	29.4
11	150	700	696.775	183.920	22.6
12	200	800	773.126	241.176	20.5
13	200	900	830.743	245.133	22.5
14	100	1000	476.942	113.852	13.8
15	200	1000	1996.273	242.022	21.0

Note:m=Number of constraints, n=Number of variables
*, ** have the same meaning as in Table I

TABLE III

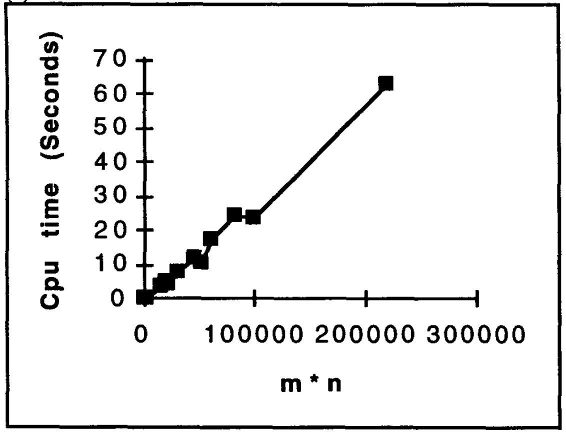
Problem Set

Problem	Integer program variables/constraints	IPA*	GA**	Optimal
A27	27/116	18	18	yes
A45	45/330	30	31	yes

Note: *IPA=interior point algorithm [16]
**GA=genetic algorithms

The processing time of these test problems is presented in Figure 1, where the x-axis represents the product of the number of constraints and the number of variables, and the y-axis represents the processing time in seconds. The results show that the CPU time is nearly proportional to the product of the number of constraints and the number of variables for the set covering problems. The CPU time for set partitioning problems is larger than that for the set covering problems. This is to be expected since the penalty function of set partitioning problems is much more complicated than that of set covering problems. We are currently working on the problem of reducing the processing time of the set partitioning problems, possibly by adopting the Annealing Genetic algorithm of Lin, Kao, and Hsu [20].

(a)



(b)

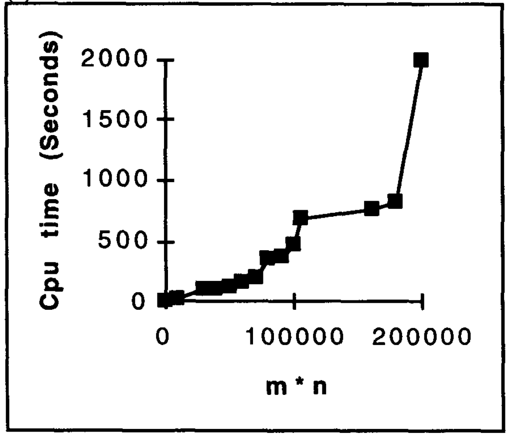


Figure 1: Average Processing time: (a) for set covering problems and (b) for set partitioning problems

As can be expected, if the product of the number of constraints and the number of variables is small, the execution time is small too. Therefore, to reduce execution time in query processing, algebraic operations should be performed as early as possible.

4 CONCLUSIONS

We described formulation of set covering and partitioning problems in SQL. We define special operators for the "WHERE" clause of SQL expressions to give a shorthand notation of such problems. We have presented genetic algorithms for the implementation of the access routines for the proposed operators. At present, our major works are centered in the extension of DBMS operators and the derivation of the penalty function procedures for the proposed extended DBMS operators. We found that our genetic algorithm approach performed quite well both on the computational effort and the solution quality through a variety of test problems. These algorithms are bounded by a polynomial time. Therefore, this approach makes it possible for a DBMS to respond to queries involving the proposed operators in a restricted amount of time in real-time applications. In the future, we plan to adopt annealing genetic approach [20] to implement the access routines for the proposed operators in query optimization. In addition, we plan to modify the greedy crossover for better performance.

References

[1] E. Balas and C.H. Martin, "Pivot and Complement - A Heuristic for 0-1 Programming," Management Science 26(1), pp. 86-96, January 1980.

[2] K. Bennett, M. C. Ferris, and Y. E. Ioannidis, "A Genetic Algorithm for Database Query Optimization," Proceedings of the Fifth International Conference on Genetic Algorithms, 1991.

[3] E.F. Codd, A Relational Model of Data for Large Shared Data Banks, CACM, 13(6):377-387, 1970.

[4] E.F. Codd, The Relational Model for Database Management, Version 2. Reading, MA: Addison-Wesley, 1990.

[5] C. J. Date, A Guide to the SQL Standard, Addison-Wesley, Reading, Mass., 1987.

[6] C. J. Date, "An Introduction to Database Systems," Volume I, Fifth Edition, Reading, MA: Addison-Wesley, 1990.

[7] R. Elmasri and S.B. Navathe, "Fundamentals of Database Systems," The Benjamin/Cummings

Publishing Company, Inc., 390 Bridge Parkway, Redwood City, California, 1989.

[8] D. R. Fulkerson, G. L. Nemhauser and L. E. Trotter Jr., "Two Computationally Difficult Set Covering Problems that Arise in Computing the 1-Width of Incidence Matrices of Steiner Triple Systems," Mathmatical Programming Study 2 (1974) 72-81.

[9] R. S. Garfinkel and G. L. Nemhauser, "Integer Programming," John Wiley & Sons, 1972.

[10] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison-Wesley, 1989.

[11] J. H. Holland, "Adaption in Natural and Artificial Systems," Ann Arbor: The University of Michigan Press, 1975.

[12] Y. E. Ioannidis and E. Wong, "Query Optimization by Simulated Annealing," in Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data, San Francisco, CA, May 1987, pp. 9-22.

[13] Y. E. Ioannidis and Y. C. Kang, "Randomized Algorithms for Optimizing large Join Queries," in Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data, Allantic City, NJ, May 1990, pp. 312-321.

[14] Y. E. Ioannidis and Y. C. Kang, "Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization," in Proc. of the 1991 ACM-SIGMOD Conference on the Management of Data, May 1991, pp. 168-177.

[15] M. Jarke and J. Koch, "Query Optimization in Database Systems," ACM Computing Surveys, 16(2):111-152, June 1984.

[16] N. Karmarker, M.G.C. Resende and K.G. Ramakrishnan, "An Interior Point Algorithm to Solve Computationally Difficult Set Covering Problems," Mathmatical Programming 52 (1991) 597-618.

[17] W. Kim, D. Reiner, and D. Batory, Query Processing in Database Systems. Springer Verlag, N. Y., 1986.

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

[19] G.E. Liepins, M.R. Hilliard, J. Richardson, and M. Palmer, "Genetic Algorithms Applications to Set Covering and Traveling Salesman Problems," in Operations Research and Artificial Intelligence: The Integration of Problem-Solving Strategies, edited by D.E. Brown, C. White, III, Kluwer Academic Publishers, 1990.

[20] F.T. Lin, C.Y. Kao, and C.C. Hsu, "Applying the Genetic Approach to Simulated Annealing in Solving Some NP-Hard Problems," To Appear in IEEE Trans. on Systems, Man, Cybernetics, 23(3), May 1993.

[21] H.M. Salkin and R.D. Koncal, "Set Covering by an All Integer Algorithm: Computational Experience," Journal of ACM, 20(2):189-193, April 1973.

[22] H.M. Salkin and K. Mathur, Foundations of Integer Programming, Elsevier Science Publishing Co., Inc., 1989.

[23] A. Swami and A. Gupta, "Optimization of Large Join Queries," In Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data, pp. 8-17, Chicago, IL, June 1988.

[24] R.F. van de Lans, "Introduction to SQL," Addison-Wesley, 1988.

[25] V. Zissimopoulos, V. Th. Paschos, and F. Pekergin, "On the Approximation of NP-Complete Problems by Using the Boltzmann Machine Method: the Cases of Some Covering and Packing Problems," IEEE Trans. on Computers, 40(12):1413-1418, December 1991.