An Evolutionary Approach to Materialized Views Selection in a Data Warehouse Environment

Chuan Zhang, Xin Yao, *Senior Member, IEEE*, and Jian Yang

***Abstract*—A data warehouse (DW) contains multiple views accessed by queries. One of the most important decisions in designing a DW is selecting views to materialize for the purpose of efficiently supporting decision making. The search space for possible materialized views is exponentially large. Therefore heuristics have been used to search for a near optimal solution. In this paper, we explore the use of an evolutionary algorithm for materialized view selection based on multiple global processing plans for queries. We apply a hybrid evolutionary algorithm to solve three related problems. The first is to optimize queries. The second is to choose the best global processing plan from multiple global processing plans. The third is to select materialized views from a given global processing plan. Our experiment shows that the hybrid evolutionary algorithm delivers better performance than either the evolutionary algorithm or heuristics used alone in terms of the minimal query and maintenance cost and the evaluation cost to obtain the minimal cost.**

***Index Terms*—Data mining, data warehousing, evolutionary algorithms, materialized view selection.**

I. Introduction

DATA warehousing is an approach to the integration of data from multiple, possibly very large, distributed, heterogeneous databases and other information sources. A data warehouse (DW) is a repository of integrated information available for querying and analysis. To avoid accessing the original data sources and increase the efficiency of the queries posed to a DW, some intermediate results in the query processing are stored in the DW. These intermediate results stored in a DW are called materialized views. On a sufficiently abstract level, a DW can be seen as a set of materialized views over the data extracted from the distributed heterogeneous databases. There are many research issues related to DWs [1], among which materialized view selection is one of the most challenging ones. On one hand, materialized views speed up query processing. On the other hand, they have to be refreshed when changes occur to the data sources. Therefore, there are two costs involved in materialized view selection: the query processing cost and the materialized view maintenance cost. The question we are interested in is: what views should be materialized in order to make the sum of the query performance and view maintenance cost minimal?

Manuscript received November 28, 1999; revised July 22, 2001.

C. Zhang was with the School of Computer Science, University College, University of New South Wales (UNSW), Australian Defence Force Academy, Canberra, Australia. He is now with Asiaonline, Ltd., Canberra, Australia (e-mail: czhang@cs.adfa.edu.au).

X. Yao is with the School of Computer Science, University of Birmingham, Birmingham, U.K. (e-mail: x.yao@cs.bham.ac.uk).

J. Yang is with Infolab, Tilburg University, Tilburg, The Netherlands (e-mail: jian.yang@kub.nl).

Publisher Item Identifier S 1094-6977(01)08910-6.

The materialized view selection involves a difficult trade-off between query performance and maintenance cost.

- Materializing all the views in a DW can achieve the best performance but at the highest cost of view maintenance.
- Leaving all the views virtual will have the lowest view maintenance cost but the poorest query performance. The word “virtual” here means that no intermediate result will be saved in the DW.
- We can have some views materialized (e.g., have those shared views materialized), and leave others virtual. In this way we may achieve an optimal (or near optimal) balance between the performance gain and maintenance cost. Unfortunately the materialized view selection design problem has been proven to be NP-hard [2]. Heuristics have to be used in practice to find a near optimal solution to this problem.

The problem considered in this paper can be described as follows. Based on a set of frequently asked DW queries, select a set of views to materialize so that the total query and maintenance cost is minimized. Our problem is related to three different issues:

1) query optimization;
2) multiple query optimization;
3) materialized view selection.

The existing algorithms for solving one or more of the above optimization problems can be classified into four categories according to [3].

Deterministic algorithms usually construct or search a solution in a deterministic manner either by applying heuristics or by exhaustive search.

Randomized algorithms pursue a completely different approach. First, a set of moves are defined. These moves constitute edges between different solutions in the solution space. Two solutions are connected by an edge if and only if they can be transformed into one another by exactly one move. Each of the algorithms performs a random walk along the edges according to certain rules, terminating as soon as no more applicable ones exist or a time limit is exceeded. The best solution encountered so far will be the result.

Evolutionary algorithms use a randomized search strategy similar to biological evolution in their search for good solutions. Although an evolutionary algorithm resembles randomized algorithms in this aspect, the approach shows enough differences to warrant a consideration of its own. The basic idea is to start with a random initial population and generate offspring by random variations (e.g., crossover and mutation). The “fittest” members of the population survive the subsequent selection; the next generation is based on these. The algorithm terminates as soon as there is no further improvement over a period or after a predetermined number of generations. The fittest individual found is the solution.

Hybrid algorithms combine deterministic and randomized algorithms in various ways, e.g., solutions obtained by deterministic algorithms can be used as starting points for randomized algorithms or as initial population members for evolutionary algorithms; a deterministic algorithm can be applied to the best solution found by an evolutionary algorithm, etc.

In [4], the technique used was to reduce the solution space by considering only the relevant elements of the multidimensional lattice. Unfortunately, potential good solutions may be lost in the reduction process. In [2] and [5], the goal was to select an appropriate set of views that minimizes the total query response time and/or the cost of maintaining materialized views, given a limited amount of resources such as materialized time, storage space, or total view maintenance time. A greedy heuristic algorithm was used. The performance of the algorithm is highly problem dependent because the greedy nature of the algorithm makes it susceptible to poor local minima.

In [6], a framework and algorithms were described for analyzing the issues in materialized view selection in order to achieve the best combination of good query performance and low view maintenance cost. The 0–1 integer programming technique was used to obtain the optimal global processing plan and then a heuristic algorithm was employed to select the materialized views based on this global processing plan. It is worth noting that the optimal global processing plan found in such a way may not lead to the best set of materialized views. It is possible that another near optimal global processing plan may lead to a better set of materialized views. The two optimization problems should not be separated.

This paper adopts a holistic approach to materialized view selection and considers local processing plans, global processing plans, and materialized view selection in an integrated framework and algorithm [7]. The relationships among the three can be explored and exploited by our algorithms. Hence algorithms proposed in this paper are more likely to find better solutions than other methods, [2], [4], [5], [6], [8]–[10].

Although evolutionary algorithms have been applied to query optimization in recent years [3], [11], [12], because of its robustness and strong global search ability, few attempts have been made to make use of evolutionary algorithm's power in solving more complex problems, such as materialized view selection. In this paper, we propose several hybrid evolutionary and heuristic algorithms for optimizing global processing plans and materialized view selection. The hybrid algorithms combine evolutionary algorithm's power in global search with heuristic's ability in fine-grained local search to find a good set of materialized views. Our experimental results show that the hybrid algorithms performed better than the existing algorithms. They also performed better than either evolutionary algorithms or heuristic algorithms alone.

In this paper, the data model is based on selection-projection-join (SPJ) model rather than the multidimensional model.

The rest of this paper is organized as follows. Section II explains and formulates the problem of materialized view selection based on global processing plans. It also describes the cost model considered in this paper. Section III presents our algorithms. A two-level framework is introduced. The evolutionary algorithm and the heuristics used are explained in detail, including crossover, mutation, and selection used. The section also proposes methods for dealing with infeasible solutions during search. Section IV gives the experimental results using our evolutionary approach. Several hybrid algorithms are tested and evaluated using a number of randomly generated problems of variable sizes. Finally Section V concludes the paper by summarizing the main results and suggesting future work.

II. Materialized View Selection

Materialized view selection consists of three optimization problems, i.e., query optimization, multiple query optimization, and materialized view selection. It should be pointed out that a set of locally optimized queries may not be optimal anymore if multiple queries are considered together. Similarly, an optimal set of multiple queries does not guarantee the optimal selection of materialized views because a different set may lead to better materialized views. It is important to consider all three problems together in materialized view selection.

A. Query Optimization

A lot of research has been done on this topic. A valuable review can be found in [13] and [14]. In query optimization, *join* operation is one of the most expensive operations. For simplicity we only consider *join* operation in this paper. That is, query optimization will be regarded as join order optimization here.

Assume that a database D is given a set of relations $R_1, R_2, \ldots, R_m$. A local processing plan is defined as a query graph, in which all relations are leaf nodes and all operations (e.g., join, projection, and selection) are specified as its inner nodes. Since we only consider *join* operation, a local processing plan for a query can be regarded as a binary *join* tree that consists of all relations as its leaves and join operations as its inner nodes. The edges are labeled with the *join predicate* and *join selectivity.* The *join predicate* maps tuples from the Cartesian product of the adjacent nodes to $\{\text{false, true}\}$ depending on whether the tuple is to be included in the result or not. The *join selectivity* is the ratio between the included and total number of tuples.

The search space for query optimization is the set of all possible local processing plans. A point in the search space is one particular plan. Every point of the search space has a cost associated with it. Since there are lots of methods, such as nested loop, sort-merge, and hash loop, to perform a join operation, there exist several cost functions with respect to the processing tree. For example, the left trees in Fig. 1 denote nested loop join method. For nested loop join with no indices available, each tuple of the outer relation must be checked against every tuple of the inner relation, so the cost of $(R_1 \bowtie R_2)$ is $\|R_1\| * \|R_2\|$.

In the solution space, the left-deep processing trees have been of special interest to researchers. The left-deep tree is a tree where inner relation of each join is a base relation. In this paper,

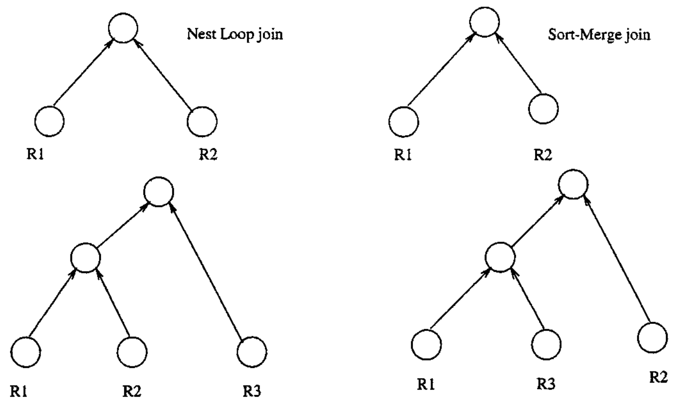

Fig. 1. Examples of join trees using nested loop join (left trees) and sort-merge join (right trees) operations.

TABLE I
POSSIBLE NESTING ORDERS FOR JOIN OPERATIONS [3]

Relations(n)	Processing Trees	solutions(Trees*n!)
1	1	1
2	1	2
3	2	12
4	5	120
5	14	1,680
6	42	30,240
7	132	665,280
8	429	17,297,280
9	1,430	518,918,400
10	4,862	17,643,225,600
11	16,796	670,442,572,800
12	58,786	28,158,588,057,600

we focus on left-deep trees. For n relations, Table I [3] illustrates how big the solution space is. In fact, it has been shown that query optimization is NP-hard [15].

In Fig. 1, the costs of two join trees in (1) may be different due to different join methods. The costs of two join trees in (2) are different as well although the structure is the same, because the join orders of relations are different. The left tree in (2) is $((R_1 \bowtie R_2) \bowtie R_3)$ while the right tree in (2) is $((R_1 \bowtie R_3) \bowtie R_2)$. Assume that the sizes for R1, R2, and R3 are 20, 30, and 40, respectively. The cost of the left tree in (2) is derived as follows. First calculate the cost of $(R_1 \bowtie R_2)$ as $||R_1||*||R_2|| = 20 * 30 = 600$. If $(R_1 \bowtie R_2)$ results in 20 tuples, then the cost of $((R_1 \bowtie R_2) \bowtie R_3)$ will be $20 * 40 = 800$. Hence the total cost is $600 + 800 = 1400$. However, the cost of the right tree in (2) will be different. The cost of $(R_1 \bowtie R_3)$ is $||R_1||*||R_3|| = 20*40 = 800$. If $(R_1 \bowtie R_3)$ results in 10 tuples, then the cost of $((R_1 \bowtie R_3) \bowtie R_2)$ will be $10 * 30 = 300$, i.e., a total cost of $800 + 300 = 1100$.

Given a set of processing plans for a query, the goal of query optimization is to find a processing plan with the lowest query processing cost. There has been some work on applying evolutionary algorithm to query optimization [3], [11], [12], [16], [17]. In this paper, query optimization is only part of a large problem—materialized view selection.

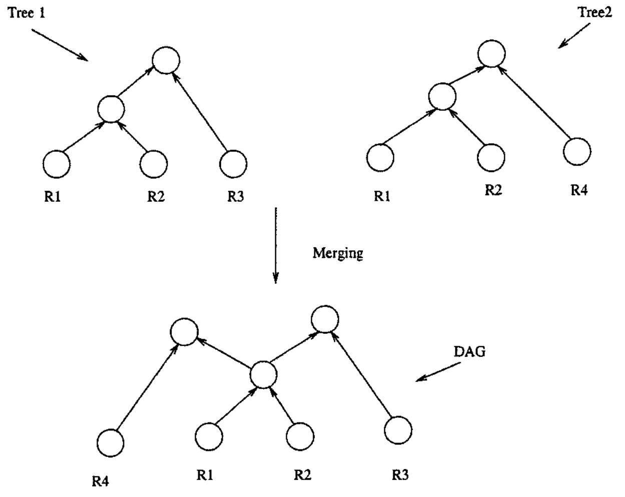

Fig. 2. Global processing plan should be a directed acyclic graph (DAG).

B. Multiple Query Optimization

A DW is a repository of integrated information available for querying analysis. One issue we have to deal with is multiple query processing. In [18]–[20], a systematic look at the problem has been presented.

Assume that a set of queries $Q = \{Q_1, Q_2, \ldots, Q_n\}$ are given. For every query Q_i, there exists at least one processing plan, called a local processing plan. A global/multiple processing plan for Q corresponds to a global plan that provides a way to compute the results for n queries. A global/multiple processing plan can be constructed by choosing one plan for each query and then merging them together. A locally optimal plan is referred to as the cost plan for processing a query Q_i individually. This corresponds to *query* optimization. The globally optimal plan is referred to as the global processing plan by merging the common parts of individual local plans.

The multiple query optimization (MQO) problem can be formulated as follows. Given n sets of local processing plans $P_1, P_2, \ldots, P_n$, with $P_i = \{p_{i1}, p_{i2}, \ldots, p_{ik_i}\}$ being the set of possible plans for query Q_i, $1 \leq i \leq n$, k_i is the number of local processing plans for Q_i. Find a global/multiple processing plan by selecting one plan from each P_i such that the cost (query cost) of the global/multiple processing plan is minimized.

In general, the union of locally optimal plans does not necessarily form a globally optimal plan. Hence, we cannot find the globally optimal plan by simply combining locally optimal plans. A heuristic algorithm is often needed in searching for a globally optimal plan. In [19] and [20], a heuristic search algorithm is proposed, which only examines a fraction of all possible global processing plans. Some potentially good plans may be lost. By using the evolutionary approach, our algorithms are capable of carrying out global search and looking into all possible combinations of individual plans.

When combining multiple query processing plans, i.e., multiple join trees, the produced global processing plan should be a directed acyclic graph (DAG) not a tree. This is shown in Fig. 2.

C. Materialized View Selection

In DW, selected information is extracted in advance and stored in a repository. A DW can therefore be seen as a set of materialized views defined over the sources. The problem we are dealing with now is how to select the views to be materialized so that the cost of query processing and view maintenance for all the nodes in a global processing plan is minimized.

An easy approach would be to use exhaustive search to find the optimal set of materialized views on the set of queries. However, this approach is impractical if the search space is big. It has been shown that materialized view selection is NP-hard [2]. Heuristic algorithms have to be used to trim the search space in order to get the results quickly [2], [4], [5]. However, the performance of a heuristic algorithm depends heavily on the quality of heuristics which may be difficult and/or costly to obtain in practice. Heuristic algorithms also get stuck easily in a local optimum. Compared with heuristic algorithms, evolutionary algorithms have many advantages, such as searching from a population of points using probabilistic transition rules. In order to avoid an exhaustive search in the whole solution space and obtain a better solution than that obtained by heuristic methods, we propose a new evolutionary approach to materialized view selection.

D. Cost Model of Materialized View Selection

1) Motivating Example: Our example is taken from a DW application which analyzes trends in sales and supply [6]. The relations and the attributes of the schema for this application are the following.

```
Item(I_id, I_name, I_price)
Part(P_id, P_name, I-id)
Supplier(S_id, S_name,P_id, City, Cost,
  Preference)
Sales(I_id, Month, Year, Amount)
```

There are five queries, as follows.

```
Q1: Select P_id, min(cost), max(cost)
From Part, Supplier
Where Part.P_id=Supplier.P_id
  And P_name in {“spark_plug,” “gas_kit”}
  Group by P_id
Q2: Select I_id,
  sum(amount*number*min_cost)
From Item, Sales, Part
Where I_name in {“MAZDA,” “NISSAN,”
  “TOYOTA”}
  And year=1996
  And Item.I_id=Sales.I_id
  And Item.I_id=Part.I_id
  And Part.P_id=
    (Select P_id, min(cost) as min_cost
    From Supplier
    Group by P_id)
  Group by I_id
Q3: Select P_id, month sum(amount)
From Item, Sales, Part
Where I_name in {“MAZDA,” “NISSAN,”
  “TOYOTA”}
    And year=1996
    And Item.I_id=Sales.I_id
    And Part.I_id=Item.I_id
    Group by P_id, month
Q4: Select I_id, Sum(amount *I_price)
From Item, Sales
Where I_name in {“MAZDA,” “NISSAN,”
  “TOYOTA”}
  And year=1996
  And Item.I_id=Sales.I_id
  Group by I_id
Q5: Select I_id, avg(amount*I_price)
From Item, Sales
Where I_name in {“MAZDA,” “NISSAN,”
  “TOYOTA”}
  and year=1996
  and Item.I_id=Sales.I_id
  Group by I_id.
```

Fig. 3 gives a possible global query processing plan for the five queries listed above, in which the local access plan for individual queries are merged based on the shared operations on common data sets. We call it the multiple view processing plan (MVPP).

The query access frequencies are labeled on the top of each query node. For simplicity, we assume that all the base relations `Item, Sales, Part`, and `Supplier` are updated only once for a certain period of time. In Fig. 3, we abbreviate one thousand as “k,” one million as “m,” and one billion as “b.” For example, the cost for obtaining `tmp3` by using `tmp1` is 36 m.

Now we have to decide which node(s) to materialize so that the total query and view maintenance cost is minimal. It is obvious from this graph that we have several alternatives for choosing the set of materialized views: e.g.,

1) materialize all the application queries;
2) materialize some of the intermediate nodes (e.g., `tmp1, tmp3, tmp7`, etc.);
3) leave all the nonleaf nodes virtual.

The cost for each alternative can be calculated in terms of query processing and view maintenance.

In order to calculate the cost, we make the following assumptions.

- There are 1 k tuples in the `Item` relation.
- On average, each item has ten parts, therefore, there are ten k tuples in `Part` table.
- There are 50 k tuples in `supplier` table.
- There are ten years worth of sales in the `Sales` table from 1987 to 1996. On average, each item is sold 100 times a month resulting in 12 m entries in the sales relation.
- The cost of answering a query Q is proportional to the number of rows in the table used to construct Q.
- The methods for implementing select and join operations are linear search and nested loop.

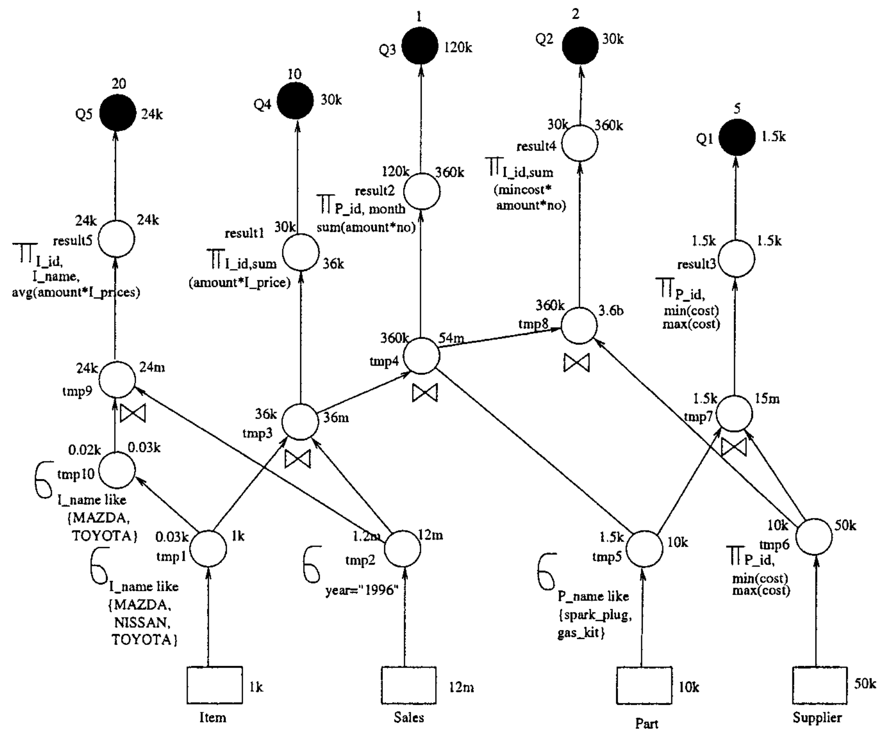

Fig. 3. Motivating example.

TABLE II
COSTS FOR DIFFERENT VIEW MATERIALIZATION STRATEGIES

Materialized views	Cost of query processing	Cost of maintenance	Total cost
Item,Sales,Part,Supplier	$8b980m860k$	0	$8b980m860k$
tmp3, tmp4, tmp8	$7b201m547k$	$1b350m125k$	$8b551m672k$
tmp3, tmp5,	$416m747k$	$16b32m204k$	$16b448m951k$
tmp3, tmp4, tmp7	$7b276m497k$	$1b220m55k$	$8b496m552k$
tmp3, tmp7	$8b281m547k$	$126m122k$	$8b407m669k$
result1,result2,result3,result4	$1m447k$	$17b384m934k$	$17b386m381k$

Based on the above assumptions, the cost for each operation node in Fig. 3 is labeled at the right-hand side of the node.

Now we can calculate the costs of different view materialization strategies. Suppose there are some materialized intermediate nodes. For each query, the cost of query processing is query frequency multiplied by the cost of query access from the materialized node(s). The maintenance cost for materialized view is the cost used for constructing this view (here we assume that recomputing is used whenever an update of involved base relation occurs). The total cost for an MVPP is the sum of all query processing and view maintenance costs. Our goal is to find a set of nodes to be materialized so that the total cost is minimal. Table II gives some materialized view design strategies and their costs. There are many other possible strategies for materializing views in this example.

2) Cost Model: A global processing plan is a DAG and can be described by $(V, A, C_a^q(v), C_m^r(v), f_q, f_u)$ where V is the set of vertices and A is the set of arcs over V. The DAG is constructed as follows.

1) Create a vertex for every relational algebra operation in a query tree, every base relation, and every distinct query.
2) For $v \in V$, $T(v)$ is the relation generated by corresponding vertex v. $T(v)$ can be a base relation, intermediate result while processing a query or the final result for a query.
3) For any leaf vertex v (that is, it has no edges coming into the vertex), $T(v)$ corresponds to a base relation. Let L be a set of leaf nodes.
4) For any root vertex v (that is, it has no edges going out of the vertex), $T(v)$ corresponds to a global query. Let R be a set of root nodes.
5) If the base relation or intermediate result relation $T(u)$ corresponding to vertex u is needed for further processing at another node v introduce an arc $u \longrightarrow v$.
6) For every vertex v, let $S(v)$ denote the source nodes which have edges pointed to v. For any $v \in L$, $S(v) = \emptyset$. Let $S^*\{v\}$ be the set of descendants of v.
7) For every vertex v, let D(v) denote the destination nodes to which v is pointed. For any $v \in R$, $D(v) = \emptyset$.
8) For $v \in V$ $C_a^q(v)$ is the cost of query q accessing $T(v)$, $C_m^r(v)$ is the cost of maintaining $T(v)$ based on changes to the base relation $S^*(v) \bigcap R$, if $T(v)$ is materialized.
9) f_q and f_u are query and maintenance frequency, respectively.

A linear cost model [21] is used to calculate the cost of query Q. The cost of answering Q is the number of rows in the table that query Q_A used to construct Q.

Let M be a set of materialized views, $C_{q_i}(M)$ be the cost to compute q_i from the set of materialized views M, and $C_m(v)$ be the cost of maintenance when v is materialized. Then the total query processing cost is $\sum_{q_i \in Q} f_{q_i} C_{q_i}(M)$. The total maintenance cost is $\sum_{v \in M} f_u C_m(v)$. The total cost of the materialized views M is

$$\sum_{q_i \in Q} f_{q_i} C_{q_i}(M) + \sum_{v \in M} f_u C_m(v).$$

Given a global processing plan, the objective is to minimize the above cost.

E. Problem Specification

The materialized view selection problem can be formulated as follows. Given n sets of local processing plans $P_1, P_2, \ldots, P_n$, where $P_i = \{p_{i1}, p_{i2}, \ldots, p_{ik_i}\}$ is the set of possible plans for query Q_i, $1 \le i \le n$, and k_i is the number of local processing plans for Q_i, select a set of views to be materialized over a global processing plan by "merging" n local processing plans (one out of each set P_i) such that the sum of query and maintenance cost is minimized. It is worth emphasizing that this problem is different from MQO [6].

1) MQO is to find an optimal processing plan for multiple queries executed at the same time by sharing some temporary results which are common subexpressions, while our problem is to find a set of relations (which can be any intermediate result from query processing) to be materialized so that the total cost (query processing and view maintenance) is minimized.
2) In MQO, a global processing plan is derived from the idea that temporary result sharing should be less expensive compared to a serial execution of queries. However, this may not be true for every possible database state. For example, sharing a temporary result may prove to be a bad decision when indices on base relations are defined. The cost of processing a selection through an index or an existing temporary result clearly depends on the size of these two structures. In our problem, if an intermediate result is materialized, we can establish a proper index on it afterwards if necessary. Therefore, it is guaranteed that there is a performance gain if an intermediate result is materialized. If the intermediate result happens to be a common subexpression which can be shared by more than one query, then there is a view maintenance gain as well.
3) In MQO, the ultimate goal is to achieve the best performance, while our problem has to take into consideration both query and view maintenance cost.
4) In MQO, the input is a set of queries and the output is a globally optimal plan. In our case, the input is a set of global queries and their access frequencies, and a set of base relations and their update frequencies. The output is one or more global processing plans with materialized views that minimize the total query and maintenance cost.

Just like we cannot guarantee to get the globally optimal processing plan from the union of locally optimal plans, we cannot guarantee to get the optimal selection of materialized views from the optimal global processing plan. This is because the optimization of global processing plans does not consider the maintenance cost. For example, an optimal global processing plan may have optimal query cost 333 m. Based on this global processing plan, we may obtain materialized views with the total cost (query cost and maintenance cost) of 370 m. However, with a nonoptimal global processing plan that has query cost 350 m, we may get materialized views with a total cost of 360 m, which is less than 370 m.

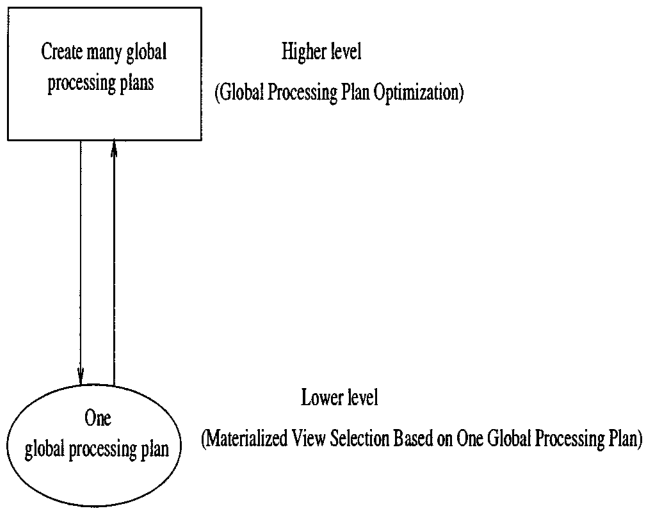
Create many global processing plans
Higher level
(Global Processing Plan Optimization)
One global processing plan
Lower level
(Materialized View Selection Based on One Global Processing Plan)

Fig. 4. Structure of our algorithms.

```
BEGIN
   Generate the initial population, G(0);
   Evaluate all individuals in G(0);
   t:=0;
   REPEAT
         t:=t+1;
         Select G(t) from G(t-1);
         Alter G(t) using variation operat
         Evaluate all individuals in G(t);
   UNTIL a satisfactory solution is found;
END;
```

Fig. 5. Abstract framework of evolutionary algorithms.

III. Algorithms for Materialized View Selection

Our algorithms are designed based on the two-level structure as shown in Fig. 4. The hierarchical structure helps to make a large problem manageable. The higher level algorithm searches for good global processing plans from local processing plans based on queries. The lower level algorithm selects the best set of materialized views with the minimal total cost for a particular global processing plan. In principle, any optimization algorithms can be used at the higher and lower levels. In practice, however, a compromise often needs to be made between the speed of optimization and the quality of the solutions. This paper investigates different hybrid algorithms where an evolutionary/heuristic algorithm is used at the higher/lower level.

When an evolutionary algorithm is implemented at the higher level to search for global processing plans, the fitness of each in-

1. Input a global processing plan represented by a DAG.
2. Use a certain graph traversal strategy, such as breadth-first, depth-first or other problem-specific strategies, to traverse through all nodes in the DAG and produce an ordered list of nodes.
3. Create a binary string according to this order, where 0 indicates that the corresponding node is not materialized and 1 represents that the corresponding node is materialized. The binary string is also called the mapping array.

Fig. 6. Mapping from a DAG (i.e., genotype) to a binary string (i.e., phenotype).

dividual (i.e., global processing plan) is defined by the total cost of the best set of materialized views (i.e., the outcome from the lower level optimization). If another evolutionary algorithm is implemented at the lower level, the fitness of the best individual from the lower level population will be used. More details about a lower level evolutionary algorithm can be found in [22] where the higher level optimization problem was not addressed.

A. Evolutionary Algorithms

Evolutionary algorithms have been shown to solve many real-world problems with success [23]–[27]. They use population-based stochastic search strategies and are unlikely to be trapped in a poor local optimum. They make few assumptions about a problem domain yet are capable of incorporating domain knowledge in the design of chromosome representation and variation operators. They are particularly suited for large and complex problems where little prior knowledge is available. The results presented in the next section illustrate that a properly designed evolutionary algorithm can be a very promising method for materialized view selection. Fig. 5 shows an abstract framework of evolutionary algorithms. Details of our implementation are described in the following subsections.

B. Representation of Solutions

Representation is one of the key issues in problem solving. Good representations often lead to a more efficient algorithm for solving a problem. Different problems usually require different representations. In our two-level structure introduced earlier, two different representations are needed to represent global processing plans and materialized views.

1) Representation of Global Processing Plans: Given n queries $Q_1, Q_2, \ldots, Q_n$, a global processing plan (i.e., an individual in a higher level evolutionary algorithm) can be represented by a vector of n integers, $P_{1i}, P_{2j}, \ldots, P_{kn}$, where P_{kn} indicates the kth local processing plan for query Q_n. For example, assume that there are three queries, Q_1, Q_2, and Q_3, and the respective number of local processing plans are 12, 120, and 120. Then {[4], [89], [70]} represents a global processing plan consisting of the fourth processing plan for Q_1, the 89th processing plan for Q_2, and the 70th processing plan for Q_3 where the range for each gene is [1 ... 12], [1 ... 120], and [1 ... 120], respectively.

2) Representation of Materialized Views: The representation of materialized views in the lower level optimization is based on DAGs. Each DAG is encoded as a binary string. Fig. 6 shows how to map a DAG into a binary string. One of the reasons of using binary strings, rather than graphs directly is to simplify the implementation of evolutionary algorithms (including crossover, mutation, and selection). It is our future work to investigate evolutionary algorithms that will operate on DAGs directly.

For example, the breadth-first traverse of the DAG in Fig. 3 results in the following ordered list: {[Q5,0], [Q4,0], [Q3,0], [Q2,0], [Q1,0], [result5,0], [result1,0], [result2,0], [result4,0], [result3,0], [tmp9,0], [tmp3,0], [tmp4,0], [tmp8,0], [tmp7,0], [tmp10,0], [tmp1,0], [tmp2,0], [tmp5,0], [tmp6,0]}. A binary string of {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0} means that no node is materialized. A string of {0,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1} means that nodes {Q4, Q1, result5, tmp2, tmp5, and tmp6} are materialized, but others are not.

C. Fitness Functions in our Evolutionary Algorithms

Since the objective in our cost model is to minimize the sum of query and maintenance cost while the fitness function of an evolutionary algorithm is usually defined as maximization, we have applied the following simple transformation to define the fitness function from the cost

$$f(x) = \begin{cases} C_{\max} - c(x), & \text{when } c(x) < C_{\max} \\ 0, & \text{otherwise} \end{cases}$$

where $c(x)$ denotes the cost function and $f(x)$ is the fitness function.

There are a lot of ways of choosing the coefficient $C_{\max}$. It can be set to the largest $c(x)$ value in the current population or the largest in the last k generations.

If an evolutionary algorithm is used at the lower level, each individual in a population represents a set of materialized views. Its fitness depends on the total query and maintenance cost as described above.

If an evolutionary algorithm is used at the higher level, each individual in a population represents a global processing plan. Its fitness is determined as follows.

1) Find a set of materialized views that minimize the total query and maintenance cost for this global processing plan. This step involves optimal selection of materialized views on a fixed global processing plan, i.e., the lower level optimization in Fig. 4.
2) Transform this minimal cost to the fitness of the individual.

D. Crossover

Crossover encourages information exchange among different individuals. It helps the propagation of useful genes in the population and assembling better individuals. One-point crossover is used in our evolutionary algorithms for its simplicity and effectiveness in our case.

In a lower level evolutionary algorithm, the crossover is implemented as a kind of cut-and-swap operator [28]. For example, given two individuals

$$L_1 = 1\,100\,100|0\,100\,100\,001\,111$$

and

$$L_2 = 0\,100\,110|1011000100111$$

where L_1 indicates that nodes {Q5, Q4, Q1, result4, tmp3, tmp1, tmp2, tmp5, and tmp6} are materialized and L_2 means that nodes {Q4, Q1, result5, result2, result3, tmp9, tmp7, tmp2, tmp5, and tmp6} are materialized. Assume the crossover point (indicated by symbol |) is chosen at random as seven, between one and 20. Then the two offspring after crossover are

$$L_1' = 1\,100\,100|1\,011\,000\,100\,111$$

and

$$L_2' = 0\,100\,110|0100100001111$$

where L_1' indicates that nodes {Q5, Q4, Q1, result2, result3, tmp9, tmp7, tmp2, tmp5, and tmp6} are materialized and L_2' shows that nodes {Q4, Q1, result5, result4, tmp3, tmp1, tmp2, tmp5, and tmp6} are materialized. Two new sets of materialized views are generated which have inherited genes from both parents.

In a higher level evolutionary algorithm, one-point crossover is such implemented that crossover points can only be between genes, but not in a gene. For example, given two individuals

$$L_1 = [4][20][30]|[10][99]$$

and

$$L_2 = [5][30][21]|[40][80]$$

where the symbol | indicates the crossover point. The two offspring are

$$L_1' = [4][20][30][40][80]$$

and

$$L_2' = [5][30][21][10][99].$$

E. Mutation

Although crossover can put good genes together to generate better offspring. It cannot generate new genes. Mutation is needed to create new genes that may not be present in any member of a population and enables the algorithm to reach all possible solutions (in theory) in the search space.

Mutation in a lower level evolutionary algorithm is implemented as a bit-flipping operator. Given an individual

$$L = 11\,001\,000\,100\,100\,001\,111.$$

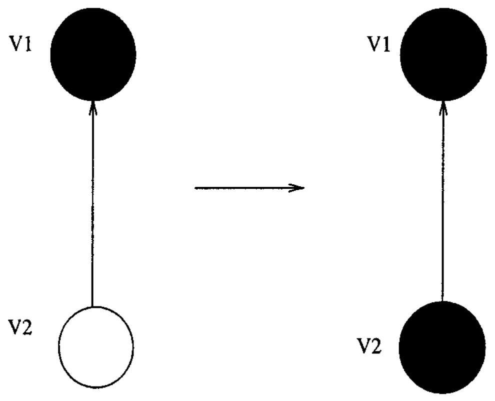

Fig. 7. Example of invalid result.

A random position between onr and 20 will be generated first. Say it is 16. Then the 16th bit will be flipped from 0 to 1 with a probability to produce the offspring

$$L' = 11\,001\,000\,100\,100\,011\,111.$$

Mutation in a higher level evolutionary algorithm is implemented differently due to a different chromosome representation. Rather than flipping a bit, a random number in a certain range will be generated as a new gene in an individual. For example, given an individual

$$L_1 = [4][20][30]|[10][99].$$

Assume the third gene is randomly selected for mutation. Further assume that the third gene has a total of 120 possible processing plans. Then a random number between one and 120 is generated, say 16. The offspring after mutation will be

$$L_1' = [4][20][16]|[10][99].$$

F. Dealing With Invalid Solutions

In the lower level of optimization as shown in Fig. 4, "invalid" solutions may be generated during search, e.g., by crossover and/or mutation. Fig. 7 shows such an example of possible invalid solutions. If `v2` has the same ancestors (excluding `v1`) as `v1`, it is unnecessary to materialize `v2`. Any solutions that materialize `v2` in such a case will be regarded as "invalid." In some cases, invalid solutions can be prevented from being generated or repaired after generation. We will describe how we deal with invalid solutions in the following subsections.

1) Claim: In Fig. 7, let `v1` and `v2` be two nodes in a global processing plan represented by a DAG. If `v1` is a parent of `v2` and `v2` has the same ancestors (excluding `v1`) as `v1`, then there is no need to materialize `v2` after `v1` has been materialized.

Proof: Let $C_{q_i}(M)$ be the cost of computing q_i from the set of materialized views M. If both `v1` and `v2` are materialized, then the total cost is

$$C_1 = \sum_{q \in O_{v_1}} f_q(q) C_q(v_1) + \sum_{q \in O_{v_2}} f_q(q) C_q(v_2) + \sum_{r \in I_{v_1}} f_u(r) C_m^r(v_1) + \sum_{r \in I_{v_2}} f_u(r) C_m^r(v_2). \quad (1)$$

Because `v1` and `v2` have the same parents, (1) can be rewritten as

$$C_1 = \sum_{q \in O_{v_1}} f_q(q) * (C_q(v_1) + C_q(v_2)) + \sum_{r \in I_{v_1}} f_u(r) * C_m^r(v_1) + \sum_{r \in I_{v_2}} f_u(r) * C_m^r(v_2).$$

Since `v1` is materialized before `v2`, `v2` cannot be reached by any queries, i.e., $C_q(v_2) = 0$. Hence the above equation becomes

$$C_1 = \sum_{q \in O_{v_1}} f_q(q) * C_a^q(v_1) + \sum_{r \in I_{v_1}} f_u(r) * C_m^r(v_1) + \sum_{r \in I_{v_2}} f_u(r) * C_m^r(v_2). \quad (2)$$

If we only materialize `v1`, the total cost will be

$$C_2 = \sum_{q \in O_{v_1}} f_q(q) * C_a^q(v_1) + \sum_{r \in I_{v_1}} f_u(r) * C_m^r(v_1).$$

Because $C_1 > C_2$, it is clear that materializing both `v1` and `v2` has a higher cost than materializing `v1` alone. This is not surprising because materializing `v2` increases the maintenance cost without reducing any query costs. Therefore, in the presence of the materialized view `v1`, we should not materialize `v2` under the conditions mentioned above.

For example, given the global processing plan in Fig. 3, assume that an offspring{`00 000 000 001 100 101 100`} is generated after crossover and mutation. The offspring indicates that {`tmp9, tmp3, tmp7, tmp1, tmp2`} should be materialized. However, {`tmp3, tmp9`} are ancestors of {`tmp1, tmp2`}. The offspring is invalid because it has a higher cost than another individual {`00 000 000 001 100 100 000`} which has fewer materialized views. It could be argued that individuals having higher costs might not harm optimization. They do reduce the efficiency of optimization.

There are several methods for dealing with an invalid solution. One is to constrain crossover and mutation such that only valid solutions are generated. Another is to allow invalid solutions but penalize them by introducing a penalty term in the fitness function. The first method can prevent invalid solutions from being generated, but may introduce a complex search landscape because valid regions may be separated by invalid regions. Moving from one valid region to the other may be difficult. Good solutions in a different valid region may not be found. The second method introduces another difficult problem, i.e., how to select an optimal penalty coefficient in order to strike the right balance between minimizing the cost and minimizing the penalty.

We use the repair approach in this paper. That is, invalid solutions are allowed to be generated, but will be repaired into valid ones before evaluating their fitness. Fig. 8 shows the repair algorithm. The algorithm was designed according to the *claim* proved in the previous section.

```
REPEAT
If a parent-child pair share the same ancestors and they are materialized, then
1. unmaterialize the child node,
2. re-calculate the total cost, and
3. replace the old solution by the new one.
UNTIL no materialized parent-child pair share the same ancestors
```

Fig. 8. Repair algorithm.

G. Selection

Selection in evolutionary algorithms determines the probability of individuals being selected for reproduction. The principle here is to assign higher probabilities to fitter individuals. Tournament selection is used in our algorithms because it can better maintain a relatively smooth selection pressure over generations. It also facilitates future parallel implementation of our algorithms since it does not require global information.

Tournament selection is implemented by conducting tournaments among a number of randomly selected individuals. The winner is selected to survive for reproduction. The tournament size (i.e., the number of individuals involved in a tournament) is an important parameter that determines the selection pressure. A large size introduces a strong pressure, which often leads to fast convergence to a local optimum. The quality of the local optimum depends quite a lot on the start conditions of the algorithm. A small tournament size introduces a weak selection pressure, which often implies slow convergence but the algorithm is less likely to be trapped in a poor local optimum. Following suggestions from the literature [29] and our own preliminary experiments, the tournament size was chosen to be between four and seven in our study.

IV. Experimental Studies

All our experiments were performed under SUN OS 5.5. The simulation software was built on the basis of the Simple Genetic Algorithm [30] and GAlib [31]. In particular, we have implemented our lower level evolutionary algorithm based on the Simple Genetic Algorithm program [30] which is a C-language translation and extension of the original Pascal code [28]. Our higher level evolutionary algorithm was implemented using the GAlib [31], which is a library containing different chromosome representation schemes, evolutionary operators, selection schemes, etc.

A. Evolving Materialized Views Based on a Given Global Processing Plan

The lower level algorithm shown in Fig. 4 optimizes materialized view selection on a given global processing plan. Yang *et al.* [6] have recently proposed a very good heuristic algorithm for optimal selection of materialized views on a fixed global processing plan. Fig. 9 compares the results from Yang *et al.*'s algorithm [6] and those produced by our evolutionary algorithm on a number of randomly generated problems with up to 50 queries. The number of source relations involved in each query varies from three to eight. The nodes of the DAG varies from 24 to 200. We compare our algorithm to Yang *et al.*'s [6] because their

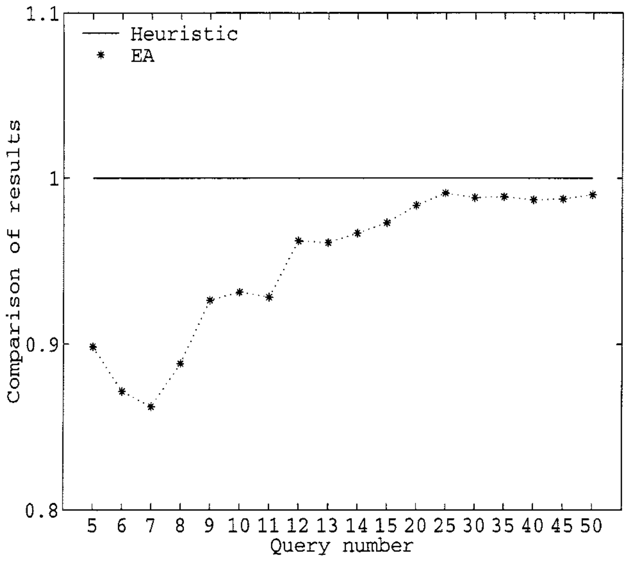

Fig. 9. Comparison between our evolutionary algorithm and Yang *et al.*'s heuristic algorithm [6].

algorithm is one of the best existing ones using the same cost model.

It is clear from Fig. 9 that the evolutionary algorithm outperformed the heuristic algorithm consistently. The advantage of the evolutionary algorithm was most prominent when the number of queries was small. This large advantage decreased gradually as the number of queries increased. However, since the total cost increases dramatically as the number of query grows, a little difference in Fig. 9 may translate into a large amount of cost saving in practice.

B. Evolving Global Processing Plans and Materialized Views

Without loss of generality, query processing plans used in our experiments are generated at random as a set of left-deep binary trees. It has been argued that good solutions are likely to exist among these trees [3]. The experiments were run over randomly generated queries. These queries share at least two relations. For our experiments, we have generated up to 60 queries. Each query has from six to 720 different query processing plans. The solution space for such problems is huge. For example, with 10 queries, there are 720^{10} possible global processing plans. For each global processing plan, there are 40 join nodes on average and the space of possible sets of materialized views is $O(2^{40})$. Hence, the size of the whole solution space is $720^{10}(O(2^{40}))$.

In order to evaluate and gain a better understanding of our evolutionary algorithm, we compare it with the following heuristic method for generating a near optimal global processing plan.

1) Create optimal global processing plans by merging locally optimal plans.
2) Compare the total query cost of each global processing plan and select the one which gives the lowest query cost.

Fig. 10 shows the results produced by different hybrid algorithms, where EA1 denotes the higher level evolutionary algorithm for optimizing global processing plans, EA2 represents the lower level evolutionary algorithm for materialized view selection given a global processing plan, H1 indicates the higher

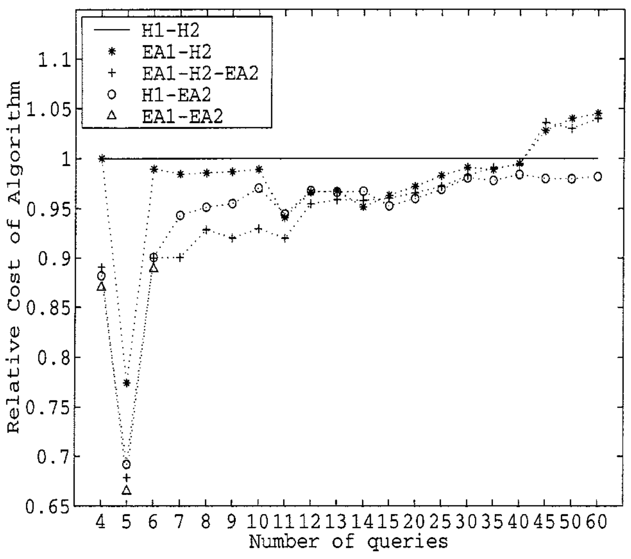

Fig. 10. Comparison of different hybrid algorithms.

level heuristic algorithm described above, and H2 is the lower level heuristic algorithm used in [6]. Since most heuristic algorithms designed for materialized view selection differ in the cost models and problem formulations used, meaningful comparisons with them are difficult. Yang *et al.*'s algorithm [6] was chosen as H2 because they used the same cost model and problem formulation as ours. Their algorithm is also one of the best under such a model.

The implementation details of each hybrid algorithm are as follows.

1) EA1-EA2 represents the algorithm where EA1 is used at the higher level for optimizing global processing plans and EA2 is used at the lower level for materialized view selection.
2) EA1-H2 uses H2 to select materialized views for each global processing plan.
3) EA1-H2-EA2 is similar to EA1-H2, but applies EA2 to further improve the best global processing plan found by EA1-H2.
4) H1-EA2 uses H1 for optimizing global processing plans and EA2 for materialized view selection.
5) H1-H2 uses H1 for optimizing global processing plans and H2 for materialized view selection for each global processing plan.

All results shown in Fig. 10 have been averaged over five independent runs. The costs have been normalized using the H1-H2 algorithm as the reference. From the results shown in Fig. 10, we can observe that EA1-H2-EA2 and EA1-EA2 seemed to perform very well when the number of queries was small. As the number of queries increased, H1-EA2 emerged as the best performer among all hybrid algorithms. It seemed to cope with the increasing number of queries very well without any sudden increase in the total cost.

Fig. 10 shows that hybrid algorithms outperformed the H1-H2 heuristic algorithm in almost all cases, which illustrates the advantage of having an evolutionary algorithm to search a larger part of a huge space and thus find a better solution. However, using both EA1 and EA2 proved to be very CPU

TABLE III
COMPARISON OF H1-H2 AND EA1-H2 BASED ON 30 INDEPENDENT RUNS OF EACH EXPERIMENT

Number of queries	H1-H2		EA1-H2		H1-H2 − EA1-H2
	Mean	Std Dev	Mean	Std Dev	T-test
5	50250	0	50266	144	-0.98
10	100854	0	99818	2594	2.19
20	217818	0	215226	5865	2.42
40	258238	0	255506	5401	2.77
60	301748	0	299583	5356	2.11

TABLE IV
COMPARISON OF H1-EA2 AND EA1-H2 BASED ON 30 INDEPENDENT RUNS OF EACH EXPERIMENT

Number of queries	H1-EA2		EA1-H2		H1-EA2 − EA1-H2
	Mean	Std Dev	Mean	Std Dev	T-test
5	44903	0	50266	144	-327.51
10	97877	0	99818	2594	-4.1
20	207494	0	215226	5865	-7.22
40	247940	0	255506	5401	-7.67
60	292456	0	299583	5356	-7.29

intensive and time-consuming. To evaluate an individual in EA1, EA2 had to be executed. This is the reason why we did not carry out experiments with EA1-EA2 for problems with seven or more queries. Hybrid algorithms that employ a mixture of heuristic and evolutionary algorithms should be adopted to strike a balance between the solution quality and the computation time needed to achieve such quality.

Out of the two hybrid algorithms, EA1-H2 and H1-EA2, H1-EA2 seemed to perform better and more consistent. Although the additional EA2 after EA1-H2 (i.e., the EA1-H2-EA2 algorithm) helped to improve EA1-H2's results when the number of queries was smaller than 13; it did not help much for problems with a larger number of queries. The difference in results from EA1-H2 and H1-EA2 suggests that the best hybrid algorithm should employ a heuristic algorithm to find a good global processing plan and use an evolutionary algorithm to explore possible materialized views on it. EA1-H2 did not perform as well probably because of inaccurate fitness evaluation by H2. In EA1, the fitness of each individual (i.e., a global processing plan) is evaluated by running H2 and using its result to calculate the fitness. Because H2 is a heuristic algorithm only, its solution may be far away from the actual optimum. When this happens, a good global processing plan may be "miscalculated" as poor and thus would be unable to survive in the EA1 population. More analysis is needed in the future to confirm whether this is the primary reason for EA1-H2's poor performance.

To better understand and further evaluate different hybrid algorithms, i.e., H1-EA2 and EA1-H2, extensive experiments have been carried out to compare them using a statistically sound method. Each experiment reported below has been repeated independently 30 times with the number of queries being five, 10, 20, 40, and 60. The results are summarized in

TABLE V
COMPARISON OF TIME TO FIND THE GLOBAL OPTIMAL SOLUTION BY DIFFERENT ALGORITHMS

Algorithm	6 queries	7 queries	8 queries
H1-H2	5 Secs	50 Secs	1.2 Mins
EA1-H2	30 Secs	10.1 Mins	20 Mins
EA1-H2-GA2	1 Mins	10 Mins	25 Mins
H1-EA2	35 Secs	10 Mins	22 Mins
EA1-EA2	4 Mins	2 Hours	7 Hours
Exhaustive	5 Mins	2.5 Hours	25.2 Hours

Tables III and IV. As shown in Table III, EA1-H2 performed significantly better than H1-H2 on all but the first problem when the number of queries was five, in which case no statistically significant difference was found. According to Table IV, H1-EA2 performed significantly better than EA1-H2 for all test cases. It is clear that H1-EA2 is the best hybrid algorithm for all the problems we have tested.

To compare execution time of different algorithms using the exhaustive search algorithm as a benchmark, we have carried out additional experiments to evaluate the time taken by each algorithm to find the global optimal solution. Table V summarizes the experimental results. It is clear from the table that hybrid algorithms are able to find the global optimal solution within a reasonable amount of time.

V. CONCLUSION

The materialized view selection based on multiple query processing plans is a hard combinatorial optimization problem. Previous work has either assumed a fixed global processing plan for materialized view selection or only examined multiple query optimization without considering materialized view selection. We have argued in this paper that a good selection of materialized views can only be found by taking a holistic approach and considering the optimization of both global processing plans and materialized view selection. A two-level structure for materialized view selection was proposed. It has facilitated greatly the development of several hybrid algorithms.

In this paper, we have studied several hybrid heuristic and evolutionary algorithms. Pure evolutionary algorithms were found to be impractical due to their excessive computation time. Pure heuristic algorithms were unsatisfactory in terms of the quality of the solutions they found. Hybrid algorithms that combine the advantages of heuristic and evolutionary algorithms seem to perform the best in our experiments. Our experimental results show that applying an evolutionary algorithm to either global processing plan optimization or materialized view selection for a given global processing plan can reduce the total query and maintenance cost significantly. Our study also shows that simply combining or merging optimal local processing plans will not produce an optimal global processing plan in most cases. Finding an optimal global processing plan with optimal materialized views requires a two level hierarchy as described by Fig. 4.

While our hybrid algorithms perform better than the heuristic algorithm in terms of cost savings, they often require longer computation time. While the heuristic algorithm typically took seconds to run, a hybrid algorithm typically took minutes, or even hours to run. Finding the suitable trade-off between the computation time and the cost saving will be a topic for future studies.

Once a data warehousing design is completed and implemented, it will be used frequently and may last for a long time. Hence it is very important to optimize the design as much as possible, even if this means a relatively long design time. It is time that is well spent. In this case, the extra computation time incurred by employing an evolutionary algorithm during the design stage is well justified. It is expected to lead to substantial cost savings when the DW is in use.

References

[1] J. Widom, "Research problems in data warehouse," in *Proc. 4th Int. Conf. Inform. Knowledge Manage.*, 1995, pp. 25–30.

[2] H. Gupta and I. S. Mumick, "Selection of views to materialize under a maintenance cost constraint," in *Proc. Int. Conf. Database Theory (ICDT)*, 1999, pp. 453–470.

[3] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *Very Large Data Base J.*, vol. 6, no. 3, pp. 191–208, 1997.

[4] E. Baralis, S. Paraboschi, and E. Teniente, "Materialized view selection in a multidimensional database," in *Proc. 23rd Int. Conf. Very Large Data Base (VLDB)*, 1997, pp. 156–165.

[5] H. Gupta *et al.*, "Index selection for olap," in *Proc. Int. Conf. Data Eng. (ICDE)*, 1997, pp. 208–219.

[6] J. Yang, K. Karlapalem, and Q. Li, "Algorithm for materialized view design in data warehousing environment," in *Proc. 23th Int. Conf. Very Large Data Bases (VLDB)*, 1997, pp. 136–145.

[7] C. Zhang, X. Yao, and J. Yang, "Evolving materialized views in data warehouse," in *Proc. IEEE Congr. Evolutionary Computat.*, Washington, D.C., 1999, pp. 823–829.

[8] W. J. Labio, D. Quass, and B. Adelberg, "Physical database design for data warehouses," in *Proc. Int. Conf. Data Eng. (ICDE)*, 1997, pp. 277–288.

[9] K. A. Ross, D. Srivastava, and S. Sudarshan, "Materialized view maintenance and integrity constraint checking: Trading space for time," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1996, pp. 447–458.

[10] D. Theodorators and T. K. Sellis, "Data warehouse configuration," in *Proc. 23rd Int. Conf. Very Large Data Bases (VLDB)*, 1997, pp. 126–135.

[11] B. Kristin, M. C. Ferris, and Y. Ioannidis, "A genetic algorithm for database query optimization," Univ. Wisconsin, Madison, Tech. Rep. TR1004, 1991.

[12] M. Gregory, "Genetic algorithm optimization of distributed database queries," in *Proc. ICEC*, 1998, pp. 271–276.

[13] Y. E. Ioannidis, "Query optimization," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 121–123, Mar. 1996.

[14] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Principles Database Syst. (PODS)*, June 1998, pp. 34–43.

[15] C. Wang and M.-S. Chen, "On the complexity of distributed query optimization," *IEEE Trans. Knowl. Data Eng.*, vol. 8, pp. 650–662, Aug. 1996.

[16] A. Ho and G. Lumpkin, "The genetic query optimizer," in *Genetic Algorithms at Stanford 1994*, J. R. Koza, Ed. Stanford, CA: Stanford Univ., 1994, pp. 67–76.

[17] M. Stillger and M. Spiliopoulou, "Genetic programming in database query optimization," in *Proc First Annu. Conf. Genetic Programming*, Stanford, CA, July 1996.

[18] U. Charkravarthy and J. Minker, "Processing multiple queries in database systems," *Database Eng.*, vol. 5, no. 3, pp. 38–44, Sept. 1982.

[19] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, Mar. 1988.

[20] K. Shim, T. K. Sellis, and D. Nau, "Improvements on a heuristic algorithm for multiple-query optimization," *Data Knowl. Eng.*, vol. 12, pp. 197–222, 1994.

[21] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Montreal, PQ, Canada, June 1996, pp. 205–216.

[22] C. Zhang and J. Yang, "Genetic algorithm for materialized view selection in data warehouse environments," in *Proc. First Int. Conf. Data Warehousing Knowledge Discovery, Lecture Notes in Computer Science*, Florence, Italy, 1999.

[23] P. J. Angeline *et al.*, Ed., *Proceedings of the 1999 Congress on Evolutionary Computation (CEC)*. Piscataway, NJ: IEEE Press, July 1999.

[24] A. Zalzala *et al.*, Ed., *Proceedings of the 2000 Congress on Evolutionary Computation (CEC)*. Piscataway, NJ: IEEE Press, July 2000.

[25] W. Banzhaf *et al.*, Ed., *Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO)*. San Francisco, CA: Morgan Kaufmann, July 1999.

[26] D. Whitley *et al.*, Ed., *Proceedings of the 2000 Genetic and Evolutionary Computation Conference (GECCO)*. San Francisco, CA: Morgan Kaufmann, July 2000.

[27] M. Schoenauer *et al.*, Ed., *Proceedings of the Sixth International Conference on Parallel Problem Solving from Nature (PPSN VI), Lecture Notes in Computer Science*. New York: Springer-Verlag, Sept. 2000, vol. 1917.

[28] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[29] T. Back, D. B. Fogel, and Z. Michalewicz, *Handbook of Evolutionary Computation*. Amsterdam, The Netherlands: IOP/Oxford Univ. Press, 1997.

[30] R. E. Smith, D. E. Goldberg, and J. A. Earickson, "SGA-C: A C-language implementation of simple genetic algorithm," TCGA, Clearing House for Genetic Algorithms, Univ. Alabama, Dept. Eng. Mech., Tuscaloosa, Rep. 91 002, Mar. 1994.

[31] Massachusetts Institute of Technology (MIT). (1998) Galib: A C++ genetic algorithms library. [Online]. Available: http://lancet.mit.edu/galib-2.4/GAlib.html.

Chuan Zhang was born in Sichuan, China, in 1966. He received the B.A. degree in computer science from Sichuan University, Chengdu, the M.S. degree in computer science from Southeast University, Nanjin, China, and the Ph.D. degree in computer science from the University of New South Wales, Australian Defence Force Academy, Canberra, Australia, in 1986, 1989, and 2000, respectively.

From 1989 to 1995, he was an Engineer and Senior Engineer with Relational Database Management Systems (RDBMS), Chengdu, China. He worked as a Research Assistant at the University of Hong Kong from 1995 to 1997. Currently, he is an Oracle Certified Professional (OCP), Oracle 8i Database Administrator (DBA), and Unix Administrator at Asiaonline, Ltd., Canberra, Australia. His current research interests include data warehousing and RDBMS.

Xin Yao (M'91–SM'96) received the B.Sc. degree in computer science from the University of Science and Technology of China (USTC), Hefei, the M.Sc. degree from the North China Institute of Computing Technologies (NCI), Beijing, and the Ph.D. degree in computer science from USTC in 1982, 1985, and 1990, respectively.

He is currently a Chaired Professor of Computer Science, University of Birmingham, Birmingham, U.K. and a Guest Professor at USTC. From 1992 to 1999, he was a Lecturer, Senior Lecturer, and Associate Professor at University College, University of New South Wales, and the Australian Defence Force Academy (ADFA), Canberra. He held postdoctoral fellowships with the Australian National University (ANU) and the Commonwealth Scientific and Industrial Research Organization (CSIRO), Melbourne, between 1990 and 1992. He is an Associate/Action Editor or a Member of the editorial board of six international journals including and an Editor/Co-Editor of nine journal special issues. His major research interests include combinations between neural and evolutionary computation techniques, evolutionary learning, co-evolution, evolutionary design and evolvable hardware, neural network ensembles, global optimization, simulated annealing, computational time complexity, and data mining.

Dr. Yao has chaired/co-chaired 16 international conferences in evolutionary computation and computational intelligence since 1996 including CEC'99, IEEE ECNN'00, PPSN VI'00, ICCIMA'01, and CEC'02. He is the recipient of the 2001 IEEE Donald G. Fink Prize Paper Award, the Chair of the IEEE NNC Technical Committee on Evolutionary Computation, and the President of the Evolutionary Programming Society. He is an Associate Editor of IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION.

Jian Yang received the Ph.D. degree in computer science from the Australian National University, Canberra, in 1995.

Currently she is an Assistant Professor with Infolab, Tilburg University, Tilburg, the Netherlands. Before joining Infolab, she was a Senior Research Scientist with the Division of Mathematical and Information Science, Commonwealth Scientific and Industrial Research Organization (CSIRO), Canberra, Australia. Her research interests include query optimization, e-commerce, web/internet databases, and workflow management systems.

Dr. Yang is a member of ACM and the IEEE Computer Society.