Speeding Up Warehouse Physical Design Using A Randomized Algorithm

Minsoo Lee and Joachim Hammer
Dept. of Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611-6120
{mslee, jhammer}@cise.ufl.edu

Abstract

A data warehouse stores information that is collected from multiple, heterogeneous information sources for the purpose of complex querying and analysis. Information in the warehouse is typically stored in the form of materialized views. One of the most important tasks when designing a warehouse is the selection of materialized views to be maintained in the warehouse. The goal is to select a set of views in such a way as to minimize the total query response time over all queries, given a limited amount of time for maintaining the views (*maintenance-cost view selection problem*). The paper focuses on an efficient solution to the maintenance-cost view selection problem using a genetic algorithm for computing a near-optimal set of views. Specifically, we explore the view selection problem in the context of OR view graphs. We show that our approach represents a dramatic improvement in time complexity over existing search-based approaches using heuristics. Our analysis shows that the algorithm consistently yields a solution that lies within 10% of the optimal query benefit while at the same time exhibiting only a linear increase in execution time. We have implemented a prototype version of our algorithm which is used to simulate the measurements used in the analysis of our approach.

Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'99)
Heidelberg, Germany, 14. - 15. 6. 1999
(S. Gatziu, M. Jeusfeld, M. Staudt, Y. Vassiliou, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-19/

1 Introduction

A *data warehouse* stores information that is collected from multiple, heterogeneous information sources for the purpose of complex querying and analysis [IK93, Wid95]. The information in the warehouse is typically processed and integrated before it is loaded in order to detect and resolve any inconsistencies and discrepancies among related data items from different sources. Since the amount of information in a data warehouse tends to be large and queries may involve hundreds of complex aggregates at a time, the organization of the data warehouse becomes a critical factor in supporting efficient online analytical query processing (OLAP) as well as in allowing periodic maintenance of the warehouse contents. Data in the warehouse is often organized in summary tables, or *materialized views* [Rou97], which represent pre-computed portions of the most frequently asked queries. In this way, the warehouse query processor avoids having to scan the large data sets for each query, a task that is even more wasteful if the query occurs frequently. However, in order to keep these materialized views consistent with the data at the sources, the views have to be *maintained*. Rather than periodically refreshing the entire view, a process that may be time consuming and wasteful, a view can be maintained in an *incremental* fashion, whereby only the portions of the view which are affected by the changes in the relevant sources are updated [GM95, ZGH+95].
Besides this so-called view maintenance or update cost, each materialized view in the warehouse also requires additional storage space which must be taken into account when deciding which and how many views to materialize. For example, given a set of frequently asked OLAP queries, materializing all possible views will certainly increase query response time but will also raise the update costs for the warehouse and may exceed the available storage capacity. Thus by trading space for time and vice versa, the warehouse administrator must carefully decide on a particular warehouse configuration which balances the three important factors given above: query response time, maintenance cost, and storage space. The problem of selecting a set of materialized views for a particular warehouse configuration which represents a desirable

balance among the three costs is known as the *view selection problem*[1].

In this paper we focus on a solution to the maintenance-cost view selection problem which minimizes query response time given varying upper bounds on the maintenance cost, assuming unlimited amount of storage space because storage space is regarded cheap and not a critical resource. Specifically, we explore the view selection problem in the context of OR view graphs, in which any view can be computed from any of its related views. Although the view selection problem has been addressed previously (e.g., see [Gup97], [GM99], [TS97]), existing algorithms do not perform well when computing warehouse configurations involving more than 20-25 views or so. In those cases, the search space becomes too large for any kind of exhaustive search method and even the best heuristics can only compute acceptable solutions for a small set of special cases of the problem. To this end, we have designed a solution involving randomization techniques which have proven successful in other combinatorial problems. We show that our solution is superior to existing solutions in terms of both its expected run-time behavior as well as the quality of the warehouse configurations found. The analysis proves that our genetic algorithm yields a solution that lies within 90% of the optimal query benefit while at the same time exhibiting only a linear increase in execution time. We expect our algorithm to be useful in data warehouse design; most importantly in those scenarios where the queries which are supported by the existing warehouse views change frequently, making it necessary to reconfigure the warehouse efficiently and quickly. Supporting data warehouse evolution in this way may increase the usefulness of the data warehousing concept even further.

The paper is organized as follows. In the Sec. 2 we present an overview of the related work. Sec. 3 describes our technical approach. Specifically, we briefly introduce the idea behind genetic algorithms (which are a special class of randomized algorithms) and how we are using the technique to find an efficient solution to the view selection problem. In Sec. 4 we describe the implementation of our prototype which was used to generate the simulation runs which we present and analyze in Sec. 5. Sec. 6 concludes the paper with a summary of our results and future plans.

2 Related Research

All of the related work on view selection uses some form of greedy strategy or heuristics-based searching technique to avoid having to exhaustively traverse the solution space in search of the optimal solution. The problem of selecting additional structures for materialization was first studied by Roussopoulos [Rou82] who proposed to materialize view indices rather than the actual views themselves. View indices are similar to views except that instead of storing the tuples in the views directly, each tuple in the view index consists of pointers to the tuples in the base relations that derive the view tuple. The algorithm is based on A* to find an optimal set of view indexes but uses a very simple cost model for updating the view which does not take into account which subviews have been selected. As a result, the maintenance cost for the selected view set is not very realistic.

More recently, Ross et al. [RSS96] and Labio et al. [LQA97] have examined the same problem using exhaustive search algorithms that make use of heuristics for pruning the search space. The work by Labio et al. is an extension of the work by Ross et al. considering indexes and also improving upon the efficiency of the algorithm. In addition, Labio et al. are the first to provide a valuable set of rules and guidelines for choosing a set of views and indexes when their algorithm cannot compute the optimal warehouse configuration within a reasonable time due to the complexity of the solution. Similarly, Theodoratos et al. [TS97] present an exhaustive search algorithm with pruning to find a warehouse configuration for answering a set of queries given unlimited space for storing the views. Their work also focuses on minimizing query evaluation and view maintenance.

[HRU96] present and analyze greedy algorithms for selection of views in the special case of "data cubes" that come within 63% of the optimal configuration. However, their calculations do not figure in the update costs for the selected views.

Our work is most closely related to that of Gupta [GM99] who has used both the greedy approach as well as the A* algorithm for solving the maintenance-cost view selection problem in the context of both OR/AND view graphs and the general case of AND-OR view graphs. His approach also balances query response time and view maintenance cost while assuming an unlimited amount of storage space.

3 Technical Approach

In [Gup97] the view selection problem is stated to be NP-hard (see, for example, [Coo71, GJ79]), as there is a straightforward reduction to the minimum set cover problem. Roughly speaking, it is very difficult to find an optimal solution to problems in this class because of the fact that the solution space grows exponentially as the problem size increases. Although some good solutions for NP-hard problems in general and the view selection problem in specific exist, such approaches encounter significant problems with performance when the problem size grows above a certain limit. More recent approaches use randomized algorithms in solving NP-hard problems. Randomized algorithms are based on statistical concepts where the large search space can be explored randomly using an evaluation function to guide the search process closer to the desired goal. Randomized algorithms can find a reasonable solution within a relatively short period of time by trading executing time for quality. Although the resulting solution is only near-optimal, this reduction is not as drastic as the reduction in execution time. Usually,

[1] Sometimes the problem is also referred to as the *view index selection problem* (VIS) when the solution includes a recommendation on which index structures should be maintained in support of the materialized views.

the solution is within a few percentage points of the optimal solution which makes randomized algorithm an attractive alternative to traditional approaches such as the ones outlined in Sec. 2, for example.

Our approach uses a genetic algorithm, which is one form of a randomized algorithm. The motivation to use genetic algorithms in solving the view selection problem was based on the observation that data warehouses can have a large number of views and the queries that must be supported may change very frequently. Thus, a fast solution is needed to provide new configurations for the data warehouse: an ideal starting point for the genetic algorithm. However, genetic algorithms do not provide a magical solution by themselves and their success (or failure) often depends on the proper problem specification, the set-up of the algorithm, as well as the outcome of the extremely difficult and tedious fine-tuning of the algorithm that must be performed during many test runs. After a brief overview of genetic algorithms, we provide details on how to apply these techniques to design an optimal solution to the view selection problem. Specifically, we elaborate on a suitable representation of the solution space as well as the necessary evaluation functions needed by our genetic algorithm.

3.1 Genetic Algorithms

The idea behind the Genetic Algorithm (GA) [Gol89] comes from imitating how living organisms evolve into superior populations from one generation to the next. The genetic algorithm works as follows. A pool of genomes is initially established. Each genome represents a possible solution for the problem to be solved. This pool of genomes is called a population. The population will undergo changes and create a new population. Each population is referred to as a generation. Starting with an initial generation *t*, the sequence of subsequent populations is referred to as generation *t+1*, *t+2*, ..., and so on. After several generations, it is expected that the population *t+k* should be composed of genomes which are superior to the genomes in population *t*. By superior genomes we mean genomes which represent a solution that is closer to the optimal solution (based on a so-called fitness evaluation).

The Genetic Algorithm repeatedly executes the following four steps:

① t = t +1
② select P(t) from P(t-1)
③ recombine P(t)
④ evaluate P(t)

In step ①, a new generation *t* is created by increasing the generation variable *t* by one. In step ② superior genomes among the previous population *P(t-1)* are selected and used as the basis for composing the genomes in the new population *P(t)*. A statistical method, for example, the *roulette wheel method* [Mic94], is used to select those genomes which are superior.

In step ③, several operations are executed on paired or individual genomes to create new genomes in the population. These operations are called *crossover* and *mutation* operations, respectively. Step ④ evaluates the population that is created. A function called the *fitness function*, which evaluates the superiority of a genome, is used in this process. The fitness of each genome can be gathered and used as a metric to evaluate the improvement made in the new generation. This fitness value is also used during the selection process (in step ②) in the next iteration to select superior genomes for the next population. Also, the genome with the best fitness so far is saved. We now explain how this algorithm can be adapted to solve the view selection problem.

3.2 An Improved Solution to the View Selection Problem

In order to apply a genetic algorithm approach to solve the view selection problem, two requirements must be met: (1) We need to find a string representation of a candidate solution and, (2) we need to be able to define the fitness function, the crossover operator, and the mutation operator as outlined above.

In [Mic94], the authors discuss several solutions to popular problems using genetic algorithms, including the 0/1 knapsack problem (see for example [Aho83]). The similarity of the view selection problem to the 0/1 knapsack problem gives us a hint on how to apply the genetic algorithm strategies in our context. However, to our knowledge nobody has yet to apply genetic algorithm techniques to solving view selection and the solutions presented here represent our own approach.

3.2.1 Problem Specification

The problem to be solved can be stated as follows [GM99]: Given an OR view graph G and a quantity representing the total maintenance time limit, select a set of views that minimizes the total query response time and also does not exceed the total maintenance time limit. An OR view graph is composed of a set of views where each view in the graph can be constructed from other source views in one or more ways, but each derivation involves only one other view. In other words, only one view among the source views is needed to compute a view. An example of an OR view graph is the data cube [GCB$^+$97] where each view can be constructed in many different ways but each derivation only involves one view. A sample OR view graph is shown in Figure 1.

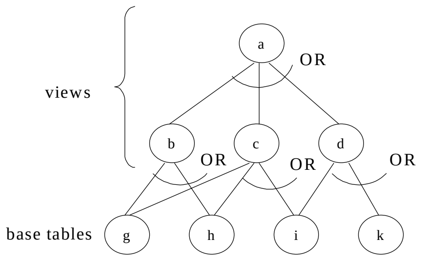

Figure 1: Sample OR view graph for four views.

For example, in this figure, the view labeled a can be computed from any of the views labeled b, c, d (the same is true for the views b, c, d).

As mentioned in the introduction, we are limiting our experiments to OR view graphs at this point since the cost model is considerably less complex than that for the other two view graph models (i.e., AND and AND-OR view graphs). However, despite their simpler cost model, OR view graphs are useful and appear frequently in warehouses used for decision support in the form of data cubes as indicated above.

Future versions of this paper will address the cases of AND view graphs as well as AND-OR view graphs. We are also deferring the problem of index selection for the next version of our algorithm. However, in Sec. 6 we briefly mention how index selection can be added into the problem domain in a straightforward manner.

3.2.2 Problem Solution

Step 1 – Representation of the Solution.

A genome represents a candidate solution of the problem to be solved. A genome can be represented appropriately as a string as follows: Either as a binary string composed of 0s and 1s, or as a string of alphanumeric characters. The content of the string is flexible, but the representation of the solution must be carefully designed so that it is possible to properly represent all possible solutions. As alphanumeric strings are good for representing solutions to ordering problems and binary strings are used for selection problems, we use a binary string to represent a solution. The views in the OR view graph may be enumerated as v_1, v_2, …, v_m where m is the total number of views. We can represent a selection of these views as a binary string of m bits. If the bit in position i (starting from the leftmost bit as position 1) is 1, view v_i is selected. Otherwise, view v_i is not selected. For example, the bit string 001101001 encodes the fact that only views v_3, v_4, v_6 and v_9 are selected.

Step 2 – Initialization of the Population.

The initial population consists of a pool of randomly generated bit strings of size m. In future implementations, however, it is straightforward to start with an initial population which represents a favorable configuration based on external knowledge about the problem and its solution. It will be interesting to see if and how this affects the quality as well as the run-time of our algorithm. For the experiments described in this paper, we have chosen a population size of 30.

Step 3 – Selection, Crossover, Mutation, Termination

The selection process uses the roulette wheel method. The crossover and mutation operators are assigned probabilities p_c and p_m, respectively. The specific values used in our simulation are 0.001 and 0.9. The *crossover* operation is applied to two genomes by exchanging information between the two, thereby creating two new genomes. Crossover works as follows:

```
Each genome is selected with a probability of $p_c$.
Pair genomes.
For each pair, do the following :
   // Assuming two genomes
   // g1 = ($b_1$ $b_2$ ... $b_{pos}$     $b_{pos+1}$ ... $b_m$) and
   // g2 = ($c_1$ $c_2$ ...  $c_{pos}$    $c_{pos+1}$ ... $c_m$)
  (1) Randomly decide a crossover point pos.
  (2) Exchange information among genomes,
      and  replace g1, g2 with g1', g2'
       // (ex)  g1' = ($b_1$ $b_2$ ... $b_{pos}$     $c_{pos+1}$ ... $c_m$)  and
       //         g2' = ($c_1$ $c_2$ ...  $c_{pos}$    $b_{pos+1}$ ... $b_m$)
```

The *mutation* operator works as follows.

```
For all genomes,
    For each bit in genome,
           mutate (flip) bit with probability of pm
```

The selection, crossover, mutation and evaluation (described in Step 4) processes will be repeated in a loop until the termination condition is satisfied. The *termination condition* is reached after 400 generations. The values used for the probabilities and termination condition are based on empirical values used in other examples, and although reference [Mic94] mentioned that no improvement was observed after 500 generations, we have reduced this value to 400 in our experiments since our algorithm converged more rapidly.

Step 4 – Evaluation Process

The *fitness function* measures how good a solution (i.e., a genome) is by providing a fitness value as follows: If the fitness is high, the solution is closer to an optimal solution; if the fitness is low, the solution is far away from the optimal solution. There are many possible fitness functions and finding the best possible one (i.e., one that can truthfully evaluate the quality of a particular warehouse configuration) requires a lot of fine-tuning.

For our problem, the fitness function has to evaluate a genome (i.e., a set of selected views to materialize) with respect to the query benefit as well as with respect to the maintenance constraint. This is similar to the goal of the 0/1 knapsack problem, for example, where the goal is to maximize the profit of the packed load while satisfying a specific capacity constraint of the knapsack. The difference is that in the view selection problem, when a view is selected, the benefit will not only depend on the view itself but also on other views that are selected. A good way to model such a complex problem is by introducing a penalty value as part of the fitness function. This penalty value will reduce the fitness if the maintenance constraint is not satisfied. When the maintenance constraint is satisfied, the penalty value will have no effect and only the query benefit should be evaluated. We have applied the penalty value in three different ways when calculating the fitness: Subtract mode (S), Divide mode (D), and Subtract & Divide mode (SD).

The subtract mode will calculate the fitness by subtracting the penalty value from the query benefit. Because the fitness value is not allowed to have a negative value, the fitness is set to 0 when the result of the calculation becomes negative (i.e., the penalty value exceeds the query benefit). The divide mode will divide the query benefit with the penalty value in an effort to reduce the query benefit. When the penalty value is less than 1, the division is not performed in order to prevent the fitness from increasing. The subtract & divide mode combines the two methods discussed above. If the query benefit is larger than the penalty value, the subtract mode is used. If the penalty value is larger than the query benefit, the divide mode is used. The penalty value can be calculated using a penalty function, which will be discussed afterwards. Thus, we have defined a fitness function, called *Eval,* as follows*:*

Subtract mode (S):

$Eval(x)=B(G,M)-Pen(x)$ (*if* $B(G,M)-Pen(x)\geq 0$) ……①
$\quad =0$ (*if* $B(G,M)-Pen(x)<0$)
$\quad$ *for all* $x[i]=1, \quad vi \in M$
$\quad$ *for all* $x[i]=0, \quad vi \notin M$

Divide mode (D):

$Eval(x)=B(G,M) / Pen(x)$ (*if* $Pen(x)>1$) ……………②
$\quad =B(G,M)$ (*if* $Pen(x)\leq 1$)

Subtract&Divide mode (SD):

$Eval(x)=B(G,M)-Pen(x)$ (*if* $B(G,M)>Pen(x)$) ………③
$\quad =B(G,M)/Pen(x)$ (*if* $Pen(x)\geq B(G,M)$ *and* $Pen(x) >1$)
$\quad =B(G,M)$ (*if* $Pen(x)\geq B(G,M)$ *and* $Pen(x) \leq 1$)

where *B* is the query benefit function, *Pen* is the penalty function, *x* is a genome, *G* is the OR view graph, and *M* is the set of selected views given by *x*.
The penalty function itself can also take on various forms. For example, we have experimented with logarithmic, linear and exponential penalty functions as shown in ④,⑤, and ⑥.

Logarithmic penalty (LG):

$Pen(x) = \log_2 (1 + \rho \; (U(M) - S))$ ………………④

Linear penalty (LN):

$Pen(x) = (1 + \rho \; (U(M) - S))$ ……………………⑤

Exponential penalty (EX):

$Pen(x) = (1 + \rho \; (U(M) - S))^2$ …………………⑥

where ρ is defined as a constant calculated from the given OR-view graph G, *U(M)* is the total maintenance cost of the set of materialized views *M*, and *S* is the maintenance cost constraint.
We combined the three types of penalty modes (i.e., S, D, SD) and the three types of penalty functions (i.e., LG, LN, EX) in our prototype to evaluate and determine the best possible strategy for solving the view selection problem. Please note, the details as well as the formulas for the query benefit function B(G,M), the total maintenance cost U(M), and ρ are provided in a full version of this paper [LH99] which is available for download via ftp and http from our publication server.

4 Prototype Implementation

We used version 2.4.3 of the genetic algorithm toolkit from MIT called *GAlib* [MIT] to develop a prototype of the algorithm described above. The toolkit supports various types of genetic algorithms including mutation and crossover operators, built-in genome types such as 1-dimensional or 2-dimensional strings, and a statistics gathering tool that can provide summarized information about each generation during a single run of the genetic algorithm. The prototype was written entirely in C++ using Microsoft Visual C++ as our development platform. Since the toolkit did not provide any libraries to encode a fitness function based on the evaluation strategies discussed above, we had to encode our own. The fitness function we developed can calculate the fitness in nine different ways by pairing each type of penalty mode with each type of penalty function; in our implementation, we can control the way the penalty is calculated and applied in the fitness function by setting the value of a variable which indicates the desired strategy. This allows us to switch back and forth between the different penalty modes when conducting our experiments. The fitness function needs to evaluate each genome using the cost values given by the OR-view graph and the maintenance cost limit (e.g., given by the warehouse administrator). For this purpose, additional cost functions which, when given a genome can calculate the total query cost and the total maintenance cost of the selected views represented by the genome, must be encoded. The OR-view graph has the related costs shown in Table 1. Each node (=view) in the graph, has associated with it a read cost of the view (RC), a query frequency (QF) and an update frequency (UF). Each edge of the graph, which denotes the relationship among the views, is associated with a query cost (QC) and a maintenance cost (MC).

Table 1: Cost parameters for OR view graphs.

	Para-meter	Description
Node (View)	RC	Read Cost of the view, also used to represent the size of the view.
	QF	Query Frequency, represents the number of queries on the view during a given time interval.
	UF	Update Frequency, represents the number of updates on the view during a given time interval.
Edge	QC	Query Cost, represents the cost for calculating a view from one of its source views.
	MC	Maintenance Cost represents the cost for updating a view using one of its source views.

The total query cost for the selected views represented by a genome is calculated by summing over all calculated minimum cost paths from each selected view to another selected view or a base table. Each minimum cost path is composed of all of the QC values of the edges on the path and the RC values of the final selected view or base table. This calculation is implemented by using a depth-first traversal of the OR view graph.
The total maintenance cost is calculated similarly, but the cost of each minimum cost path is composed of only the UC values of the edges. The detailed formulae and examples are given in [LH99]. An OR-view graph generator, which can randomly generate OR-views based on the density and using the parameter ranges given for each parameter of the graph, was also developed for experimental purpose. In addition, we implemented an exhaustive search algorithm to find the optimal solution (at least for small warehouse configurations) in order to be able to compare the quality of our GA-based solution to the optimal one for each test case.

5 Evaluation of the Algorithm

Our genetic algorithm was developed and evaluated using a Pentium II 450 MHz PC running Windows NT 4.0. We performed two kinds of evaluations. First, the nine strategies for the fitness functions (see Sec. 3.2.2) were compared in terms of the quality of the generated solutions with respect to the optimal solutions. Second, we compared the run-time behavior of the genetic algorithm to the exhaustive search algorithm in order to gain insight into the efficiency of our approach.
The OR-view graphs that were used in the experiments were as follows. The number of base tables was fixed to 10 tables. The number of views varied from 5 to 20 views. The edge density of the graph varied from 15% to 30% to 50% to 75%. The ranges for the values of all the important parameters of the OR-view graphs are shown in Table 2. The maintenance cost constraint for the problem was set to 50, 100, 300, and 500. Please note that one possible interpretation of these values is to view them as time limits on how long the warehouse is expected to be unavailable due to maintenance or as the amount of data that must be read etc.

Table 2: Range of parameter values for the simulated OR-view graphs

	Para-meter	Description
Node (View)	RC	100-10,000 for base tables (RC for views are calculated from source views)
	QF	0.1 - 0.9
	UF	0.1- 0.9
Edge	QC	10 - 80 % of RC of source view
	MC	10 – 150% of QC

5.1 Quality of Solutions

Initially, we used all nine different fitness functions to conduct the experiments. The quality of the solutions was measured as a ratio of the optimal total query cost (obtained using the exhaustive search) over the computed total query cost (obtained using the genetic algorithm). The ratio was computed and averaged over several runs. It was initially expected that the ratio would always be less than 100%. However, we observed that the solutions produced by the genetic algorithm sometimes resulted in a higher than specified maintenance cost but lower than expected overall query cost: in those cases, the total query cost obtained was lower than the query cost obtained by strictly adhering to the maintenance constraint value (i.e., the ratio exceeded 100%). This was very interesting in the sense that although a maintenance cost constraint may be given, it may be beneficial to use it more as a guideline (within certain limits) rather than as a strict policy. Actually, in [GM99] the inverted-tree greedy heuristic also does not guarantee a strict maintenance cost constraint, but satisfies a limit within twice the constraint value.
The nine different strategies used in our initial set of experiments are denoted LG-S, LG-D, LG-SD, LN-S, LN-D, LN-SD, EX-S, EX-D, EX-SD, where LG, LN, EX denote the different penalty functions and S, D, SD denote the different ways of applying those penalty functions (as described in Sec. 3.2.2).
After an initial experiment, the logarithmic penalty functions (LG-S, LG-D, LG-SD) did not perform well, especially LG-S and LG-SD. The reason was that the logarithmic penalty function makes the penalty value too small to have a noticeable effect on the fitness value. Thus, for LG-S and LG-SD, our algorithm always tried to maximize the query benefit while ignoring the maintenance cost constraint by yielding a solution that materializes *all* of the views. LG-D and several others such as LN-S, EX-S did not result in such extreme solutions but tended to fluctuate wildly over the maintenance cost limit, sometimes exceeding it by as much as 10,000%! Therefore, we disregard these strategies in our figures and only show the results from the remaining strategies, namely LN-D, LN-SD, EX-D, EX-SD as depicted in Figures 2 and 3. Figure 2 shows the results of averaging over the ratios of optimal total query cost (based on a strict maintenance constraint) over GA total query costs. Figure 3 shows the results of averaging over the ratios of GA total maintenance cost over the maintenance constraint. The values are arranged in tuples in lexicographical order:

`(density, number of views, maintenance constraint)`.

The density changes occur at the points 1, 65, 129 and 193 on the x-axis, each increasing the densities. The numbers of views are shown in increasing order within a given density. The maintenance cost is also shown in increasing order within each set of views.
The results in Figure 3 show that the LN-D and LN-SD still exhibit a considerable amount of fluctuation (about

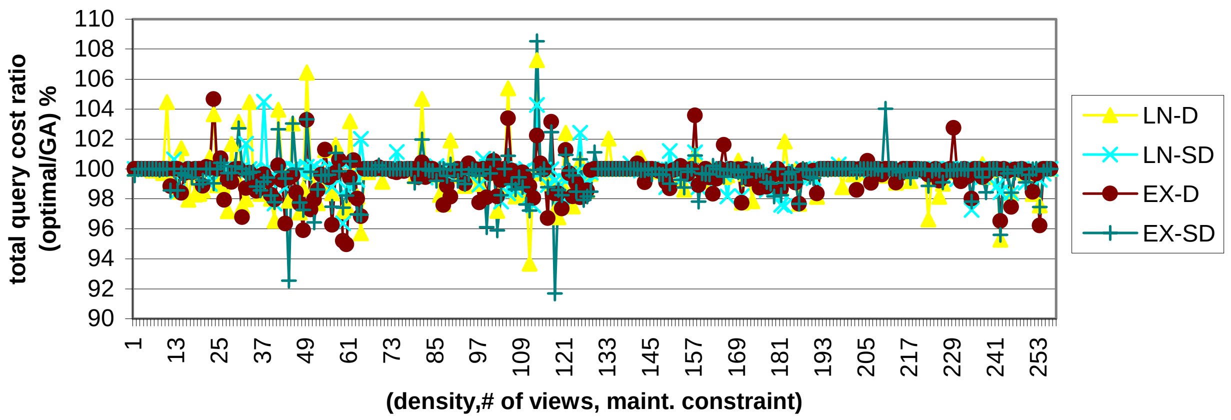

Figure 2: Average ratios of (optimal total query cost/GA total query cost)

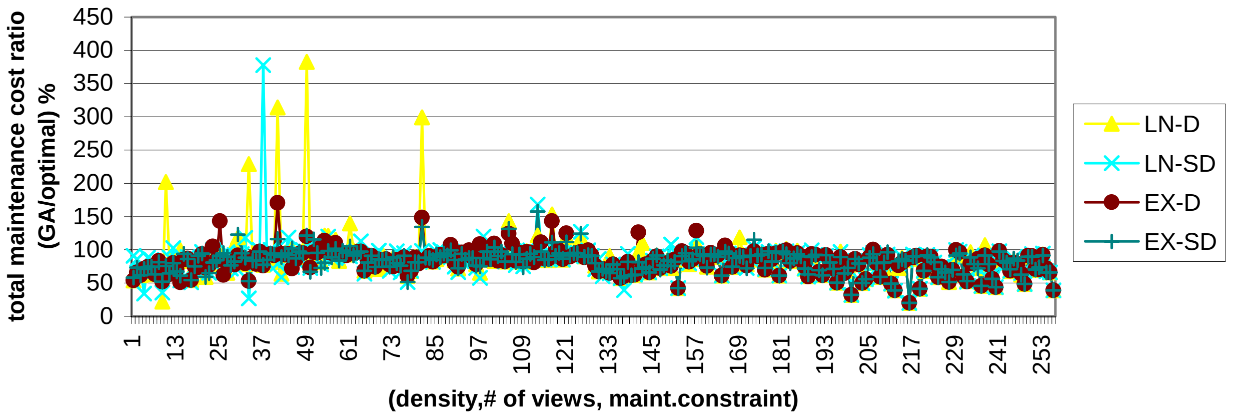

Figure 3: Average ratios of (GA total maintenance cost/maintenance constraint)

380%) for the maintenance cost. This was especially noticeable for low density OR-view graphs where the penalty values resulted in values which were too small to enforce the maintenance constraint. If we discard these two strategies from our consideration, Figure 2 shows that the remaining EX-D and EX-SD strategies obtain a total query cost ratio that is guaranteed to always be over 90% which is very close to the optimal solution. Furthermore, the maintenance cost is always within two times the value of the maintenance cost. Thus, EX-D and EX-SD represent good fitness functions for our genetic algorithm. Note that this result is also very close to the one that was verified in theory in the inverted-tree greedy heuristics proposed by [GM99].

5.2 Execution Time

Figures 4 and 5 show a comparison in execution time between our genetic algorithm and the exhaustive search averaged over the sample OR view graphs. The exhaustive search algorithm was developed to obtain the optimal solutions for comparing the qualities of the solutions, and by using this algorithm, we have actual proof that it is extremely time consuming to obtain an optimal solution when the number of views exceeds 20 views. As a result, we limited our experiments to only 20 views. Although better heuristics exist (which still have polynomial time complexity), this particular experiment is intended to give the reader a feel for the performance capabilities of our genetic algorithm. From the figures we can see that the execution time for the exhaustive algorithm increases exponentially within each given density as it goes up to 20 views. Our genetic algorithm on the other hand exhibits linear behavior. As the density grows, the slope of the linear graph increases only slightly. The genetic algorithm took approximately 1/80th of the time of an exhaustive search of an OR-view graph with density of 75% and 20 views. As the number of views goes up to 30 and beyond, this ratio is expected to be much more impressive. Furthermore, the quality of the solution generated by the genetic algorithm remains very close to the optimal.

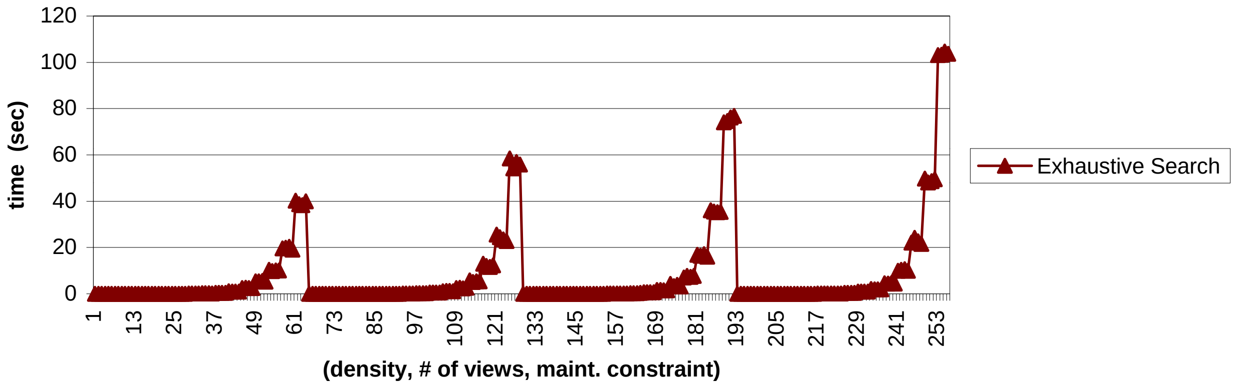

Figure 4: Execution time for exhaustive search algorithm

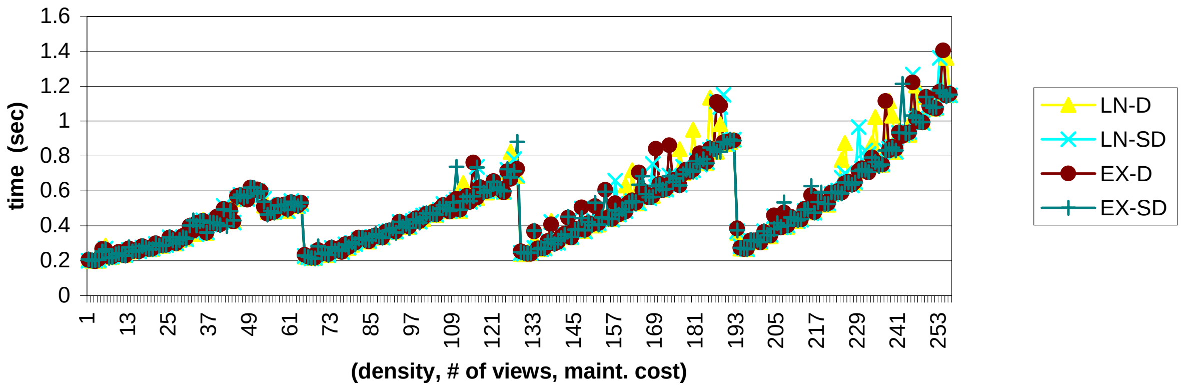

Figure 5: Execution time for genetic algorithm

6 Conclusion

In this paper we have shown that our genetic algorithm is superior to existing solutions to the view selection problem in the context of OR view graphs. Specifically, our genetic algorithm consistently yields a solution that comes within 10% of the optimal solution (which we verified by running an exhaustive search on OR view graphs with up to 20 views) while at the same time exhibiting a linear run-time behavior. A penalty function has been included in the fitness function, and experimental results show that the EX-D and EX-SD penalty functions produce the best results for the maintenance cost view selection problem. We believe that this algorithm can become an invaluable tool for warehouse evolution, especially for those data warehouses with a large number of views and frequent changes to the queries which are supported by the given warehouse configuration.

In the future, we are considering the following improvements:

- Generate an initial population based on knowledge of a possible solution rather than using random configurations.
- Experiment with several other crossover or mutation operators to speed up convergence even further.
- Implement a more flexible termination condition that can interrupt the algorithm when the solution lies with a certain threshold instead of always computing all generations.
- Expanding our approach to include AND-OR view graphs as well as indexes. The first is straightforward. The latter is more complicated as we have to modify the problem representation. One possible approach may be as follows: Add the indexes related to a view immediately after the bit position of the view. However, crossover and mutation operations need to be carefully redesigned since only when a view is selected for materialization can the associated indexes be selected. See [LH99] for more details on how we plan on incorporating index selection into our algorithm.
- Lastly, genetic algorithms are well suited for exploiting parallelism. For further improvement in the performance of the devised algorithm, a parallel version may be devised.

References

[Aho83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1983.

[Coo71] S.A. Cook, “The Complexity of Theorem Proving Procedure,” *Annual ACM SIGACT Symposium on Theory of Computing*, pp. 151-158, 1971.

[GJ79] M.R. Garey and D.S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness*, San Francisco, 1979.

[GCB$^+$97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals,” *Data Mining and Knowledge Discovery*, **1**:1, pp. 29-53, 1997.

[GM95] A. Gupta and I.S. Mumick, “Maintenance of Materialized Views: Problems, Techniques, and Applications,” *Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, **18**:2, pp. 3-18, 1995.

[Gup97] H. Gupta, “Selection of Views to Materialize in a Data Warehouse,” in *Proceedings of the International Conference on Database Theory*, pp. 98-112, Delphi, Greece, January 1997.

[GM99] H. Gupta and I. Mumick, "Selection of Views to Materialize Under a Maintenance Cost Constraint," in *Proceedings of the International Conference on Database Theory*, Jerusalem, Israel, pp. 453-470, January 1999.

[Gol89] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, p.412, Addison-Wesley, Mass., 1989.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman, “Implementing data cubes efficiently,” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **25**:2, pp. 205-216, 1996.

[IK93] W.H. Inmon and C. Kelley, *Rdb/VMS: Developing the Data Warehouse*, QED Publishing Group, Boston, London, Toronto, 1993.

[LQA97] W. Labio, D. Quass, and B. Adelberg, “Physical Database Design for Data Warehouses,” in *Proceedings of the International Conference on Data Engineering*, Birmingham, England, pp. 277-288, March 1997.

[LH99] M. Lee and J. Hammer, “Speeding Up Warehouse Physical Design Using A Randomized Algorithm,” University of Florida, Gainesville, FL, Technical Report April 1999.

[Mic94] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Sringer-Verlag, New York, New York, NY, 1994.

[MIT] MIT Technology Lab, “GAlib: A C++ Library of Genetic Algorithm Components”, URL, `http://lancet.mit.edu/ga/`.

[RSS96] K.A. Ross, D. Srivastava, and S. Sudarshan, “Materialized view maintenance and integrity constraint checking: Trading space for time,” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **25**:2, pp. 447-458, 1996.

[Rou82] N. Roussopoulos, “View Indexing inrelational Databases,” *ACM Transactions on Database Systems*, **7**:2, pp. 258-290, 1982.

[Rou97] N. Roussopoulos, “Materialized Views and Data Warehouses,” in *Proceedings of the KRDB*, December 1997.

[TS97] D. Theodoratos and T.K. Sellis, “Data Warehouse Configuration,” in *Proceedings of the Twenty-third International Conference on Very Large Databases*, Athens, Greece, pp. 126-135, August 1997.

[Wid95] J. Widom, “Research Problems in Data Warehousing,” in *Proceedings of the Fourth International Conference on Information and Knowledge Management*, Baltimore, Maryland, pp. 25-30, November 1995.

[ZGH$^+$95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, “View Maintenance in a Warehousing Environment,” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **24**:2, pp. 316-27, 1995.