Review of Algorithms for the Join Ordering Problem in Database Query Optimization

S. Vellev

Key Words: *Query optimization; join ordering, relational databases; query execution plan; deterministic algorithms; randomized algorithms; genetic algorithms; hybrid algorithms.*

Abstract. *Finding the optimal join ordering for a database query is a complex combinatorial optimization problem which has been approached by a wide variety of strategies and algorithms, ranging from simple deterministic search to complex hybrid algorithms based on genetic search and incorporating domain-specific heuristics. In this paper we review a set of join ordering algorithms and classify them according to the nature of the search strategy they implement. We also briefly discuss the relative advantages and applicability of different algorithms.*

Introduction

The problem of finding the optimal join ordering executing a query to a relational database management system (RDBMS) is a combinatorial optimization problem. Queries in an RDBMS are defined in a declarative, non-procedural language, such as SQL. This raises the need to transform the declarative query into a procedural, effective plan for its execution. Each query can be mapped to a set of execution plans which are equivalent in terms of the result they generate but the execution cost of the different plans can vary by many orders. The execution plan is selected from the set of all alternatives by a dedicated RDBMS module - the Query Optimizer.

Due to the high processing cost, the evaluation of joins and their ordering are the primary focus of query optimization. Traditionally, the optimization of such expressions is done by complete traversal of the solution space (possibly utilizing some pruning techniques). This is a feasible approach for most of the classic database applications, where the size of the query (measured in number of joined relations) rarely exceeds 8-10, but it is completely inapplicable to some contemporary databases (Object-Oriented Databases, Multimedia Databases) and database applications such as Decision Support Systems (DSS), Online Analytical Processing (OLAP), Data Warehousing, Geographical Information Systems (GIS), etc. Queries in such applications may involve tens or even hundreds of joined relations.

The Join Ordering Problem (JOP) has been approached by several classes of algorithms. It is a generalization of the classical combinatorial Traveling Salesman Problem (TSP) - the problem of finding the shortest Hamiltonian cycle in a complete graph. The TSP is among the best-studied combinatorial optimization problems and dozens of algorithms have been proposed for it. Most of these algorithms are directly applicable to the JOP (which is considerable newer). In this review however, we consider only algorithms already applied to the JOP. An extensive survey of general global optimization algorithms is not in the scope of this work.

Complexity of the Problem

The JOP in its general form is $\mathcal{NP}$-complete. A formal proof of this fact for cyclic queries was first presented in [8]. For small queries, it is still possible to do a complete traversal and find the global optimum, however, as the size of the solution space grows exponentially in the number of joined relations, for larger queries the JOP can no longer be solved exactly in its general form.

The approaches for breaking the $\mathcal{NP}$-completeness can be classified into two groups:

- **Sacrificing Generality**

Restrictions are imposed on the size or form of the solution space, by setting additional requirements to the structure of the solutions, the connectedness of the query graph (e.g. considering only acyclic graphs), the form of the cost function, the implementation of the relational algebra operators, etc.

- **Sacrificing Exactness**

Instead of looking for the global optimum, acceptably good suboptimal solutions are found. The portion of the explored solution space is limited by using some heuristics constructing the solutions or by applying randomized and genetic search algorithms.

These approaches (or a combination of them) usually reduce the complexity of the algorithms from exponential to polynomial under the imposed restrictions.

Join Ordering Strategies

There are three main strategies for optimizing join orders:

- **Bottom-up optimization.**

This is a synthetic approach in which the query execution plan is generated starting from the base relations and generating step by step more and more complete partial execution plans until finally an execution plan for the whole query is obtained.

- **Top-down optimization.**

This is a „divide and conquer" approach in which the query is divided into parts, each part is optimized separately and finally the different partial execution plans are aggregated to form the complete query execution plan.

•Transformation optimization.
This approach starts with some valid complete execution plan which is transformed into another valid complete plan, improving the solution step by step.

Query optimizers are usually using elements of more than one of the above strategies.

The JOP can be approached by four classes of algorithms:

1. Deterministic Algorithms. The algorithms of this class perform some sort of deterministic search of the solution space, either through complete traversal, or by applying some heuristics pruning the space.

2. Randomized Algorithms. These algorithms perform a random *walk* in the solution space, moving from point to point in the solution space. A move is possible if the solution represented by the first point can be transformed into the solution represented by the second point by applying a single transformation rule (from a set of predefined valid transformation rules) to it. The algorithm execution ends either when no more valid moves can be done from the current solution space point or when a predefined run time has elapsed. The best solution found during the random walk is the result of the optimization.

3. Genetic Algorithms. Genetic algorithms mimic the biological evolution in their search for the optimum solution [1, 2]. The main idea is, starting from some initial set (*population*) of solutions to generate offspring by random crossover and mutation. The best individuals (by the cost function) in the population survive on each generation and form the new population. The algorithm stops either after some (finite) number of generations or when the population becomes homogeneous above some threshold according to the cost function [3].

4. Hybrid Algorithms. Hybrid algorithms combine elements of two or more of the above strategies. The solutions found by some deterministic heuristic become the starting point of a randomized search or the initial population of a genetic algorithm (an approach known as *seeding*), a genetic algorithm is enhanced by local search techniques, etc.

Classification of Join Ordering Algorithms

In this section, we will briefly describe the most popular algorithms proposed for the JOP and we will classify them according to the search strategy they implement.

1. Deterministic Algorithms

1.1. Dynamic Programming

This is the algorithm used in practically all existing commercial RDBMS systems. The Dynamic programming has been first suggested as a query optimization strategy in IBM's classical System R by Selinger [4]. The algorithm performs a complete traversal with dynamic pruning of the solution space. It constructs all alternative join trees (fulfilling three classical heuristic constraints) by iterating over the already joined relations and possibly pruning some suboptimal solutions.

The Dynamic Programming algorithm is guaranteed to find the optimum in the solution space constrained by the three heuristics. In many cases it manages to avoid the complete traversal by dynamically pruning part of the suboptimal plans on each step. Although it is still exponential in the general case, for some particular query types the complexity of the algorithm is only $O(N^3)$. In the general case however, the memory and CPU requirements of the Dynamic Programming grow exponentially in the number of joined relations because all concurrent plans generated in the previous algorithm step must be kept. That is why, most of today's database management systems impose restrictions on the size of the relation (usually up to about 15 joins). For relations with less than ten joins, the algorithm has proved its high effectiveness. Today it is considered a standard among the query optimization strategies.

The three classical heuristics applied by the Dynamic Programming algorithm are the following:

•Selection-Projection Heuristic. Selections and projections are processed „on the fly" and almost never generate transitional relations. Selections are processed upon first relation access. Projections are processed while generating the output of other operations. This heuristic prunes only suboptimal solutions - the separate processing of selections and projections would incur additional computational costs.

•Cartesian Product Heuristic. Cartesian products are never formed, except for the case when they are contained in the original query. Relations are always combined through joins. This constraint almost always eliminates suboptimal solutions due to the high cardinality of a typical Cartesian product of two relations. The exceptions are very few and occur in the cases when the cardinality of the particular Cartesian product happens to be small [5].

•Tree Form Heuristic. The third constraint is about the form of the execution plan trees - the internal operand of every join is always a base relation and never a transitional result. Such trees are called *left-deep* (while arbitrary-form trees are referred to as *bushy* and their set is denoted by $\mathcal{A}$) and the subspace of all left-deep solutions is denoted by $\mathcal{L}$. This heuristic can eliminate the optimum plan and is the most controversial of the three. It is claimed that, in most cases, the optimal left-deep tree has cost that is pretty close to the global optimum. There are two heuristic arguments for this - first, having the base relations as internal join operands maximizes the use of the existing indexes and second, having the transitional results as external join operands allows the sequences of nested-loop joins to be flow processed[1]. Flow processing means that the complete sequence of operations are executed on each retrieved tuple instead of executing each single operation on each tuple.

1.2. Iterative Dynamic Programming

This is one of the newest deterministic optimization algorithms [6]. It combines the classical Dynamic Programming with a greedy search strategy. The main advantage of the Dynamic Programming is that it always finds the optimum solution in $\mathcal{L}$. As we already observed however, it has high time complexity and consumes a lot of memory. Suppose that the clas-

[1] A similar argument holds for right-deep trees with respect to the hash-join.

sical Dynamic Programming algorithm has generated all joins of k relations and at this point it has used up all available memory. At this step, instead of trying to generate all join combinations of k+1 relations, the Iterative Dynamic Programming chooses one of the k-relation join plans, discards all other plans containing any of the relations of the selected plan and then restarts the Dynamic Programming algorithm to obtain the join combinations of k+1 relations, then k+2 relations and so on, using the selected k-relation join plan as an atomic „building block".

Such a strategy obviously has far more modest memory requirements. The length k of the partial plans at which the Dynamic Programming is interrupted by the greedy strategy is a parameter of the algorithm. The time complexity is $O(n^k), 2<k<n$.

1.3. Minimum Selectivity Heuristic

Optimal or near-optimal solutions are often characterized by transitional relations with small cardinality. This heuristic constructs left-deep trees, choosing at each step the relation that minimizes the cardinality of the intermediate result [7].

1.4. Top-Down Heuristic

This heuristic is based on the observation that the last joins in a query influence its cost the most. This is explained by the fact that the transitional result usually grows dramatically towards the end of the query evaluation. At each step, this heuristic selects the relation with the minimum cost to join to the intermediate result [7].

1.5. IK Algorithm

Ibaraki and Kameda [8] introduce an algorithm called IK which takes advantage of the special form of the nested-loop cost function. The IK algorithm finds the optimal left-deep tree of an acyclic graph by assigning ranks to relations and ordering the relations according to their rank.

1.6. Krishnamurthy-Boral-Zaniolo (KBZ) Algorithm

This algorithm is presented in details in [9] and is based on the IK algorithm. It finds the minimum spanning tree of an acyclic graph and applies the IK algorithm to the resulting tree. The minimum spanning tree considers the minimum product of edge weights (selectivities). It is important to note that the KBZ algorithm imposes restrictions on the form of the cost function. The algorithm has been successfully used for queries with up to 15 joins.

1.7. Relational Difference Calculus

This is a new heuristic developed from a method called Boolean Difference Calculus [10]. The main idea is to find the most influential relation in a join expression. A detailed description of the method, accompanied by examples can be found in [7].

1.8. Augmentation Heuristic

This algorithm is the generalization of all greedy heuristic algorithms [11]. It is another greedy heuristic which joins relations one by one, selecting the relation according to some greedy criterion on each step. Experiments have been done with five different criteria. The best results were achieved using the selectivity criterion - choosing the relation with minimum selectivity.

1.9. Local Improvements Algorithm

The algorithm makes local improvements on a given left-deep execution plan. A window of size c is moving through the relations permutation. At each step, the group of c relations (called a *cluster*) is locally optimized, substituting the existing permutation with the optimal one. It is easily proven that following such strategy can only improve the initial permutation. The clusters can also overlap each other. If c is the cluster size and o is the overlap size, experiments show that the best combinations (c, o) in decreasing order are (5, 4), (4, 3), (3, 2), (2, 1) and (2, 0), depending on the available optimization time (they are ordered in decreasing time complexity). Increasing the cluster size, the algorithm complexity tends to $O(N!)$, so clusters with size above 5 are not used [11].

1.10. A* Algorithm

In the domain of Artificial Intelligence, the heuristic algorithm called A* is extensively applied to complex search problems. A* has also been proposed for query optimization and may become the direct successor of the traditional Dynamic Programming [12]. Instead of step processing and using all plans with n relations to generate all plans with n+1 relations, the A* algorithm starts developing one of the generated plans based on its expected proximity to the optimal plan.

1.11. Optimal Top-Down Join Enumeration

By taking existing algorithms for the minimal cut problem and tuning them for the join enumeration context, the Optimal Top-Down Join Enumeration algorithm is the first top-down join enumeration algorithm with space and time complexity that is optimal with respect to the join graph [13]. The algorithm can be easily integrated with branch-and-bound pruning or demand-driven interesting orders.

2. Randomized Algorithms

Due to the inability of the classical deterministic algorithms to optimize large-size queries which become more and more common in contemporary database applications, various non-deterministic approaches have been developed. Different variations of randomized optimization algorithms have been proposed [14].

2.1. Random Walk Algorithm

The simplest randomized algorithm performs a random *walk* in the solution space, starting from a randomly chosen point in it. On each step, a random move is done, if it leads to a point with lower cost [14]. The effectiveness of such a strategy highly depends on the ratio between the „good" and the „bad" solutions in the solution space, as well as on the size of the random sample that is examined. This approach is obviously quite na□ve because only a small neighborhood of the starting point is examined and no attempt is done to search a path approaching an (at least local) optimum [15].

2.2. Iterative Improvement Algorithm

A more sophisticated randomized approach is offered by the Iterative Improvement algorithm [16, 11, 18]. It is a variation of a greedy search strategy similar to the *hill-climbing* algorithm [19]. The difference with the classical hill-climbing is that no attempt is done to find the neighbor with minimum cost because in the general case there are too many neighbors to check. Similarly, instead of checking the cost of all neighbors to determine whether a point is a local optimum, a point is considered a local optimum if no better neighbor could be found for a predefined number of attempts.

2.3. Simulated Annealing Algorithm

The Simulated Annealing algorithm is an improvement to the Iterative Improvement, which allows also moves that lead to points with higher cost than the current point. This lowers the chances of the algorithm to get trapped in a poor local optimum. Moves are accepted with a probability which depends on the cost ratio between the current and the destination point and an algorithm parameter that determines the likelihood for search continuation at a given point of time. Query optimization by Simulated Annealing was proposed in [20].

2.4. Two-Phase Optimization Algorithm

This algorithm is a combination of the Iterative Improvement and the Simulated Annealing algorithms which benefits from the advantages of both [18]. The Iterative Improvement, if applied multiple times, can cover a great portion of the solution space, while the Simulated Annealing is very suitable for thorough search of a point neighborhood.

2.5. QuickPick Algorithm

This is a probabilistic bottom-up join ordering technique performing a biased random sampling of the solution space. It uses the following mapping between a join query's predicates and a query plan. For each new predicate a join is added to the tree and, if not present yet, the base relation required. In case both of the predicate's join arguments are already present, i.e. already connected by a previous join, the predicate is added to this very join. At each step, the algorithm randomly chooses a query predicate to add to the partial execution plan. If the cost of the partial plan exceeds the cost of the best plan found so far, the current plan is discarded. This procedure is repeated until some stopping criterion is fulfilled [21].

3. Genetic Algorithms

Designed to mimic the natural evolution process, genetic algorithms nowadays enjoy an increasing popularity and are being applied to various complex optimization problems. As in Nature, where the best fit individuals in a population have greatest survival probability and highest opportunity to have their features inherited by the offspring, genetic algorithms breed and combine solutions to obtain even better ones [22].

The query execution plan can be considered a program in an abstract tree representation which is evaluated bottom up. The relations are the terminals and the joins are the functions in the genetic program. Thus, the query execution plan satisfies the structural requirements of the genetic programming method, which applies the paradigm of search through genetic algorithms, developed in [23, 24]. The input and the output of each join operator in the plan are relations, therefore the closure requirement defined in [25] is satisfied.

Each particular genetic algorithms is a concretization of the canonical genetic optimization algorithm, that is, different genetic algorithms differ from each other in their coding method (converting a solution into an internal representation upon which the genetic operators can be applied) and by the choice of the three genetic operators - selection, mutation and crossover.

Genetic algorithms have been first applied to the JOP in [26] and [27]. The fitness function used requires backward transformation from chromosome to tree representation, which is complex and with high computational cost. The chosen crossover operators have a serious flaw - they disrupt the chromosome structure, transforming two valid parent chromosomes into an invalid one, which then needs to be „repaired" to become a correct solution encoding. Despite these shortcomings, the achieved results are promising.

Later in [28] some of these disadvantages have been overcome. The fitness function is based on the cost of the query execution plan, which is defined as the total execution time from the first retrieval of a relation from the database to the completion of the output generation (the query result). The model also considers multiprocessor environment and implements parallel processing.

Currently, probably the most popular non-experimental genetic SQL query optimizer is the GEQO (GEnetic Query Optimizer) in the Postgres (PostgreSQL) RDBMS. It considers only left-deep solutions, implements an Elitist selection operator, a simple edge recombination crossover and does not apply mutation. The population size is fixed. Postgres has two optimizer implementations, a classical deterministic optimizer and a genetic optimizer, the latter being used for queries with more than 10 joins. A genetic query optimizer was also introduced in the Microsoft SQL Server 2005.

3.1. Coding

The coding methods for the JOP can be classified by the form of the trees they operate on - left-deep or bushy. The choice of coding strongly influences the choice of the three genetic operators. Left-deep codings are prevalent with genetic algorithms, mostly because they allow for simpler and more efficient mutation and crossover implementations. Note that the solution space must be closed under the genetic operators, which means mutation and crossover should produce valid solutions only with respect to the selected coding.

3.1.1. Simple Left-Deep Tree Coding

Each left-deep tree can be represented in a unique way as an ordered sequence of its leaves:

$$((((R_1 \bowtie R_5) \bowtie R_3) \bowtie R_4) \bowtie R_2) \rightarrow 15342$$

This is probably the most popular coding – it has been used in numerous genetic algorithm implementations,

including PostgreSQL's GEQO [29]. Two of its recent applications to the JOP can be found in [30] and [31].

3.1.2. Traveling Salesman Coding

An ordered list of all relations participating in the join is created. The solution is scanned from left to right, each relation is substituted by its index in the ordered list and then it is removed from the list. Such a coding has been successfully used in the solving of the TSP with genetic algorithms.

$$L = [1, 2, 3, 4, 5], ((((R_1 \bowtie R_5) \bowtie R_3) \bowtie R_4) \bowtie R_2) \rightarrow 14221$$

3.1.3. Bushy Tree Coding

The coding of bushy trees is less straightforward. It must be designed so as not to overly complicate the implementation of the crossover and mutation operators. A good coding algorithm has been proposed in [26], where the symbols in the code represent the edges of the query graph. One useful property of this coding is that it cannot represent Cartesian products, which means that the application of any crossover and mutation operators cannot lead to a Cartesian product in the execution plan.

3.2. Selection

3.2.1. Roulette Selection

Each individual in the population corresponds to a disc sector whose area is inverse-proportional to its cost. The disc can be thought of as a roulette. N turns of the roulette determine the N individuals of the new generation. This algorithm considers the relative fitness of the individuals in the population, which means that a „super" individual may cause the early extinction of other individuals. The classical Roulette selection is thus characterized by fast convergence.

3.2.2. Magnitude Roulette Selection

This is a variation of the classical Roulette selection, in which the disc areas are determined not by the fitness of the individuals, but by the magnitude of their fitness. Experiments show that the Magnitude Roulette selection is characterized by slower evolution progress but the risk of premature convergence is much lower [32].

3.2.3. Rank Selection

In the domain of query optimization, the fitness of the individuals in a population may vary by 10^{100}. In selection algorithms based on the relative fitness of the individuals, some will have no chances to survive while others will quickly dominate the population, leading to quick convergence. The rank selection algorithm assigns ranks from N for the best-fit individual to 1 for the worst-fits individual in a population with size N. Then each individual with rank R survives with probability $R / ((N + 1) * N / 2)$. This selection scheme was suggested in [30].

3.2.4. Elitist Selection

The individuals in the population are sorted in decreasing order of their fitness and the first N individuals are preserved in the new generation. This is the most popular selection operator and it is characterized by relatively fast convergence. It is the selection algorithm implemented in GEQO [29].

3.2.5. Adaptive Selection

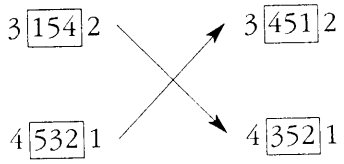
Self-adaptation in genetic algorithms (population size adaptation in particular) is a topic that is receiving considerable attention recently. The classical selection algorithms keep the population size fixed. This simplifies the algorithms but it is an artificial restriction and does not follow any analogy to biological evolution, where the number of individuals in a population varies continuously in time, increasing when there are high-fit individuals and abundant resources and decreasing otherwise. Intuition hints that it may be beneficial for the population to expand in the early generations when there is high phenotype diversity and there is opportunity to „experiment" with different characteristics of the individuals, and to shrink with the increase of population convergence, when the unification of the individuals in terms of structure and fitness no longer justifies the maintenance of a large population and the higher computational costs associated with it.

An adaptive selection operator with dynamic population size has been recently applied to the JOP in [31] and its comparison against the classical fixed-size Elitist selection seem promising.

3.3. Crossover

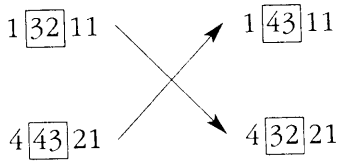
3.3.1. Subsequence Exchange Crossover I

This crossover algorithm is applicable to the Simple Left-Deep Tree Coding and the Bushy TreeCoding. A random subsequence of the code characters of both parents is chosen. Then the subsequence is substituted by another one containing the same characters but arranged in the order of their occurrence in the other parent [7].



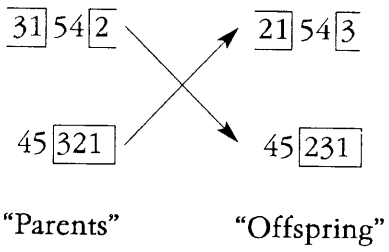
3.3.2. Subsequence Exchange Crossover II

This crossover algorithm is applicable to the Traveling Salesman coding. Two random subsequences with equal length are chosen in both parents. Then the two subsequences are exchanged to form the offspring. This coding is applicable only to the Traveling Salesman coding because it allows duplicate symbols in the code [7].



3.3.3. Subsets Exchange Crossover

This crossover algorithm is applicable to the Simple Left-Deep Tree Coding and the Bushy Tree Coding. Two random subsets of characters with equal cardinality are chosen in the codes of both parents such that the two subsets contain the same elements. Then these subsets are exchanged to form the offspring [7].



3.3.4. Order Crossover

Offspring are generated by choosing two random split points in the parent chromosomes, inheriting a gene subsequence from one of the parents and filling up the missing genes in the relative order they occur in the second parent [30].

(1 3 | 5 7 9 10 | 2 8 6 4)→(7 10 | 2 1 6 9 | 8 4 3 5)
(3 8 | 2 1 6 9 | 4 10 7 5)→(1 6|5 7 9 10 | 4 3 8 2)

A variant of this algorithm using a single split point is used in [31].

3.3.5. Modified Two Swap (M2S) Crossover

This is one of the two crossover operators proposed in the first work to apply genetic optimization to the JOP [26]. Two genes are randomly chosen in the first parent and are replaced by the corresponding genes from the second parent, preserving their order in the second parent.

(**1** 3 2 **4** 6 5) → (**4** 3 2 **1** 6 5)
(2 **3** 4 6 5 **1**) → (2 **1** 4 6 5 **3**)

3.3.6. CHUNK Crossover

This is the second crossover operator proposed in [30] for bushy encoding. A random chunk of genes in the first parent is chosen, the chunk is copied into the offspring (in the same position it occurs in the parent) and the rest of the genes in the offspring are filled up in the order they occur in the second parent.

A good algorithm for bushy tree crossover generating valid offspring solutions and ensuring that most of the structural characteristics of the parent trees are inherited by their successors is presented in [28].

3.4. Mutation

3.4.1. Reciprocal Exchange Mutation

The genes in (random) position i and $(i+1)$ mod N, where N is the length of the chromosome, are swapped [30]. Applied to the simple left-deep tree coding, this mutation operator obtains a new chromosome that is a valid solution.

3.4.2. Exhaustive Mutation

With left-deep codings where a gene contains information about a relation and a join method, the following simple mutation can be considered: two random genes i and j are swapped and the join method of another randomly selected gene k is modified [31]. Such a mutation operator, even if applied alone, guarantees that every point in the solution space is reachable for any choice of the starting point. This is a useful characteristic of genetic operators since it is a prerequisite for the convergence of the genetic algorithm.

4. Hybrid Algorithms

A number of hybrid optimization algorithms have been suggested and comparative experiments have been done with them [11].

4.1. Toured Simulated Annealing

This is an algorithm similar to the Two-Phase Optimization proposed by Lanzelotte et al. [34] in the context of distributed databases. Several *tours* in the solution space are traversed via the Simulated Annealing algorithm, each tour starting from a different initial point. The initial points are obtained by some deterministic greedy algorithm which builds solutions using some augmentation-type heuristic (e.g. minimum selectivity).

4.2. AB Algorithm

This is an evolution of the KBZ algorithm [35]. It allows the use of two join methods - sort-merge and nested loop. The sort-merge cost model has been simplified in order to satisfy the constraints of the KBZ algorithm. The AB algorithm includes both heuristic and randomized elements. The inner loop searches heuristically for a local minimum, while in the external loop several starting points are randomly generated using an idea similar to the Iterative Improvement.

4.3. Improved A* Algorithm

The Improved A* algorithm is a recent improvement to the classical deterministic A* algorithm which takes advantage of the fact that the original A* builds a list of promising nodes, i.e. nodes that are probably part of the best path [36]. The Improved A* algorithm uses this additional information by examining, with some probability, the list of promising nodes, running the original A* using some of these nodes as a starting point.

Discussion

As the review suggests, there is a considerable number of exact and approximate optimization algorithms for the JOP. The different algorithms exploit different characteristics of the problem instances and perform better for different forms and sizes of the solution space, cost functions, join methods, etc.

The No Free Lunch (NFL) theorem [37] states that all algorithms searching for an extremum of a cost function perform exactly the same over all possible cost functions. The NFL suggests that there are classes of problems and for each class there exists an algorithm that solves the problems of that class

most efficiently. Applying the NFL to the JOP, we can conclude that there is no „best" algorithm but that it is necessary to study the relationships between the spaces of problems and the spaces of algorithms.

The major aspects that have to be taken into account comparing the different JOP algorithms are their applicability and restrictions, the impact on performance of query size, the connectedness of the join graph, and the cost model used.

For small queries (ones with up to 8-10 joins), the size of the solution space allows the application of both deterministic and non-deterministic algorithms. The main advantage of deterministic algorithms is that they are guaranteed to find the global optimum.

A good study of the relative performance of the deterministic heuristic algorithms against different join graph topologies and cost models is presented in [7].

The performance of the Minimum Selectivity heuristic is relatively good for low connectivities of the join graph and for the nested loop cost model. For higher connectivities of the join graph and for the asymmetric hash loop cost model, the heuristic performs poorly.

The Relational Difference Calculus has very good overall performance, the heuristic yielding particularly good results for the asymmetric hash loop cost model even for high connectivities.

The Top-Down heuristic has very good performance for almost all join graph / cost model combinations (the exclusion being clique join graph / nested loop cost model) and it is usually able to find near-optimal left-deep solutions.

As a summary, the Relational Difference Calculus and Top-Down are the best option, however for highly connected join graphs both have disappointing performance - Top-Down with the nested loop and Relational Difference Calculus with the hash loop. These findings suggest that better performance can be expected by combining different heuristics, adapting the optimizer to the shape of the join graph and the join method used [7].

The A* algorithm, which can be viewed as a successor of the classical Dynamic Programming, generates a complete execution plan at a much earlier stage than the Dynamic Programming and it prunes suboptimal solutions more aggressively. For small queries, A* has very good performance.

The IK algorithm takes advantage of the special form of the nested-loop cost function and optimizes a query with N joins with time complexity $O(N^2 \log_2 N)$. Ibaraki and Kameda also propose an algorithm which is applicable even to cyclic queries and finds a good (although not always optimal) solution with time complexity $O(N^3)$.

The KBZ algorithm uses basically the same techniques but is more general and more complex and has time complexity $O(N^2)$ for tree queries, where it directly constructs the optimal left-deep solution [9]. For cyclic graphs, the minimum spanning tree has to be computed first. As with the IK algorithm, the applicability of the KBZ algorithm depends on the form of the cost function - the nested loop and hash loop cost functions satisfy the constraints but, in general, the sort-merge does not. KBZ has good performance on low-connectivity join graphs and very poor performance for the completely connected clique graph.

The AB algorithm mixes deterministic and randomized techniques and has time complexity $O(N^4)$ [35]. It utilizes the KBZ algorithm as a subroutine with complexity $O(N^2)$ and runs it $O(N^2)$ times over randomly chosen spanning trees of the query graph. Thanks to an interesting separation of the sort-merge cost function into a part that affects the optimization and a part that does not, the AB algorithm is applicable to all join methods despite the limitations of the KBZ algorithm.

With finite run time, the effectiveness of randomized algorithms depends on the characteristics of the cost function and the connectedness of the query graph. These results have been studied in detail, compared against each other as well as against the results of Dynamic Programming [11, 18]. The results from comparing the relative performance of the Iterative Improvement and the Simulated Annealing are pretty controversial - some authors [16] suggest the Iterative Improvement is superior for non-recursive large join queries, while others [18] (later backed by [7]) show the opposite, that the Simulated Annealing almost always outperforms the Iterative Improvement.

Among randomized algorithms, the Iterative Improvement finds a reasonably good solution for a very short time, while the Simulated Annealing takes more time but is able to find better solutions. The Two-Phase Optimization benefits from the advantages of both approaches and finds the best results for the shortest time [38].

In contrast to transformation-based algorithms such as the Iterative Improvement and Simulated Annealing which traverse the solution space state by state, the QuickPick algorithm converges quicker and delivers more stable results. In addition, transformation-based algorithms depend to a certain degree on the quality of the starting solution which affects the stability of the results obtained and requires careful parameter tuning: if convergence is too fast, the algorithms may get prematurely stuck in a poor local optimum [21]. Algorithms like the Toured Simulated Annealing and the Two-Phase Optimization have been developed to address this issue.

The comparative experiments of randomized and genetic algorithms have been mostly limited to empirical evaluations over particular test problems. Some research indicates that genetic algorithms perform better than the Simulated Annealing, which on its turn outperforms the Iterative Improvement. The comparison in run time is also in favor of genetic (versus randomized) algorithms [7]. Results from performance experiments comparing two genetic algorithms against the simplest randomized one, the Random Walk, show that for small queries the two genetic algorithms have a well-pronounced superiority, while for large queries the Random Walk has performance comparable to that of the two genetic strategies [31]. In general, the results vary considerably, which shows that one of the two approaches may domineer over the other in some class of problems while in another class of problems the results can be just the opposite.

The theoretical comparison between the Simulated Annealing and the genetic algorithms [39] shows that many genetic algorithms (the canonical genetic algorithm in particular) are characterized by better probability of finding a good solution than the Simulated Annealing, provided that the solution space satisfies certain constraints. These constraints however are weak and hold true for almost any choice of the genetic operators.

An important aspect in favor of randomized and genetic

algorithms is that they are easily hybridized, incorporating domain-specific knowledge. This takes them from black-box to problem-aware optimization, which is expected to be beneficial for performance as hinted by the NFL theorem. Besides, from design and implementation perspective, both randomized and genetic algorithms can be easily adapted to take advantage of parallel computer architectures.

The precision of the suboptimal solutions found by randomized and genetic algorithms (i.e. their proximity to the global optimum) is proportional to the number of solutions the algorithm has considered (which in turn is proportional to its running time) and is strongly influenced by the properties of the solution space for the particular problem instance. The solution quality also depends on the convergence characteristics of the algorithms and their ability to escape poor suboptimal plateaus in the solution space. Studying the correlation between the properties of the solution space and the hardness of the JOP instance for a particular class of optimization algorithms is currently in its infancy - the knowledge is still limited and mostly based on empirical results for small queries (up to 10 joined relations). Still there are no analytical tools or respective theory that could help answer the question what will the performance (e.g. deviation from the global optimum as a function of the number of solutions explored) of a particular algorithm be for a fixed JOP instance.

One clear factor influencing the effectiveness of non-deterministic search algorithms is the shape of the solution space, the cost distribution of the local minima and the ratio between the „good" and „bad" solutions. For example, search spaces with the form of a *well* [2] are easier for randomized and genetic algorithms and good-quality solutions can be expected. On the other hand, solution spaces that are not smooth and have great deviations of the cost function present a considerably greater challenge.

Empirical results evaluating the cost distribution for the spaces of left-deep and bushy trees show that, for the nested-loop cost model, about 10% of the left-deep solutions are no worse than twice (2^1) the global minimum, and 45% are better or equal to sixteen (2^4) times the global minimum. On the other hand, the „quality ratio" in the space of the bushy trees is much lower - only about 5% of the solutions have costs less than two times the global minimum, and 25% are better or equal than 16 times the global minimum [27].

A comprehensive experimental study on the influence of connectivity of the query graph, predicate selectivities and relation sizes on the shape of the search space and the performance of heuristics and probabilistic optimization algorithms is presented in [33]. It is concluded that the shape of the search space is clearly determined by these parameters and that the performance of the probabilistic optimization algorithms is directly tied to the shape of the search space. Heuristics, on the other hand, exhibit unstable performance which is only partially dependent on the form of the search space.

[2] Such spaces have two characteristics - the cost difference of any two local minima is small and there exists a path between any two local minima whose elements have costs similar to that of the minima it connects (i.e. the area of the low-cost solutions is smooth) [17].

Conclusion

Finding the optimal join ordering for a database query is a complex combinatorial optimization problem, which, in its general form, is $\mathcal{NP}$ -complete. The algorithms proposed for the JOP can be classified into four basic types - deterministic, randomized, genetic and hybrid. The applicability and effectiveness of the four classes of algorithms depend mostly on the query size, the cost function and the properties of the solution space it defines.

For small queries (with up to 10 joins) deterministic algorithms are the strongest contender, mostly because they are guaranteed to find the global optimum. For larger queries however, the combinatorial explosion makes the exhaustive search of the solution space impossible. Large solution spaces are the domain of non-deterministic algorithms - randomized, genetic and hybrid.

Today's RDBMSs still rely mostly on deterministic query optimization algorithms such as the Dynamic Programming, however, with the advent of some unconventional databases and specific database applications, non-deterministic query optimizers steadily force their way. Now already in its third decade, query optimization continues to be an active field of research.

References

1. Bäck, Th. Optimization by Means of Genetic Algorithms. In Proc. of the 36th International Scientific Colloquium,163-169. Technische Universität Ilmenau, 1991.
2. Yao, X. Global Optimization by Evolutionary Algorithms. The University of New South Wales, 1997.
3. Koza, J. Genetic Evolution and Co-evolution of Computer Programs. In Proc. of 2nd Conference on Artificial Life, 1990.
4. Selinger, P., M. Astrahan, D. Chamberlin, R. Lorie, T. Price. Access Path Selection in a Relational Database Management System. In Proc. of ACM SIGMOD Conference, 1979.
5. Vance, B., D. Maier. Rapid Bushy Join-Orderoptimization with Cartesian Products. In Proc. of ACM SIGMOD,1996.
6. Kossmann, D., K. Stocker. Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. In Proc of ACM, 2000.
7. Steinbrunn, M., G. Moerkotte, A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. In Proc. of VLDB. Springer-Verlag, 1997.
8. Ibaraki, T., T. Kameda. On the Optimal Nesting Order for Computing N-relational Joins. In Proc. of ACM, 1984.
9. Krishnamurthy, R., H, Boral. C. Zaniolo. Optimization of Nonrecursive Queries. In Proc. of the 12th International VLDB Conference. Kyoto Japan, 1986.
10. Schneeweis, W. Boolean Functions with Engineering Applications and Computer Programs., Springer Verlag, 1989.
11. Swami, A. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. – *In Proc. of the ACM SIGMOD*, 18, 1989, 2, 367-376.
12. Yoo, H., Lafortune S. An Intelligent Search Method for Query Optimization by Semijoins. In Proc. of IEEE Transactions on Knowledge and Data Engineering, 1989.
13. DeHaan, D., F. Tompa. Optimal Top-down Jin Enumeration. In Proc. of the ACM SIGMOD, Beijing, China, 785-796, 2007.
14. Galindo-Legaria, C., A. Pellenkoft, M. Kersten. Fast, Randomized

Join-order Selection – Why Use Transformations? In Proc. of Conference on Very Large Data Bases (VLDB), Santiago, Chile, 1994.
15. Duvivier, D., Ph. Preux, E-G. Talbi. Climbing up NP-hard Hills. In Proc. of Parallel Problem Solving from Nature (PPSN IV), 1996.
16. Swami, A., A. Gupta. Optimization of Large Join Queries. In Proc. of ACM SIGMOD International Conference on Management of Data, 8-17, Chicago, Illinois, United States, 1988.
17. Loannidis, Y., Y. Kang. Left-deep Vs. Bushy Trees: an Analysis of Strategy Spaces and its Implications for Query Optimization. In Proc. of ACM SIGMOD Conference on the Management of Data, 1991.
18. Loannidis, Y., Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. In Proc. of ACM, 1990.
19. Yuret, D., M. de la Maza. Dynamic Hill Climbing: Overcoming the Limitations of Optimization Techniques. MIT. In The Second Turkish Symposium on Artificial Intelligence and Neural Networks, 1993.
20. Ioannidis, Y., E. Wong. Query Optimization by Simulated Annealing. In Proc. of the ACM SIGMOD, 9-22, 1987.
21. Waas, F., A. Pellenkoft. Probabilistic Bottom-up Join Ordering Selection - Breaking the Curse of NP-completeness. In Report INS-R9906, ISSN 1386-3681, 15-32, Centrum voor Wiskunde en Informatica, 1999.
22. Khuri, S., Th. Bäck, J. Heitkotter. An Evolutionary Approach to Combinatorial Optimization Problems. In Proc. of ACM, 1994.
23. Holland, J. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.
24. Goldberg, D. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, Mass., 1989.
25. Koza, J. Genetic Programming. The MIT Press, Cambridge, Massachusetts, 1991.
26. Bennett, K., M. Ferris, Y. Ioannidis. A Genetic Algorithm for Database Query Optimization. In Proc. of the 4th International Conference on Genetic Algorithms, 1991.
27. Steinbrunn, M., G. Moerkotte, A. Kemper. Optimizing Join Orders. Technical report MIP-9307, Universität Passau, 1993.
28. Stillger, M., M. Spiliopoulou. Genetic Programming in Database Query Optimization. In Proc. of the 1st Annual Conference on Genetic Programming, 1996.
29. Genetic Query Optimization (GEQO) in PostgreSQL. In PostgreSQL 8.3.3 Documentation, Chapter 49, 2008.
30. Zhou, Z. Using Heuristics and Genetic Algorithms for Large-scale Database Query Optimization.–*Journal of Information and Computing Science*, 2, 2007, No. 4, 261-280.
31. Vellev, St. An Adaptive Genetic Algorithm with Dynamic Population Size for Optimizing Join Queries. In IBS Information Science and Computing, Book 2 (Advanced Research in Artificial Intelligence), 82-88, 2008.
32. Bennett, F., J. Koza, M. Keane, D. Andre. Genetic Programming – Biologically Inspired Computation that Exhibits Creativity in Solving Non-trivial Problems. In Proc. of DIMACS Workshop on Evolution as Computation, 1999.
33. König-Ries, B., S. Helmer, G. Moerkotte. An Experimental Study on the Complexity of Left-deep Join Ordering Problems for Cyclic Queries. In Technical Report 95-4 of the RWTH Aachen, Germany, 1995.
34. Lanzelotte, R., P. Valduriez. M. Zait. On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. In Proc. of the 19th International VLDB Conference, Dublin, Ireland, 1993.
35. Swami, A., B. Iyer. A Polynomial Time Algorithm for Optimizing Join Queries. In Proc. of IEEE International Conference on Data Engineering, Vienna, Austria, 1993.
36. Goyal, A., A. Thakral, G. Sharma. Improved A* Algorithm for Query Optimization. In Proc. of the 20th European Conference on Modeling and Simulation, Bonn, Germany, 2006.
37. Wolpert, D., W. Macready. No Free Lunch Theorems for Optimization. In IEEE Transactions on Evolutionary Computation, 67-82, 1997.
38. Hart, W. A Theoretical Comparison of Evolutionary Algorithms and Simulated Annealing. Sandia National Laboratories. In Proc. of the 5th Annual Conference of Evolutionary Programming, 1996.
39. Haynes, T. A Comparison of Random Search Versus Genetic Programming as Engines for Collective Adaptation, 1997.

Manuscript received on 07.04.2008

Stoyan Vellev *was born in 1976 in Sofia, Bulgaria. He graduated from Sofia University in 2001 with a M.Sc. degree in Computer Science. Currently he is a Ph.D. student at the Faculty of Mathematics and Informatics at Sofia University, working in the area of database query processing. His research interests are in the field of database systems and artificial intelligence - query optimization, information retrieval, declarative languages, computational linguistics, cognitive science, evolutionary computing, etc.*

Contacts:
Sofia University
Faculty of Mathematics and Informatics
Department of Computer Informatics
7 Raiko Alexiev Str. 1113 Sofia Bulgaria
e-mail: stoyan.vellev@sap.com