Algorithms for Materialized View Design in Data Warehousing Environment

Jian Yang
School of Computer Science
University College, UNSW
Australian Defence Force Academy
Canberra ACT 2600, Australia.
jian@csadfa.cs.adfa.oz.au

Kamalakar Karlapalem
Dept of Computer Science
Univ. of Science & Technology
Clear Water Bay, Kowloon
Hong Kong
kamal@cs.ust.hk

Qing Li
Dept of Computing
Hong Kong Polytechnic Univ.
Hung Hom, Kowloon
Hong Kong
csqli@comp.polyu.edu.hk

Abstract

Selecting views to materialize is one of the most important decisions in designing a data warehouse. In this paper, we present a framework for analyzing the issues in selecting views to materialize so as to achieve the best combination of good query performance and low view maintenance. We first develop a heuristic algorithm which can provide a feasible solution based on individual optimal query plans. We also map the materialized view design problem as 0-1 integer programming problem, whose solution can guarantee an optimal solution.

1 Introduction

There are two approaches towards providing integrated access to multiple, distributed, heterogeneous databases: (1) *lazy* or *on-demand* approach to data integration, which often uses *virtual view(s)* techniques; and (2) *data warehousing* approach, where the repository serves as a warehouse storing the data of interest. One of the techniques this approach uses is *materialized view(s)*.

The virtual view approach may be better if the information sources are changing frequently. On the other hand, the materialized approach would be superior if the information sources change infrequently and very fast query response time is needed. The virtual and materialized view approaches represent two ends of vast spectrum of possibilities. We believe that it may be more efficient not to materialize all the views, but rather to materialize certain "shared" portions of the base data, from which the warehouse views can be achieved. In this paper, we present significantly new techniques for selecting views to be materialized.

1.1 Related Work

Finding common subexpressions among multiple queries has been examined in the past in various contexts. [Jar84] discussed the problem of common subexpression isolation. [Hal74, Hal76] used operator trees to represent the queries and a bottom-up traversal procedure to identify common parts. A lot of research has focused on the problem of multiple-query optimization (MQO). The effort in this area has been to find an optimal execution plan for multiple queries executed at the same time, based on the idea that the temporary result sharing should be less expensive compared to a serial execution of queries.

What distinguishes our problem from common subexpression and MQO is the following:

- MQO is to find an optimal execution plan for multiple queries executed at the same time by sharing some temporary results which are common subexpressions, whereas our problem is to find a set of relations (which can be any intermediate result from query processing) to be materialized so that that the total cost (query execution plus view maintenance) is minimal;
- In MQO, a global access plan is derived from

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

the idea that temporary result sharing should be less expensive compared to a serial execution of queries. However, this cannot be true for every possible database state. In our approach, 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;

- In MQO, the ultimate goal is to achieve the best performance, whereas our problem has to take into consideration of both query and view maintenance cost;

- In MQO, the input is a set of queries and the output is a global optimal plan; in our problem, however, the inputs are: a set of global queries and their access frequencies, and a set of base relations and their update frequencies and the output is a set of views to be materialized.

In summary, some of the techniques used in common subexpression and MQO can be applicable to our problem, however our problem is more general and thus more complicated than MQO [YKL97b].

Recently, some research has been done in the area of selecting views to materialize in the data warehousing environment. In [HRU96], the authors provide algorithms to select views to materialize when there are queries with only aggregate functions involved for OLAP applications. While in our work, we are dealing with more general SQL queries which include select, project, join, and aggregation operations. In [Gup97], the author presented several heuristic algorithms for selecting views. Their work provides (1) a near-optimal polynomial time greedy algorithm for two special cases, i.e., AND graphs (with each query having a unique execution plan) and OR graphs (with each query having multiple execution plans); (2) a near-optimal exponential time greedy algorithm for the combined AND-OR graphs. They do not provide an evaluation of the algorithms in terms of the quality of solutions. In this paper, we consider combined cases where each query has multiple query execution plans, employ both query processing and view maintenance cost functions, and provide an optimal solution using 0-1 integer programming formulation and a near-optimal heuristics based algorithm.

1.2 Contributions and Organization of Our Paper

The specific contributions of our paper are as follows:

- A framework is presented to highlight issues of materialized view design in a distributed data warehouse environment. This framework is based on the specification of *Multiple View Processing Plan (MVPP)* which is used to present the problem formally.

- The cost model for materialized view design is provided and analyzed in terms of query performance as well as view maintenance.

- The algorithms for generating MVPPs and determining the selection of views to materialize are presented and analyzed. We provide two algorithms to generate MVPP(s): one can generate a feasible solution expeditiously, the other can provide an optimal solution by mapping the optimal MVPP generation problem as a 0-1 integer programming problem.

The rest of the paper is organized as follows: Section 2 provides an example to illustrate different alternatives for materialized view design and analyzes the existing work. Section 3 presents the formal specification of the problem and the cost models used in our algorithms. In Section 4, we describe several algorithms for addressing materialized view design problem. We conclude in section 5 by summarizing our results and suggesting some ideas for future work.

2 Issues of Materialized View Design

2.1 Motivating Application

Our examples are taken from a data warehouse application which analyzes trends in sales and supply. We have simplified the presentation for ease of exposition. The relations and the attributes of the schema for this application are:

```
Item(I_id, I_name, I_price)
Part(P_id, P_name, I_id)
Supplier(S_id, S_name, P_id, city, cost)
Sales(I_id, month, year, amount)
```

Suppose we have four frequently asked data warehouse queries as listed in Figure 1. We observe that these queries are defined over overlapping portions of the base data or intermediate query results. For example, Q1 and Q2 can share the intermediate result of

```
Q1: Select   I_id, sum(amount*I_price)
    From     Item, Sales
    Where    I_name like {MAZDA, NISSEN, TOYOTA}
    And      year=1996
    And      Item.I_id=Sales.I_id
    Group by I_id

Q2: Select   P_id, month, sum(amount*no)
    From     Item, Sales, Part
    Where    I_name like {MAZDA, NISSEN, TOYOTA}
    And      year=1996
    And      Item.I_id=Sales.I_id
    And      Part.I_id=Item.I_id
    Group by P_id, month

Q3: Select   P_id, min(cost), max(cost)
    From     Part, Supplier
    Where    Part.P_id=supplier.P_id
    And      P_name like {spark_plug, gas_kit}
    Group by Pid

Q4: Select   I_id, sum(amount*number*min_cost)
    From     Item, Sales, Part
    Where    I_name like {MAZDA, NISSEN, 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
```

Figure 1: Example Queries

joining `Item` and `Sales`, and `Q4` can use the intermediate result of `Q3` which calculates the minimal cost of a part.

2.2 Example MVPPs

Figure 2 represents a possible global query access plan for the motivative example described above, in which the local access plan for individual queries are merged based on the shared operations on common data sets. We call it *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 `Supperlier` are updated only once for a certain period of time.

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 non-leaf nodes virtual. The cost for each alternative shall be calculated

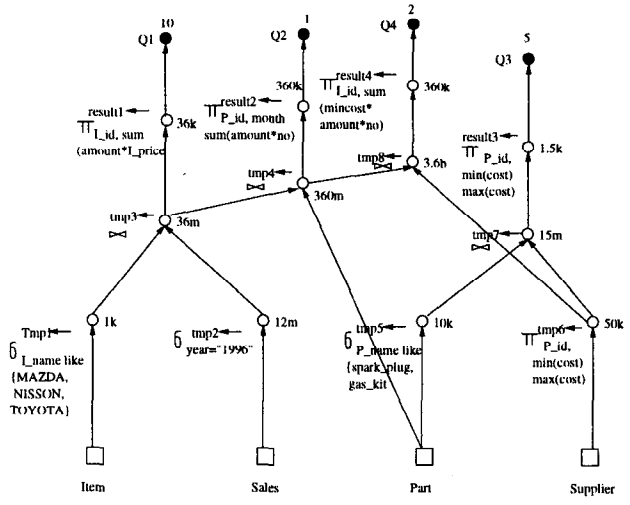

Figure 2: A MVPP for Example Queries

in terms of query processing and view maintenance. (In the annotations, we abbreviate thousand as "k", million as "m", and billion as "b".)

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

- There are 1k tuples in the `Item` relation;
- On average each item has 10 parts, therefore there are 10k tuples in `Part` table;
- There are 50k tuples in `supplier` table;
- There are 10 years worth of sales in the `Sales` table from 1987 through 1996. On average each item is sold 100 times a month, resulting in 12m entries in the sales relation;
- The cost of answering a query Q is the number of rows present in the table used to construct Q. Note that similar cost calculation is used in [HRU96].
- The methods for implementing select and join operation are linear search and nested loop.

Based on the above assumptions, the cost for each operation node in Figure 2 is labeled at the right side of the node.

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 re-computing is used whenever an update of an involved base relation occurs). For example, if `tmp3` is materialized, the query processing cost for `Q1` is $10 * 36.03m$. The view maintenance cost is $2 * (36m + 12m + 1k)$. 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.

In Table 1, we list some materialized view design strategies based on the above example, and their costs. From this table, we have the following observations:

- materializing all the application views in the data warehouse can achieve the best performance at the highest cost of maintenance;
- leaving all the application views virtual will have the poorest performance but the lowest maintenance cost;
- if we have the intermediate results of some operations materialized, and some virtual, especially when there are some shared operations on common data involved, then we can achieve an optimal result with both performance and maintenance taking into account (e.g., materializing `tmp3` and `tmp7` are the best among all the listed strategies).

3 Specifications and Cost Analysis of Materialized View Design

Materialized view design can be achieved with the help of an MVPP. A MVPP specifies the views that the data warehouse will maintain (either materialized or virtual). As will be defined formally below, the MVPP is a directed acyclic graph that represents a query processing strategy of warehouse views. The leaf nodes correspond to the base relations, and the root nodes correspond to warehouse queries. Analogous to query execution plans, there can be more than one MVPP for the same set of views depending upon the access characteristics of the applications and physical data warehouse parameters.

3.1 Specifications

An MVPP is a labeled dag $\mathcal{M} = (V, A, C_a^q, C_m^r, f_q, f_u)$ where V is a set of vertices, A is a set of arcs over V, such that

- for every relational algebra operation in a query tree, for every base relation, and for every distinct query, we create a vertex;
- for $v \in V$, $T(v)$ is the relation generated by corresponding vertex v. $T(v)$ can be a base relation, an intermediate result while processing a query, or the final result for a query;
- for any leaf vertex v, (that is, it has no edges coming into the vertex), $T(v)$ corresponds to a base relation, and is depicted using the $\square$ symbol. Let L be a set of leaf nodes. For any vertex $v \in L$, $f_u(v)$ represents the update frequency of v;
- for any root vertex v, (that is, it has no edges going out of the vertex), $T(v)$ correspond to a global query, and is depicted using the $\bullet$ symbol. Let R be a set of root nodes. For every vertex $v \in R$, $f_q(v)$ represents the query access frequency of v;
- if the base relation or intermediate result relation $T(u)$ corresponding to vertex u is needed for further processing at a node v, we introduce an arc $u \rightarrow v$;
- 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 $S^*\{v\} = S(v) \cup \{\cup_{v' \in S(v)} S^*\{v'\}\}$ be the set of descendants of v;
- for every vertex v, let $D(v)$ denote the *destination* nodes to which v is pointed; for any $v \in R$, $D(v) = \emptyset$. Let $D^*\{v\} = D(v) \cup \{\cup_{v' \in D(v)} D^*\{v'\}\}$ be the set of ancestors of v;
- 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 $S^*(v) \cap R$, if $T(v)$ is materialized.

Now the problem for materialized view design can be described as follows:

1. **Selection of Views to be Materialized from an MVPP:** determine a set of vertices $M \subseteq V$, such that if $\forall v \in M, T(v)$ is materialized, then the sum cost of processing all the queries and maintaining all the views is the smallest possible.
2. **MVPP Generation:** find all pairs of distinct vertices $u, v \in V$, such that $S^*(u) = S^*(v)$ and $T(u) = T(v)$, then $T(u)$ and $T(v)$ are common subexpressions which can be merged to form an MVPP.

For simplicity in notation, we denote the relation $T(v)$ corresponding to a vertex v as just v for the rest of the paper.

3.2 Cost Analysis

3.2.1 Cost of an MVPP

Let M be a set of views in an MVPP to be materialized, f_q, f_u the frequency of executing queries and fre-

Table 1: 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

quency of updating base relations, respectively. Furthermore for each $v \in M$, let $C_a^q(v)$ and $C_m^r(v)$ denote the cost of access for query q using view v and the cost of maintenance of view v based on changes to base relation r, respectively (where $v \in R$ is the set of queries and $r \in L$ is the set of base relations). Note that $C_a^q(v) = 0$ if query q does not access view v, and $C_m^r(v) = 0$ if view v does not have base relation r as its descendant.

Then the query processing cost will be
$C_{queryprocessing}(v) = \sum_{q \in R} f_q C_a^q(v)$

And the materialized view maintenance cost will be
$C_{maintenance}(v) = \sum_{r \in L} f_u C_m^r(v)$

Thus the total cost of materializing a view v is
$C_{total}(v) = \sum_{q \in R} f_q C_a^q(v) + \sum_{r \in L} f_u C_m^r(v)$

Therefore, the total cost of materializing a set of views M is C_{total}[1]:

$C_{total} = \sum_{v \in M} C_{total}(v)$

Our goal is to find the set M so that if the members of M are materialized then the value of C_{total} will be the smallest among all the feasible sets of materialized views. It is obvious from the last formula that the determination of M depends on four factors: (1) frequencies of global query access, (2) frequencies of member database relation update, (3) costs of query processing from materialized view(s), and (4) costs for materialized view maintenance from base relations.

3.2.2 Cost for Shared Views

For every view v in individual query access plan, we introduce an *Ecost(v)* function which represents the benefit of sharing a view among multiple views, and is defined on each view as follows:

$Ecost(v) = \{\sum_{q \in R} f_q C_a^q(v)\}/n_v$

where n_v is the number of queries which can share view v. For example, n_v of tmp3 in Figure 2 is 3.

This formula implies that the smaller the value is, the more likely this view should be selected in the final MVPP and materialized due to its high sharability and being cheap to produce. We will use *Ecost* function in the algorithm for generating best MVPP in section 4.2.3.

[1]Note that in a distributed date warehouse environment, the cost should also incorporate the costs of data transferring among different sites.

4 Algorithms for Materialized View Design

As discussed in Section 3, there are two issues in materialized view design: (1) selecting views to materialize and, (2) MVPP design. In this section, we first present an algorithm to select views to materialize when an MVPP is given, followed by two algorithms for generating and constructing MVPP(s).

4.1 Algorithm for selecting views to be materialized

Given an MVPP, we shall find a set of materialized views such that the total cost for query processing and view maintenance is minimal by comparing the cost of every possible combination of nodes. Suppose that there are n nodes in an MVPP excluding leaf nodes, then we have to try 2^n combinations of nodes. However, we can use some heuristics to reduce the search space.

Before we present our heuristic algorithm, we shall introduce all the notations used in this algorithm:

- O_v denotes the global queries which uses v, $O_v = R \cap D^*\{v\}$; where R is the set of root nodes and $D^*\{v\}$ is the set of ancestors of v as defined in Section 3.
- I_v denotes the base relations which are used to produce v, $I_v = L \cap S^*\{v\}$; where L is the set of leaf nodes and $S^*\{v\}$ is the set of descendants of v as defined in Section 3.
- $w(v)$ denotes the weight of a node, which is calculated as
$w(v) = \sum_{q \in O_v} f_q(q) * C_a^q(v) - \sum_{r \in I_v} f_u(r) * C_m^r(v)$.
The first part of this formula indicates the saving

begin
1. $M := \emptyset$;
2. create list LV for all the nodes
   (with positive value of weights)
   based on the descending order of their weights;
3. pick up the first one v from LV;
4. generate O_v, I_v, and S_v;
5. calculate $C_s = \sum_{q \in O_v}\{f_q(q) * (C_a^q(v) - \sum_{u \in S_v \cap M} C_a^q(u))\} - \sum_{r \in I_v}\{f_u(r) C_m^r(v)\}$;
6. if $C_s > 0$, then
   6.1. insert v into M;
   6.2. remove v from LV;
7. else remove v and all the nodes
   listed after v from LV who are in
   the subtree rooted at v;
8. repeat step 3 until $LV = \emptyset$;
9. for each $v \in M$, if $D(v) \subset M$, then
   remove v from M;

end;

Figure 3: HA_{MVD} - Materialized View Design Algorithm

if node v is materialized, the second part indicates the cost for materialized view maintenance.

- LV is the list of nodes based on descending order of $w(v)$;
- $S_v = S^*\{v\}$ is the set of nodes (leaf nodes and intermediate nodes) which are used to produce v;

Without losing generality, we assume there are altogether k queries. Let M be the set of materialized views. The algorithm in Figure 3 for determining M is based on the following idea: whenever a new node is considered to be materialized, we calculate the saving it brings in accessing all the queries involved, subtracting the cost for maintaining this node. If this value is positive, then this node will be materialized and added into M.

In Step 5, the first part of C_s is the saving if v is to be materialized. The second part is the view maintenance cost for v. $C_s > 0$ indicates that there is a cost gain if v is materialized. $\sum_{u \in S_v \cap M} C_a^q(u))$ is the replicated saving in case of some descendants of v are already chosen to be materialized. After applying transformation, C_s becomes:

$$C_s = \sum_{q \in O_v} f_q(q) * C_a^q(v) - \sum_{r \in I_v} f_u(r) * C_m^r(v) - \sum_{q \in O_v} f_q(q)(\sum_{u \in D(v) \cap M} C_a^q(u))$$
$$= w(v) - \sum_{q \in O_v}(f_q(q) * \sum_{u \in D(v) \cap M} C_a^q(u))$$

For example, if v_1 is a descendant of v_2, and $w(v_1) > w(v_2)$, then the second part of the above formula for v_1 and v_2 are the same. Therefore, if materializing v_1 will not gain anything, then definitely there will not be any gain to materialize v_2. Applying Step 7 in the algorithm HA_{MVD} we can save some search space.

A fuller explanation and discussion of this algorithm run by this example is presented in [YKL97a].

4.2 Algorithms for multiple MVPPs design

Normally for one query, there are several processing plans, among which there is one optimal plan. Therefore, we will have multiple MVPPs based on different combinations of individual plans. In the following subsections, we first discuss some transformation rules for query plans which are needed for our algorithms. We then present two algorithms for multiple MVPP design: the first one can provide a feasible solution by dealing with optimal plan instead of all possible plans for each query; the second one provides an algorithm which considers all possible plans for each query to generate a single optimal MVPP by applying 0-1 integer programming technique.

4.2.1 Transformations

Since *join* is one of the most expensive operations, we like to find the sharable join operations among queries as early as possible. To do this, we have to pull up all the *select, project,* and *aggregation* operations along the query tree, and push down these operations when an MVPP is generated by merging common join operations.

Pulling up select and project operations are straight forward based on relational algebra or calculus. Some rewrite rules have been previously given in [Day87]. As for pushing down transformations after an MVPP is generated by merging common join operations, we introduce and/or adopt the following rules:

- *pushing down select operations:* if there is more than one query sharing a join operation, and these queries have different select conditions on the attributes of two base relations of the join operation, then the select condition for a base relation attribute is the disjunction of all the select conditions on that attribute;
- *pushing down project operations:* the attributes which should be projected for a base relation should be the union of the projection attributes of queries which share the common join operation, plus the join attribute(s) (if required);

- *pushing down aggregations with identical group-by attributes:* we apply the rewrite rules given in [CS94, Yan94]: if different queries which share the same join operation have different aggregation functions with the same group-by attributes, then the new aggregation functions will include all the individual aggregation functions;

- *pushing down aggregations with different group-by attributes:* If different queries which share the same join operation have different group-by attributes on the same base relation, then we have to use the combination of individual group-by attributes as the new group-by attribute against the base relation. This will generate multiple distinct group-by operations on the same join result or base relation for queries.

4.2.2 A Feasible Solution

In order to reduce the search space, we start with individual query optimal plans, and order them based on query access frequency times query processing cost. Once the order of the optimal query plans is fixed, we pick up the first optimal plan, and incorporate the second one into it based on the idea of using the common subexpressions if there is any. After the first two are merged, the next one is picked up to be incorporated with the merged plan. We keep doing it until all the plans are merged. Then, we repeat this procedure of incorporating all other plans with the second expensive plan, so on and so forth, until all the plans have been considered. If there are k global plans, we will end up with k MVPPs. For every MVPP generated, we run HA_{MVD} (described in the previous subsection 4.1), compare the total cost of each MVPP, and select the one which gives the lowest cost.

The algorithm for generating MVPPs is presented in Figure 4. In this algorithm, step 4.3 is to merge the current MVPP with the elements of list l (of optimal query execution plans *op*'s) based on the join pattern of current *op*. The idea is to reserve the join pattern of current $MVPP$, and then try to find the join operation nodes in the $MVPP$ which can be used in the individual optimal query plan *op*. If there is any such node, evolve the MVPP; otherwise the join pattern in the *op* shall be used.

The detailed explanation of the algorithm run by this example is presented in [YKL97a].

After each MVPP is derived, we have to optimize it by pushing down the select and project operations as far as possible. What differentiates MVPP optimization with traditional heuristic query optimization is that in an MVPP several queries can share some in-

---

begin

1. for each query q_i, generate an optimal query processing plan *op*;
2. for any query involving join operations, push up all the select and project operations;
3. create a list $l =< op_1, op_2, \ldots, op_k >$, in which the elements are in the descending order of the values of $f_q(op_i) * C_a(op_i)$;
4. for $n = 1$ to k do
   - 4.1. pick up the first element from l, $l(1)$, maintain the order of the joins in (1);
   - 4.2. $MVPP(n) := l(1)$;
   - 4.3. for $m = 2$ to k do
     - 4.3.1. divide the leaf node set of op_m into several disjoint subsets, according to the following order of
       (1) the set of leaf nodes that are already joined conjunctively in $MVPP(n)$, and one of the leaf nodes in this set is the first node of the join;
       (2) the set of leaf nodes that are not joined in $MVPP(n)$, but joined in op_m;
     - 4.3.2. find the common ancestor node of elements of each subset either in in $MVPP(n)$ or in op_m, create new node(s) to join these ancestors nodes, replace the final join operation node in op_m with the root node of these new node(s), delete all the unused nodes and associated edges in op_m;
     - 4.3.3. $MVPP(n) := MVPP(n) \cup l(m)$;
     - 4.3.4. $m := m + 1$;
   - 4.4. $n := n + 1$;
   - 4.5. move $l(1)$ to the end of the list;
5. for every leaf node $v \in L$ of every $MVPP$, find all the relevant select conditions of queries which are members of $R \cap S^*\{v\}$, take the disjunction of select conditions, push it down to v;
6. for every leaf node $v \in L$ of every $MVPP$, find all the relevant project operation of queries which are members of $R \cap S^*\{v\}$, of queries which are members of $R \cap S^*\{v\}$, take the union of the relevant project attributes, plus the join attribute(s), push it down to v;

end;

Figure 4: HA_{mvpp} - A Heuristic Algorithm for Generating Multiple MVPPs

termediate nodes, therefore there can be several select conditions and aggregations on base relations which are combined (as discussed in subsection 4.2.1.)

For each MVPP obtained, we run the HA_{MVD} algorithm to select views for materializing, followed by calculating the total cost for each MVPP using the model defined in Section 4.3. After that, we can select the best MVPP which has the optimal combination of query processing and view maintenance cost.

Note that the algorithm described in Figure 4 may not guarantee that an optimal MVPP can always be obtained (and not missed), since only a subset of the possible MVPPs (for a given set of queries) has been considered. Nevertheless, we believe it captures a reasonable subset of MVPPs, out of which a satisfactory (and balanced) solution can be found efficiently.

4.2.3 A 0-1 integer programming solution

In this subsection, we try to overcome the limitations of the HA_{mvpp} by looking into all possible combinations of individual query plans and then selecting the most beneficial MVPP(s). In order to do this systematically while incorporating the cost of processing the queries, we model the optimal MVPP selection problem as 0-1 integer programming (IP) problem. This approach has two advantages. the first is the automatic formulation of IP problem given a set of queries and base relations. Secondly, IP problem has been well studied and there is lot of software available that solves IP problems quite efficiently. This approach is better than applying ad-hoc heuristics [Gup97] which are difficult to evaluate or provide an intuition as to why the heuristics work. Further, IP solution procedure provides an optimal or near-optimal solution. A point to note is that the optimal MVPP selection problem lends itself to be specified as an IP problem, a technique which has been applied in generating optimal query execution plans [BNNS96, ETB96] for distributed and deductive database systems. For simplicity we assume that all the select, project and aggregate operations have been pushed up and we only consider join operations.

Before we present our algorithm, we again introduce all the notations used here first:

- there are k number of queries: $q_1, q_2, \cdots, q_k$;
- for any query q, there is a set of possible join plans, which can be represented as binary trees with join operations as root nodes of (sub)tree(s). See Figure 5;
- given a *join plan tree* of a query q, an in-order traversal of this plan generates an ordered list p, in the form of (R, S), such that R and S are two children of a join node in this plan tree. We denote a join plan tree by p, and s denotes a *join pattern* or a subtree of a plan p as shown in Figure 5.

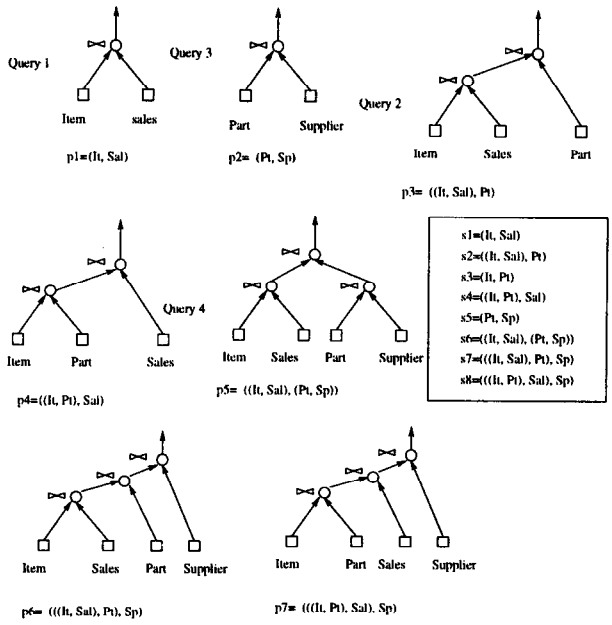

Figure 5: Join Plan Trees

- let $p(q) = \{p_i : p_i \text{ is a join plan tree of } q\}$ be the set of all possible join plan trees of query q;
- let $P = \cup_{i=1}^{k} p(q_i)$ be the set of all possible join plan trees for all the k queries;
- let $s(p) = \{s_i : s_i \text{ is a pattern contained in plan } p\}$ be the set of all join patterns of p;
- let $S = \cup_{i=1}^{l} s(q_i)$ be the set of all possible join patterns for all the plans.

According to the above definitions, we can get for example the following information from Figure 5 (Note: we use abbreviation `It, Sal, Pt, Sp` for `Item, Sales, Part`, and `Supplier`, respectively):

Join Plan Trees

$p(q_1) = \{(It, Sal)\} = \{p_1\}$
$p(q_2) = \{((It, Sal), Pt), ((It, Pt), Sal)\} = \{p_3, p_4\}$
$p(q_3) = \{(Pt, Sp)\} = \{p_2\}$
$p(q_4) = \{((It, Sal), (Pt, Sp)), (((It, Sal), Pt), Sp),$
$(((It, Pt), Sal), Sp)\} = \{p_5, p_6, p_7\}$

Join Patterns

$s(p_1) = \{(It, Sal)\} = \{s_1\}$
$s(p_2) = \{(Pt, Sp)\} = \{s_5\}$
$s(p_3) = \{(It, Sal), ((It, Sal, Pt)\} = \{s_1, s_2\}$

$s(p_4) = \{(It, Pt), ((It, Pt), Sal)\} = \{s_3, s_4\}$
$s(p_5) = \{(It, Sal), (Pt, Sp), ((It, Sal), (Pt, Sp))\}$
$= \{s_1, s_5, s_6\}$
$s(p_6) = \{(It, Sal), ((It, Sal), Pt), (((It, Sal), Pt), Sp)\}$
$= \{s_1, s_2, s_7\}$
$s(p_7) = \{(It, Pt), ((It, Pt), Sal), (((It, Pt), Sal, Sp)\}$
$= \{s_3, s_4, s_8\}$

List of Join Plan Trees and Join Patterns

$P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$
$S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$

Here we consider all the possible join plan trees for each query including those join plan trees whose pattern cannot be shared by any other queries; only those join patterns which will generate cartesian products of relations are not considered. The IP problem formulation states: select a subset of the join plan trees such that all queries can be executed and the total query processing cost is the minimum. The join plan trees are selected such that the cost of processing the join patterns (s) in the join plan trees (p) is the least. We shall first present the notation and variables used in the formulation, and then present the IP formulation of the problem.

After we get S, P, p_i, and s_j, we construct two matrices A and B as follows:

Let $p_1, p_2, \ldots, p_l \in P$ be the l join plan trees for k queries $q_1, q_2, \ldots, q_k$. Let $s_1, , s_2, \ldots, s_m \in S$ be m patterns derived from l join plan trees.

Let A be a $k \times l$ matrix whose element $a_{ij} = 1$ if query q_i can be answered by join plan tree p_j. Let x_i be a binary variable which takes value 1 if join plan tree p_i is selected, else 0.

Let B be a $m \times l$ matrix whose element $b_{ij} = 1$ if pattern s_i is contained in join plan tree p_j. Then, the problem of selecting an optimal MVPP reduces to selecting a subset of l join plan trees $\{p_1, p_2, \ldots, p_l\}$ so as to:

minimize $x_0 = \sum_{i=1}^{i=m} Ecost(s_i) * \{\sum_{j=1}^{j=l} b_{ij} * x_j\}$

subject to

$\sum_{j=1}^{j=l} a_{ij} * x_i = 1$ for every query i

where each $x_i = 0$ or 1.

Note that $Ecost(s_i)$ is the estimated cost of pattern (node) s_i defined in Section 4.3. The constraint for each query q_i, namely, $\sum_{j=1}^{j=l} a_{ij} * x_i = 1$ states that the query q_i should be answered by exactly one of the join plan trees selected. The solution to the above 0-1 integer programming formulation gives the set of join plan trees which form the optimal MVPP. Thus the problem of selecting an optimal MVPP is solved by 0-1 integer programming.

For the example in Figure 5, we have the following two matrixes:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Based on our assumption in Section 2 and Figure 2, we get the costs and the estimated cost for each join pattern as shown in Table 2.

After solving the 0-1 integer programming problem we get the optimal solution that selects join plan trees p_1, p_2, p_3, p_6 to form the optimal MVPP, which is the same as that generated by HA_{mvpp}.

4.2.4 The comparison of the algorithms for MVPP generation

By comparing the results obtained by using the above two algorithms, we can get the following conclusions:

- The HA_{mvpp} algorithm is to get multiple MVPPs regardless of their query cost; while the 0-1 integer programming approach is to get a best MVPP in terms of query access efficiency.
- Although the results generated by HA_{mvpp} may include the best MVPP, which is the case for our example, it cannot guarantee that it is always the case. The reason for this is because HA_{mvpp} only works with optimal plans. For instance, if we assign different cost values for patterns in Table 2, the best MVPP may turn out to be the merging join plan trees of p_1, p_2, p_4, p_5, which will not be picked up by HA_{mvpp}, as p_4 is not an optimal plan for query 2. In contrast, the 0-1 integer programming approach works with all the possible join plan trees, therefore it can definitely get the best MVPP.
- The complexity of HA_{mvpp} is $O(n)$ if there are n number of queries, while the complexity of the 0-1 integer programming approach is $O(2^n)$. Therefore, if we just need a reasonable solution, we can

Table 2: Costs of Patterns

Patterns	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
n_v	3	2	2	2	2	1	1	1
$Cost = \sum_1^{n_v} f_q C_a^q(s_i)$	1E7	1.2E14	1.2E10	1.2E14	6E11	1.2E18	1.2E18	1.2E18
ECosts	3.3E6	6E13	6E9	6E9	3E11	1.2E18	1.2E18	1.2E14

use HA_{mvpp}. With current 0-1 linear programming software package, we can get the answer within an acceptable period of time, as the MVPP generation problem can be done off line. Thus if we need an optimal MVPP, we could use the 0-1 integer programming approach.

5 Conclusions

We have addressed and designed algorithms for the materialized view design problem, i.e., how to select a set of views to be materialized so that the sum cost of processing a set of queries and maintaining the materialized views is minimized. Our approach relies on analyzing the queries so as to derive common intermediate results which can be shared among the queries. The cost model takes into consideration of not only query access frequencies and base relation update frequencies, but also query access costs and view maintenance costs. The algorithms for generating MVPPs uses the techniques from single query optimization, coupled with query tree merging techniques which aims to incorporate the individual optimal query plans as much as possible in the MVPP. We are also able to successfully map the optimal MVPP generation problem as a 0-1 integer programming problem so that we are guaranteed to have an optimal solution.

The work presented here is the outcome of the first stage of research in Materialize View Design project. We are currently working on the combined index selection and materialized view design problem. We are also extending this work towards a cost-based approach to migrate a legacy database system onto a data warehousing platform based on the HODFA architecture [KLS95], wherein the cost of migrating the consistency between the legacy database and the base relations while maintaining consistency has to be incorporated. Finally, we will focus on developing an analytical model for a multiple view processing environment to simulate different scenarios to evaluate the solutions for the materialized view design problem.

References

[BNNS96] C. Bell, A. Nerode, R. T. Ng, and V. S. Subrahmanian. Implementing deductive databases by mixed integer programming. *ACM Transactions on Database Systems*, 21(2):p238–69, 1996.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. *In VLDB*, 1994.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. *In VLDB*, 1987.

[ETB96] C. J. Egyhazy, K. P. Triantis, and B. Bhaskar. A query processing algorithm for a system of heterogeneous distributed databases. *Distributed and Parallel Databases*, 4(1):p49–79, January 1996.

[Gup97] H. Gupta. Selection of views to materialized in a data warehouse. *in ICDT*, 1997.

[Hal74] P.V. Hall. Common subexpression identification in general algebraic systems. *Tech. Rep. UKSC 0060, IBM United Kingdom Scientific Centre*, Nov. 1974.

[Hal76] P.V. Hall. Optimization of a single relation expression in a relational data base system. *IBM J. Res. Dev. 20, 3*, pages 244–257, May 1976.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. *In Proc. of the ACM SIGMOD International Conference of Management of Data, Canada*, June 1996.

[Jar84] M. Jarke. Common subexpression isolation in multiple query optimization. *Query Processing in Database Systems*, pages 191–205, 1984.

[KLS95] K. Karlapalem, Q. Li, and C. Shum. Hodfa: An architectural framework for homogenizing heterogeneous legacy databases. *SIGMOD RECORD*, 24(1):15–20, March 1995.

[Yan94] W. P. Yan. Performing group-by before join. *In ICDE*, 1994.

[YKL97a] J. Yang, K. Karlapalem, and Q. Li. A framework for designing materialized views in data warehousing environment. *in the Proc. of ICDCS'97 International Conference in Distributed Computing Systems, Baltimore, Maryland, USA*, 1997.

[YKL97b] J. Yang, K. Karlapalem, and Q. Li. Tackling the challenges of materialized view design in data warehousing environment. *in the Proc. of Int'l Workshop on Research Issues in Data Engineering (RIDE'97), IEEE Computer Sociaty, UK*, 1997.