

# SPEEDING UP MATERIALIZED VIEW SELECTION IN DATA WAREHOUSES USING A RANDOMIZED ALGORITHM

MINSOO LEE<sup>†</sup>

*Dept. of Computer and Information Science and Engineering, University of Florida, 301 CSE Building  
Gainesville, FL 32611-6120, U.S.A.*

JOACHIM HAMMER

*Dept. of Computer and Information Science and Engineering, University of Florida, 301 CSE Building  
Gainesville, FL 32611-6120, U.S.A.*

Received

Revised

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, which represent pre-computed portions of frequently asked queries. 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*).

In this paper, we propose 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 maintenance-cost 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.

*Keywords:* Data warehouse, genetic algorithm, view maintenance, view materialization, view selection, warehouse configuration

## 1. Introduction

A data warehouse stores information that is collected from multiple, heterogeneous information sources for the purpose of complex querying and analysis.<sup>1,2</sup> 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<sup>3</sup>, 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

---

<sup>†</sup> Author's current address: Oracle Corporation, 200 Oracle Parkway, Box 695210, Redwood Shores, CA 94065

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.<sup>4,5</sup>

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*.<sup>a</sup>

In this paper we propose a new algorithm for 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; storage space is cheap and not regarded as a critical resource anymore. Specifically, we explore the maintenance-cost view selection problem in the context of OR view graphs, in which any view can be computed from any of its related views. Although this problem has been addressed previously (e.g., see Labio et al.<sup>6</sup>, Theodoratos and Sellis<sup>7</sup>), existing algorithms do not perform well when computing warehouse configurations involving more than 20-25 views or more. 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.<sup>8,9</sup> 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 cost 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 article is organized as follows. In Section 2 we present an overview of the related work. Section 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 maintenance-cost view selection problem. In Section 4 we describe the implementation of our prototype which was used to generate the simulation runs which we present and analyze in Section 5. Section 6 concludes the article with a summary of our results and future plans.

## 2. Related Research

The majority of the related work on view selection uses a form of greedy strategy or heuristics-based searching technique to avoid having to exhaustively traverse the solution

---

<sup>a</sup> 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.

space in search of the optimal solution. The problem of selecting additional structures for materialization was first studied by Roussopoulos<sup>10</sup> 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 the A\* algorithm<sup>11</sup> 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.<sup>12</sup> have examined the same problem using exhaustive search algorithms and provide optimizations as well as heuristics for pruning the search space. The authors have shown that the problem cannot be solved by optimizing the selection of each subview locally and must instead be addressed using global optimization. The work by Labio et al.<sup>13</sup> is also based on A\* and represents an extension of the previous work by considering indexes and also improving upon the optimality 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. Also, by running experiments, the authors were able to study how a constrained space can be used most efficiently by trading off supporting views for indexes. The results show that building indices on key attributes in the primary view will save maintenance cost while requiring only small additional amounts of storage.

Similarly, Theodoratos et al.<sup>7</sup> 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. The algorithm not only computes the set of views but also finds a complete rewriting of the queries over it.

Harinarayan et al.<sup>14</sup> present and analyze several greedy algorithms for selection of views in the special case of “data cubes”<sup>15</sup> that come within 63% of the optimal configuration. The problem of deciding which cell in a cube (i.e., view) to materialize is addressed in the form of space constraints. However, the authors’ calculations do not figure in the update costs for the selected views.

In their very detailed and informative report on view selection,<sup>6</sup> Gupta discusses the view selection problem by constraining the total space needed to materialize the selected views. Three types of view configurations are identified and form the basis for applying different kinds of greedy algorithms: the AND view graph, OR view graph, and AND-OR view graph. For each view configuration, different algorithms are devised to handle the different cases in increasing order of complexity by first considering no updates, then including updates, and finally considering both updates and indexes. The greedy algorithms are analytically proven to guarantee a solution whose performance is again within 63% of the optimal solution. The algorithms have polynomial time complexity with respect to the number of views. Although using the greedy algorithm has proven to guarantee a reasonably good solution, it still suffers from the potential problem of computing only local optima, because the initial selections influence the solution greatly. Also, the greedy algorithm considers only one view at a time. As a result, if two views may help each other, considering one view at a time will cause a further deviation from the optimal solution.

Our work is most closely related to that of Gupta et al.,<sup>16</sup> where the authors have used both the greedy approach as well as the A\* algorithm for solving the maintenance-cost view selection problem in the context of OR/AND view graphs and the general case of

AND-OR view graphs. Their approach also balances query response time and view maintenance cost while assuming a fixed amount of storage space. Since the maintenance-cost view selection problem is much harder to solve than the view selection problem without updates and considering only space constraints, approximation algorithms were designed. In the case of OR view graphs, the Inverted-Tree Greedy Algorithm was proposed. For the AND-OR view graphs, the A\* heuristic was suggested. The inverted-tree set concept used by the Inverted-Tree Greedy Algorithm can only be applied to OR view graphs. When higher complexity problems need to be solved (e.g., AND-OR view graphs), the greedy algorithm is abandoned in favor of the traditional A\* algorithm. The Inverted-Tree Greedy Algorithm provides a solution that comes within 63% of the optimal solution, while the A\* heuristic finds the optimal solution. However, in terms of the time it takes to generate a solution, the author's experimental results on OR view graphs showed that the Inverted-Tree Greedy Algorithm is generally much faster than the A\* heuristic.

Rule-based systems using the RETE, TREAT, and A-TREAT models have dealt with similar issues, namely determining which nodes to materialize in a discrimination network.<sup>17,18</sup> RETE networks materialize the selection and join nodes, while TREAT networks materialize only selection nodes to avoid the high cost of materializing join results. Wang and Hanson<sup>18</sup> compare the performance of these two networks for rule condition testing in the database environment. A-TREAT materializes nodes based on a heuristic which makes use of the selectivity.

The data warehouse configuration problem also relates to the important problem of how to answer queries using materialized views (see, for example, Chaudhuri et al.,<sup>19</sup> Larson and Yang,<sup>20</sup> and Tsatalos et al.<sup>21</sup>). Optimizing query evaluation in the presence of materialized views is the main goal of the problem. The problem of maintaining the materialized views has also been actively researched. Several incremental maintenance algorithms have been proposed (e.g., Blakely et al.,<sup>22</sup> and Gupta et al.<sup>23</sup>), and self-maintainable views are also recently being researched (e.g., Quass et al.<sup>24</sup>).

In the report by Shukla et al.,<sup>25</sup> the problem of selecting the aggregate views to pre-compute on some subsets of dimensions for multidimensional database sets is dealt with by proposing a simpler and faster algorithm named PBS over the leading existing algorithm called BPUS.<sup>14</sup> The PBS algorithm can select a set of aggregates for pre-computation even when BPUS misses good solutions. Chang et al.<sup>26</sup> suggest an adapted greedy algorithm for selection of materialized views used to design the data warehousing system for an engineering company. Their cost model optimizes the total of the maintenance, storage and query costs. The report by Zhang and Yang<sup>27</sup> deals with the dynamic environment of the data warehouse. When the user requirement changes, the materialized views must evolve to meet the new user requirements. A framework to determine if and how the materialized views are affected is provided in order to efficiently obtain the new set of materialized views.

The use of randomized algorithms in the database area has so far only been researched in the context of query optimization. More specifically, large combinatorial problems such as the multi-join optimization problem have been the most actively applied areas (see, for example, the work done by Swami<sup>28</sup>). Other areas such as data mining are also using genetic algorithms to discover an initial set of rules within the data sets (e.g., Augier et al.,<sup>29</sup> Flockhart and Radcliffe<sup>30</sup>). As more complex problems dealing with large or even unlimited search spaces emerge, these algorithms are expected to become more widely used.

### 3. Technical Approach

The view selection problem as stated in the introduction is NP-hard,<sup>31,32</sup> since one can produce 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 respect to performance when the problem size grows above a certain limit. More recent approaches use randomized algorithms to help solve 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, the solution is within a few percentage points of the optimal solution which makes randomized algorithms an attractive alternative to traditional approaches such as the ones outlined in Section 2, for example.

Among the randomized algorithms, the most well-known algorithms are *hill-climbing methods*,<sup>33</sup> *simulated annealing*,<sup>34</sup> and *genetic algorithms*.<sup>8</sup> Hill-climbing methods are based on the iterative improvement technique which is applied to a single point in the search space and continuously tries to search its neighbors to find a better point. If no other better point in the neighborhood is found, the search is terminated. This approach has the disadvantage of providing only a local optimum which is dependent on the starting point.

Simulated annealing eliminates the disadvantage of hill-climbing by using a probability for acceptance to decide whether or not to move to a neighboring point. The technique originates from the theory of statistical mechanics and is based on the analogy between the annealing of solids (i.e., the process that occurs when heating and subsequently cooling a substance) and solving optimization problems. The probability for acceptance is dynamically calculated based on two factors: (1) how good the neighboring point is, and (2) a so-called temperature value. In simulated annealing, it is possible to move to a neighboring point that is further away from the optimum than the previous one in expectation that its neighbors will represent a better solution. The lower the temperature value, the harder it is to move to a neighboring point that is worse than the previous point. As the algorithm proceeds, the temperature is lowered to stabilize the search on a close to optimum point. Using a probability for acceptance eliminates the dependency on the starting point of the search.

The genetic algorithm in contrast, uses a multi-directional search by maintaining a pool of candidate points in the search space. Information is exchanged among the candidate points to direct the search where good candidates survive while bad candidates die. This multi-directional evolutionary approach will allow the genetic algorithm to efficiently search the space and find a point near the global optimum.

The motivation to use genetic algorithms in solving the maintenance-cost view selection problem was based on the observation that data warehouses can have a large number of views and that the queries that must be supported may change very frequently. Thus, a solution is needed to provide new materialized view and index configurations for the data warehouse quickly and efficiently: an ideal scenario for the genetic algorithm. However, genetic algorithms do not magically provide a good solution and their success (or failure) often depend on the proper problem specification: the set-up of the algorithm, as

well as 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 in the next section, we provide details on how we have applied the genetic algorithm to the maintenance-cost view selection problem. Specifically, we elaborate on a suitable representation of the solution as well as on the necessary evaluation functions that are needed by our genetic algorithm to guide its exploration of the solution space.

### 3.1. Genetic algorithms

The idea behind the genetic algorithm 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 are 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 in this sequence is referred to as a generation. Generations are labeled, for example, generation  $t$ , generation  $t+1$ , generation  $t+2$ , ..., and so on. After several generations, it is expected that the population in generation  $t+k$  should be composed of superior genomes (i.e., those that have survived the evolution process) than the population in generation  $t$ . Figure 1 shows a sequence of populations.

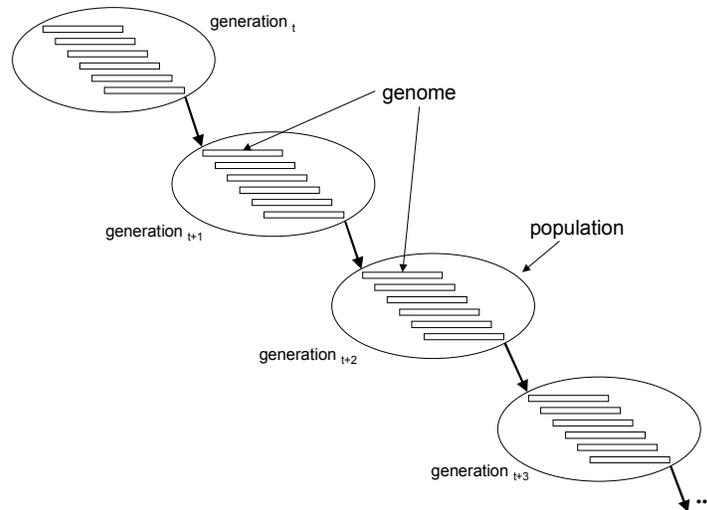


Figure 1: Sequence of populations.

Starting at  $t=0$ , the initial population  $P(0)$  – also referred to as *generation<sub>0</sub>* – is established by either randomly generating a pool of genomes or replicating a particular genome. The genetic algorithm then 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 indexed by the value of  $t$  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,<sup>9</sup> is used to select those genomes which are superior.

In step ③, the population is recombined by performing several operations on paired or individual genomes to create new genomes in the population. These operations are called *crossover* and *mutation* operations, respectively. The crossover operation allows a pair of genomes to exchange information between themselves expecting that the superior properties of each genome can be combined. The mutation operation applies a random change to a genome expecting the introduction of a new superior property into the genome pool. Step ④ evaluates the population that is created. A so-called *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 maintenance-cost view selection problem.

### **3.2. A new algorithm for the maintenance-cost view selection problem**

In order to apply a Genetic Algorithm (GA) approach to the maintenance-cost view selection problem the following three requirements must be met: (1) We need to find a formal representation of a candidate solution, preferably one that can be expressed using simple character strings. (2) We need to decide on a method to initialize the population, perform the crossover and mutation operations, and determine the termination condition for the genetic algorithm. (3) Most importantly, we need to define the fitness function as outlined above.

Michalewicz<sup>9</sup> discusses several solutions to popular problems using genetic algorithms, including the 0/1 knapsack problem (see, for example, Aho et al.<sup>35</sup>). The similarity of the maintenance-cost view selection problem to the 0/1 knapsack problem gives us a hint on how to apply the Genetic Algorithm in our context. Both problems try to maximize a certain property while satisfying one or more constraints. The goal of the 0/1 knapsack problem is to select a set of items that will maximize the total profit while satisfying a total weight constraint. The goal of the maintenance-cost view selection problem on the other hand is to select a set of views to be materialized in order to maximize the query benefit (i.e., minimize query response time) while adhering to a total maintenance cost limit. The difference is that the items selected in the 0/1 knapsack problem do not affect each other in terms of the profit/weight computation, while each view selection in the maintenance-cost view selection problem will affect the cost computation of the parent views resulting in a more complex problem. To our knowledge, nobody has yet to apply genetic algorithm techniques to solving the maintenance-cost view selection and the solutions presented here represent our own approach.

#### **3.2.1. Problem specification**

Summarizing Gupta and Mumick,<sup>16</sup> the problem to be solved can be stated informally as follows: Given an OR view graph  $G$  and a quantity representing the total maintenance time limit, select a set of views to materialize that minimizes the total query response time and also does not exceed the total maintenance time limit. An OR view graph is constructed

based on the queries involved in the data warehouse application. Each query will form a view which is a node in the graph. Additional views may also be included as nodes if these views can contribute to constructing the views related to the queries of the data warehouse application. The relationship among the views are modeled as edges between the nodes. An OR view graph is composed of a set of views where each view in the graph can be derived from a subset of other views (i.e., source views) within the graph in one or more ways, but each derivation involves only one other view. In other words, only a single view among the source views is needed to compute a view. An example of an OR view graph is the data cube<sup>15</sup> where each view can be constructed in many different ways but each derivation only involves one view.

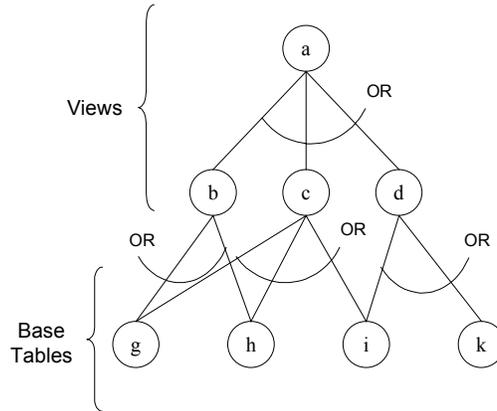


Figure 2: Sample OR view graph for four views.

A sample OR view graph is shown in Figure 2. For example, in this figure, view *a* can be computed from any of the views *b*, *c*, *d*. View *b* can again be computed from any of the views *g* or *h*. The same applies to computing the views *c* and *d* from their child views. If all of the ORs were converted to ANDs, meaning that all of the child views are needed to compute the parent view, the graph becomes an AND view graph. If there exists a mix of ORs and ANDs in the view graph, it is called an AND-OR view graph. Note that although our problem does not deal with the issue of constructing the OR view graph, parameters related to a given OR view graph are used in our cost model for generating a solution. The complexity of the problem can be addressed by examining the search space involved. Assuming that there are  $n$  views in the graph, the total number of the subset of views that can exist are  $2^n$  (i.e., exponential to the number of views).

As mentioned in the introduction, at this point we are focusing our attention on OR view graphs since the cost model is considerably less complex and therefore more accurately definable than that for the other two view graph models (i.e., AND and AND-OR view graphs) whose cost model relies on a wide range of parameters. The simplicity of the cost model allows us to isolate our experimental results from the complex issues related to a complex cost model. 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<sup>15</sup> as indicated above.

We now formally restate our maintenance-cost view selection problem as follows.

**Definition: Maintenance-cost view selection problem for OR view graphs.** Given an OR view graph  $G$ , and a total maintenance cost limit  $S$ , select the set of views  $M$  to materialize that satisfy the following two conditions: (1) minimizes the total query cost  $\tau$  of the OR view graph  $G$  (i.e., maximizes the benefit  $B$  of the total query cost obtained by materializing the views) and (2), the total maintenance cost  $U$  of the set of materialized views  $M$  is less than  $S$ .

In our future research, we will address the cases of AND view graphs as well as AND-OR view graphs. We are also deferring the problem of index selection. However, in Section 6 we outline how index selection can be added into in a straightforward manner.

### 3.2.2. Design of our genetic algorithm solution

#### STEP 1 - REPRESENTATION OF THE SOLUTION

A genome represents a candidate solution of the problem to be solved. The solution of the problem must be represented in a string format. The string can be a binary string composed of 0s and 1s, or 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 within the search space. Alphanumeric strings are suitable for representing a solution to an ordering problem, where an optimal order of the items need to be decided, such as the Traveling Salesman Problem. On the other hand, binary strings are more suitable for representing solutions to selection problems where an optimal subset of the candidate items need to be identified.

In the solutions presented here, we use a binary string representation. The views in the OR view graph may be enumerated as  $v_1, v_2, \dots, v_m$  where  $m$  is the total number of views. We can represent a selection of these views as a binary string of  $m$  bits as follows. If the bit in position  $i$  (starting from the leftmost bit as position 1) is 1, the view  $v_i$  is selected. Otherwise, the view  $v_i$  is not selected. For example, the bit string 001101001 encodes the fact that from a possible set of nine views total, only views  $v_3, v_4, v_6$ , and  $v_9$  are selected. In general, the encoding of a genome can be formalized as:

$$\text{genome} = (b_1 b_2 b_3 \dots b_m), \text{ where } \begin{aligned} b_i &= 1 \text{ if view } v_i \text{ is selected for materialization} \\ b_i &= 0 \text{ if view } v_i \text{ is not selected for materialization} \end{aligned}$$

#### STEP 2 - INITIALIZATION OF THE POPULATION

The initial population is a pool of randomly generated bit strings of size  $m$ . In future implementations, however, we will start with an initial population which represents a favorable configuration based on external knowledge about the problem and its solution rather than a random sampling. External knowledge such as the OR view graph configuration information can be used to drive the initial selection of materialized views when creating a genome. An example heuristic for this could be that the views with a high query frequency are most likely selected for materialization. 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 report, we have chosen a population size of 30, which is a commonly used value in the literature based on studies conducted by De Jong.<sup>8</sup>

#### STEP 3 - SELECTION, CROSSOVER, MUTATION, TERMINATION

The selection process will use the popular roulette wheel method.<sup>9</sup> The crossover and mutation operators are assigned probabilities  $p_c$  and  $p_m$ , respectively. The specific values

for those probabilities in our simulation are 0.9 and 0.001. We have chosen these values because of the success other researchers had when using these probabilities to solve related problems.<sup>9</sup> A study on parametric values<sup>8</sup> used for genetic algorithms by De Jong in 1975 suggested a high crossover probability, low mutation probability and a moderate population size. As we have mentioned in the beginning, a considerable amount of time must be spent fine-tuning the parameters controlling the various steps of the genetic algorithm. We have experimented with several different starting values (based on our findings in the literature on random algorithms) and found that the parameter values presented in this report resulted in a system that produced the most favorable results when applied to solving the maintenance-cost view selection problem. In addition, we have also experienced that picking the correct seed values is as much of an art as it is science!

The *roulette wheel method* makes use of the fitness values of each genome. The fitness value indicates how good the genome is as a solution to the problem. The details of the fitness value calculation are deferred until later in our explanations. The roulette wheel method works as follows:

```
// Assume that each genome is assigned a number between 1 and pop_size.
// The fitness value for each genome is  $f_i$ .

Calculate the total fitness  $F$  of the population by adding the fitness
values of the genomes.


$$F = \sum_{i=1, \text{pop\_size}} f_i$$


Calculate the selection probability  $p_i$  of each individual genome by
dividing each genome's fitness by  $F$ .


$$p_i = f_i / F$$


Calculate the cumulative probability  $c_i$  for each genome by adding up the
 $p_i$  values for the genomes 1 through  $i$ .


$$c_i = \sum_{j=1, i} p_j$$


Repeat the following steps pop_size times, which simulates spinning the
roulette wheel pop_size times:

    Generate a random number  $R_n$  within 0 to 1.

        
$$R_n = \text{random}(0,1)$$


    If  $R_n < c_1$  then select the first genome 1.

    Else select the genome  $i$  where  $c_{i-1} < R_n \leq c_i$ .
```

The *crossover* operation is applied to two genes by exchanging information between the two, thereby creating two new genes. Crossover works as follows:

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

Exchange information among genomes, and replace  $g1, g2$  with  $g1', g2'$ 

// (ex)  $g1' = (b_1 \ b_2 \ \dots \ b_{pos} \ \quad c_{pos+1} \ \dots \ c_m)$  and
//  $g2' = (c_1 \ c_2 \ \dots \ c_{pos} \ \quad b_{pos+1} \ \dots \ b_m)$ 
```

The *mutation* operator makes changes to a single genome and works as follows:

```
For all genomes,  
  For each bit in the genome,  
    mutate (flip) the bit with a probability of  $p_m$ 
```

The selection, crossover, mutation and evaluation (described in Step 4 below) processes are repeated in a loop until the termination condition is satisfied. In our approach, the *termination condition* is 400 generations. The termination condition is an adjustable value that can be decided from experiments using the designed genetic algorithm. Although studies in Michalewicz<sup>9</sup> reported no noticeable improvements for the genetic algorithm designed for the 0/1 knapsack problem after roughly 500 generations, we were able to reduce 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 satisfies the goal; if the fitness is low, the genome is not a suitable solution. Although the solutions with high fitness values are desirable and usually selected for the next generation, those with a low fitness value are included in the next generation with a small probability. This allows the genetic algorithm to explore various possibilities for evolution. As the evaluation of a genome forms the basis for evaluating the combined fitness of a population, correctly defining the fitness function is critical to the success of the genetic algorithm. The fitness function for evaluating a genome can be devised in many different ways depending on how the effectiveness of the solution is measured. Finding the best possible fitness function (i.e., one that can truthfully evaluate the quality of a particular warehouse configuration) requires a lot of fine-tuning which can only be done through experimental results. We describe the outcome of this fine-tuning in detail in Section 5.

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 (i.e., reduction in the query cost due to materialization of query results in the form of views) and maintenance constraint. This is similar to the 0/1 knapsack problem, 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 maintenance-cost 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 function will reduce the fitness if the maintenance constraint is not satisfied. When the maintenance constraint is satisfied, the penalty function 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. Since the fitness value cannot assume 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 by 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 we discuss later.

Assume that  $B$  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  $x$  (if the view  $v_i$  represented by the bit  $i$  in  $x$  is set, the view is included in  $M$ ; otherwise, the view is not included in  $M$ ). We have defined a fitness function, called  $Eval$ , as follows:

$$\begin{aligned} \textbf{Subtract mode (S): } Eval(x) &= B(G,M) - Pen(x) && \text{(if } B(G,M) - Pen(x) \geq 0) \\ &= 0 && \text{(if } B(G,M) - Pen(x) < 0) \\ &&& \text{for all } x[i] = 1, \quad v_i \in M \\ &&& \text{for all } x[i] = 0, \quad v_i \notin M \end{aligned} \quad (3.1)$$

$$\begin{aligned} \textbf{Divide (D): } Eval(x) &= B(G,M) / Pen(x) && \text{(if } Pen(x) > 1) \\ &= B(G,M) && \text{(if } Pen(x) \leq 1) \end{aligned} \quad (3.2)$$

**Subtract&Divide (SD):**

$$\begin{aligned} Eval(x) &= B(G,M) - Pen(x) && \text{(if } B(G,M) > Pen(x)) \\ &= B(G,M) / Pen(x) && \text{(if } Pen(x) \geq B(G,M) \text{ and } Pen(x) > 1) \\ &= B(G,M) && \text{(if } Pen(x) \geq B(G,M) \text{ and } Pen(x) \leq 1) \end{aligned} \quad (3.3)$$

The penalty function itself can also have various forms. For example, we have experimented with logarithmic penalty, linear penalty and exponential penalty functions as shown in Eq. (3.4), Eq. (3.5), and Eq. (3.6). Note that the penalty that is applied by the different functions increases: the logarithmic penalty function applies the lowest, the exponential penalty function the highest penalty value.

The function  $U$  calculates the total maintenance cost for the set of materialized views  $M$ . The value  $\rho$  is a constant which is calculated using the query benefit function and the total maintenance cost function.  $S$  is the total maintenance time constraint.

$$\textbf{Logarithmic penalty (LG): } Pen(x) = \log_2 (1 + \rho (U(M) - S)) \quad (3.4)$$

$$\textbf{Linear penalty (LN): } Pen(x) = (1 + \rho (U(M) - S)) \quad (3.5)$$

$$\textbf{Exponential penalty (EX): } Pen(x) = (1 + \rho (U(M) - S))^2 \quad (3.6)$$

We have combined the three penalty modes (i.e., S, D, SD) with the three penalty functions (i.e., LG, LN, EX) in our prototype to evaluate and determine the best possible strategy for solving the maintenance-cost view selection problem. Among the nine combinations (i.e., LG-S, LG-D, LG-SD, LN-S, LN-D, LN-SD, EX-S, EX-D, EX-SD), our evaluation has identified several promising strategies.

### 3.2.3. Cost model and formulae

The details as well as the formulae for the query benefit function  $B(G,M)$ , the total maintenance cost  $U(M)$ , and  $\rho$  are described next. Before explaining the formulae, we first illustrate the costs that are assigned to an OR-view graph. Table 1 provides a list of cost parameters.

Figure 3 depicts a sample OR-view graph. Each node, which represents a view in the graph, has associated with it a read cost (RC), a query frequency (QF), and an update

frequency (UF). In addition, 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.

	Parameter	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.

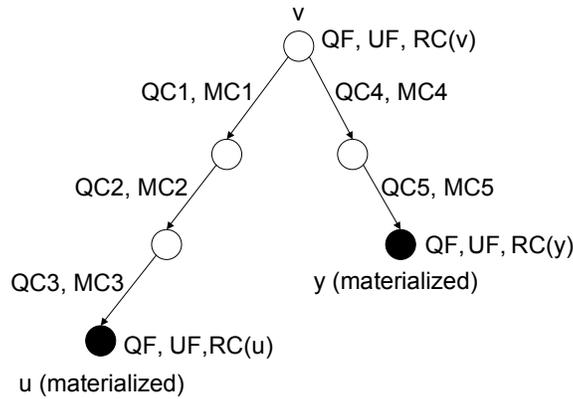


Figure 3: Sample OR view graph with corresponding cost parameters.

Each edge in the graph has a query cost (QC1 to QC5) and a maintenance cost assigned (MC1 to MC5). The white nodes in the graph denote non-materialized views (i.e., views that are not selected for materialization) while the black nodes denote the views that are selected for materialization. Both non-materialized and materialized nodes have a read cost assigned to them. However, for the simplifying the subsequent explanations we only show those read costs which are used in the text. As shown in the graph, the materialized nodes  $u$  and  $y$ , and the non-materialized node  $v$  are assigned with read costs  $R(u)$ ,  $R(y)$ , and  $R(v)$ , respectively. Note, the query frequency (QF) and the update frequency (UF) are also assigned to each node but are just shown for the nodes  $u$ ,  $v$ , and  $y$  to keep the figure simple.

In order to define the query benefit function  $B(G, M)$ , we first need to clarify the definitions of the two functions  $Q(v, M)$  and  $\tau(G, M)$ . In the following definitions, remember that the set of materialized views  $M$  is equivalent to a single genome configuration in the genetic algorithm.

**Definition 1:**  $Q(v, M)$  is the *cost of answering a query* on  $v$  (either a view or a base table) in the presence of a set of materialized views  $M$ . This function will calculate the *minimum query-length* of a path from  $v$  to some  $u \in (M \cup L)$  where  $L$  is the set of sinks (base tables) in  $G$ . The query-length is calculated as follows:

$$\begin{array}{ll} \text{Query-length} & = RC(u) + \text{sum (query-costs)} \\ \text{"of path from } v \text{ to } u\text{"} & \quad \text{"associated with edges on path from } v \text{ to } u\text{"} \end{array}$$

Assuming that base tables are always available,  $Q(v, \phi)$  is the cost of answering a query directly from the base tables. In the example shown in Figure 3, assume that  $RC(u)+QC3+QC2+QC1 < RC(y)+QC5+QC4$ . Then the *cost of answering a query* on  $v$  in the presence of a set of materialized views  $M$  is computed as  $Q(v, M) = RC(u) + QC3 + QC2 + QC1$ .

**Definition 2:** The *total query cost*  $\tau(G, M)$  is defined over the OR view graph  $G$  in the presence of a set  $M$  of materialized views. This is the quantity that we want to minimize in the maintenance-cost view selection problem.

$$\tau(G, M) = \sum QF_v * Q(v, M), \text{ where } v \in V(G)$$

In this formula,  $QF_v$  is the query frequency of the view  $v$ ,  $V(G)$  denotes *all of the views* (materialized and not materialized) in the OR-view graph  $G$ , and  $Q(v, M)$  is the *cost of answering a query* on  $v$  in the presence of a set  $M$  of materialized views as defined above:

**Definition 3:** Using the definitions 1 and 2, the cost formula for the *query benefit function*  $B(G, M)$  is defined below assuming that  $G$  is a given OR-view graph, and  $M$  is the set of selected views to be materialized:

$$B(G, M) = \tau(G, \phi) - \tau(G, M)$$

Note that Gupta and Mumick<sup>16</sup> refer to  $B$  as the *absolute benefit* of  $M$ . Our notation of  $B(G, M)$  is equivalent to their notation of  $B(M, \Phi)$ . The penalty function uses the *total maintenance cost function*  $U(M)$ . To define  $U(M)$ , the function  $UC(v, M)$  needs to be specified first.

**Definition 4:**  $UC(v, M)$  is the *cost of maintaining a materialized view*  $v$  in the presence of a set of materialized views  $M$ . This is calculated as the *minimum maintenance-length* (sum of maintenance costs associated with edges) of a path from  $v$  to some  $y \in (M \cup L) - \{v\}$  where  $L$  is the set of sinks (base tables) in  $G$ .

For example, in Figure 3, assume that  $MC4+MC5 < MC1+MC2+MC3$ . Then  $UC(v, M)$  is calculated as  $UC(v, M) = MC4 + MC5$ .

**Definition 5:** Assume that  $M$  is a set of views selected for materialization, and  $UF_v$  is the update frequency of the view  $v$ . Then the *total maintenance cost function*  $U(M)$  is defined as :

$$U(M) = \sum UF_v * UC(v, M), \text{ where } v \in M$$

Note that this cost is calculated *only* over *materialized views*, not all views.

**Definition 6:** The formula to calculate  $\rho$  as used in the penalty function is:

$$\rho = \text{Max}\left(\frac{B(G, \{v_i\})}{U(\{v_i\})}\right), \text{ where } v_i \in V(G)$$

#### 4. Prototype Implementation

We have used version 2.4.3 of the genetic algorithm toolkit from MIT called Galib<sup>36</sup> to develop a prototype of the algorithm described above. The toolkit supports various types of genetic algorithms, 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 in the graph, has associated with it a read cost (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).

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. During the depth-first traversal, intermediate results are stored on the nodes of the OR view graph to eliminate redundant calculations.

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. This is also calculated simultaneously while the total query cost is calculated within a single run of the depth-first traversal. The detailed formulae and examples were discussed in Section 3.2.3.

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. The parameters that control the OR view graph configuration are the number of base tables, number of views, density, query cost range, maintenance cost range, read cost range for base tables, query frequency range, and update frequency range. In addition, we implemented an exhaustive search algorithm to find the

optimal solution in order to be able to compare the quality of our GA 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 runtime 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. Although we tested our genetic algorithm on warehouses containing significantly more than 20 views, we did not include those numbers in this analysis since we had no benchmark values for comparison. The extremely long running times for the exhaustive algorithm when given a configuration of more than 25 views made such comparisons infeasible. For example, using a Pentium II PC with 128MB of RAM, a single execution of the exhaustive algorithm on a warehouse with 30 views took over 24 hours to complete.<sup>b</sup> The edge density of the graph varied from 15% to 30% to 50% to 75%. The ranges for the values of all of the important parameters of the OR-view graphs are shown in Table 2. The maintenance cost constraints for the problem were set to 50, 100, 300, and 500. Please note that one may interpret these values as time limits on how long the warehouse may be down for maintenance or as upper values for the amount of data that must be read etc.

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

Nodes (Views)			Edges	
<i>RC</i>	<i>QF</i>	<i>UF</i>	<i>QC</i>	<i>MC</i>
100-10,000 for base tables (RC for views are calculated from source views)	0.1 - 0.9	0.1- 0.9	10 - 80 % of RC of source view	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 total computed query cost (obtained using the genetic algorithm). The ratio was computed and averaged over several runs. It was expected that the ratio would always be less than 100%. However, we observed that the *genetic algorithm sometimes relaxes the maintenance cost constraint in order to trade off maintenance cost with a lower and better overall query cost*. In those cases, the total query cost obtained was actually lower than the optimal query cost which was computed with a strict 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 interpreted as a guideline (within certain limits) rather than as a strict value. Actually, the

<sup>b</sup> One also has to keep in mind that in order to gather enough data for our analysis, several runs are needed for each warehouse configuration.

inverted-tree greedy heuristic by Gupta and Mumick<sup>16</sup> also does not guarantee a strict maintenance cost constraint, but satisfies a limit within twice the constraint value. The nine different strategies are denoted LG-S, LG-D, LG-SD, LN-S, LN-D, LN-SD, EX-S, EX-D, EX-SD, where LG, LN, S, etc. denote the different penalties and functions as described in Sec. 3.2.2.

After an initial run of experiments, we realized that 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 enforce a strong enough penalty on the fitness value. Thus, for LG-S and LG-SD, it 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 4 and 5.

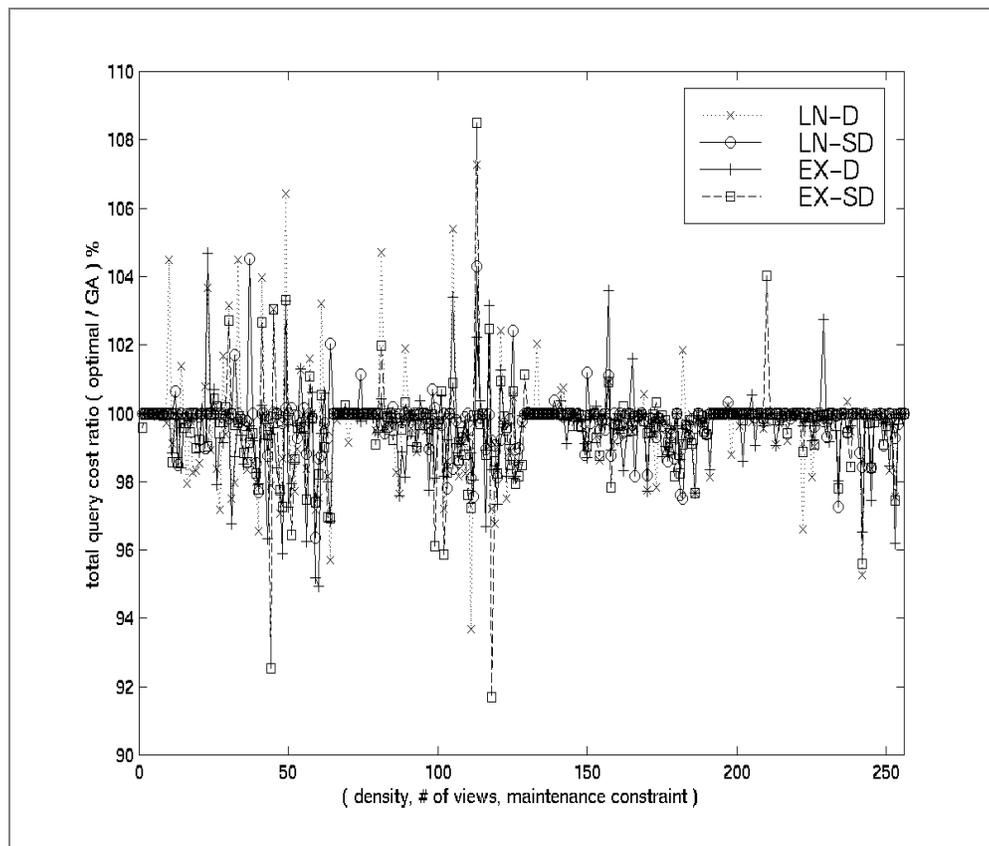


Figure 4: Average ratios of optimal total query cost over GA query cost.

Figure 4 shows the results of averaging over the ratios of optimal total query cost (based on a strict maintenance constraint) over GA total query costs. The values are arranged in tuples in lexicographical order as follows:

(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 plotted in increasing order within a given density. The maintenance cost is provided in increasing order within each set of views.

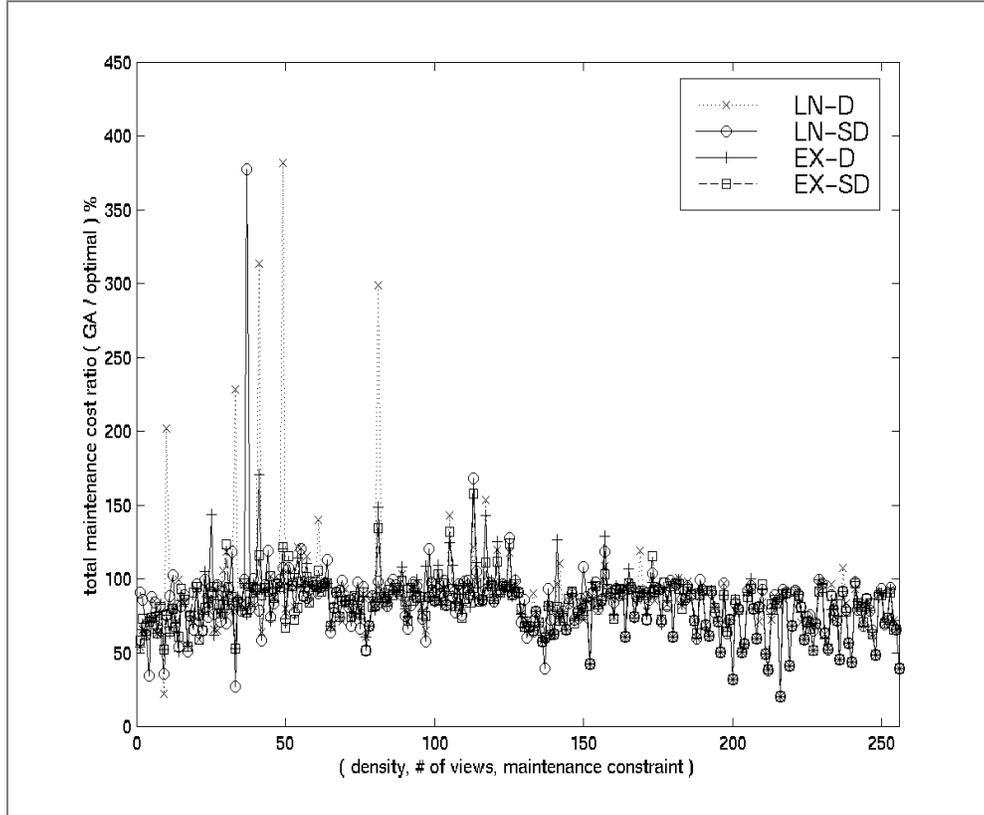


Figure 5: Average ratios of GA total maintenance cost over maintenance constraint.

Figure 5 shows the results of averaging over the ratios of GA total maintenance cost over the maintenance constraint. The results show that the LN-D and LN-SD still have a considerably large fluctuation (about 380%) for the maintenance cost. These behaviors were exhibited especially for low density OR-view graphs where the penalty values resulted in small values which were not enough to enforce the maintenance constraint. If we discard these two mechanisms from our consideration, Figure 4 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. It is interesting to note that this result is also very close to the one that was verified in theory in the inverted-tree greedy heuristics proposed by Gupta and Mumick<sup>16</sup> where the solution returned has a maintenance cost within twice the limit and comes within 63% of the optimal solution.

## 5.2. Execution time

Figures 6 and 7 show the execution times for the exhaustive search algorithm and our genetic algorithm averaged over the sample OR view graphs. The exhaustive search algorithm shown in Figure 6 was developed to obtain a benchmark for measuring the quality of the solutions provided by the genetic algorithm. 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 provide a feel for the performance.

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. The results of other heuristic approaches can be found in the literature. For example, Labio et al.<sup>13</sup> provide a comparison of their work on the A\* heuristic against the exhaustive search algorithm in terms of the search space pruned. Gupta and Mumick<sup>16</sup> also provide experimental results based on the A\* heuristic and Inverted-tree Greedy heuristics. Both show polynomial performance in the best case using the heuristics.

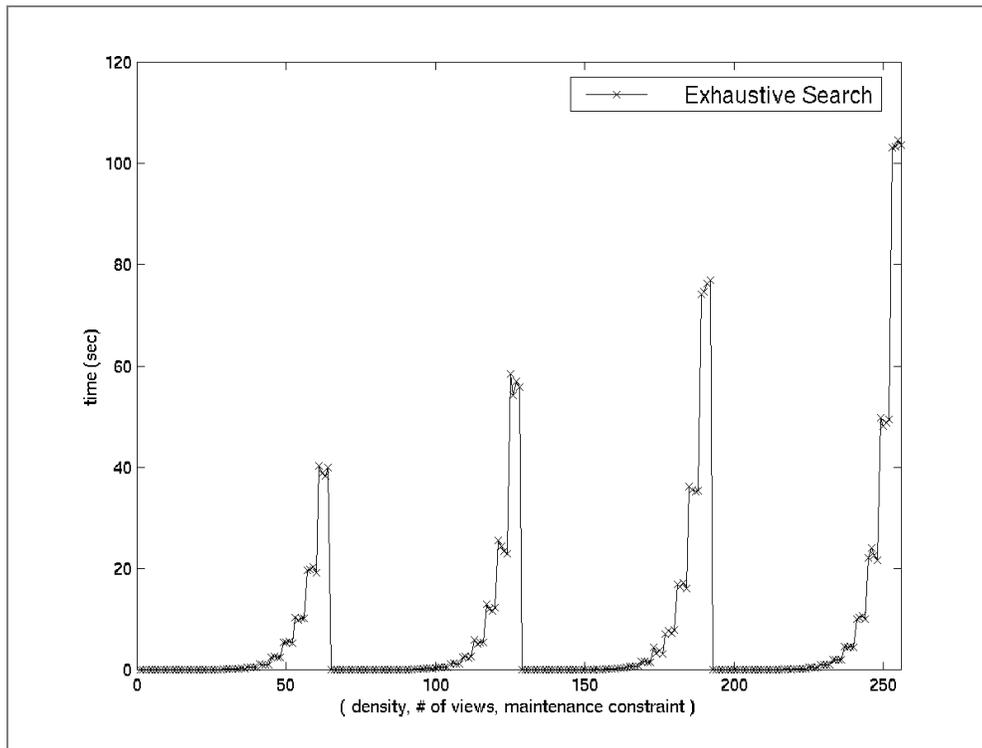


Figure 6: Execution time for Exhaustive Search Algorithm.

Our genetic algorithm on the other hand, which is shown in Figure 7, exhibits linear behavior. When the number of views are small (e.g., less than 5 views), the overhead of the genetic algorithm will actually make it slower than using an exhaustive search. However, its advantage over existing solutions can be clearly seen when the number of views in the

warehouse is large. As the density grows, the slope of the linear graph increases only slightly. The genetic algorithm took approximately  $1/80^{\text{th}}$  of the time taken by the exhaustive search to compute an OR-view graph with 20 views and 75% density. As the number of views goes up to 30 and beyond, this ratio is expected to be much more impressive. Yet the quality of the solution generated by the genetic algorithm is still very close to the optimal.

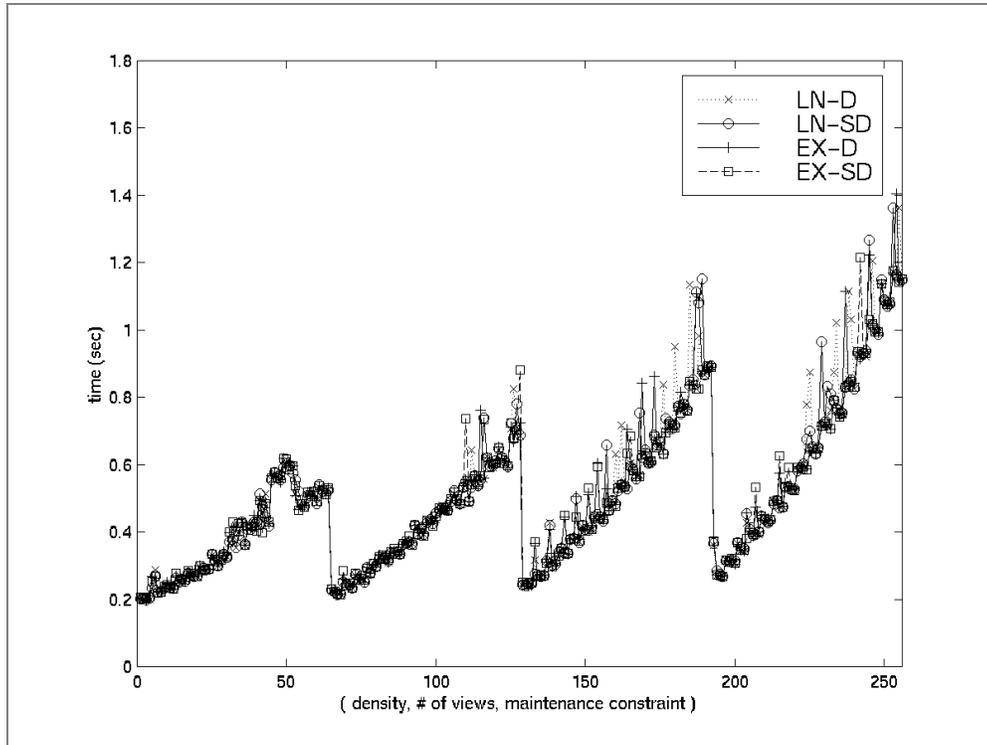


Figure 7: Execution time for Genetic Algorithm.

## 6. Conclusion

### 6.1. Summary

In this paper we have shown that our genetic algorithm is superior to existing solutions to the maintenance-cost 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 while at the same time exhibiting a linear run-time behavior. We have verified this by running an exhaustive search on OR view graphs with up to 20 views. A penalty function has been included in the fitness function, and experimental results show that the EX-D and EX-SD strategies for applying penalty functions produce the best results for the maintenance-cost view selection problem. We believe that our algorithm can become an important tool for warehouse evolution, especially for those data warehouses

that contain a large number of views and must accommodate frequent changes to the queries which are supported by the given warehouse configuration.

## 6.2. *Future improvements*

For future research, we are investigating the following improvements:

- Generate an initial population based on knowledge of a possible solution rather than using random configurations. This should allow the algorithm to converge more quickly.
- Experiment with several other crossover or mutation operators to create better genomes more effectively and speed up convergence of the algorithm even further.
- Implement a more flexible termination condition that can interrupt the algorithm when the solution lies within a certain threshold instead of always computing all 400 generations.
- Expand our approach to include AND-OR view graphs as well as indexes. The first is straightforward as it only needs a new, well-defined cost model. The latter is more complicated as we have to modify the solution representation (see Sec. 6.3 for a possible approach).
- Genetic algorithms are well suited for exploiting parallelism. For further improvement of the performance of our algorithm, we are looking into developing a parallel version on an 8-node, 32 processor IBM RS/6000 SP.

## 6.3. *View selection with indexes*

In the remainder of this article, we sketch out our solution to the maintenance-cost view selection problem which includes indexes. In our solution representation, we begin by adding the indexes related to a view immediately after the bit position of the view. As an example, assuming view  $v_1$  has indexes  $i_1$ ,  $i_2$  and  $i_3$ , and view  $v_2$  has indexes  $i_4$  and  $i_5$ . The positions of these views and related indexes can be fixed in the order of  $(v_1 i_1 i_2 i_3 v_2 i_4 i_5)$ . However, crossover operations and mutation operations may need to be carefully redesigned since an index can be selected only when the associated view has been selected for materialization.

One possible way to modify the crossover is to adjust the crossover point to be always before a view position in the genome. For example, if the initial crossover point is  $k$  in the genomes and position  $k$  is an index position related to view  $v_i$  (where the bit position of  $v_i$  is  $j$ , and  $j < k$ ), then adjust the crossover point to be  $j-1$  instead of  $k$ . This should allow the related view and index information to not be separated and thus consistently maintained. The mutation operation must be changed to select a bit (i.e., flip the bit from 0 to 1) representing an index only when the view to which the index is related is selected. Furthermore, a bit representing a view may only be deselected (i.e., flipped from 1 to 0) when it is assured that all of the indexes related to the view are not selected.

The cost model for the OR-view graph which allows us calculate the total query cost and the total maintenance cost also needs to be modified in order to consider the benefits and costs related to index materialization. The candidate indexes for materialization will not only be associated with each view but also with edges connecting the view to a parent view since the index will affect the query cost related to the edge. If an index is materialized, the index will decrease the query cost related to the edge by a certain percentage specified by the index. On the other hand, the index materialization will increase the maintenance cost related to the edge by a certain percentage specified by the

index. These additional cost calculations are included in the total maintenance cost calculation. Including indexes will further increase the possible number of physical data warehouse configurations, making it even more difficult to find a good configuration within a reasonable time using traditional methods as the number of views and candidate indexes increase.

We have already added these extensions to the prototype described in this paper and are performing a series of new experiments to help us analyze our approach. It is worth pointing out that a significant amount of verification is needed in this new algorithm to verify if the genomes correctly represent a data warehouse configuration in terms of the selected views and indexes as discussed above for the new crossover and mutation operations. One approach that we are considering is to leave the crossover and mutation operations relatively simple while increasing the complexity of the fitness function. The goal is to heavily penalize those genomes that do not reflect a correct configuration in order to force them out of the populations. We will report on our findings in subsequent conference papers and articles.

## References

1. W. H. Inmon and C. Kelley, *Rdb/VMS: Developing the Data Warehouse* (QED Publishing Group, Boston, London, Toronto, 1993).
2. J. Widom, Research Problems in Data Warehousing, in *Proceedings of the Fourth International Conference on Information and Knowledge Management*, Baltimore, Maryland, 1995, 25-30.
3. N. Roussopoulos, Materialized Views and Data Warehouses, in *Proceedings of the Workshop on Knowledge Representation meets Databases (KRDB)*, Athens, Greece, 1997, 12.1-12.6.
4. 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**, (1995) 3-18.
5. 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** (1995) 316-27.
6. H. Gupta, Selection of Views to Materialize in a Data Warehouse, in *Proceedings of the International Conference on Database Theory*, Delphi, Greece, 1997, 98-112.
7. D. Theodoratos and T. K. Sellis, Data Warehouse Configuration, in *Proceedings of the Twenty-third International Conference on Very Large Databases*, Athens, Greece, 1997, 126-135.
8. D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning* (Addison-Wesley, Reading, Mass, 1989).
9. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs* (Springer-Verlag, New York, New York, NY, 1994).
10. N. Roussopoulos, View Indexing in relational Databases, *ACM Transactions on Database Systems* **7:2** (1982) 258-290.
11. N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence* (MacGraw-Hill, New York NY, 1971).
12. 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** (1996) 447-458.

13. W. Labio, D. Quass, and B. Adelberg, Physical Database Design for Data Warehouses, in *Proceedings of the International Conference on Database Engineering*, Birmingham, England, 1997, 277-288.
14. V. Harinarayan, A. Rajaraman, and J. D. Ullman, Implementing data cubes efficiently, *SIGMOD Record (ACM Special Interest Group on Management of Data)* **25:2** (1996) 205-216.
15. 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** (1997) 29-53.
16. H. Gupta and I. Mumick, Selection of Views to Materialize Under a Maintenance Cost Constraint, in *Proceedings of the International Conference on Management of Data*, Jerusalem, Israel, 1999, 453-470.
17. E. Hanson, Rule Condition Testing and Action Execution in Ariel, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, 1992, 49-58.
18. Y.-W. Wang and E. Hanson, A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions, in *Proceedings of the Eighth International Conference on Database Engineering*, Tempe, AZ, 1992, 88-97.
19. S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, Optimizing Queries with Materialized Views, in *Proceedings of the International Conference on Data Engineering*, Taipei, Taiwan, 1995, 190-200.
20. P.-A. Larson and H. Yang, Computing Queries from Derived Relations, in *Proceedings of the Eleventh International Conference on Very Large Databases*, Stockholm, Sweden, 1985, 259-269.
21. O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis, The GMAP: A Versatile Tool for Physical Data Independence, in *Proceedings of the Twentieth International Conference on Very Large Databases*, Santiago de Chile, Chile, 1994, 367-378.
22. J. A. Blakeley, P.-A. Larson, and F. W. Tompa, Efficiently Updating Materialized Views, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., 1986, 61-71.
23. A. Gupta, I. Mumick, and V. Subrahmanian, Maintaining Views Incrementally, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., 1993, 157-166.
24. D. Quass, A. Gupta, I. S. Mumick, and J. Widom, Making Views Self-maintainable for Data Warehousing, in *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, 1996, 158-169.
25. A. Shukla, P. Deshpande, and J. F. Naughton, Materialized View Selection for Multidimensional Datasets, in *Proceedings of the Twenty-fourth International Conference on Very Large Databases*, New York City, New York, USA, 1998, 488-499.
26. G. K. Y. Chan, Q. Li, and L. Feng, Design and selection of materialized views in a data warehousing environment: a case study, in *Proceedings of the Information and Knowledge Management*, Kansas City, MO, USA, 1999, 42-47.
27. C. Zhang and J. Yang, Materialized View Evolution Support in Data Warehouse Environment, in *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, Hsinchu, Taiwan, 1999, 247-254.
28. A. Swami, Optimization of large join queries: combining heuristics and combinational techniques, *SIGMOD Record* **18:2** (1989) 367-76.

29. S. Augier, G. Venturini, and Y. Kodratoff, Learning First Order Logic Rules with a Genetic Algorithm, in *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95)*, Montreal, Canada, 1995, 21-26.
30. I. W. Flockhart and N. J. Radcliffe, A Genetic Algorithm-based Approach to Data Mining, in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, Portland, Oregon, 1997, 299-302.
31. S. A. Cook, The Complexity of Theorem Proving Procedure, *Annual ACM SIGACT Symposium on Theory of Computing* **3** (1971) 151-158.
32. M. R. Garey and D. S. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness* (Freeman, San Francisco, 1979).
33. E. H. L. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines* (John Wiley, Chichester, UK, 1989).
34. P. J. M. v. Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications* (Kluwer, Dordrecht, Holland, 1987).
35. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1983).
36. MIT Technology Lab, GALib: A C++ Library of Genetic Algorithm Components, URL, <http://lancet.mit.edu/ga/>.