Abstract

Query optimization is a complex task. It is search for best solution from among the semantically equivalent solutions that can be generated for any given query. It therefore seems logical to consider query optimization in terms of search algorithm. As queries getting more and more complex the search complexity is increasing. Available query optimization techniques are inadequate to support some of emerging database applications. In this day’s genetic algorithm becoming a solid comparator for various algorithms that are used and accepted method for difficult optimization problem. This work is the review of studies widely carried out on the application of genetic algorithm to database query optimization. From studies reviewed it turn out that genetic algorithm are viable alternative to existing query optimizers for optimization of very large queries.

Keywords: Query parser, Query optimizer, Query evaluator, Sailors-Reserves schema, Sailors-Reserve schema, genetic algorithm

1. Introduction

At present we are in the age of information technology, databases have become a necessary and fundamental tool for managing and exploiting the power of information. Because the amount of data in a database grows from larger to largest as time passes, one of the most important characteristics of a database is its ability to maintain a consistent and acceptable level of performance. The principal mechanism through which a database maintains an optimal level of performances is known as the database query optimizer; without a well-designed query optimizer, even small databases would be noticeably sluggish. The query optimizers for some of the most popular commercial-quality databases are estimated to have required about 50 man-years of development. It should therefore go without saying that the specific processes involved in designing the internal structure of a real-world optimizer can be overwhelmingly complex. Nevertheless, because of the optimizer's paramount importance to the robustness and flexibility of a database, it is worthwhile to engage in a survey of the theory behind the rudimentary components of a basic, cost-based query optimizer.

Throughout this paper, the canonical Sailors-Reserves schema will be used to provide concrete examples of how an optimizer generates, evaluates, and selects query evaluation plans. This schema models the structure of the data kept at a hypothetical watercraft rental service that allows sailors with various attributes to make reservations for boats on different days. The specific instance of the Sailors-Reserves schema that will be used here is defined as follows:

Sailors(sid: integer, sname: string, rating: integer, age: real)

Reserves(sid: integer, bid: integer, day: date, rname: string)

2. Query Processing

Whenever a SQL query is issued, the query is first parsed and then presented to the database’s query optimizer before being executed. In order to visualize what the main components of a database query optimizer are and how these components interact in order to produce a query plan that is ready for evaluation, it may be helpful to consider the following figure:
This paper will focus specifically on some of the details involved in generating query evaluation plans and in estimating the costs of such plans. Before proceeding any further, let us define exactly what constitutes a query evaluation plan: in general, a query evaluation plan is a tree with relational operators at the intermediate nodes and relations at the leaf nodes. In broad terms, the purpose of a database query optimizer is to find a good evaluation plan for any given query. Typically, an optimizer will consider only a subset of all the possible plans because the number of possible plans can be extremely large. Thus, considering each plan in turn and executing the most optimal one would actually be more time consuming than considering only one plan and executing it—even if the plan was exceedingly sub-optimal. As a matter of practice, then, many query optimizers are designed simply to avoid the poorest of evaluation plans. Although the actual implementation of a database's query optimizer varies from system to system, in theory, optimizing a SQL query involves three basic steps. First, the SQL must be rewritten in terms of relational algebra. Specifically, the query is treated as a collection of projections ($p$), selections ($s$), and Cartesian products ($x$), with any remaining operators carried out on the result of the given $p$-$s$-$x$ expression. Next, once such a relational algebra expression has been formed, the optimizer must enumerate various alternative plans for evaluating the expressions. Again, a typical optimizer does not consider every possible evaluation plan of a given expression since this would require excessive overhead, thereby rendering any possible timesaving optimizations moot. Finally, the optimizer must estimate the cost associated with each of the enumerated plans, and choose the plan with the lowest estimated cost (or at least avoid the plans with the highest estimated costs) [2] [8]. Now let us consider the query optimization process with a concrete example of a SQL query example using our Sailors-Reserves schema:

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100
AND S.rating > 5
```

We can represent the previous query in relational algebra as follows:

$p\{sname\} \{ bid=100 \wedge rating>5 \} (Reserves\{join\{sid=sid\}\}Sailors)$

This expression is straightforward to evaluate: first we join the Reserves and Sailors relations along the sid attribute, then we select only those tuples that satisfy the conditions that $bid=100$ and $rating>5$, and finally we project the sname attribute of the resulting tuples. In terms of an extended relational algebra tree, the above expression evaluates to:
3. QUERY OPTIMIZER IMPLEMENTATION

Technically, in order to obtain a fully specified evaluation plan, the optimizer must also decide on an implementation method for each of the algebraic expressions in the tree. The implementation of such algebraic expressions as well as the implementation of the underlying relational operators is a lengthy and complex topic that is outside the scope of this survey. Furthermore, although the process of translating SQL queries into relational algebra expressions has received a great deal of scrutiny from researchers, for the sake of brevity we decide to focus this survey on examining some of the more basic procedures used in generating alternative query evaluation plans and in assigning an estimated cost to these plans once they have been generated. Because they cannot possibly generate and exhaustively evaluate every semantically equivalent expression tree for any given plan, many optimizers are designed to follow a set of heuristics that help them to generate plans that are likely to be fairly low in cost, thereby avoiding the lurking possibility of choosing an especially costly plan later on. Although this will be the approach that we follow in this survey, it should be noted that not all optimizers function in this manner. Especially when the given query is relatively simple or small, some optimizers do actually generate every possible query evaluation plan and assign each of these plans a cost. The inferior plans are then eliminated based on the value of this estimated cost. The drawback to this method, of course, is that if a query is very complex, it could require a significant amount of time to generate and assign cost to every possible plan. To put it differently, why spend time enumerating and assigning cost to poor plans that are just going to be eliminated anyway? Thus, for a general-purpose query optimizer, it is a good idea to develop a set of heuristics that will allow the optimizer to avoid enumerating plans that are likely to be inferior.

One of the most basic considerations that an optimizer must be aware of when enumerating alternative plans is whether or not pipelining is present. Pipelining is where the result of one operator is immediately directed to another operator without creating a temporary relation to hold the intermediate results. If, on the other hand, the result of one operator is saved in a temporary relation to be referenced by a later operator, we say that the output is materialized. Thus, in Figure 2, the output of the join between Sailors and Reserves is pipelined into the operators that follow, and no materializing occurs. Pipelining the output of one operator into the next saves the cost of creating and populating a relation to carry the intermediate result. Frequently, the savings created by pipelining are significant; especially when the plan involves evaluating multiple, large joins. Thus in general, pipelined evaluation has lower overhead costs than materialization, so only plans that take advantage of pipelining should be generated [6][7].

Another important guideline that an optimizer must follow involves the order in which selections and joins are performed. Because the join operator potentially involves iterating one-by-one through each element in two different sets of tuples, joining two relations can be a relatively costly operation. Therefore, a good heuristic is to reduce as much as possible the sizes of the relations that are to be joined. One approach is called pushing selections—that is, executing any applicable selections on the relations as soon as possible so that there will be fewer tuples in the result set. As an example, consider the following query evaluation plan:

![Figure 3 Query Evaluation plan](image)

Note that this plan is semantically equivalent to the plan in Figure 2. In terms of minimizing cost, on the other hand, Figure 3 is superior since we perform our selections on the Sailors and Reserves relations before they are joined. Thus, in most cases, plans in which selections (and projections--indeed any operator that reduces the number of tuples in a relation) are pushed ahead of joins should be constructed whenever possible.

As a final heuristic, an optimizer must consider whether indexes are available for any fields in the given relations of the query, and if so, how these indexes should interact with the heuristics of pipelining and pushing selections [4][6][7]. For example, suppose that there exists an index on the bid attribute of the Reserves relation, and an index on the sid attribute of the Sailors relation. Given these two indexes, consider the following query valuation plan:
Here, we are able to perform the selection \( bid=100 \) using the index on \( bid \) to retrieve only those tuples that match the given criterion. For each selected tuple, we next retrieve the matching tuples from the \( Sailors \) relation using the index on \( sid \). As Figure 4 illustrates, the selected \( Reserves \) tuples are not materialized, and the join is pipelined. For each tuple resulting from the join, we perform the selection using \( rating > 5 \) and then pipeline the result set into the projection. Perhaps the most noticeable aspect of this plan is that the selection \( rating > 5 \) is not pushed ahead of the join--a clear violation of one of our basic heuristics. The reason for this violation is that since there is no given index on the rating attribute of \( Sailors \), if the selection were performed before the join, the selection would involve scanning through every tuple of the \( Sailors \) relation. Making matters worse, once the selection has been performed, we no longer have an index on the \( sid \) field of the result of the selection. This, in turn, would increase the cost of the subsequent join. Thus, pushing selections ahead of joins is a good general heuristic, but it is not always the best approach; in most cases, the existence of a useful index on an attribute takes precedence over pushing selections.

3.1 Heuristic optimizer cost estimation

Now we have investigated three of the basic heuristics that optimizers use in order to enumerate plans that are likely not to be exorbitantly costly, it is worthwhile to explore how an optimizer estimates the cost of each plan. The term \textit{cost} is very broad, and the specific formula used to estimate the cost of a plan varies from system to system. However, for the purposes of this survey, we will use a cost model that incorporates the amount of disk I/O (usually measured as the number of memory pages that must be fetched from the disk) and the amount of time used by the CPU while performing the various necessary calculations (such as comparing two pieces of data or copying data from one location into another). Of the two factors that comprise our cost model, the amount of page I/Os is by far the most significant. Because executing page I/Os involves the physical movements of the mechanical read-write head around the disk (thereby incurring rotational latency and seek time), even a small number of page I/Os can take a relatively long time, making the cost estimate very large. Nonetheless, for the sake of accuracy, page I/Os and CPU usage both are usually taken into account when estimating the cost of a plan. Given this cost model above, for an optimizer to assign a cost to a plan, it must be able to make two estimations for each part of the relational algebra expression tree. First, for each node in the tree, the optimizer must estimate the size of performing the corresponding relational operation. The size of the operation is determined most notably by whether pipelining is used or whether temporary relations must be created to pass the output of an operator to its parent. In addition, for each node in the tree, the optimizer must estimate the size and sort order are essential since they will in turn affect the estimation of the size, sort order, and cost for the parent. To accurately perform both of these important estimations, the optimizer must have an intimate knowledge of the various parameters of the input relations--knowledge such as the number of memory pages required for a relation, or the availability of indexes on certain fields. Fortunately, (for most production-level databases at least) such metadata is readily available since it is maintained in the database's system catalogs. However, for this reason it can be difficult to give a general formula that calculates cost because the basis of any given cost estimation depends on what statistics are actually stored in the system catalogs--and the contents of a database's catalogs varies depending on the system. As an example of how the contents of the system catalogs can affect how the cost model is applied, let us examine how a typical optimizer estimates the size of the result computed by an operator on a given input. Consider a basic query of the following form:

\[
\text{SELECT} \text{attri} \text{FROM} \text{relation list WHERE clause}
\]
The maximum number of tuples in the result of the query—without taking into account the possible elimination of duplicate rows—is the product of the cardinalities of all the relations listed in the FROM clause. Clearly, this maximum result can be potentially very large; fortunately, we know that every term in the WHERE clause will eliminate some of the potential result tuples. The effect of the WHERE clause on the result size can be modeled by associating a reduction factor to each term in the clause. A term's reduction factor is simply the ratio of the expected result size to the input size (considering, of course, only the selection represented by the term in question). The actual size of the result can then be estimated by multiplying the maximum result size times the product of the reduction factors for each term in the WHERE clause. Depending on the format of the terms, different equations can be used to estimate the particular reduction factor. All of these equations, however, make assumptions about the presence of indexes on at least one column of the relation referenced by the term under consideration, as well as assumptions that there are uniform and independent distributions of tuples along the index key values. If there is not an index in the relation, many optimizers simply assign the reduction factor to a modest, predefined constant. If, on the other hand, the system catalog happens to maintain statistics such as the number of distinct values present for any attribute even when there is no index on the attribute, then better reduction factors can be obtained. This is one example showing that databases that keep more detailed statistics in their system catalogs are usually able to use more accurate cost estimations. In short, an optimizer is only as good as the plan it eventually decides to evaluate; this plan, in turn, is only as good as the cost model used by the system, and this cost model is itself only as good as the metadata that is kept in the database's system catalogs.

It should be clear from the discussion above that database query optimization—even in a relatively simplified form—is a remarkably complex subject. In abstract terms, however, query optimization is essentially a matter of performing a complex search—that is, a search for the best possible plan among the set of semantically equivalent plans that can be generated for any given query. Conceptually, then, it seems sensible to consider query optimization in terms of the algorithms used to perform various searches. Two of the most common methods for navigating through a search space involve using calculus to make either indirect or direct estimations. Indirect calculus-based methods attempt to detect local extreme by solving a nonlinear set of equations that result from setting the slope of the objective function equal to zero. Direct calculus-based methods, on the other hand, seek local optima by testing the function at an arbitrary initial point and then moving in a direction related to the local slope. This method is also known as hill-climbing—that is, finding the local optimum by climbing the function in the steepest possible direction. Both of these methods are relatively easy to implement and they work extremely well for uni-modal search spaces such as this:

Calculus-based methods are not, however, particularly well suited to multimodal search spaces that consist of several local extrema multiple local extrema are problematic because both direct and indirect search methods may settle into one of the local extrema, totally missing a crucial global extrema that exists elsewhere in the function. It is reasonable to assume that query optimization exhibits a highly multimodal search space, and as such, precludes the use of either direct or indirect calculus-based search methods. Fortunately, there do exist a number of clever, robust search methods that work especially well for complex multimodal search spaces; one such method is known as a genetic algorithm.

### 3.2 Genetic algorithm

Before proceeding any further, it is important to explain exactly what is meant by the term genetic algorithm. Simply put, a genetic algorithm is a search algorithm based on the mechanics of natural selection and genetics [1] [5]. In order to be more effective than calculus-based search methods, genetic algorithms—or GAs for short—must differ in some very fundamental ways. First, GAs work with a special, abstracted encoding of the set of input parameters, not the parameters themselves. Also genetic algorithms search using a large population of points not just a single point. GAs, in addition, uses an objective function to determine payoff information, as opposed to using derivatives or other knowledge auxiliary to the input function. Finally, GAs, use probabilistic transition rules, not deterministic rules. Let us now explore how genetic algorithms work by undertaking a simple searching example: suppose we want to maximize the value of the function \( f(x) = x^2 \) on the interval \([0, 31]\). It should be noted that this problem presents a decidedly unimodal search space and that calculus-based methods would work quite well; however, for the purposes of illustration, we will still use a GA, even though this is the algorithmic equivalent of using a flame-thrower to kill a mosquito. To begin implementing our genetic algorithm, we first require the parameter set of the problem to be encoded as a finite-length string over some finite alphabet. Canonically, a string of bits is used for such a representation, and this will work well for our example. Thus, our representation should be the set of all possible strings of length five that contain any permutation of ones and zeroes. Converting these binary digits into decimal form shows that this representation is perfectly suited to our problem. For example, the string 00000 corresponds to 0, the lowest value in our input parameter set, while the string 11111 corresponds to \( 1(2^4) + 1(2^3) + 1(2^2) + 1(2^1) + 1(2^0) = 31 \), the largest value in our input set. Continuing, we now select at random some arbitrary number of the possible strings; these selected strings will become our initial population. For our example, we have chosen the following initial population:
Next, we must define a set of simple operations that take this initial population and generate successive populations that, hopefully, improve over time. The most basic genetic algorithms that yield good results require at least two and usually three operators: reproduction, crossover, and mutation. Reproduction is the process in which individual strings are copied with respect to their value according to an objective function known as the fitness function. This fitness function is defined so that it represents some measure of profit or effectiveness or goodness that we want to maximize. Copying strings according to this fitness model ensures that strings with higher values have a better probability of contributing one or more offspring to the next generation. This operator is essentially an artificial version of Darwin's concept of natural selection. Referring back to our example, we have a relatively straightforward concept of fitness—the strings with the highest binary value are the most fit in terms of maximizing the value of our input range, so we will make this our fitness function. Deciding which strings should be selected for reproduction involves giving preferential treatment to those strings that responded well to our fitness function. If we imagine selecting the mating pool as spinning a roulette wheel that has been weighted according to the percentages, it is clear that our fittest strings—numbers 2 and 4—have the highest chance of being selected for reproduction. After spinning the wheel four times, suppose that we landed on string number 1’s space once, string number 2’s space twice, string number 3’s space zero times, and string number 4’s space once. Our mating pool would then look like this:

\[
01101, \ 11000, \ 11000, \ 10011
\]

At this point, we need to consider how the crossover operation is used to interact with the strings in our mating pool. In essence, the crossover operator is the artificial equivalent of a parent passing its inherent genetic characteristics onto an offspring during reproduction. In the context of our example, we pair off the strings in our mating pool and choose at random a position in the string at which to exchange the values. Thus, if we chose the cross-site to be between the fourth and last positions of the string, we would have the following:

\[
\begin{align*}
S1 &= 0 \ 1 \ 1 \ 0 \ | \ 0 \ 1 \ 1 \ 0 \\
   &= 1 \ 0 \ 0 \ 0 \ | \ 0 \ 1 \ 1 \ 0 \\
S2 &= 1 \ 1 \ 0 \ 0 \ | \ 1 \ 1 \ 1 \ 0 \\
   &= 1 \ 1 \ 0 \ 0 \ | \ 1 \ 1 \ 0 \ 0 \\
\end{align*}
\]

Finishing the crossover on the other two strings in the mating pool gives us the data about the second generation of our population. Clearly, even after just a single generation, our population has improved in terms of total, average, and maximum fitness, and it does not seem presumptuous to suggest that after several more generations the population can be expected to stabilize and converge around the string 11111, which is the optimal answer for which we were searching.

The mechanisms of reproduction and crossover are surprisingly simple to implement; they involve nothing more than random number generation, string copying, and some partial string exchanging. Nonetheless, the combined emphasis of reproduction and the structured—though randomized—information exchange of crossover give genetic algorithms most of their power. However, if fitness-based reproduction combined with crossover provides the bulk of a genetic algorithms processing power, then what is the purpose of the mutation operator? Many computer scientists (and even biologists) disagree as to the degree of importance of the mutation operator, but classically, it plays a decidedly secondary role in the algorithmic process. Mutation is, in fact, needed because even though reproduction and crossover effectively search through and recombine specific members the population, occasionally they may become overzealous and lose some potentially useful genetic material. In our previous example, this genetic material corresponds to various arrangements of 1’s and 0’s at particular locations. Therefore, in artificial genetic systems, the mutation operator protects against the irrevocable loss of important genetic material that may eventually be needed to lead the algorithm to the globally optimal solution. The mutation operator, in practical terms, is implemented by the occasional random alteration of the value of a string position. In the previous example, this equates simply to occasionally changing a 1 to a 0 or vice versa. Although significant research has gone into abstracting other genetic operators and reproductive schemes from the biological realm, the three operators discussed above—reproduction, crossover, and mutation—have continually proved to be both computationally simple and statistically effective in attacking the majority of important optimization problems [3].

3.3 Specific characteristics and snag of genetic query optimization

3.3.1 Usage of a steady state GA

(Replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time.
3.3.2 Mutation as genetic operator

It is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

4. CONCLUSION

To conclude, we reflect on the fact that the purpose of this survey was twofold. One purpose was to explore the fundamental concepts behind the two topics at hand--database query optimization and genetic algorithms--and then to condense and explain these key concepts. The other purpose was, given this newly acquired information, to look toward the possibility of integrating these two topics into a challenging and substantive capstone project. To this end, it is encouraging to see that these two seemingly disparate topics do indeed have a point of intersection. Specifically, GAs are especially well suited to tackling the inherently complex process of searching for and selecting an optimal plan out of the population of relational algebra trees that are generated by the query optimizer. Henceforth, our focus on query optimization and genetic algorithms must be shifted from information gathering to integration and implementation in the context of a large-scale development project.

We have to find a compromise for the parameter settings to satisfy two competing demands:

- Optimality of the query plan
- Computing time

In the current implementation, the fitness of each candidate join sequence is estimated by running the standard planner's join selection and cost estimation code from scratch. To the extent that different candidates use similar subsequences of joins, a great deal of work will be repeated. This could be made significantly faster by retaining cost estimates for sub-joins. The problem is to avoid expending unreasonable amounts of memory on retaining that state.

At a more basic level, it is not clear that solving query optimization with a GA algorithm designed for TSP is appropriate. In the TSP case, the cost associated with any substring (partial tour) is independent of the rest of the tour, but this is certainly not true for query optimization. Thus it is questionable whether edge recombination crossover is the most effective mutation procedure.

References


[7] Lohman, Guy M, Dean Daniels, Laura M. Haas, Ruth Kistler, Patricia G. Selinger "Optimization of Nested Queries in a Relational Database" IMB Research Laboratory RJ4260, San Jose, Calif, April 1983.