International Journal of Computers, Communications & Control
Vol. II (2007), No. 3, pp. 288-298

Optimization of Queries with Conjunction of Predicates

Nicoleta Livia Tudor

Abstract:
A method to optimize the access at the objects of a relational database is through the optimization of the queries. This article presents an approach of the cost model used in optimization of Select-Project-Join (SPJ) queries with conjunction of predicates and proposes a join optimization algorithm named System RO-H (System Rank Ordering Heuristic). The System RO-H algorithm for optimizing SPJ queries with conjunction of predicates is a System R Dynamic Programming algorithm that extends optimal linear join subplans using a rank-ordering heuristic method as follows: choosing a predicate in ascending order according to the h-metric, where the h-metric depends on the selectivity and the cost per tuple of the predicate, using an expression with heuristic constants.
The System Rank-Ordering Heuristic algorithm finds an optimal plan in the space of linear left deep join trees. The System RO-H algorithm saves not a single plan, but multiple optimal plans for every subset, one for each distinct such order, termed interesting order. In order to build an optimal execution plan for a set S of i relations, the optimal plan for each subset of S, consisting of i-1 relations is extended, using the Lemma based on a h-metric for predicates. Optimal plans for subsets are stored and reused. The optimization algorithm chooses a plan of least cost from the execution space.
Keywords: optimal join subplans, cost function, query tree, join optimization algorithm, h-metric, heuristic method

1 Introduction

This article proposes a Select-Project-Join query optimization algorithm, based on a heuristic function and suggests an approach of the cost model used in optimization of queries with conjunction of predicates.

The paper is organized as follows:

- the section Prior work presents the improved alternatives of join optimization algorithms
- the section System Rank Ordering Heuristic Algorithm: Algorithm shows a different improved System R Dynamic Programming algorithm based on utilization of a heuristic function and the mathematical expression of the System Rank Ordering Heuristic algorithm
- the section Performance Evaluation presents the performance results and implementation details.

2 Prior Work

There are known many improved alternatives for optimizing queries with user defined predicates:

System R dynamic programming algorithm [1] illustrates the System R dynamic programming algorithm that finds an optimal plan in the space of linear (left-deep) join trees [2]. The algorithm proceeds by building optimal execution plans for increasingly larger subsets of the set of all relations in the join. In order to build an optimal plan for a set S of i+1 relations, the optimal plan for each subset of S, consisting of i relations is extended, and the cheapest of the extended plans is chosen. The System-R algorithm

saves not a single plan, but multiple optimal plans for every subset S, one for each distinct such order, termed interesting order. The enumeration complexity of the algorithm is $O(n2^{n-1})$, where R_1, R_2,... , R_n - relations.

The optimization algorithm for the space of bushy join trees is similar to the System-R algorithm, except that the both inputs of a join operator can be an intermediate result. The number of optimal subplans that must be stored for a join with n tables is 2n times the number of interesting orders. The complexity is $O(3^n)$.

LDL algorithm was used in the LDL project at MCC [3] and subsequently at the Papyrus project at HP Laboratories [4]. The LDL algorithm treats expensive predicates and relations alike and may produce plans that are significantly worse than plans produced by the traditional optimization algorithm where all selections are evaluated as early as possible. LDL algorithm cannot consider all plans in the space of unconstrained linear execution trees. Hellerstein [5] shows that the LDL algorithm fails to consider plans that evaluate expensive predicates on operands of a join prior to taking the join. In order to optimize a query that consists of a join of n relations and k expensive predicates, the dynamic programming algorithm will need to construct 2^{n+k} optimal subplans.

The predicate migration algorithm improves on the LDL approach. Predicate migration approach, which given a linear join tree, chooses a way of interleaving the join and the user-defined predicates and is integrated with a System R style optimizer [5]. The algorithm places the user-defined predicates in their optimal position relative to the join nodes. This approach has serious drawbacks that limit its applicability [6]: it cannot guarantee an optimal plan because it uses a heuristic to estimate the rank of join predicates that influence the choice of the plan.

The naive optimization algorithm for the space of unconstrained linear join trees behaves exactly like the System R algorithm. The total number of stored plans per each distinct set of relations increases to 2^k, and the number of plans that need to be stored increases to 2^{k+n}, where k is the number of user-defined predicates. The complexity of this algorithm is exponential in the number of user-defined predicates and in the number of relations in the query.

Optimization algorithms with complete rank-ordering [7] use the ability to order the execution of predicates (called ranks or rank-order), that were applied prior to application of any other operators. When the join methods are regular, the algorithm restricts the sequence in which the user-defined predicates may be applied and reduces the complexity of enumeration from exponential to polynomial in the number of user-defined predicates.

Optimization algorithm with pruning [7] compares and prunes plans that have different tags. The "udp-pushdown" rule provides a sufficient condition for a predicate to be pushed down and it can be used to pin the selections as soon as they are evaluable and helps avoid constructing plans where the predicates are pulled up. The "udp-pullover" rule allows avoid generating alternative plans that push down user-defined predicates and are suboptimal.

The conservative local heuristic algorithm can choose among plans that result from application of a sequence of udp-pushdown and udp-pullover rules. The two plans picked by the conservative local heuristic complement each other, and the heuristic can guard against the choice of a plan resulting from greedily pushing down a predicate by the Pull-Rank algorithm. Thus, conservative local heuristic can find optimal plans that Pull-Rank and other global heuristics fail to find due to their greedy approach, but incurs only low computational overhead. As the following lemma states, the conservative local heuristic algorithm produces an optimal plan in several important special cases [7].

In the next section, we present a different improved System R Dynamic Programming algorithm for optimizing Select-Project-Join queries with conjunction of predicates, based on a heuristic method. We implement the optimization algorithm by extending a System R style optimizer.

3 System Rank Ordering Heuristic Algorithm

In this section, we discuss an approache proposed for optimizing Select-Project-Join (SPJ) queries with conjunction of predicates, using an algorithm System RO-H (System Rank Ordering Heuristic), based on a heuristic method.

This algorithm extends optimal linear join subplans by choosing one predicate in ascending order according to the h-metric (heuristic metric). H-metric determines minimum value between the rank of the predicate and the ratio between selectivity minus 1 and cost per tuple.

To define the heuristic method for extending optimal linear join subplans, the following shall be considered.

3.1 Regular Join Methods

Let us consider a class of SPJ queries on relations R_1, R_2, ..., R_n, n ∈N* and an implementation of the JOIN operator with conjunction of k predicates p_1, ..., p_k, k∈N*.

Definition 1. A join method is called regular if the cost f(R_1, R_2) of joining two relations of sizes R_1 and R_2 depends on the sizes of the relations as follows: $f(R_1,R_2)$ = a + bR_1 + cR_2 + dR_1R_2 where the constants a, b, c, d are independent of the sizes of relations R_1 and R_2 [7].

If join operators follow the assumption of being regular joins, we can restrict enumeration to execution trees where all predicates are ordered by rank order heuristic.

3.2 Tags for Plan Representation

The following definition states formally how we associate a tag with a join tree [7]:

Definition 2. Let T be an unconstrained linear join tree that consists of a join among a set R of relations and evaluation of a set U ⊆ S of user defined predicates where S is the set of all user defined predicates in the query that can be evaluated over the subexpression of the query that consists of the join among relations in R. Then, the tag associated with the tree T is the ordered set of predicates S−U, sorted by rank order.

Figure 1 illustrates the execution plans that need to be considered when there are three relations R_1, R_2, R_3 and two selection predicates p_1, p_2 on R_1. T and T' are possible plans for $R_1 \bowtie R_2 \bowtie R_3$ (each with differing tags). The tags for T and T' are $< >$ and $< p_1 >$ respectively.

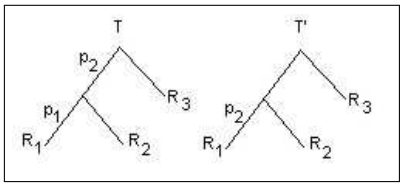

Figure 1: The execution plans

3.3 Predicate Order in Optimal Plans

Let us consider a class of SPJ queries on relations R_1, R_2, ... , R_n and an implementation of the JOIN operator with conjunction of k predicates p_1, ..., p_k. The problem of join ordering, addressed in [8], [9], [10] and [11] utilizes the notion of a rank. The rank of a predicate p_i, (i = 1, ..., n), rank(p_i) = costpertuple(p_i) / (1 − selectivity(p_i)). Chaudhuri and Shim refer to such execution trees as rank ordered [7], as defined below:

Definition 3. The user-defined predicates in an unconstrained execution tree T are rank ordered if for any two user-defined predicates p and p' in T such that rank(p) < rank(p'), either p precedes p' in the tree T, or p is not evaluable in the tree T' obtained by exchanging the positions of p and p' in T.

Hellerstein et al. [12] consider expensive predicates, i.e., where the computation needed for evaluating whether the predicate is true or false dominates the overall cost [12]. In that context, it is shown that predicates should be ranked in ascending order according to the metric $(selectivity-1)/(costpertuple)$. Hellerstein et al. consider that the processor is perfect in its prediction, and it predicts the branch to the next iteration of the query will be taken when the selectivity ≤ 0.5, and will not be taken when selectivity > 0.5.

In System Rank Ordering Heuristic algorithm, we refer to such execution trees as rank ordered, according to the h-metric based on a heuristic method.

Definition 4. We call the h-metric (heuristic metric) of predicate p_i, i = 1, ... ,k having selectivity s_i, the pair

$$\left(s_i, \frac{c_1 * s_i + c_2 * (1 - s_i)}{costpertuple(p_i)}\right),$$

where c_1 and c_2 are heuristic constants, defined as follows: c_1 = 0, c_2 = -1, if $0.5 < s_i \leq 1$ and c_2 = 0, c_1 = -1, if $0 \leq s_i \leq 0.5$.

Observations:

1. The utility of the h-metric is that we avoid generating a large number of intermediate-quality plans, that improve on the currently computed best cost, without being optimal.

2. The heuristic method is not guaranteed to find the optimal solution, but we will demonstrate that it finds good solutions.

Lemma 5. *Let T be an unconstrained linear join tree that consists of a join among a set of n relations and let s_i and s_j be the selectivities for p_i and p_j respectively. The plan of the execution tree T cannot be optimal if $s_i \geq s_j$ and*

$$\frac{c_1 * s_i + c_2 * (1 - s_i)}{costpertuple(p_i)} > \frac{c_1 * s_j + c_2 * (1 - s_j)}{costpertuple(p_j)},$$

where c_1 and c_2 are heuristic constants, defined as follows: c_1 = 0, c_2 = -1, if $0.5 < s_i \leq 1$ and c_2 = 0, c_1 = -1, if $0 \leq s_i \leq 0.5$.

Proof:

Let T be an unconstrained linear join tree that consists of a join among a set R of n relations, two predicates p_1, p_2 with $s_1 \geq s_2$ and let τ be a subexpression among execution tree T and let us refer to this subexpression by τ(R), where the parameter R refers to the input relation of τ. Then:

$$\tau(R) = \tau(\sigma_{p1}(R1))$$

We can relate the execution trees in Figure 2 and the following correspondence holds:

$$T = \sigma_{p_2}(\tau)$$

The following parameters shall be defined:

- cost(p_1) = cost of predicate τ per tuple
- $|R_1|$ = number of tuples in relation R_1

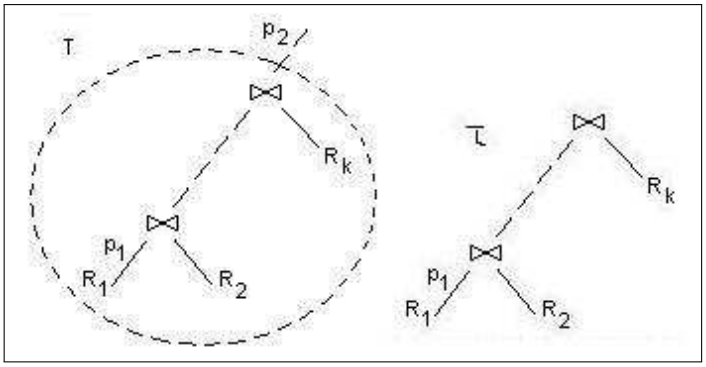

Figure 2: τ: subexpression of execution tree T

- cost(τ) = cost of execution tree τ
- size(τ) = size of the output of execution tree τ
- s_i = selectivity of the predicate p_i
- $\tau_0 = \tau\ (R_1)$

We now estimate the cost of the execution tree T as follows:

$$Cost(T) = cost(p_1)|R_1| + cost(\tau) + cost(p_2)size(\tau),$$

where size(τ) = size(τ_0) s_1, then Cost(T) = cost(p_1) $|R_1|$ + cost(τ) + cost(p_2) size(τ_0) s_1

We can represent that the cost of an expression $\tau\ (R_1)$ to be the sum of the following three components:

1. Cost of evaluating predicates in the expression $\tau\ (R_1)$: the sum of all such costs is

$$\sum_i cost_i size_i(R_1)$$

where $cost_i$ is the cost of applying the predicate per tuple and $size_i$ is the size of the relation preceding the i-th application of a predicate.

2. Cost of evaluating join nodes in the expression $\tau\ (R_1)$ that are ancestors of R_1: the sum of all such costs is

$$\sum_j cost(Join)_j(R_1)$$

3. Cost of evaluating all other operators that are not affected by the input relation R_1: We denote this cost by $cost_0$. Then the cost of an expression $\tau\ (R_1)$ can be computed as follows:

$$Cost(\tau(R_1)) = \sum_i cost_i size_i(R_1) + \sum_j cost(Join)_j(R_1) + cost_0$$

$$Cost(T) = cost(p_1)|R_1| + \sum_i cost_i size_i(R_1) + \sum_j cost(Join)_j(R_1) + cost_0 + cost(p_2)size(\tau_0)s_1$$

If s1 $\geq$ s2 and cost(p_1) = cost(p_2), then

$$\frac{c_1 * s_1 + c_2 * (1 - s_1)}{costpertuple(p_1)} > \frac{c_1 * s_2 + c_2 * (1 - s_2)}{costpertuple(p_2)}$$

If we change p_1 with p_2, let T' be a join tree that consists of a join among a set R of n relations, two predicates p_1, p_2 with $s_1 \geq s_2$, then

$$Cost(T') = cost(p_2)|R_1| + \sum_i cost_i size_i(R_1) + \sum_j cost(Join)_j(R_1) + cost_0 + cost(p_1) size(\tau_0) s_2$$

cost (T) > cost (T') $\Rightarrow$ the plan of the execution tree T cannot be optimal

A corollary of this lemma is that whenever two consecutive terms appear anywhere as conjunctions in an optimal plan, then the one with lower selectivity must appear first if it has the same h-metric.

We use Lemma 5. in the System Rank-Ordering Heuristic algorithm below.

3.4 System Rank-Ordering Heuristic Algorithm

The System Rank-Ordering Heuristic algorithm finds an optimal plan in the space of linear (left-deep) join trees. The cost function assigns a real number to any given plan in the execution space and satisfies the principle of optimality [13]. An optimal plan for a set of relations must be an extension of an optimal plan for some subset of the set. The optimization algorithm chooses a plan of least cost from the execution space.

Definition 6. The System Rank-Ordering Heuristic algorithm for optimizing SPJ queries with conjunction of predicates is a System R Dynamic Programming algorithm that extends optimal linear join subplans using a rank-ordering heuristic method as follows:

- choosing a predicate in ascending order according to the h-metric
- h-metric depends on the selectivity and the cost per tuple of the predicate, using an expression with heuristic constants.

The System Rank-Ordering Heuristic algorithm saves not a single plan, but multiple optimal plans for every subset, one for each distinct such order, termed interesting order [14]. In order to build an optimal execution plan for a set S of i relations, the optimal plan for each subset of S, consisting of i - 1 relations is extended, using the Lemma 1 based on a h-metric for k predicates. Optimal plans for subsets are stored in the OptPlan() array and are reused.

The System Rank-Ordering Heuristic algorithm is presented as follows:

```
Procedure System Rank-Ordering Heuristic
for i = 2 to n do
   for all  S  from  {R1, ..., Rn} with | S | = i
     BestPlan = a plan with infinite cost
     for all Rj, Sj, where S =
                union ({Rj}, Sj),intersect({Rj},Sj) = null
     for all P from OptPlan(Sj, t) with all different tag t
              nr1 = |P|; nr2 = | Rj |
              r1 = array with evaluable predicates on P
              r2 = array with evaluable predicates on Rj
              EuristicOrder(r1); EuristicOrder(r2)
              // ascending ordering of predicates
              // in the P, Rj according to the h-metric
              for i = 0 to nr1
              for j = 0 to nr2
                   p' = ExtendJoinPlan( P, Rj, r1[i], r2[j])
                   if cost (p') < cost(BestPlan[tag(p')])
```

```
                        BestPlan[tag(p')] = p'
                    endif
                repeat
                repeat
     repeat
     repeat
     OptPlan(S) = BestPlan
   repeat
repeat
finalPlan = a plan with infinite cost
for all plan P from OptPlan({R1, Ě, Rn})
    if complete_cost(P) < cost(finalPlan)
          finalPlan = completed plan of P
    endif
repeat
return(finalPlan)
end

Function ExtendJoinPlan( P, Rj, r1[i], r2[j])
  let r1[i] be a predicate applied to the plan P for Sj, t
  r2[j] = a predicate for the relation Rj
  p' = the join plan between Sj, t and Rj,
    for the subset of predicates (r1[i], r2[j])
  extend OptPlan(Sj, t) with the plan p'
return

Procedure EuristicOrder(r1)
  for i = 0 to nr1 - 1
  for j = i + 1 to nr1
    // si = selectivity for r1[i]
    // cost (pi) = cost of the predicate r1[i]
    if r1[i] > r1[j] and
     (c1*si + c2*(1-si)) / cost(pi) > (c1*sj + c2*(1-sj)) / cost(pj)
      change r1[i] with r1[j]
    endif
  repeat
  repeat
return
```

Observation: Arrays r1 and r2 are useful for ascending ordering of the predicates according to the h-metric.

When the join methods are regular, the System Rank-Ordering Heuristic algorithm enables us to further restrict the sequence in which the predicates may be applied and reduces the complexity of enumeration in the number of predicates.

4 Performance Evaluation

In this section we present the results of performance evaluations on our implementation.

4.1 Experimental Framework

We used an experimental framework similar to that in [15] and [16] and [7]. We performed experiments using an AMD Athlon(tm)XP 1600+ machine with 256 MB of RAM and running Windows XP Professional version 2002.

The algorithms were run on queries consisting of equijoins. Relation cardinalities ranged from a hundred to a thousand tuples, and the numbers of unique values in join columns varied from $25^o/_o$ to $100^o/_o$ of the corresponding relation cardinality. The selectivities of predicates were randomly chosen from 10^{-4} to 1.0 and the cost per tuple of predicates was represented by the number of I/O accesses and selected randomly from 1 to 1000.

We considered nested-loop, merge-scan, and simple and hybrid hash joins as join methods [17]. In our experiments, only the cost for number of I/O accesses was accounted for. For our experiments, we generated 3 join (join among four relations) queries, 5 join queries, and 7 join queries.

We performed two sets of experiments. In the first set, we varied the number of selection predicates that apply on one relation. In the second set, we varied the distribution of the selection predicates among multiple relations in the query, i.e., we kept the number of selection predicates fixed, but varied how these predicates are distributed among the relations in a query.

4.2 Candidate Algorithms

For each query instance, we ran the following optimization algorithms:

- System R Dynamic Programming algorithm: The system R style optimization algorithm that evaluates all predicates as early as possible
- Optimization Algorithms With Complete Rank-Ordering: It compares plans that have the same tag over the same set of relations
- Opt-rank-conservative algorithm: This algorithm uses conservative local heuristic with complete rank-ordering
- System Rank-Ordering Heuristic algorithm that extends optimal linear join subplans using a rank-ordering heuristic method.

4.3 Effect of Number of Predicates

In this set of experiments, the number of predicates was varied from 1 to 5 and the number of join queries was varied from 3 to 7 (7 joins for 8 relations). The results presented for each data point represents an average over 100 queries. These queries were generated by randomly choosing one relation on which all the predicates apply and then randomly picking the cost and selectivities of the predicates as well.

Table 1 shows the average number of enumerated plans for the algorithms: System R Dynamic Programming, Optimization With Complete Rank-Ordering, Opt-rank-conservative and System Rank-Ordering Heuristic algorithm.

Figures 3, 4 and 5 show a comparison of the performances (average number of enumerated plans) for the 4 algorithms (System R Dynamic Programming, Optimization With Complete Rank-Ordering, Opt-rank-conservative and System Rank-Ordering Heuristic algorithm).

The results obtained for queries with 3, 5 or 7 joins show a similar trend:

- the enumerations necessary in the System R Dynamic Programming algorithm is independent of the number of predicates

Number of predicates / Number of enumerated plans	1	2	3	4	5
Complete Rank-Ordering	250.85	811.93	1148.57	1377.53	1603.26
System Rank-Ordering Heuristic	251.45	285.47	429.33	507.84	527.85
Opt-Rank-Conservative	117.66	139.69	169.37	194.30	225.40
System R Dynamic Programming	90.17	90.17	90.17	90.17	90.17

Table 1. Worst-Case Estimates for Enumerated Plans (3 Join Query)

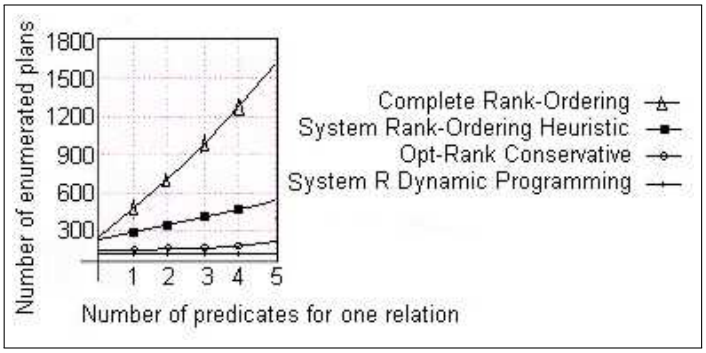

Figure 3: 3 Join Query

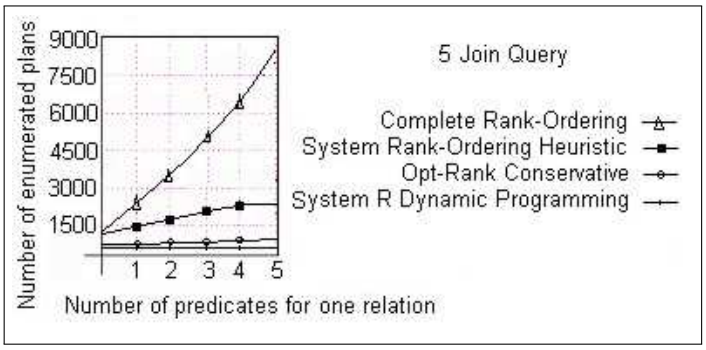

Figure 4: 5 Join Query

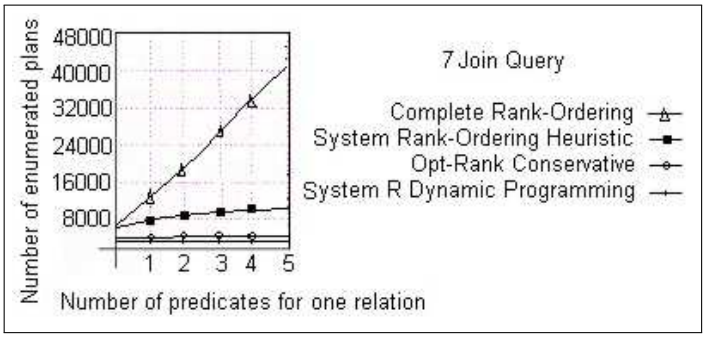

Figure 5: Performance on a varying number of predicates

- the Optimization Algorithms With Complete Rank-Ordering generated more plans than the Opt-rank-conservative algorithm and System Rank-Ordering Heuristic algorithm
- the average number of enumerated plans for the System Rank-Ordering Heuristic algorithm is approximately linear
- the gap in the number of enumerated plans between the Complete Rank-Ordering algorithm and System Rank-Ordering Heuristic algorithm increases significantly as the number of predicates grows

5 Conclusion

This article presents an approach of the cost model used in optimization of Select-Project-Join (SPJ) queries with conjunction of predicates and proposes a join optimization algorithm named System RO-H (System Rank Ordering Heuristic). The System RO-H algorithm is a System R Dynamic Programming algorithm that extends optimal linear join subplans using a rank ordering heuristic method. The comparison of the performances of algorithms shows that our proposed System Rank-Ordering Heuristic techniques are extremely effective and are guaranteed to generate optimal plans.

Bibliography

[1] S. Ganguly, W. Hasan, R. Krishnamurthy, Query optimization for parallel execution, *In Proceedings of the ACM SIGMOD International Conference on Management of Data SIGMOD*, Ed. ACM Press, New York, pp. 9-18, 1992.

[2] P. G. Selinger, M. M. Astrahan, R. A. Lorie, T. G. Price, Access path selection in a relational database management system, *In Proceedings of ACM SIGMOD International Conference on Management of Data SIGMOD, Boston, MA,* ACM Press, New York, pp. 23-34, 1979.

[3] D. Chimenti, R. Gamboa, R. Krishnamurthy, Towards an open architecture for LDL, *In Proceedings of the 15th International Conference on Very Large Data Bases, VLDB, Netherlands,* Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, pp. 195-203, 1989.

[4] S. Chaudhuri, K. Shim, Query optimization in the presence of foreign functions, *In Proceedings of the 19th International Conference on Very Large Data Bases, Ireland,* Morgan Kaufmann Publishers Inc., San Francisco, CA, pp. 526-541, 1993.

[5] J. M. Hellerstein, Practical predicate placement, *In Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapolis,* ACM Press, New York, pp. 325-335, 1994.

[6] J. M. Hellerstein, M. Stonebraker, Predicate migration: Optimizing queries with expensive predicates, *In Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington,* ACM Press, New York, pp. 267-276, 1993.

[7] S. Chaudhuri, K. Shim, Optimization of Queries with User-Defined Predicates, *ACM Transactions on Database Systems,* Vol. 24, No. 2, pp. 177-228, 1999.

[8] J. M. Hellerstein, J. M., M. Stonebraker, Predicate migration: Optimizing queries with expensive predicates, *In Proceedings of the ACM SIGMOD International Conference onManagement of Data, Washington, DC,* Eds. ACM Press, New York, NY, pp. 267-276, 1993.

[9] R. Krishnamurthy, H. Boral, C. Zaniolo, Optimization of nonrecursive queries, *In Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan,* VLDB Endowment, Berkeley, CA, pp. 128-137, 1986.

[10] C. L. Monma, J. Sidney, Sequencing with series-parallel precedence constraints, *Math. Oper. Res. 4,* pp. 215-224, 1979.

[11] K. Y. Whang, R. Krishnamurthy, Query optimization in a memory-resident domain relational calculus database system, *ACM Trans. Database Syst. 15,* pp. 67-95, 1990.

[12] J. M. Hellerstein, M. Stonebraker, Predicate migration: Optimizing queries with expensive predicates, *In Proceedings of the ACM SIGMOD Conference,* 1993.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms,* MIT Press, Cambridge, MA, 1990.

[14] P. G. Selinger, M. M. Astrahan, R. A. Lorie, T. G. Price, Access path selection in a relational database management system, *In Proceedings of ACM SIGMOD International Conference on Management of Data, Boston, MA,* ACM Press, New York, pp. 23-34, 1979.

[15] Y. E. Ioannidis, Y. C. Kang, Randomized algorithms for optimizing large join queries, *In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data,* Ed. ACM Press, New York, NY, pp. 312-321, 1990.

[16] S. Chaudhuri, K. Shim, Including group-by in query optimization, *In Proceedings of the 20th International Conference on Very Large Data Bases,* VLDB Endowment, Berkeley, CA, 1994.

[17] L. D. Shapiro, Join processing in database systems with large main memories, *ACM Trans. Database Syst. 11,* pp. 239-264, 1986.

Tudor Nicoleta Liviana
Petroleum-Gas University of Ploiesti
Department of Computer Science
39 Bucuresti Avenue, 100680, Ploiesti, Romania
E-mail: tudorl@upg-ploiesti.ro
Received: January 25, 2007
Revised: June 13, 2007

Nicoleta Liviana Tudor (born on July 15, 1968) graduated the Faculty of Mathematics of the Bucharest University in 1992. She has 15 years experience of teaching in the field of computer science both in Petroleum-Gas University of Ploiesti, and in other educational institutions of Romania. Since 2006 she is Lecturer in the Computer Science Department at Petroleum-Gas University of Ploiesti, Romania. She is a PhD student at the Petroleum-Gas University of Ploiesti, Control Engineering and Computers Department. Her main research fields are middle-tier business objects, XML web services, data processing, relational databases. She has authored 2 books and more than 14 research papers in the area of relational databases, data structure, and published in international journals, and in the proceedings of prestigious international conferences.