Selection of Views to Materialize Under a Maintenance Cost Constraint

Himanshu Gupta[1*] and Inderpal Singh Mumick[2]

[1] hgupta@db.stanford.edu, Stanford University, Stanford, CA 94305
[2] mumick@savera.com, Savera Systems, Summit, NJ 07901

Abstract. A data warehouse stores materialized views derived from one or more sources for the purpose of efficiently implementing decision-support or OLAP queries. One of the most important decisions in designing a data warehouse is the selection of materialized views to be maintained at the warehouse. The goal is to select an appropriate set of views that minimizes total query response time and/or the cost of maintaining the selected views, given a limited amount of resource such as materialization time, storage space, or total view maintenance time. In this article, we develop algorithms to select a set of views to materialize in a data warehouse in order to minimize the total query response time under the constraint of a given total view maintenance time. As the above maintenance-cost view-selection problem is extremely intractable, we tackle some special cases and design approximation algorithms. First, we design an approximation greedy algorithm for the maintenance-cost view-selection problem in OR view graphs, which arise in many practical applications, e.g., data cubes. We prove that the query benefit of the solution delivered by the proposed greedy heuristic is within 63% of that of the optimal solution. Second, we also design an A^* heuristic, that delivers an optimal solution, for the general case of AND-OR view graphs. We implemented our algorithms and a performance study of the algorithms shows that the proposed greedy algorithm for OR view graphs almost always delivers an optimal solution.

1 Introduction

A *data warehouse* is a repository of integrated information available for querying and analysis [IK93, HGMW+95, Wid95]. Figure 1 illustrates the architecture of a typical warehouse [WGL+96]. The bottom of the figure depicts the multiple *information sources* of interest. Data that is of interest to the client(s) is derived or copied and integrated into the data warehouse, depicted near the top of the figure. These views stored at the warehouse are often referred to as *materialized views*. The *integrator*, which lies in between the sources and the warehouse, is responsible for maintaining the materialized views at the warehouse in response to changes at the sources [ZGMHW95, CW91, GM95]. This incremental maintenance of views incurs what is known as *maintenance cost*. We use the term maintenance cost interchangeably with *maintenance time*.

* Supported by NSF grant IRI-96-31952.

One of the advantages of such a system is that user queries can be answered using the information stored at the warehouse and need not be translated and shipped to the original source(s) for execution. Also, warehouse data is available for queries even when the original information source(s) are inaccessible due to real-time operations or other reasons. Widom in [Wid95] gives a nice overview of the technical issues that arise in a data warehouse.

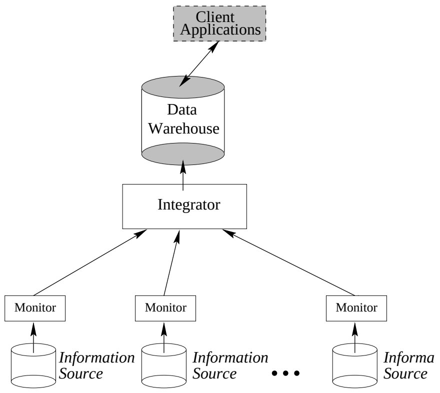


Fig. 1. A typical data warehouse architecture

The selection of which views to materialize is one of the most important decisions in the design of a data warehouse. Earlier work [Gup97] presents a theoretical formulation of the general "view-selection" problem in a data warehouse. Given some resource constraint and a query load to be supported at the warehouse, the *view-selection* problem defined in [Gup97] is to select a set of derived views to materialize, under a given resource constraint, pthat will minimize the sum of total query response time and maintenance time of the selected views. [Gup97] presents near-optimal polynomial-time greedy algorithms for some special cases of the general problem where the resource constraint is disk space.

In this article, we consider the view-selection problem of selecting views to materialize in order to optimize the total query response time, under the constraint that the selected set of views incur less than a given amount of total maintenance time. Hereafter, we will refer to this problem as the *maintenance-cost view-selection* problem. The maintenance-cost view-selection problem is much more difficult than the view-selection problem with a disk-space constraint, because the maintenance cost of a view v depends on the set of other materialized views. For the special case of "OR view graphs," we present a competitive greedy algorithm that provably delivers a near-optimal solution. The OR view graphs, which are view graphs where exactly one view is used to derive another view, arise in many important practical applications. A very important application is that of OLAP warehouses called data cubes, where the candidate views for precomputation (materialization) form an "OR boolean lattice." For the general maintenance-cost view-selection problem that arises in a data warehouse, i.e., for the general case of AND-OR view graphs, we present an A^* heuristic that delivers an optimal solution.

The rest of the paper is organized as follows. The rest of this section gives a brief summary of the related work. In Section 2, we present the motivation for the maintenance-cost view-selection problem and the main contributions of this article. Section 3 presents some preliminary definitions. We define the maintenance-cost view-selection problem formally in Section 4. In Section 5, we present an approximation greedy algorithm for the maintenance-cost view-selection problem in OR view graphs. Section 6 presents an A^* heuristic that delivers an optimal set of views for the maintenance-cost view-selection problem in general AND-OR view graphs. We present our experimental results in Section 7. Finally, we give some concluding remarks in Section 8.

1.1 Related Work

Recently, there has been a lot of interest on the problem of selecting views to materialize in a data warehouse. Harinarayan, Rajaraman and Ullman [HRU96] provide algorithms to select views to materialize in order to minimize the total query response time, for the case of data cubes or other OLAP applications when there are only queries with aggregates over the base relation. The view graphs that arise in OLAP applications are special cases of OR view graphs. The authors in [HRU96] propose a polynomial-time greedy algorithm that delivers a near-optimal solution. Gupta et al. in [GHRU97] extend their results to selection of views *and* indexes in data cubes. Gupta in [Gup97] presents a theoretical formulation of the general view-selection problem in a data warehouse and generalizes the previous results to (i) AND view graphs, where each view has a unique execution plan, (ii) OR view graphs, (iii) OR view graphs with indexes, (iv) AND view graphs with indexes, and some other special cases of AND-OR view graphs. All of the above mentioned works ([HRU96, GHRU97, Gup97]) present approximation algorithms to select a set of structures that minimizes the total query response time under a given *space* constraint; the constraint represents the maximum amount of disk space that can be used to store the materialized views.

Other recent works on the view-selection problem have been as follows. Ross, Srivastava, and Sudarshan in [RSS96], Yang, Karlapalem, and Li in [YKL97], Baralis, Paraboschi, and Teniente in [BPT97], and Theodoratos and Sellis in [TS97] provide various frameworks and heuristics for selection of views in order to optimize the sum of query response time and view maintenance time without any resource constraint. Most of the heuristics presented there are either exhaustive searches or do not have any performance guarantees on the quality of the solution delivered.

Ours is the first article to address the problem of selecting views to materialize in a data warehouse under the constraint of a given amount of total view maintenance time.

2 Motivation and Contributions

Most of the previous work done ([HRU96, GHRU97, Gup97]) on designing polynomial-time approximation algorithms that provably deliver a near-optimal solution for the view-selection problem suffers from one drawback. The designed algorithms apply only to the case of a disk-space constraint.

Though the previous work has offered significant insight into the nature of the view-selection problem, the constraint considered therein makes the results less applicable in practice because disk-space is very cheap in real life. In practice, the real constraining factor that prevents us from materializing everything at the warehouse is the maintenance time incurred in keeping the materialized views up to date at the warehouse. Usually, changes to the source data are queued and propagated periodically to the warehouse views in a large batch update transaction. The update transaction is usually done overnight, so that the warehouse is available for querying and analysis during the day time. Hence, there is a constraint on the time that can be alloted to the maintenance of materialized views.

In this article, we consider the maintenance-cost view-selection problem which is to select a set of views to materialize in order to minimize the query response time under a constraint of maintenance time. We do not make any assumptions about the query or the maintenance cost models. It is easy to see that the view-selection problem under a disk-space constraint is only a special case of the maintenance-time view-selection problem, when maintenance cost of each view remains a constant, i.e., the cost of maintaining a view is independent of the set of other materialized views. Thus, the maintenance-cost view-selection problem is trivially `NP`-hard, since the space-constraint view-selection problem is `NP`-hard [Gup97].

Now, we explain the main differences between the view-selection problem under the space constraint and the maintenance-cost view-selection problem, which makes the maintenance-cost view-selection optimization problem more difficult. In the case of the view-selection problem with space constraint, as the query benefit of a view never increases with materialization of other views, the query-benefit per unit space of a non-selected view always decreases monotonically with the selection of other views. This property is formally defined in [Gup97] as the monotonicity property of the benefit function and is repeated here for convenience.

Definition 1. (Monotonic Property) A benefit function B, which is used to prioritize views for selection, is said to satisfy the *monotonicity property* for a set of views M with respect to distinct views V_1 and V_2 if $B(\{V_1, V_2\}, M)$ is less than (or equal to) either $B(\{V_1\}, M)$ or $B(\{V_2\}, M)$.

In the case of the view-selection problem under space constraint, the query-benefit per unit space function satisfies the above defined monotonicity property for all sets M and views V_1 and V_2. However, for the case of maintenance-cost view-selection problem, the maintenance cost of a view can decrease with selection of other views for materialization and hence, the query-benefit per unit of maintenance-cost of a view can actually increase. Hence, the total maintenance cost of two "dependent" views may be much less than the sum of the maintenance costs of the individual views, causing the query-benefit per unit maintenance-cost of two dependent views to be sometimes much greater than the query-benefit per unit maintenance-cost of either of the individual views. The above described non-monotonic behavior of the query-benefit function makes the maintenance-problem view-selection problem intractable. The non-monotonic behavior of the query-benefit per unit maintenance-cost function is illustrated in Example 2 in Section 5, where it is shown that the simple greedy approaches presented in previous works for the space-constraint view-selection problem could deliver an arbitrarily bad solution when applied to the maintenance-cost view-selection problem.

Contributions In this article, we have identified the maintenance-cost view-selection problem and the difficulty it presents. We develop a couple of algorithms to solve the maintenance-cost view-selection problem within the framework of general query and maintenance cost models. For the maintenance-cost view-selection problem in general OR view graphs, we present a greedy heuristic that selects a *set* of views at each stage of the algorithm. We prove that the proposed greedy algorithm delivers a near-optimal solution. The OR view graphs, where exactly one view is used to compute another view, arise in many important practical applications. A very important application is that of OLAP warehouses called data cubes, where the candidate views for precomputation (materialization) form an "OR boolean lattice." We also present an A^* heuristic for the general case of AND-OR graphs. Performance studies indicate that the proposed greedy heuristic almost always returns an optimal solution for OR view graphs. The maintenance-cost view-selection was one of the open problems mentioned in [Gup97]. By designing an approximate algorithm for the problem, this article essentially answers one of the open questions raised in [Gup97].

3 Preliminaries

In this section, we present a refinement of a few definitions from [Gup97] used in this article. Throughout this article, we use $V(G)$ and $E(G)$ to denote the *set of vertices* and *edges* respectively of a graph G.

Definition 2. (Expression AND-DAG) An *expression AND-DAG* for a view, or a query, V is a directed acyclic graph having the base relations (and materialized views) as "sinks" (no outgoing edges) and the node V as a "source" (no incoming edges). If a node/view u has outgoing edges to nodes $v_1, v_2, \ldots, v_k$, then u can be computed from *all* of the views $v_1, v_2, \ldots, v_k$ and this dependence is indicated by drawing a semicircle, called an *AND arc*, through the edges $(u, v_1), (u, v_2), \ldots, (u, v_k)$. Such an AND arc has an operator[3] and a query-cost

[3] The operator associated with the AND arc is actually a k-ary function involving operations like join, union, aggregation etc.

associated with it, which is the cost incurred during the computation of u from $v_1, v_2, \ldots, v_k$.

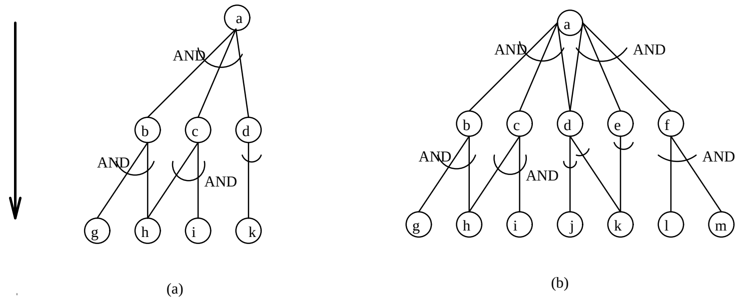


Fig. 2. a) An expression AND-DAG b) An expression ANDOR-DAG

Definition 3. (Expression ANDOR-DAG) An *expression ANDOR-DAG* for a view or a query V is a directed acyclic graph with V as a source and the base relations as sinks. Each non-sink node v has associated with it one or more AND arcs, where each AND arc binds a *subset* of the outgoing edges of node v. As in the previous definition, each AND arc has an operator and a cost associated with it. More than one AND arc at a node depicts multiple ways of computing that node.

Figure 2 shows an example of an expression AND-DAG as well as an expression ANDOR-DAG. In Figure 2 (b), the node a can be computed either from the set of views $\{b, c, d\}$ or $\{d, e, f\}$. The view a can also be computed from the set $\{j, k, f\}$, as d can be computed from j or k and e can be computed from k.

Definition 4. (AND-OR View Graph) A directed acyclic graph G having the base relations as the sinks is called an *AND-OR view graph* for the views (or queries) $V_1, V_2, \ldots, V_k$ if for each V_i, there is a subgraph[4] G_i in G that is an expression ANDOR-DAG for V_i. Each node v in an AND-OR view graph has the following parameters associated with it: query-frequency f_v (frequency of the queries on v), update-frequency g_v (frequency of updates on v), and reading-cost R_v (cost incurred in reading the materialized view v). Also, there is a

[4] An AND-OR view graph H is called a subgraph of an AND-OR view graph G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$, and each edge e_1 in H is bound with the same set of edges through an AND-arc as it is bound through an AND-arc in G. That is, if $e_1, e_2 \in E(G)$, $e_1 \in E(H)$, and e_1 and e_2 are bound by an AND-arc (which may bind other edges too) in G, then $e_2 \in E(H)$, and e_1 and e_2 are bound with the same AND-arc in H. For example, Figure 2 (a) is a subgraph of Figure 2 (b), but Figure 2 (a) without the edge (c, h) is not.

maintenance-cost function UC[5] associated with G, such that for a view v and a set of views M, $UC(v, M)$ gives the cost of maintaining v in presence of the set M of materialized views.

Given a set of queries $Q_1, Q_2, \ldots, Q_k$ to be supported at a warehouse, [Gup97] shows how to construct an AND-OR view graph for the queries.

Definition 5. (Evaluation Cost) The *evaluation cost* of an AND-DAG H embedded in an AND-OR view graph G is the sum of the costs associated with the AND arcs in H, plus the sum of the reading costs associated with the sinks/leaves of H.

Definition 6. (OR View Graphs) An OR view graph is a special case of an AND-OR view graph, where each AND-arc binds exactly one edge. Hence, in OR view graphs, we omit drawing AND arcs and label the edges, rather than the AND arcs, with query-costs. Also, in OR view graphs, instead of the maintenance cost function UC for the graph, there is a maintenance-cost value associated with each edge (u, v), which is the maintenance cost incurred in maintaining u using the materialized view v. Figure 3 shows an example of an OR view graph $\mathcal{G}$.

4 The Maintenance-Cost View-Selection Problem

In this section, we present a formal definition of the maintenance-cost view-selection problem which is to select a set of views in order to minimize the total query response time under a given maintenance-cost constraint.

Given an AND-OR view graph G and a quantity S (available maintenance time), the *maintenance-cost view-selection problem* is to select a set of views M, a subset of the nodes in G, that minimizes the total query response time such that the total maintenance time of the set M is less than S.

More formally, let $Q(v, M)$ denote the cost of answering a query v (also a node of G) in the presence of a set M of materialized views. As defined before, $UC(v, M)$ is the cost of maintaining a materialized view v in presence of a set M of materialized views. Given an AND-OR view graph G and a quantity S, the maintenance-cost view-selection problem is to select a set of views/nodes M, that minimizes $\tau(G, M)$, where

$$\tau(G, M) = \sum_{v \in V(G)} f_v Q(v, M),$$

under the constraint that $U(M) \leq S$, where $U(M)$, the *total maintenance time*, is defined as

$$U(M) = \sum_{v \in M} g_v UC(v, M).$$

The view-selection problem under a disk-space constraint can be easily shown to be `NP`-hard, as there is a straightforward reduction [Gup97] from the **`minimum set cover`** problem. Thus, the maintenance-cost view-selection problem, which is a more general problem as discussed in Section 2, is trivially `NP`-hard.

---

[5] The function symbol "UC" denotes *update cost.*

Computing Q(v,M) The cost of answering a query v in presence of M, $Q(v, M)$, in an AND-OR view graph G is actually the evaluation cost of the cheapest AND-DAG H_v for v, such that H_v is a subgraph of G and the sinks of H_v belong to the set $M \cup L$, where L is the set of sinks in G. Here, without loss of generality, we have assumed that the nodes in L, the set of sinks in G, are always available for computation as they represent the base tables at the source(s). Thus, $Q(v, \phi)$ is the cost of answering a query on v directly from the source(s). For the special case of OR view graphs, $Q(v, M)$ is the minimum query-length of a path from v to some $u \in (M \cup L)$, where the *query-length* of a path from v to u is defined as R_u, the reading cost of u, plus the sum of the query-costs associated with the edges on the path.

Computing UC(v,M) in OR view Graphs The quantity $UC(v, M)$, as defined earlier, denotes the maintenance cost of a view v with respect to a selected set of materialized views M, i.e., in presence of the set $M \cup L$. As before, we assume that the set L of base relations in G is always available. In general AND-OR view graphs, the function UC, which depends upon the maintenance cost model being used, is associated with the graph. However, for the special case of OR view graphs, $UC(v, M)$ is computed from the maintenance costs associated with the edges in the graph as follows. The quantity $UC(v, M)$ is defined as the minimum maintenance-length of a path from v to some $u \in (M \cup L) - \{v\}$, where the *maintenance-length* of a path is defined as the sum of the maintenance-costs associated with the edges on the path.[6] The above characterization of $UC(v, M)$ in OR view graphs is without any loss of generality of a maintenance-cost model, because in OR view graphs a view u uses at most one view to help maintain itself.

In the following example, we illustrate the above definitions of Q and UC on OR view graphs.

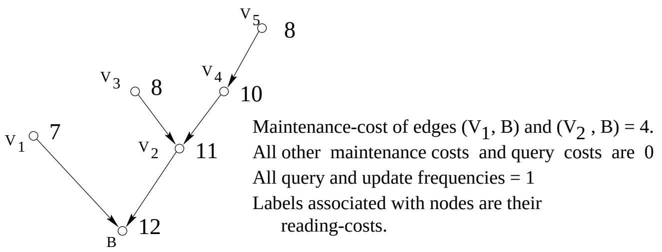


Fig. 3. $\mathcal{G}$: An OR view graph

Example 1. Consider the OR view graph $\mathcal{G}$ of Figure 3. In the given OR view graph $\mathcal{G}$, the maintenance-costs and query-costs associated with each edge is

[6] Note that the maintenance-length doesn't include the reading cost of the destination as in the query-length of a path.

zero, except for the maintenance-cost of 4 associated with the edges (V_1, B) and (V_2, B). Also, all query and update frequencies are uniformly 1. The label associated with each of the nodes in $\mathcal{G}$ is the reading-cost of the node. Also, the set of sinks $L = \{B\}$.

In the OR view graph $\mathcal{G}$, $Q(V_i, \phi) = 12$ for all $i \leq 5$, because as the query-costs are all zero, the minimum query-length of a path from V_i to B is just the reading-cost of B. Note that $Q(B, \phi) = 12$. Also, as the minimum maintenance-length of a path from a view V_i to B is 4, $UC(V_i, \phi) = 4$ for all $i \leq 5$.

Knapsack Effect We simplify the view-selection problem as in prior discussions ([HRU96, GHRU97, Gup97]) by allowing that a solution may consume "slightly" more than the given amount of constraint. This assumption is made to ignore the `knapsack` component of the view-selection problem. However, when proving performance guarantee of a given algorithm, we compare the solution delivered by the algorithm with an optimal solution that consumes the same amount of resource as that consumed by the delivered solution.

5 Inverted-Tree Greedy Algorithm

In this section, we present a competitive greedy algorithm called the *Inverted-Tree Greedy Algorithm* which delivers a near-optimal solution for the maintenance-cost view-selection problem in OR view graphs.

In the context of view-selection problem, a greedy algorithm was originally proposed in [HRU96] for selection of views in data cubes under a disk-space constraint. Gupta in [Gup97] generalized the results to some special cases of AND-OR view graphs, but still for the constraint of disk space. The greedy algorithms proposed in the context of view-selection work in stages. At each stage, the algorithm picks the "most beneficial" view. The algorithm continues to pick views until the set of selected views take up the given resource constraint. One of the key notions required in designing a greedy algorithm for selection of views is the notion of the "most beneficial" view.

In the greedy heuristics proposed earlier ([HRU96, GHRU97, Gup97]) for selection of views to materialize under a space constraint, views are selected in order of their "query benefits" per unit space consumed. We now define a similar notion of benefit for the maintenance-cost view-selection problem addressed in this article.

Most Beneficial View

Consider an OR view graph G. At a stage, when a set of views M has already been selected for materialization, the *query benefit* $B(C, M)$ associated with a set of views C with respect to M is defined as $\tau(G, M) - \tau(G, M \cup C)$. We define the *effective maintenance-cost* $EU(C, M)$ of C with respect to M as $U(M \cup C) - U(M)$.[7] Based on these two notions, we define the view that has

[7] The effective maintenance-cost may be negative. The results in this article hold nevertheless.

the most query-benefit per unit effective maintenance-cost with respect to M as the *most beneficial view* for greedy selection at the stage when the set M has already been selected for materialization.

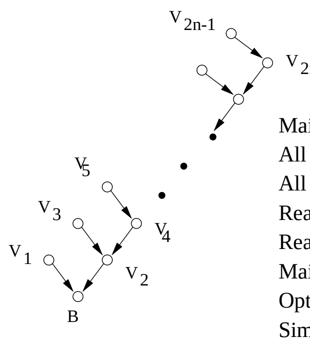
We illustrate through an example that a *simple greedy* algorithm, that at each stage selects the most beneficial view, as defined above, could deliver an arbitrarily bad solution.

Example 2. Consider the OR view graph $\mathcal{G}$ shown in Figure 3. We assume that the base relation B is materialized and we consider the case when the maintenance-cost constraint is 4 units.

We first compute the query benefit of V_1 at the initial stage when only the base relation B is available (materialized). Recall from Example 1 that $Q(V_i, \phi) = 12$ for all $i \leq 5$ and $Q(B, \phi) = 12$. Thus, $\tau(\mathcal{G}, \phi) = 12 \times 6 = 72$, as all the query frequencies are 1. Also, $Q(V_1, \{V_1\}) = 7$, as the reading-cost of V_1 is 7, $Q(V_i, \{V_1\}) = 12$ for $i = 2, 3, 4, 5$, and $Q(B, \{V_1\}) = 12$. Thus, $\tau(G, \{V_1\}) = 12 \times 5 + 7 = 67$ and thus, the initial query benefit of V_1 is $72 - 67 = 5$. Similarly, the initial query benefits of each of the views V_2, V_3, V_4, and V_5 can be computed to be 4.

Also, $U(\{V_i\}) = UC(V_i, \{V_i\}) = 4$ as the minimum maintenance-length of a path from any V_i to B is 4. Thus, the solution returned by the simple greedy algorithm, that picks the most beneficial view, as defined above, at each stage, is $\{V_1\}$.

It is easy to see that the optimal solution is $\{V_2, V_3, V_4, V_5\}$ with a query benefit of 11 and a total maintenance time of 4. To demonstrate the **non-monotonic** behavior of the benefit function, observe that the query-benefits per unit maintenance-cost of sets $\{V_2\}, \{V_3\}, \{V_2, V_3\}$ are 1, 1, and 7/4 respectively. This non-monotonic behavior is the reason why the simple greedy algorithm that selects views on the basis of their query-benefits per unit maintenance-cost can deliver an arbitrarily bad solution. Figure 4 illustrates through an extended example that the optimal solution can be made to have an arbitrarily high query benefit, while keeping the simple greedy solution unchanged.



Maintenance-cost of edges (V_1, B) and (V_2 , B) = 4.
All other edge maintenance costs and query costs are 0
All query and update frequencies = 1
Reading-cost of B = 12.
Reading-cost of V_i = 3, if i is odd, else 12.
Maintenance-time constraint = 4
Optimal solution is the set of all views except V_1
Simple Greedy solution = { V_1 }

Fig. 4. An OR view graph, $\mathcal{H}$, for which simple greedy performs arbitrarily bad

Note that the nodes in the OR view graphs $\mathcal{G}$ and $\mathcal{H}$, presented in Figure 3 and Figure 4 respectively, can be easily mapped into **real queries** involving aggregations over the base data B. The query-costs associated with the edges in $\mathcal{G}$ and $\mathcal{H}$ depict the *linear cost model*, where the cost of answering a query on v using its descendant u is directly proportional to the size of the view u, which in our model of OR view graphs is represented by the reading-cost of u. Notice that the minimum query-length of a path from u to v in $\mathcal{G}$ or $\mathcal{H}$ is R_v, the reading-cost of v. As zero maintenance-costs in the OR view graphs $\mathcal{G}$ and $\mathcal{H}$ can be replaced by extremely small quantities, the OR view graphs $\mathcal{G}$ and $\mathcal{H}$ depict the plausible scenario when the cost of maintaining a view u from a materialized view v be negligible in comparison to the maintenance cost incurred in maintaining a view u directly from the base data B.

Definition 7. (Inverted Tree Set) A set of nodes R is defined to be an inverted tree set in a directed graph G if there is a subgraph (not necessarily induced) T_R in the transitive closure of G such that the set of vertices of T_R is R, and the inverse graph[8] of T_R is a tree.[9]

In the OR view graph $\mathcal{G}$ of Figure 3, any subset of $\{V_2, V_3, V_4, V_5\}$ that includes V_2 forms an inverted tree set. The T_R graph corresponding to the inverted tree set $R = \{V_2, V_3, V_5\}$ has the edges (V_2, V_5) and (V_2, V_3) only.

The motivation for the inverted tree set comes from the following observation, which we prove in Lemma 1. In an OR view graph, an arbitrary set O (in particular an optimal solution O), can be partitioned into inverted tree sets such that the effective maintenance-cost of O with respect to an already materialized set M is greater than the sum of effective-costs of inverted tree sets with respect to M.

Based on the notion of an inverted tree set, we develop a greedy heuristic called *Inverted-tree Greedy Algorithm* which, at each stage, considers all inverted tree sets in the given view graph and selects the inverted tree set that has the most query-benefit per unit effective maintenance-cost.

Algorithm 1 <u>**Inverted-Tree Greedy Algorithm**</u>

Given: An OR view graph (G), and a total view maintenance time constraint S
BEGIN
  $M = \phi;\ B_C = 0;$
  repeat
    for each inverted tree set of views T in G such that $T \cap M = \phi$
      if $(EU(T, M) \leq S)$ **and** $(B(T, M)/EU(T, M) > \mathrm{B_C})$
        $B_C = B(T, M)/EU(T, M);$
        $C = T;$
      endif
    endfor

[8] The inverse of a directed graph is the graph with its edges reversed.
[9] A tree is a *connected* graph in which each vertex has exactly one incoming edge.

$M = M \cup C$;
until ($U(M) \geq S$);
return M;
END. $\diamond$

We prove in Theorem 1 that the Inverted-tree greedy algorithm is guaranteed to deliver a near-optimal solution. In Section 7, we present experimental results that indicate that in practice, the Inverted-tree greedy algorithm almost always returns an optimal solution. We now define a notion of update graphs which is used to prove Lemma 1.

Definition 8. (Update Graph) Given an OR view graph G and a set of nodes/views O in G. An update graph of O in G is denoted by U_O^G and is a subgraph of G such that $V(U_O^G) = O$, and $E(U_O^G) = \{(v, u) \mid u, v \in O$ and $v (\in O)$ is such that $UC(u, \{v\}) \leq UC(u, \{w\})$ for all $w \in O\}$. We drop the superscript G of U_O^G, whenever evident from context.

It is easy to see that an update graph is an embedded forest in G. An update graph of O is useful in determining the flow of changes when maintaining the set of views O. An edge (v, u) in an update graph U_O signifies that the view u uses the view v (or its delta tables) to incrementally maintain itself, when the set O is materialized. Figure 5 shows the update graph of $\{V_1, V_2, V_5\}$ in the OR view graph $\mathcal{G}$ of our running example in Figure 3.

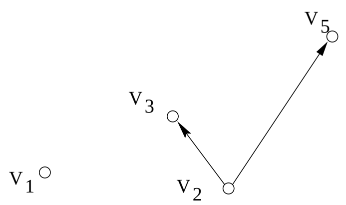


Fig. 5. The update-graph for $\{V_1, V_2, V_3, V_5\}$ in G

Lemma 1 *For a given set of views M, a set of views O in an OR view graph G can be partitioned into inverted tree sets $O_1, O_2, \ldots, O_m$, such that $\sum_{i=1}^{m} EU(O_i, M) \leq EU(O, M)$.*

Proof. Consider the update graph U_O of O in G. By definition, U_O is a forest consisting of m trees, say, $U_1, \ldots, U_m$ for some $m \leq |O|$. Let, $O_i = V(U_i)$, for $i \leq m$.

An edge (y, x) in the update graph U_O implies presence of an edge (x, y) in the transitive closure of G. Thus, an embedded tree U_i in the update graph U_O is an embedded tree in the transitive closure of the inverse graph of G. Hence, the set of vertices O_i is an inverted tree set in G.

For a set of views C, we use $UC(C,M)$ to denote the maintenance cost of the set C w.r.t. M, i.e., $UC(C,M) = \sum_{v \in C} g_v UC(v, M \cup C)$, where $UC(v,M)$ for a view v is as defined in Section 3. Now, the effective maintenance-cost of a set O_i with respect to a set M can be written as $EU(O_i, M) = (UC(O_i, M) + UC(M, O_i)) - U(M) = UC(O_i, M) - (U(M) - UC(M, O_i)) = UC(O_i, M) - Rd(M, O_i)$, where $Rd(M,C)$ is used to denote the reduction in the maintenance time of M due to the set of views C, i.e., $Rd(M,C) = U(M) - UC(M,C)$.

As, no view in a set O_i uses a view in a different set O_j for its maintenance, $UC(O,M) = \sum_{i=1}^{m} UC(O_i, M)$. Also, the reduction in the maintenance cost of M due to the set O is less than the sum of the reductions due to the sets $O_1, \ldots, O_m$, i.e., $Rd(M,O) \leq \sum_{i=1}^{m} Rd(M, O_i)$.

Therefore, as $EU(O,M) = UC(O,M) - Rd(M,O)$, we have $EU(O,M) \geq \sum_{i=1}^{m} EU(O_i, M)$. □

Theorem 1 *Given an OR view graph G and a total maintenance-time constraint S. The Inverted-tree greedy algorithm (Algorithm 1) returns a solution M such that $U(M) \leq 2S$ and M has a query benefit of at least $(1 - 1/e) = 63\%$ of that of an optimal solution that has a maintenance cost of at most $U(M)$, under the assumption that the optimal solution doesn't have an inverted tree set O_i such that $U(O_i) > S$.* ∎

The simplifying assumption made in the above algorithm is almost always true, because $U(M)$ is not expected to be much higher than S. The following theorem proves a similar performance guarantee on the solution returned by the Inverted-tree greedy algorithm without the assumption used in Theorem 1.

Theorem 2 *Given an OR view graph G and a total maintenance-time constraint S. The Inverted-tree greedy algorithm (Algorithm 1) returns a solution M such that $U(M) \leq 2S$ and $B(M,\phi)/U(M) \geq 0.5B(O,\phi)/S$, where O is an optimal solution such that $U(O) <= S$.* ∎

Dependence of Query and Update Frequencies Note that we have not made any assumptions about the independence of query frequencies and update frequencies of views. In fact, the query frequency of a view may decrease with the materialization of other views. It can be shown that the above performance guarantees hold even when the query frequency of a view decreases with the materialization of other views.

Time Complexity Let G be an OR view graph of size n and A_v be the number of ancestors of a node $v \in V(G)$. The number of inverted tree sets in G that are formed by a node $v \in V(G)$ as its root is 2^{A_v}, because any set of ancestors of v (which become a set of descendants in the inverse graph) form an inverted tree with v and any inverted tree set that has v as its root is formed from v and a subset of its ancestors. Therefore, the total number of inverted tree sets in an OR view graph G and also, the total time complexity of a stage of the Inverted-tree greedy algorithm is $\sum_{v \in V(G)} (2^{A_v})$, which is in the worst case exponential in n.

We note that for the special case of an OR view graph being a *balanced binary tree*, each stage of the Inverted-tree greedy algorithm runs in polynomial time $O(n^2)$, where n is the number of nodes in the graph. The number of inverted tree sets, $T(h)$, in a general balanced tree of height h can also be computed using the following recursion: $T(h) = ((2r)^h - 1)/(r-1) = ((n+1)^2 - 1)/(r-1) = O(n^2/r)$, where $r > 1$ is the branching factor of the tree.

As discussed in Section 7, our experiments show that the Inverted-tree greedy approach takes substantially less time than the A^* algorithm presented in the next section, especially for sparse graphs. Also, the space requirements of the Inverted-tree greedy algorithm is polynomial in the size of graph while that of the A^* heuristic is exponential in the size of the input graph.

6 A^* Heuristic

In this section, we present an A^* heuristic that, given an AND-OR view graph and a quantity S, deliver a set of views M that has an optimal query response time such that the total maintenance cost of M is less than S. Recollect that an A^* algorithm [Nil80] searches for an optimal solution in a search graph where each node represents a candidate solution. Roussopoulos in [Rou82] also demonstrated the use of A^* heuristics for selection of indexes in relational databases.

Let G be an AND-OR view graph instance and S be the total maintenance-time constraint. We first number the set of views (nodes) N of the graph in an inverse topological order $<v_1, v_2, \ldots, v_n>$ so that all the edges (v_i, v_j) in G are such that $i > j$. We use this order of views to define a binary tree T_G of candidate feasible solutions, which is the search tree used by the A^* algorithm to search for an optimal solution. Each node x in T_G has a label $<N_x, M_x>$, where $N_x = \{v_1, v_2, \ldots, v_d\}$ is a set of views that have been considered for possible materialization at x and $M_x (\subset N_x)$, is the set of views chosen for materialization at x. The root of T_G has the label $<\phi, \phi>$, signifying an empty solution. Each node x with a label $<N_x, M_x>$ has two successor nodes $l(x)$ and $r(x)$ with the labels $<N_x \cup \{v_{d+1}\}, M_x>$ and $<N_x \cup \{v_{d+1}\}, M_x \cup \{v_{d+1}\}>$ respectively. The successor $r(x)$ exists only if $M_x \cup \{v_{d+1}\}$ has a total maintenance cost of less than S, the given cost constraint.

We define two functions[10] $g : V(T_G) \mapsto \mathcal{R}$, and $h : V(T_G) \mapsto \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers. For a node $x \in V(T_G)$, with a label $<N_x, M_x>$, the value $g(x)$ is the total query cost of the queries on N_x using the selected views in M_x. That is,

$$g_x = \sum_{v_i \in N_x} f_{v_i} Q(v_i, M_x).$$

The number $h(x)$ is an estimated lower bound on $h^*(x)$ which is defined as the remaining query cost of an optimal solution corresponding to some descendant of x in T_G. In other words, $h(x)$ is a lower bound estimation of $h^*(x) = \tau(G, M_y) -$

[10] The function g is not to be confused with the update frequency g_v associated with each view in a view graph.

$g(x)$, where M_y is an optimal solution corresponding to some descendant y of x in T_G.

Algorithm 2 <u>A^* **Heuristic**</u>

Input: G, an AND-OR view graph, and S, the maintenance-cost constraint.
Output: A set of views M selected for materialization.
BEGIN
  Create a tree T_G having just the root A. The label associated with A is $<\phi, \phi>$.
  Create a priority queue (heap) $L = <A>$.
  repeat
    Remove x from L, where x has the lowest $g(x) + h(x)$ value in L.
    Let the label of x be $<N_x, M_x>$, where $N_x = \{v_1, v_2, \ldots, v_d\}$ for some $d \leq n$.
    if $(d = n)$ **RETURN** M_x.
    Add a successor of x, $l(x)$, with a label $<N_x \cup \{v_{d+1}\}, M_x>$ to the list L.
    if $(U(M_x) < S)$
      Add to L a successor of x, $r(x)$, with a label $<N_x \cup \{v_{d+1}\}, M_x \cup \{v_{d+1}\}>$.
  until (L is empty);
  RETURN NULL;
END. $\diamond$

We now show how to compute the value $h(x)$ for a node x in the binary tree T_G. Let $N = V(G)$ be the set of all views/nodes in G. Given a node x, we need to estimate the optimal query cost of the remaining queries in $N - N_x$. Let $s(v) = g_v UC(v, N)$, the minimum maintenance time a view v can have in presence of other materialized views. Also, if a node $v \in V(G)$ is not selected for materialization, queries on v have a minimum query cost of $p(v) = f_v Q(v, N - \{v\})$. Hence, for each view v that is not selected in an optimal solution M_y containing M_x, the remaining query cost accrues by at least $p(v)$. Thus, we fill up the remaining maintenance time available $S - U(M_x)$ with views in $N - N_x$ in the order of their $p(v)/s(v)$ values. The sum of the $f_v Q(v, N - \{v\})$ values for the views left out will give a lower bound on $h^*(x)$, the optimal query cost of the remaining queries. In order to nullify the knapsack effect as mentioned in Section 4, we start with leaving out the view w that has the highest $f_v Q(v, N - \{v\})$ value.

Theorem 3 *The A^* algorithm (Algorithm 2) returns an optimal solution.* ∎

The above theorem guarantees the correctness of A^* heuristic. Better lower bounds yield A^* heuristics that will have better performances in terms of the number of nodes explored in T_G. In the worst case, the A^* heuristic can take exponential time in the number of nodes in the view graph. There is no better bounds known for the A^* algorithm in terms of the function $h(x)$ used.

7 Experimental Results

We ran some experiments to determine the quality of the solution delivered and the time taken in practice by the Inverted-tree Greedy algorithm for OR view graphs that arise in practice. We implemented both the algorithms, Inverted-tree Greedy and A^* heuristic, and ran them on random instances of OR view graphs that are balanced trees and directed acyclic OR view graphs with varying edge-densities, and random query and update frequencies. We used a linear cost model [HRU96] for the purposes of our experiments.

We made the following observations. The Inverted-tree Greedy Algorithm (Algorithm 1) returned an optimal solution for almost all (96%) view graph instances. In other cases, the solution returned by the Inverted-tree greedy algorithm had a query benefit of around 95% of the optimal query benefit. For balanced trees and sparse graphs having edge density less than 40%, the Inverted-tree greedy took substantially less time (a factor of 10 to 500) than that taken by the A^* heuristic. With the increase in the edge density, the benefit of Inverted-tree greedy over the A^* heuristic reduces and for very dense graphs, A^* may actually perform marginally better than the Inverted-tree greedy. One should observe that OR view graphs that are expected to arise in practice would be very sparse. For example, the the OR view graph corresponding to a data cube having n dimensions has $\sum_{i=1}^{n}(\binom{n}{i}2^i) = 3^n$ edges and 2^n vertices. Thus, the edge density is approximately $(0.75)^n$, for a given n.

The comparison of the times taken by the Inverted-tree greedy and the A^* heuristic is briefly presented in Figure 6. In all the plots shown in Figures 6, the different view graph instances of the maintenance-cost view-selection problem are plotted on the x-axis. A view graph instance G is represented in terms of N, the number of nodes in G, and S, the maintenance-time constraint. The view graph instances are arranged in the lexicographic order of (N, S), i.e., all the view graphs with smallest N are listed first, in order of their constraint S values. In all the graph plots, the number N varied from 10 to 25, and S varied from the time required to maintain the smallest view to the time required to maintain all views in a given view graph.

8 Conclusions

One of the most important decisions in design of a data warehouse is the selection of views to materialize. The view-selection problem in a data warehouse is to select a set of views to materialize so as to optimize the total query response time, under some resource constraint such as total space and/or the total maintenance time of the materialized views. All the prior work done on the view selection problem considered only a disk-space constraint. In practice, the real constraining factor is the total maintenance time. Hence, in this article, we have considered the maintenance-cost view-selection problem where the constraint is of total maintenance time.

As the maintenance-cost view-selection problem is intractable, we designed an approximation algorithms for the special case of OR view graphs. The OR view graphs arise in many practical applications like data cubes, and other OLAP applications. For the general case of AND-OR view graphs, we designed an A^* heuristic that delivers an optimal solution.

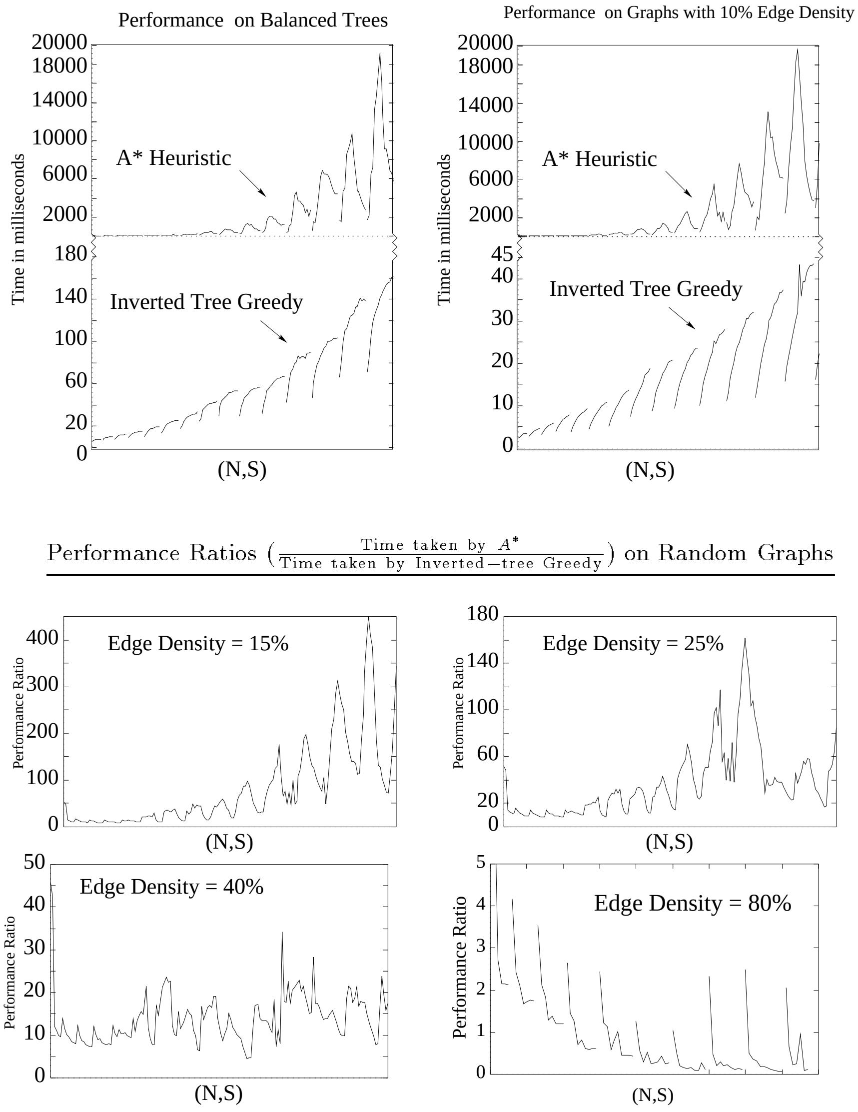


Fig. 6. Experimental Results. The x-axis shows the view graph instances in lexicographic order of their (N, S) values, where N is the number of nodes in the graph and S is the maintenance-time constraint.

Our preliminary experiment results are very encouraging for the Inverted-tree Greedy algorithm. Also, the space requirement of the Inverted-tree greedy heuristic is polynomial in size of the graph, while that of the A^* heuristic grows exponentially in the size of the graph.

References

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proc. of the VLDB*, 1997.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the VLDB*, 1991.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection in OLAP. In *Proc. of the Intl. Conf. on Data Engineering*, 1997.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2), 1995.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the Intl. Conf. on Database Theory*, Delphi, Greece, 1997.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Eng. Bulletin, Special Issue on Materialized Views and Data Warehousing*, 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD*, June 1996.

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

[MQM97] I. Mumick, D. Quass, and B. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM SIGMOD International Conference of Management of Data*, 1997.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., 1980.

[Rou82] N. Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[RSS96] K. A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the ACM SIGMOD*, 1996.

[TS97] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proceedings of the 23rd Intl. Conf. on Very Large Data Bases*, 1997.

[WGL+96] J. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Workshop on Materialized Views: Techniques and Applications*, 1996.

[Wid95] J. Widom. Research problems in data warehousing. In *Proc. of the Intl. Conf. on Information and Knowledge Management*, 1995.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proc. of the VLDB*, 1997.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1995.

This article was processed using the LaTeX macro package with LLNCS style