Selecting Views with Maintenance Cost Constraints: Issues, Heuristics and Performance

Jeffrey Xu Yu
The Chinese University of Hong Kong, Hong Kong, China
email: yu@se.cuhk.edu.hk

Chi-Hon Choi
The Chinese University of Hong Kong, Hong Kong, China
email: chchoi@se.cuhk.edu.hk

Gang Gou
The Chinese University of Hong Kong, Hong Kong, China
email: ggou@se.cuhk.edu.hk

Hongjun Lu
Hong Kong Uni. of Science and Technology, Hong Kong, China
email: luhj@cs.ust.hk

In order to efficiently support a large number of on-line analytical processing (OLAP) queries, a data warehouse needs to precompute or materialise some of such OLAP queries. One of the important issues is how to select such a set of materialised views in order to minimise the total processing cost for OLAP queries. The maintenance-cost view-selection problem is to select a set of materialised views under a maintenance cost constraint (such as maintenance time), in order to minimise the total query processing cost for a given set of queries. This problem is more difficult than the view selection problem under a disk-space constraint, because a selected view may make the previously selected views less beneficial, due to the fact that the total maintenance cost for a set of views may decrease when more views are materialised while the maintenance cost always increases under disk-space constraint. The problem has recently received significant attention. Several greedy/heuristic algorithms were proposed. However, the quality of the greedy/heuristic algorithms has not been well analysed. In this paper, in a multidimensional data warehouse environment, we re-examine the greedy/heuristic algorithms in various settings, and provide users with insights on the quality of these heuristic algorithms.

ACS Classification: H.2 (Database Management)

1. INTRODUCTION

Business landscape is quickly evolving, and markets are much more competitive and dynamic than ever. Businesses in every segment of the industry realise that their corporate and customer databases are gold mines of information that could give them a critical edge by helping them to manage

Manuscript received: 20 February 2003
Communicating Editor: John Roddick

investment, map market development, identify new customer prospects, anticipate demands on banking services, and predict consumer preferences and habits. As a collection of decision support techniques that aim at enabling executives, managers and analysts to make better and faster decision, data warehouse and on-line analytical processing (OLAP) have been successfully deployed in many industries such as manufacturing, retail, financial services, transportation, telecommunications, health-care, etc. The amount of potential data that needs to be maintained tends to be hundreds of gigabytes or terabytes in size. User queries include ad-hoc queries and complex queries that need to conduct statistical analysis by performing aggregate operations against millions of records. Therefore, OLAP query processing time becomes critical since executives, managers and analysts need to make decisions in a short time.

Precomputing OLAP queries – materialising views with aggregate functions – has been widely used as a common technique in data warehouses (Gupta and Mumick, 1999a). In a multidimensional data warehouse that consists of a fact table and a collection of dimension tables (Kimball, 1997), OLAP queries (views) can be formalised as a dependent lattice (Harinarayan, Rajaraman and Ullman, 1996), which can be represented as a directed acyclic graph. Take a real store chain application that only has four dimensions, namely, Product (50 attributes), Store (20 attributes), Time (10 attributes) and Promotion (10 attributes) given in Baralis, Paraboschi and Teniente (1997) and Kimball (1997), as an example. The corresponding graph may have over 2^{90} vertices. The potential graphs to be dealt with can be very large. The materialised view selection problem is to select an appropriate set of materialised views under a maintenance resource constraint, in order to minimise the total query processing cost for all queries. The disk-space view-selection problem is to select a set of interrelated views that minimises the total query processing cost under a given disk-space constraint (Harinarayan, Rajaraman and Ullman, 1996; Gupta *et al*, 1997; Gupta, 1997; Shukla, Deshpande and Naughton, 1998). For the disk-space view-selection problem, the benefit of the view having been chosen will remain unchanged in the subsequent view-selection process. It is formally defined as a monotonic property.

However, in real applications, the real constraint is more likely to be the maintenance-cost incurred in maintaining the materialised views up to date in a data warehouse. The maintenance-cost view-selection problem has been proven to be NP-hard (Gupta and Mumick, 1996b). In Gupta and Mumick (1996b), it claimed that the greedy algorithms that select views on the basis of query-benefit per unit maintenance-cost can deliver an arbitrarily bad solution due to the non-monotonic property of the maintenance-cost view-selection problem. In other words, a selected view may make the previous selected views less beneficial, because the total maintenance cost for a set of selected views may decrease when more views are materialised. Gupta and Mumick proposed two algorithms, namely, inverted-tree greedy and A*-heuristic to solve this intractable problem in OR view graph and AND-OR view graph respectively (Gupta and Mumick, 1996b). In Liang *et al* (2001), Liang *et al* proposed two algorithms, two-phase greedy and integrated greedy, to solve the maintenance-cost view-selection problem. The two algorithms are designed on the basis of query-benefit per unit maintenance-cost. Liang *et al* (2001) claimed that the two algorithms are able to find feasible solutions in polynomial time. But, no analytical and performance studies were given in Liang *et al* (2001).

To deal with the maintenance-cost view-selection problem, algorithms that provide a near optimal solution in polynomial time are highly desirable. But, the arguments presented in Gupta and Mumick (1996b) and Liang *et al* (2001) are not consistent. On the basis of query-benefit per unit maintenance-cost, the former indicated that greedy algorithms can generate an arbitrarily bad solution. The latter argued that greedy algorithms can possibly generate feasible solutions. The algorithms cannot be used without any systematic study on the quality.

Our main contribution, in this paper, is to provide users with insights on the heuristic algorithms in terms of both processing time for the algorithms to find a solution and the effectiveness of the algorithms to minimise query processing cost. We investigated inverted-tree greedy, A*-heuristic, two-phase greedy and integrated greedy algorithms in various settings for the general case of dependence lattice. The academic significance of this work is of twofold. First, it provides a view on how to use the heuristic algorithms. Second, it assists us to design new heuristic algorithms. Based on our extensive studies, we observe that greedy algorithms perform well under certain conditions. Our finding explored that the existing A*-heuristic (Gupta and Mumick, 1996b) cannot always find optimal solutions[1].

The rest of the paper is organised as follows. The maintenance-cost view-selection problem will be defined in Section 2 with a general cost model. Motivation will be addressed in Section 3 including examples. Section 4 discusses the four existing algorithms: the inverted-tree greedy, the A*-heuristic, the two-phase greedy and the integrated greedy for solving the maintenance-cost view-selection problem. Discussions are included to outline the strength and weakness of these four algorithms. We conducted extensive experience studies, and will report the result in Section 5. We conclude this paper in Section 6.

2. THE MAINTENANCE-COST VIEW-SELECTION PROBLEM

Like Harinarayan *et al* (1996), we denote a lattice with a set of elements (queries and views) L and a dependence relation $\preceq$ (derived-from, be-computed-from) by $(L, \preceq)$. Given two queries q_i and q_j. We say q_i is dependent on q_j, $(q_i \preceq q_j)$, if q_i can be answered using the results of q_j. The lattice $(L, \preceq)$ is called *dependent lattice*. For elements a and b of a lattice $(L, \preceq)$, $a \prec b$ means $a \preceq b$ and $a \neq b$. The ancestors and descendents of an element of a lattice (L, $\preceq$), are defined as *ancestor* $(a) = \{b \mid a \preceq b\}$ and *descendant*$(a) = \{b \mid b \preceq a\}$, respectively. A dependent lattice can be represented as a directed acyclic graph in which the lattice elements are vertices and there is an edge from a to b, if $b \prec a$ and $\nexists\, c(b \prec c \wedge c \prec a)$. There is a path downward from a to b if and only if $b \preceq a$.

A Multidimensional Database (MDDB) is a collection of relations, $D_1, \ldots, D_m, F$, where D_i is a dimension table and F is a fact table as described in Baralis, Paraboschi and Teniente (1997) and Kimball (1996). It is worth noting that, in most real applications, an MDDB consists of multiple dimensions each of them in turn can be organised as hierarchies of attributes. Consider the large grocery store chain example given in Baralis, Paraboschi and Teniente (1997) and Kimball (1996). The store chain has four dimensions, namely, `Product`, `Store`, `Time` and `Promotion`. The `Product` dimension has more than 50 attributes such as brand, category, diet-type, package type, weight, case size and a merchandise hierarchy. The `Store` dimension has more than 20 attributes including store address, telephone number, manager, description of store services, and sizes of different departments. It also has a geographic hierarchy. The `Time` dimension is characterised by the granularity of day: day in the month, in the quarter and in the year, holiday, special events, as many as more than 10 attributes. There are different granularity of the time hierarchy, namely, day, week and month, year. The `Promotion` dimension contains 10 attributes such as the promotion type, promotion cost and start/end data, etc.

Suppose that an MDDB has m dimensions and the i-th dimension has n_i attributes. Assume that each dimension is characterised as a *dimensional dependent lattice*. The dependent lattice for the i-th dimension will have 2^{n_i} elements. If queries/views can be issued/made by grouping any subsets of attributes from any or no member of m dimensions, the total number of elements for the MDDB

[1] The preliminary work was reported in CYG02.

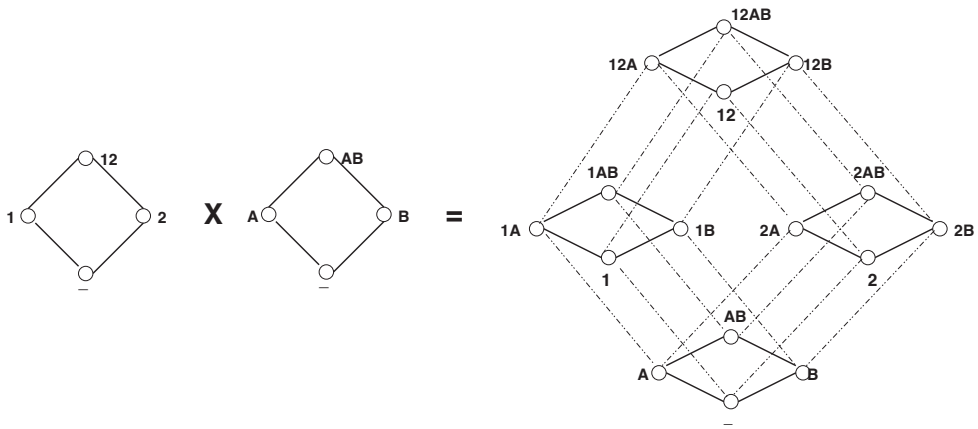


Figure 1: A simple direct product example

will be $\prod_i^m (2^{n_i} + 1)$. Therefore, the dependent lattice $(L, \preceq)$ in question is much more complex than a hypercube lattice. As for the store chain example, the number of elements is greater than 2^{90}. Here, let $(a_1, a_2, \ldots, a_m)$ be an m-tuple where each a_i is a point in the hierarchy of the i-th dimension, the dependence relation $\preceq$ can be defined as $(a_1, a_2, \ldots, a_m) \preceq (b_1, b_2, \ldots b_m)$ if and only if $a_i \preceq b_i$ for all i. This is called the *direct product* of the dimensional lattices in Harinarayan *et al* (1996). A simple direct product of two dimensional lattices is shown in Figure 1.

A direct-product of the dimensional lattice is represented as a directed acyclic graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$. We use $V(G)$ for the set of vertices of a graph G. The graph G has the following weights associated with vertices and edges.

- three weights on a vertex v:
  - r_v: initial data scan cost.
  - f_v: query frequency.
  - g_v: update frequency.
- two weights on an edge (v, u)
  - $w_{qu,v}$: query processing cost of u using v.
  - $w_{mu,v}$: updating cost of u using v.

In a general setting, given a query u and a selected materialised view v, a function $q(u, v)$ is the sum of the query processing costs associated with edges on the shortest path from v to u plus initial data scan cost of the vertex v, r_v. With $q(u, v)$, the raw table will be used instead of v, if and only if the view v cannot answer the query u. In a similar fashion, $m(u, v)$ is the sum of the maintenance-costs associated with the edges on the shortest path from v to u. It is important to know that we attempt to adopt a more general cost model than the linear cost model (Harinarayan *et al*, 1996) used in most of the existing work. The linear cost model states that the cost of answering a query, u, using one of its ancestors, v, is the number of rows present in the table v (r_v). Here, as shown in the two functions $q()$ and $m()$, we assume a general query processing cost and maintenance cost model. First, a query processing cost can be different from a maintenance cost for a pair of vertices. Mumick *et al* (1997). proposed a method of maintaining materialised aggregate views, called the summary-delta table

method, to efficiently maintain a materialised view (Mumick, Quass and Mumick, 1997). The maintenance costs can possibly be much lower than the query processing costs. Second, we also assume that query processing cost may involve other query processing costs (associated with edges) in addition to the initial table scan costs (associated with vertices). For example, given two dimensions, D_1 and D_2. Assume that the table for D_1D_2 is sorted on D_1. The cost of performing aggregate D_1 using D_1D_2 is different from the cost of performing aggregate D_2 on D_1D_2 due to sorting order. The cost differences need to be addressed as weights associated with edges. Third, there are multiple paths from a view to a query. In our settings, for simplicity, we select the shortest path as its cost.

We adopt the similar notations and definitions used in Gupta and Mumick (1996b) to define the maintenance-cost view-selection problem. Given an above-mentioned graph $G = (E, V)$ and a set of queries, Q ($\subseteq V(G)$). The maintenance-cost view-selection problem is to select a set of views M ($\subseteq V(G)$) that minimises total query processing, $\tau(G, M)$, where

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

under the constraint that the total maintenance cost, $U(M)$, is less than a maintenance cost S, such as $U(M) \leq S$, where $U(M)$ is defined as

$$U(M) = \sum_{v \in M} g_v \cdot m(v, M)$$

Here, $q(v, M)$ denotes the minimum cost of answering a query v ($\in V(G)$) in the presence of the set of materialised views, M, and $m(v, M)$ is the minimum cost of maintaining a materialised view v in the presence of the set of materialised views, M.

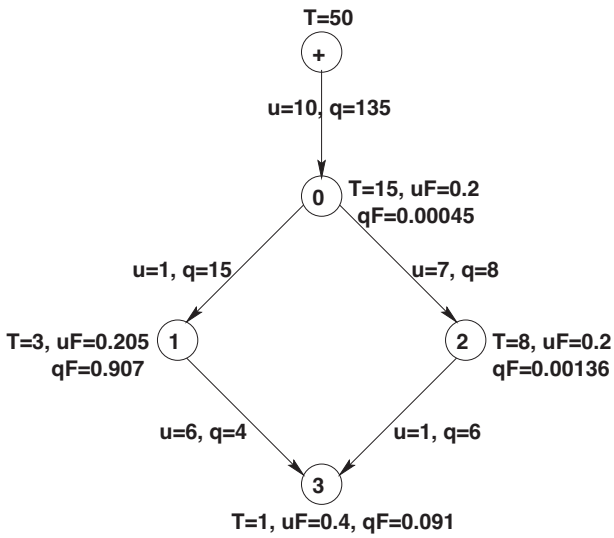


Figure 2: An example

3. MOTIVATIONS

The main idea of the greedy heuristics proposed in Harinarayan *et al* (1996), Gupta *et al* (1997), Gupta (1997) and Liang *et al* (2001) is to select materialised views, in *order* of query benefit per unit space/time consumed, which is given below.

$$QBPU(v,M) = \Delta Q_v / \Delta T_v \quad (1)$$

Here, M is a set of selected views, v is a view to be added, $\Delta Q_v = \pi(G, M) - \pi(G, M \cup \{v\})$, and $\Delta T_v = U(M \cup \{v\}) - U(M)$.

The question is whether it is possible to use such an order for the problem that does not have the so-called monotonic property. We address the related issues below. Let C_{min} be the minimum maintenance-cost constraint that allows all views to be selected as materialised views. Some examples are shown in Figure 2 and Figure 3. Here, the root (+) represents the raw table. `T`, `u`, `q`, `uF` and `qF` are, table size (initial data scan cost (r_v)), maintenance cost ($w_{m_{u,v}}$), query processing cost ($w_{q_{u,v}}$), update frequency (g_v) and query frequency (q_v), respectively.

Issue 1: *The total maintenance cost may decrease when a new materialised view is selected. However, a greedy algorithm always selects the greatest query benefit per time constraint.*

Consider an example in Figure 2. Initially, v_1 has the largest query benefit per unit time consumed. At the second stage, the greedy heuristic considers all the remaining views, $\{v_0, v_2, v_3\}$ one by one. The result is shown in the following table.

M	v	ΔT_v	$QBPU(v, M)$	Remaining Constraint
$\{v_1\}$	v_0	-0.05	-6.171	1.80
$\{v_1\}$	v_2	3.40	0.074	-1.65
$\{v_1\}$	v_3	2.40	0.227	-0.65

It shows that ΔT_v becomes negative when v_0 is added to the set of materialised views, M. However, v_3 will be selected because it has the most query benefit per unit time consumed. Due to the total maintenance cost will go beyond the C_{min}, neither v_3 nor v_2 will be selected. The resulting set of materialised views is $\{v_1\}$. But the optimal solution is $\{v_0, v_1, v_2, v_3\}$.[2]

The quality of heuristics varies dramatically in different settings. Consider another example in Figure 3. Table 1, Table 2 and Table 3 show the quality of the four algorithms: inverted-tree greedy, A*-heuristic, two-phase greedy and integrated greedy in three cases. In these tables, the column of `views` gives the resulting set of materialised views found by the algorithm specified in the column of `Algorithm`. `Q-cost` and `M-cost` are the total query processing cost and total maintenance cost, respectively, when the set of views have been materialised. These tables show that none of these four algorithms can always outperform others.

Issue 2: *Given a large maintenance-cost constraint, heuristic solutions may not be able to select all vertices as views.*

As can be seen in Table 1, when the maintenance-cost constraint is C_{min}, the optimal solution is to select all views, because the maintenance-cost constraint allows to do so. A*-heuristic and the integrated greedy are able to materialise all views. But the inverted-tree greedy and the two-phase greedy cannot achieve the optimal.

Issue 3: *A greater query benefit per unit time consumed does not necessarily lead to the minimum total query processing cost. Two subissues are: (a) selecting a view may make the further view*

[2] Note: most greedy heuristic algorithms use zero as a lower bound of the query benefit per unit time consumed.

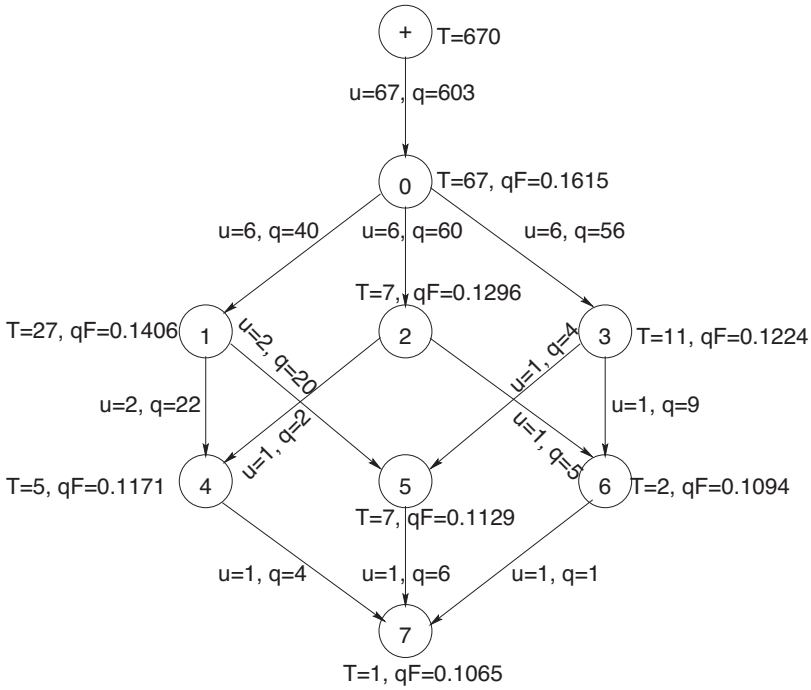


Figure 3: Another example (all update frequences are 0.125)

selections unsuccessful, and (b) not selecting the potential best view at one stage may make it unable to be selected again.

Table 2 shows an example when the maintenance-cost constraint is $0.96 \times C_{min}$. The integrated greedy might not achieve the optimal, because a potentially beneficial view might not be selected again, if it cannot be selected at the stage at which it must be selected. Suppose the set of materialised views is $\{v_0, v_2, v_3, v_6\}$ (Figure 3). The selection of v_1 fails, because the total maintenance cost will go beyond C_{min}, if v_1 is selected. The resulting set of materialised views becomes $\{v_0, v_2, v_3, v_6, v_5, v_4, v_7\}$. But the optimal solution is $\{v_0, v_1, v_2, v_3\}$ including v_1.

Issue 4: *A solution provided by A*-heuristic is not always optimal.*

Algorithm	Views	Q-cost	M-cost
Optimal	$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$	18.571	11.125
Inverted-Tree	$\{v_0\}$	115.429	8.375
A*-heuristic	$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$	18.571	11.125
Two-Phase	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Integrated	$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$	18.571	11.125

Table 1: Performance for Figure 3 with constraint = C_{min}

Algorithm	Views	Q-cost	M-cost
Optimal	$\{v_0, v_1, v_2, v_3\}$	22.314	10.625
Inverted-Tree	$\{v_0\}$	115.429	8.375
A*-heuristic	$\{v_0, v_1, v_2, v_3\}$	22.314	10.625
Two-Phase	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Integrated	$\{v_0, v_2, v_3, v_4, v_5, v_6, v_7\}$	29.818	10.375

Table 2: Performance for Figure 3 with constraint = 0.96 × C_{min}

Algorithm	Views	Q-cost	M-cost
Optimal	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Inverted-Tree	$\{v_0\}$	115.423	8.375
A*-heuristic	$\{v_0, v_1, v_2, v_6\}$	37.474	10.000
Two-Phase	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Integrated	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000

Table 3: Performance for Figure 3 with constraint = 0.9 × C_{min}

As seen in Table 3, when the maintenance-cost constraint is 0.90 × C_{min}, A*-heuristic cannot provide an optimal solution.

4. EXISTING ALGORITHMS

In this section, in brief, we introduce four algorithms: the inverted-tree greedy (Gupta and Mumick, 1999b), the A*-heuristic (Gupta and Mumick, 1999b), the two-phase greedy (Liang *et al*, 2001), and the integrated greedy (Liang *et al*, 2001). Observations will be given.

Algorithm 1: A*-Heuristics (Gupta and Mumick, 1999b)

Input: A graph $G(V, E)$ and a maintenance-cost constraint S.
Output: a set of materialised views.

1. **begin**
2. Create a tree T_G having just the root A. The label associated with A is $\langle \phi, \phi \rangle$.
3. Create a priority queue (heap) $L = \langle A \rangle$
4. **repeat**
5. Remove x from L, where x has the lowest $g(x) + h(x)$ value in L
6. Let the label of x be $\langle N_x, M_x \rangle$, where $N_x = \{v_1, v_2, \ldots, v_d\}$ for some $d \leq n$.
7. **if** $d = n$ **then**
8. **return** M_x
9. **end if**
10. Add a successor of x, $l(x)$, with a label $\langle Nx \cup \{v_{d+1}\}, M_x \rangle$ to the list L.
11. **if** $(U(M_x) < S)$ **then**
12. Add to L a successor of x, $r(x)$, with a label $\langle N_x \cup \{v_{d+1}\}, M_x \cup v_{d+1} \rangle$
13. **end if**
14. **until** (L is empty);
15. **return** $\varnothing$;
16. **end**

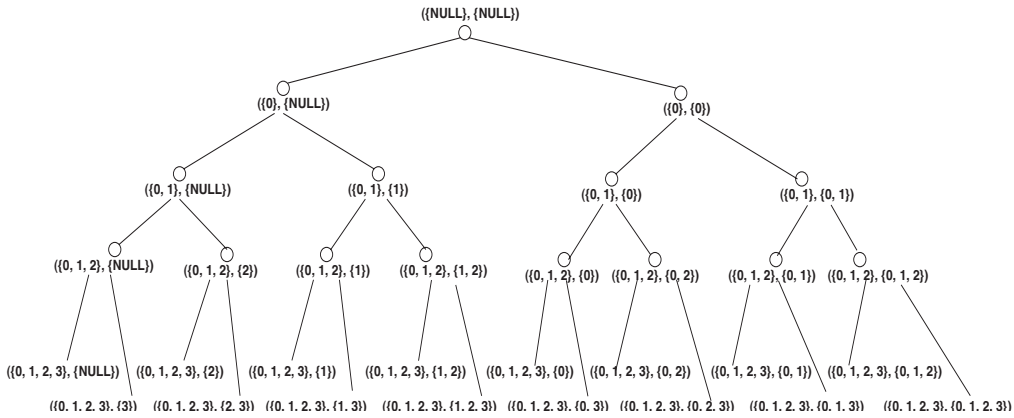


Figure 4: The binary search tree T_g of candidate solutions for Figure 2

4.1 A*-Heuristic

The A*-heuristic is shown in Algorithm 1. The A*-heuristic uses an *inverse topological order* to find a set of materialised views. It defines a binary tree T_G whose leaf vertices are the candidate solutions of this problem. At each stage of searching, A*-heuristic evaluates the benefit of remaining downward branches, and selects the branch of the greatest benefit to go down. A binary search tree is shown in Figure 4. Each vertex in this binary search tree has a label $\langle N_x, M_x \rangle$ ($M_x \subseteq N_x$), where M_x is the set of views which have been chosen to materialise and considered to answer the set of queries N_x. The search space is $2^{|V(G)|}$, where $V(G)$ is the set of vertices of the graph G. They estimated the benefit of the downward branches by summing up two functions $g(x)$ and $h(x)$. $g(x)$ is the total query processing cost of the queries on N_x using the selected views in M_x. $h(x)$ 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 (Gupta and Mumick, 1999b).

4.1.1 Discussions

In Table 1 and Table 2, A*-heuristic can reach an optimal solution. However, in Table 3, A*-heuristic can only reach a near-optimal feasible solution, not the optimal solution. The reason is that the expected benefit, $h(x)$, is very difficult to estimate accurately. It shows that A*-heuristic delivers an optimal solution only when $h(x) \leq h^*(x)$. The A*-heuristic may not reach an optimal solution under some critical maintenance-cost constraint. The A*-heuristic can take exponential time, in the worst case, with respect to the number of vertices in the graph (Gupta and Mumick, 1999b).

4.2 Inverted-Tree Greedy

The inverted-tree greedy uses a concept called an inverted tree set. Given a vertex v in a directed graph, an inverted tree set contains the vertex v and any subset of vertices reachable from v. The inverted-tree greedy is shown in Algorithm 2, where $B(C,M)$ is the query benefit associated with a set of vertices C with respect to M as $\tau(T, M) - \tau(T, M \cup C)$, and $EU(C, M)$ is the effective maintenance-cost of C with respect to M as $U(M \cup C) - U(M)$. At each stage, this algorithm considers all inverted tree sets of views, T, in the given graph G, such that $T \cap M = \varnothing$, and selects the inverted tree set that has the most query-benefit per unit effective maintenance-cost.

Algorithm 2: Inverted-Tree Greedy (Gupta and Mumick, 1999b)

Input: A graph $G(V, E)$ and a maintenance-cost constraint S.
Output: a set of materialised views.

```
1. begin
2. M ← φ; B_C ← 0;
3. repeat
4.    for each inverted-tree set of views T in G such that T ∩ M= φ do
5.       if (EU(T, M) ≤ S) and (B(T, M) / EU(T, M) > B_C) then
6.          B_C ← B(T, M) / EU(T, M); C ← T;
7.       end if
8.    end for
9.    M ← M ∪ C;
10. until (U(M) ≥ S) ;
11. return M;
12. end
```

4.2.1 Discussions

The steps for selecting a set of views for the example in Figure 3 is given below. In Table 1, we assume that the maintenance-cost constraint is the minimum cost that allows all vertices to be selected as materialised views. In the first step, the algorithm selects v_0, because it has the maximum $B(T, M) / EU(T, M)$. In the following steps, it cannot select any more vertices. The reason is that the query-benefit per unit effective-maintenance-cost does not increase by adding any new vertices.

Some observations can be made below for the inverted-tree greedy. First, the inverted-tree greedy does not guarantee a strict maintenance-cost constraint, it satisfies a limit within twice the maintenance-cost constraint. Second, the total time complexity of a stage of the inverted-tree greedy is $\Sigma_{v \in V(G)}(2^{Av})$, where $V(G)$ is the set of vertices of the graph and Av is the number of descendants of a vertex. In the worst case, it is exponential with respect to $|V(G)|$ as shown in Gupta and Mumick (1999b). Third, in our extensive experimental studies, we found that the inverted-tree greedy always chooses the first vertex as a part of its solution. The reason is that the algorithm calculates both the effective maintenance-cost and the query-benefit per unit effective-maintenance-cost. After selecting the first vertex, query-benefit per unit effective-maintenance-cost of other vertices is smaller than the first vertex. Therefore, the algorithm cannot effectively select any other views, simply because the query-benefit per unit effective-maintenance-cost of the first vertex is the highest in a sense that other vertices can be derived from it. Finally, by adding a new view into the set of views, the query benefit will increase. However, the query-benefit per unit effective-maintenance-cost tends to decrease, which possibly makes the further selection of vertices fail as shown in the examples.

4.3 Two-Phase Greedy

The two-phase greedy (Laing *et al*, 2001) is illustrated in Algorithm 3, the basic idea is that it selects a subset of the materialised views, M_1, to minimise the total query processing cost without considering the maintenance-cost constraint. If the total maintenance cost, $U(M_1)$, for all the views in M_1, is less than or equal to the given cost constraint, S, then, all the views in M_1 will be materialised. Otherwise, we need to further find a subset of M_1, denoted M_2, such that (i) all the views in M_1–M_2 can be derived from M_1, and (ii) $U(M_2) \leq S$ and the total query processing cost is minimised. The item (i) guarantees that all the queries can be answered using the views in M_1.

Algorithm 3: Two-Phase Greedy (Laing *et al*, 2001)

Input: A graph $G(V, E)$ and a maintenance-cost constraint S.
Output: a set of materialised views.

1. **begin**
2. Find a set of materialised views, M_1, to minimise the total query processing cost;
3. **if** $U(M_1) \leq S$ **then**
4. **return** M_1;
5. **else**
6. Find a subset of M_1, denoted as M_2, that minimises the total query processing cost and satisfies S.
7. **return** M_2;
8. **end if**
9. **end**

In order to make the query cost function to be consistent when comparing the two-phase greedy with other algorithms, we revise the linear query cost function in Laing *et al* (2001) as follows.

$$g(v) = \frac{\sum_{\exists v_j \in M_1 - M_2} [q(v_j, M_2) - q(v_j, M_2 \cup \{v\})] w(v_j)}{\Delta T_v} \quad (2)$$

The function $g(v)$ for obtaining a gain value for a vertex v takes the following factors into consideration. First, adding a new vertex v into M_2 incurs additional maintenance cost for v. However, the newly added vertex v is possible to reduce maintenance cost for the vertices that have already been selected in M_2, because those vertices may be able to use the vertex v to reduce the maintenance cost. Therefore, ΔT_v may be negative. Second, it considers effectiveness of this selection (vertex v) for all unselected views. It gives a query benefit for selecting this vertex v. Third, the weight for a vertex $v \in M_1$ is the sum of query frequencies for all the queries that choose v as its view. The weight gives a good estimation on the importance of a view $v \in M_1$, and is different from the query frequency for v itself.

In the first step, two-phase greedy heuristic do not consider maintenance cost. It reduces the problem to a minimum weighted maximum cardinality matching problem on a weighted bipartite graph, which can be solved in polynomial time. An example is shown in Figure 5. The vertex v_0 is possible to answer v_1, v_2 and v_3. The algorithm creates three copies of v_0, each of which has an edge associated with corresponding query vertex it can be used to answer. For every edge (v_i, q_j), there is one weight assigned to it. Then, the problem of finding M_1 is reduced to a *minimum weighted*

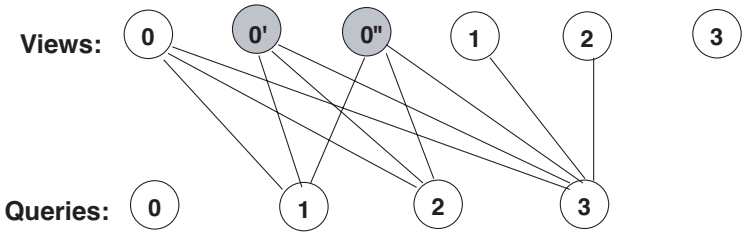


Figure 5: A bipartite graph example for Figure 2

maximum cardinality matching problem on a bipartite graph G_B based on G, which can be solved in polynomial time. Apparently, *minimum weighted* ensures that the sum of query processing cost is minimal while *maximum cardinality* ensures that the cost of all the queries is considered as optimisation target. In the second step, it further selects a subset of the set of materialised views, M_1, selected in the first step.

To avoid delivering a local optimal solution, Liang *et al* (2001) also supposed some variants to improve the performance of two-phase greedy. We call them Minimise-Lost-Benefit variant and Maximise-Benefit variant.

- **Minimise-Lost-Benefit:** At any stage of iteration, when there is no enough remaining maintenance time to select another materialised view, v, a materialised view u with the minimum *lost benefit* in the selected materialised view set is selected as a victim to be replaced with v. Consequently, it only supports soft constraint. In other words, it allows the maintenance time for the selected views to be greater than the given constraint.
- **Maximise-Benefit:** The maximise-benefit is based on a *compatible class*. Let the view set M' be the materialised views which has been found by applying the two-phase greedy. It further improves the solution by replacing a view $v \in M'$, with a minimum gain benefit, with a view $u \notin M'$ in the compatible class of v such that (i) u has never been materialised; (ii) the gain benefit of u is the maximum; (iii) the total view maintenance time is bounded in the maintenance time.

4.3.1 Discussions

The two-phase greedy takes $O(m^2+mn^{3/2})$ time, usually substantially better than the inverted-tree greedy and A*-heuristic, where m and n are the number of views and queries in the bipartite graph G_B. However, it gives neither quantitative analysis of quality of the solution nor experiment results. As our running example shows, in Table 1, the maintenance-cost constraint is the minimum cost constraint that allows all vertices to be selected as materialised view. However, two-phase greedy delivers an approximate solution instead of a full set of materialised views, it is because during the minimum weighted maximum cardinality matching, it cannot fully select all the views. In addition, during the minimum weighted maximum cardinality matching, the view cannot match to itself. For example, in Figure 5, there is no edge from v_0 (a view) to v_0 (a query). It is because, if they do so, then the minimum weighted maximum cardinality matching solver will always select a view to answer itself, and therefore the resulting M_1 is equal to $V(G)$.

4.4 Integrated Greedy

The integrated greedy is summarised in Algorithm 4. The basic idea of the algorithm is given below. When no views are selected, the total query processing cost for all the queries is very large. Then the algorithm will reduce the query processing cost by materialising views, one-by-one, as long as the total maintenance cost is bounded within the cost constraint.

Let M be the set of materialised views having been selected, and $U(M)$ be the total maintenance cost for the views in M. Recall that $\tau(G, M)$ is the total query processing cost for answering all the queries. When considering a view $v \in V(G) - M$ to be materialised, the net increase in the maintenance cost is ΔT_v, = $U(M \cup \{v\}) - U(M)$ and the amount of query processing cost reduction is $\tau(G, \mathrm{M}) - \tau(G,M \cup\{v\})$ by spending ΔT_v unit costs. Thus, each time, it chooses a view $v \notin M$ to materialise such that the gain benefit $g(v)$ brought by v is the maximum. The function $g(v)$ is defined as follows.

$$g(v) = \frac{\tau(G,M) - \tau(G,M \cup \{v\})}{\Delta T_v} \tag{3}$$

The gain benefit is similar to that used in the inverted-tree greedy in (Gupta and Mumick (1999b).

In brief, in the integrated greedy (Algorithm 4), it first selects a vertex v_0 that gives the maximum benefit when there is no view being selected. That vertex is the first vertex in the set of views, M. Next, in each iteration, it selects a vertex that will give the maximum benefit in the current iteration. Selection of a vertex in an iteration is independent from other selections. The integrated greedy can reach an optimal solution in Table 1 and 3. However, it only reaches a near-optimal solution in Table 2. The reason is that it has to give up the greatest gain benefit vertex, v_1, at one stage, i, because the total maintenance cost exceeds the given cost constraint. But, in the later selections $j > i$, v_1 will never be able to be selected again.

Algorithm 4: A Integrated Greedy Heuristics (Laing *et al*, 2001)

Input: A graph $G(V, E)$ and a maintenance-cost constraint S.
Output: a set of materialised views.

```
1. begin
2. M ←φ;
3. Let v0 be the first vertex with maximum g(v0);
4. M ←{v0};
5. τ(G,M ← S – U (M)
6. while ΔS > 0 do
7.    gain ← 0;
8.    for each v ∈ V(G) – M do
9.       g(v) = (τ(G,M) – τ(G,M ∪ {v}))/ ΔTv;
10.      if g(v) > gain then
11.         gain = g(v); vo = v;
12.      end if
13.   end for
14.   if (ΔS – ΔTvo) > 0 then
15.      ΔS = ΔS – ΔTvo;
16.      M ←M ∪{v0};
17.   end if
18. end while
19. return M;
20. end
```

4.4.1 Discussions

The integrated greedy is very similar to inverted-tree greedy. We reexamine the integrated greedy in comparison with the inverted-tree greedy by considering the following two issues: (a) the inverted-tree greedy needs to consider every inverted tree sets, and (b) the inverted-tree greedy requires the query-benefit per unit effective-maintenance-cost for the newly selected views to be greater than the previously selected view set. The item (a) makes the time complexity of the inverted-tree greedy to be exponential in the number of vertices, in the worst case. As for the item (b), because the query-benefit per unit effective-maintenance-costs tends to decrease when more vertices are added into the view set, the inverted-tree greedy is difficult to select more vertices. Instead, the integrated greedy uses the query-benefit per unit maintenance-costs. It attempts to add a vertex that will give the maximum gain into the view set. So it is weaker than the above item (b). Besides, the integrated greedy selects views one-by-one, which will significantly reduce the view-selection time.

5. A PERFORMANCE STUDY

We present some results of our extensive performance study in this section. All the algorithms were implemented using `gcc`. We use the maximum-weight matching solver implemented by Ed Rotherg who implemented H. Gabow's N-cube weighted matching algorithm (Gabon, 1973). We use it to find the minimum weighted matching by replacing a cost, c, on an edge with $c_{max} - c$, where c_{max} is a maximum value for all costs. All the algorithms used the same function, $q(v, M)$, to compute query processing cost and the same function, $m(v, M)$, to compute maintenance cost.

For a small dependence lattice (up to 16 elements), we compare five different algorithms: the optimal, the inverted-tree greedy, the A*-heuristic, the two-phase greedy, and the integrated greedy. We also report the scalability of the two algorithms, the two-phase greedy and the integrated greedy, using a large dependence lattice (up to 256 elements). These experiments were done on a Sun Blade/1000 workstation with a 750MHz UltraSPARC-III CPU running Solaris 2.8. The workstation has a total physical memory of 512M. The notations and definitions, together with the default values, for all the parameters are summarized in Table 4.

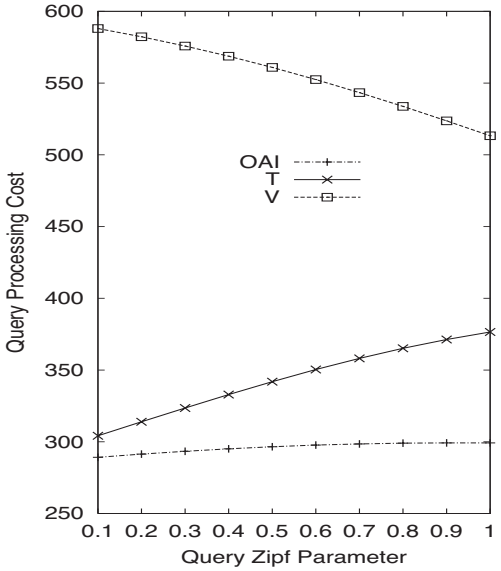
Notation	Definition (Default Values)
N	the number of vertices (16)
T	the cost constraint
θ_q	Zipf distribution factor for query frequency (0.2)
θ_u	Zipf distribution factor for update frequency (0.0)
R_v	table sizes for a vertex v
$Q_{(v,u)}$	query processing cost for a vertex u using v
$U_{(v,u)}$	maintenance cost for a vertex u using v

Table 4: System parameters

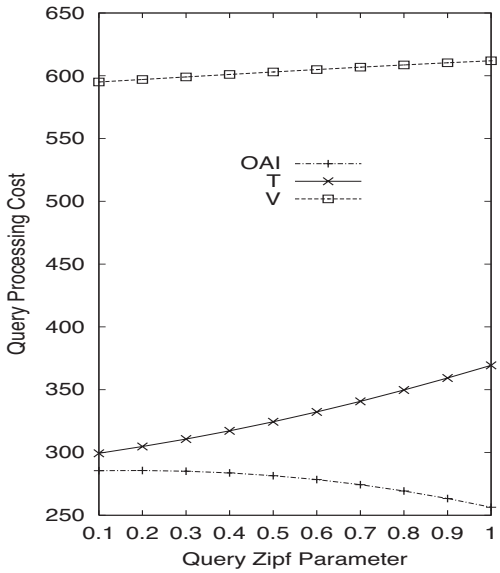
Given a dependent lattice $(L, \prec)$ of size N, we construct a directed acyclic graph $G(V, E)$. A vertex, v, has three weights, R_v, its update frequency and query frequency. An edge, from v to u, has two weights: $Q_{(u,v)}$ and $U_{(u,v)}$. We assign these weights to the graph $G(V, E)$ as follows. First, we randomly generate N distinctive table sizes (R_v). The N table sizes are randomly picked up and assigned to the vertices on a condition that the table sizes of ancestors of a vertex are greater than that of the vertex. We assume that query frequencies follow a Zipf distribution.We also assume that all vertices have the same relative update frequencies such that all materialised views need to be updated when the raw table is updated. Query frequencies are randomly assigned to all vertices. Given an edge from v to u, (v, u), we assume that the maintenance cost of u using v is smaller than the query processing cost of u using v. We also assume that maintenance cost is more related to the table size of u. In this set of tests we reported in this paper, $Q_{(v, u)}$ is a number smaller than the table size of v, (R_v). $U_{(v, u)}$ is about one 10-th of the table size u, (R_u). It is important to know that the cost function $q(v,u)$ $(m(v, u))$ considers both table sizes and query processing costs (maintenance costs), associated with edges.

In order to compare the performance of the inverted-tree greedy, A*-heuristic, two-phase greedy and integrated greedy, we implemented an algorithm for finding the optimal solution. To find the optimal set of materialised views to precompute, we enumerate all possible combinations of views, and find a set of views by which the query processing cost is minimised. Its complexity is $O(2^N)$.

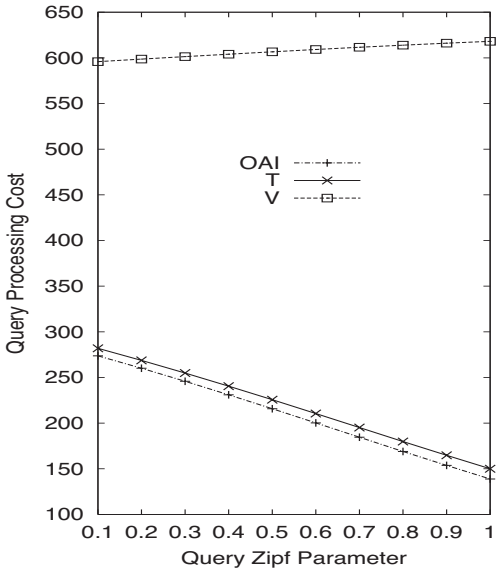
We abbreviate the optimal algorithm as `O`, the inverted-tree greedy as `V`, A*-heuristic as `A`, two-phase greedy as `T` and integrated greedy as `I` in the following figures.



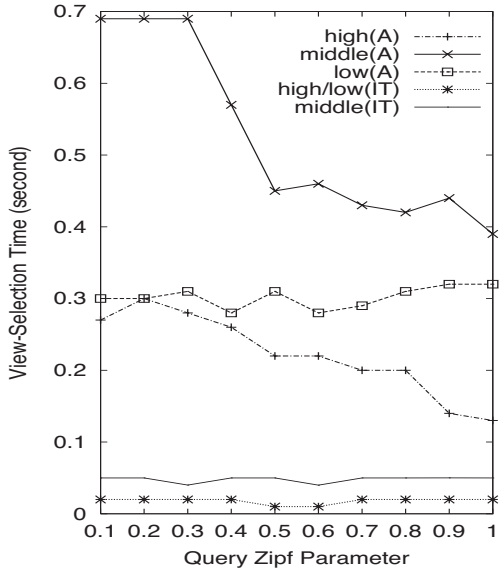
(a) Query frequency changes (high level)



(b) Query frequency changes (middle level)



(c) Query frequency changes (low level)



(d) View selection time v.s. query frequency changes (high/low/middle(0) is over 200 high/low/middle(v) is over 460)

Figure 6: The impacts of query frequencies

Exp-1: The impacts of query frequencies

First, we investigate the impacts of query frequencies. Query frequencies follow a Zipf distribution. In Figure 6, the number of vertices is 16, and the maintenance-cost constraint is $0.8 \times C_{min}$, where C_{min} is the minimum maintenance-cost constraint for all vertices to be selected as views. We vary query frequencies by increasing the Zipf factor from 0.1 to 1.0. We assign the high query frequencies to the vertices in three ways: (*i*) high level (close to the top), (*ii*) middle level, and (*iii*) low level, which are shown in Figure 6(a), (b) and (c), respectively. The assignment of high query frequencies in the graph will affect the set of views to be materialised. In this testing, the A*-heuristic and integrated greedy reach an optimal solution in all cases.

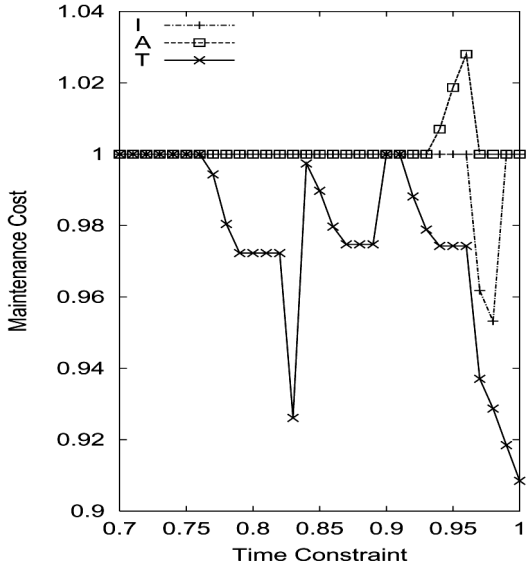
- The high query frequencies are assigned to the vertices at the high level (close to top) (Figure 6(a)): The increase of query processing cost for the four algorithms is due to the fact that the high level vertices have large query cost. In contrast, the query processing cost for the inverted-tree greedy decreases. It is because that it always attempts to select high level vertices, and they are frequently retrieved.
- The high query frequencies are assigned to the vertices at the low level (Figure 6(c)): The query processing cost for all algorithms is opposite to Figure 6(a).
- The high query frequencies are assigned to the vertices at the middle level (Figure 6(b)): The decrease of query processing cost for the three algorithms is due to the fact that the middle level vertices have lower query cost and high query frequencies. The query processing cost for the two-phase greedy and inverted-tree greedy increases, because the query frequencies of the high level vertices increase.

Figure 6(d) illustrates the relationship between view selection time and Zipf factor. We found that the five algorithms spend longer time while the high query frequencies are assigned to the vertices at the middle level. All algorithms spend the same amount of time on both high and low cases except for the A*-heuristic. It is because that, at each search stage, A*-heuristic needs to calculate the $g(x)$ and $h(x)$ of downward branches. When the high query frequencies are assigned at the top level, it is faster for A*-heuristic to reach the leaf vertices as the difference of query cost between vertices is large. However, at the middle level, the difference of query cost at each vertex becomes small, the A*-heuristic needs longer time to search other branches. Compared with the top case, A*-heuristic spends more time in the low level case. The other algorithms, the integrated greedy and two-step greedy take nearly constant time.

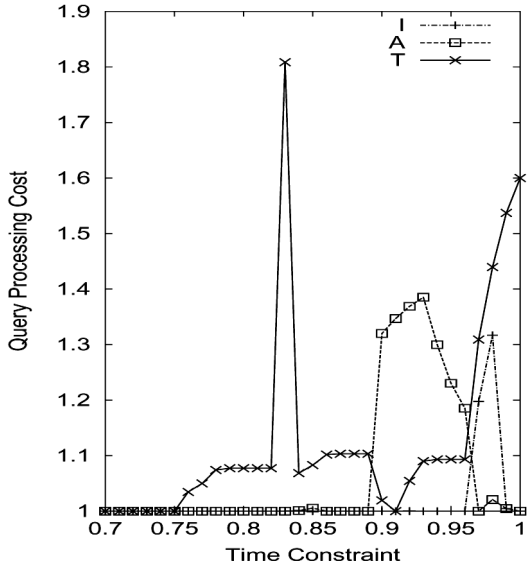
Exp-2: The impact of the maintenance-cost constraint

In this testing, we investigate the performance of the four algorithms by varying the maintenance-cost constraint. The number of vertices is 16 and the Zipf factor is 0.2. The high query frequencies are assigned at the high level. The minimum maintenance-cost constraint, denoted C_{min}, allows all vertices to be selected as materialised views.

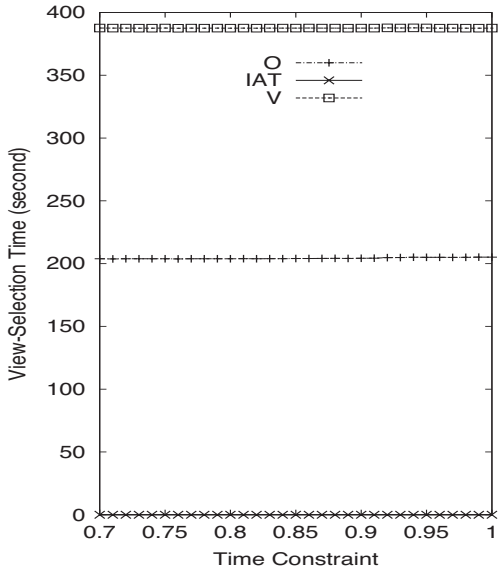
The results are shown in Figure 7(a), (b) and (c), where the maintenance-cost constraint used is $p \times C_{min}$ where p varies from 0.7 to 1.0. (Note: when $p < 0.7$, none of the algorithms can select any views.) A larger p implies that it is likely to select more views. When $p = 1$, it means that all vertices are possibly selected. In Figure 7(a) and (b), the optimal is chosen as the denominator to compare. We did not include the inverted-tree greedy (▽) in these figures, because that makes the other differences less legible. For reference, the maintenance costs for the inverted-tree greedy are, as pairs of (p, maintenance-cost), (0.70, 1), (0.80, 0.88), (0.90, 0.77) and (1.0, 0.69). The query processing costs for the inverted-tree greedy are, as pairs of (p, query-processing-cost), (0.70, 1),



(a) Maintenance cost v.s. cost constraint (0.7–1)



(b) Query processing cost v.s. cost constraint (0.7–1)



(c) View-selection time v.s. cost constraint

Figure 7: The impacts of the maintenance-cost constraint

(0.80, 1.99), (0.90, 6.42), (1.0, 11.25). The inverted-tree greedy is inferior to all others. The query processing cost is the reciprocal of the maintenance-cost (Figure 7(a) v.s. Figure 7(b)).

We found, in our performance study, that the maintenance-cost constraint is the most critical factor that affects the quality of the heuristic algorithms. Some observations are given below.

- **Issue 1:** As Figure 7(a) and (b) suggested, in a multidimensional data warehouse environment, Issue 1 has less impacts on greedy algorithms. We explain it below. When selecting a view v, the total maintenance cost, $U(M \cup \{v\})$, depends on two factors: update cost, $m(v, M)$, and update frequency of the vertex v, g_v. Recall $m(u, v)$ is the sum of the maintenance-costs associated with the edges on the shortest path from v to u, so the total maintenance cost will be greater than zero, when a new vertex is added. On the other hand, since u is derived from v, the update frequency of v should be greater than or equal to u's update frequency in a multidimensional data warehouse environment. As a result, $\Delta T_v > 0$. Therefore, Issue 1 is not a real issue in a multidimensional environment.
- **Issue 2:** The two-step greedy and the inverted-tree greedy cannot select all views to materialise, even though the maintenance-cost constraint allows it. Two-step greedy only gives an approximate solution.
- **Issue 3:** The greedy algorithms become unstable when the maintenance-cost constraint is over $0.9 \times C_{min}$. The integrated greedy is impossible to select proper views. The reasons are given in Section 3.
- **Issue 4:** The A*-heuristic cannot always find optimal solutions, in particular, when it is over $0.9 \times C_{min}$. For instance, when $p = 0.95$, A*-heuristic selected $\{v_0, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{12}\}$ of a 16 vertex graph. Its maintenance-cost is 27.25, and its query processing cost is 93.18. But, the optimal solution included $\{v_0, v_1, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}\}$. The maintenance-cost and query processing cost for the optimal solution are, 26.75 and 75.54, respectively. It is because A*-heuristic estimates the expected benefit, $h(x)$, which might not be accurate. It points out a very important fact for a greedy algorithm, if it misses selecting a vertex, (v_2 in this case), it will affect the other selections.

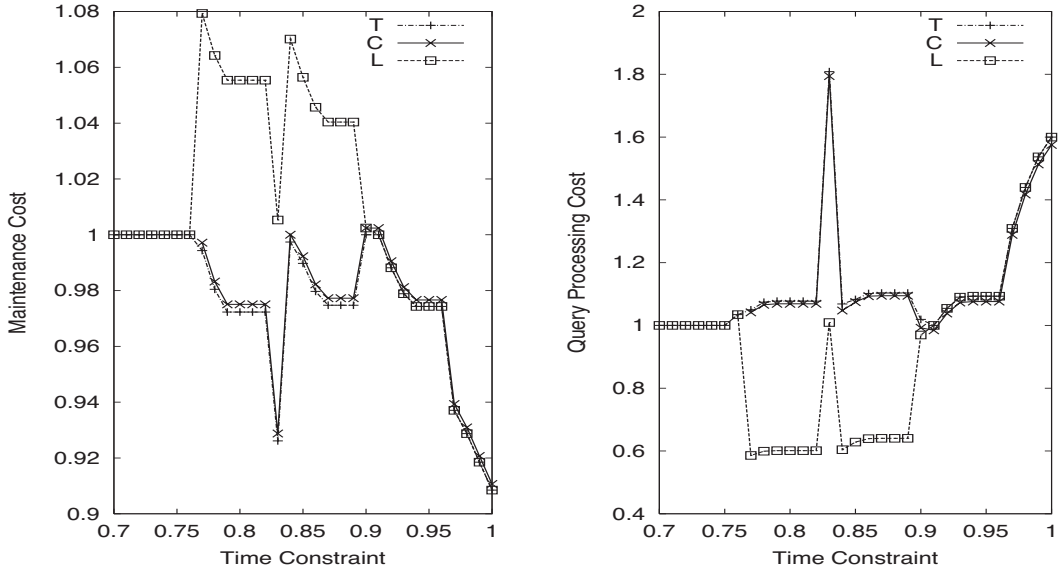
Figure 7(c) shows that the view selection time for the five algorithms. Because the number of vertices is fixed ($N = 16$), all the view selection time for all the cases are the same. The inverted-tree greedy consumes more view selection time than the optimal (the middle). It is because that, for computing the optimal solution, we only check all 2^{16} subsets once. Recall that the inverted-tree greedy needs to check the maximum query-benefit per unit effective-maintenance-costs at every stage, and check powersets repeatedly. All the other three algorithms: the A*-heuristic, the integrated greedy and the two-step greedy can be efficiently processed.

Exp-3: Two-Phase and its Variants

Figure 8(a) and (b) show the comparison amongst the two-phase greedy (`T`) and its variants, minimise-benefit-lost (`L`) and maximise-benefit (`C`). The minimise-benefit-lost variant reaches a better solution by *releasing* a certain amount of maintenance time. In other words, it does not support hard constraint and allows the maintenance time of the selected views to be greater than the given constraint. Because, in the paper, we mainly studied the maintenance-time view-selection under the hard constraint, we do not compare it with other greedy algorithms further. The maximise-benefit variant can only improve the two-phase greedy marginally. It is important to note that both variants cannot select a full set of materialised view even the maintenance cost allow them to do so.

Exp-4: Scalability

In this experimental study, we fixed all the parameters except for the number of vertices. We show two sets of results. Figures9(a), (b) and (c) show a comparison of the five algorithms: the optimal, the integrated greedy, the two-phase greedy, the inverted-tree greedy and the A*-heuristic by



(a) Maintenance cost v.s. cost constraint (two-phase and its variants)

(b) Query cost v.s. cost constraint (two-phase and its variants)

Figure 8: Two-phase and its variants

varying the number of vertices, N, from 4 to 16. The maintenance-cost constraint, C_{min}, is the minimum maintenance-cost constraint that allows all vertices to be selected. In Figure 9(d), (e) and (f), we compare the integrated greedy with the two-phase greedy by varying the number of vertices, N, from 4 to 256. The maintenance-cost constraint is $0.8 \times C_{min}$.

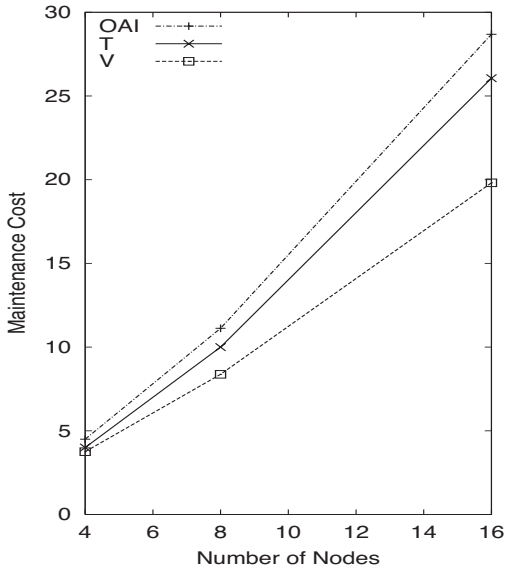
Figure9(b) shows the query processing costs. Due to the number of views to be selected, as shown in the previous testings, the A*-heuristic and integrated greedy always give an optimal solution. The two-phase greedy gives a feasible and good approximation. The A*-heuristic, integrated greedy and two-step greedy outperform the inverted-tree greedy significantly. Figure 9(c) shows the view-selection time of these algorithms. The optimal algorithm is exponential to the number of N. The inverted-tree greedy is also exponential to N, and takes longer time than the optimal. The A*-heuristic is exponential to N in the worst case. When the number of vertices is over 120, the view selection time for the integrated greedy increases exponentially. On the other hand, the view selection time for the two-phase greedy is small. In addition, the query processing time for the two-phase greedy is acceptable when the number of vertices is large. In conclusion, when $N \leq 120$, the integrated greedy is recommended to use. When $N > 120$, the two-step greedy is a reasonable choice in practice.

6. CONCLUSION

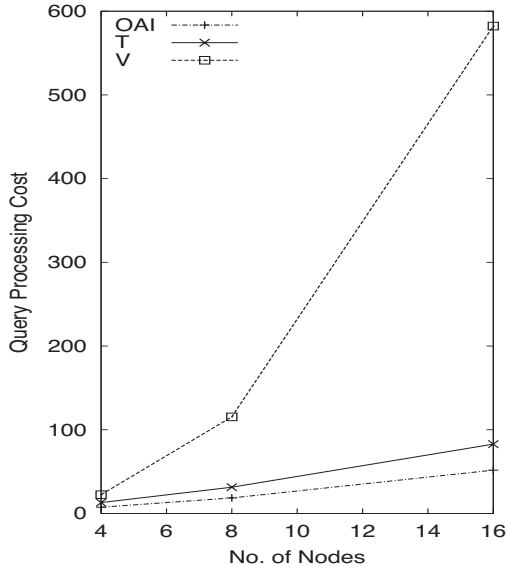
The selection of views to materialise is one of the most important issues in designing a data warehouse. We reexamine the maintenance-cost view-selection problem under a general cost model. Heuristic algorithms can provide optimal or near optimal solutions in a multidimensional data warehouse environment under certain conditions: the update cost and update frequency of any ancestor of a vertex is greater than or equal to the update cost and update frequency of the vertex, respectively.

In our extensive performance studies, the A*-heuristic, integrated greedy and two-step greedy significantly outperformed the inverted-tree greedy. The greedy algorithms are not stable when the maintenance-cost constraint is over 90% of the minimum maintenance-cost constraint that allows all views to be selected.

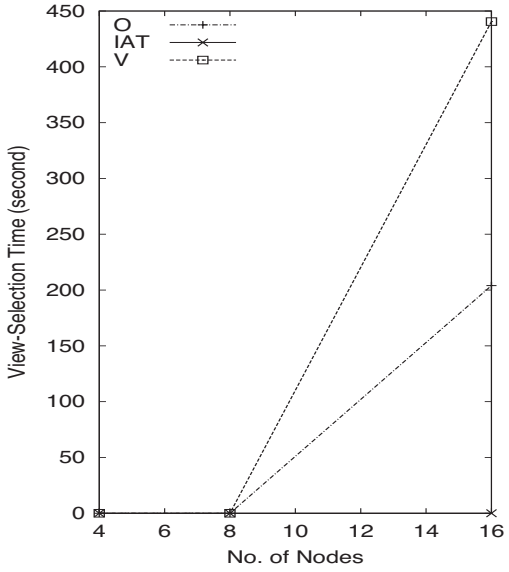
The two-phase greedy and the integrated greedy are scalable. When the number of vertices in a graph is less than or equal to 120, the integrated greedy can compute fast and give an optimal



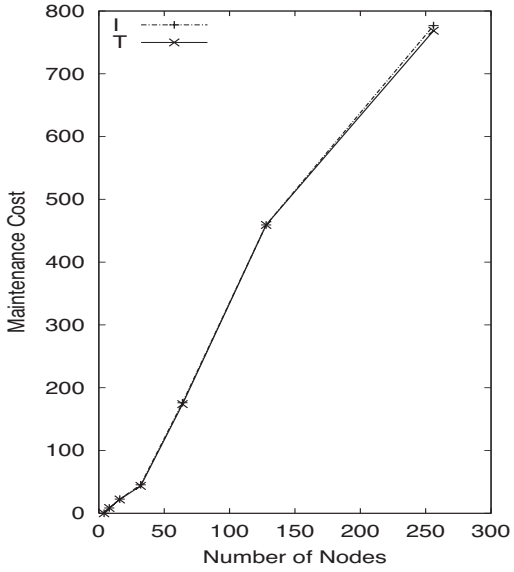
(a) Maintenance cost v.s. number of vertices



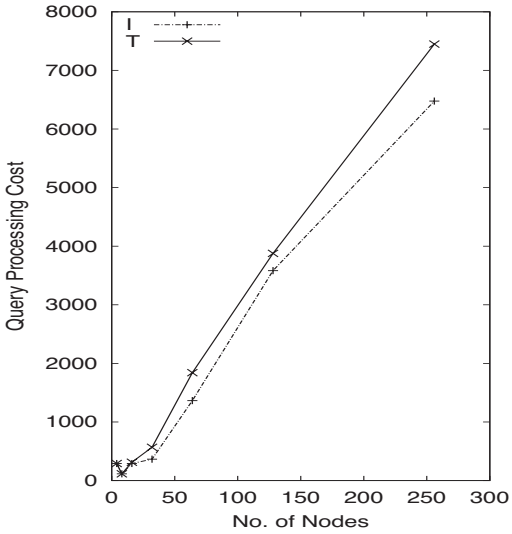
(b) Query processing cost v.s. number of vertices



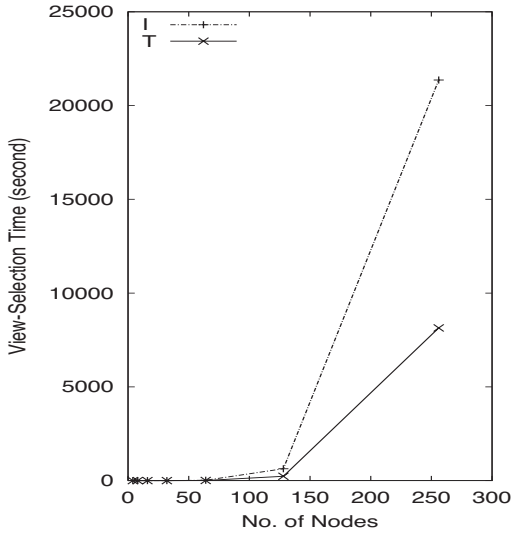
(c) View-selection time v.s. number of vertices



(d) Maintenance cost v.s. number of vertices



(e) Query processing cost v.s. number of vertices



(f) View-selection time v.s. number of vertices

Figure 9: Scalability

solution. When the number of vertices is greater than 120, the two-phase greedy is recommended to use due to the efficiency. The two-phase greedy gives a good approximate solution, which is close to the optimal solution, in our testing for a small number of vertices (16).

As our future work, we plan to study the view-selection issues using real large databases.

ACKNOWLEDGEMENT

This work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region. (Project Nos. CUHK4198/00E and HKUST6184/02E).

REFERENCES

BARALIS, E., PARABOSCHI, S and TENIENTE, E. (1997): Materialized views selection in a multidimensional database. In *Proceedings of 23rd International Conference on Very Large Data Bases*, 156–165.

CHOI, C.H., YU, J.X. and GOU, G. (2002): What difference heuristics make: Maintenance-cost view-selection revisted. In *Proc. of WAIM '02*.

GABOW, H. (1973): *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs*. PhD thesis, Stanford University.

GUPTA, H., HARINARAYAN, V., RAJARAMAN, A. and ULLMAN, J.D. (1997): Index selection for OLAP. In *Proceedings of the Thirteenth International Conference on Data Engineering*, 208–219.

GUPTA, A. and MUMICK, I.S. (1999a): *Materialized Views: Techniques, Implementations and Applications*. The MIT Press.

GUPTA, H. and MUMICK, I.S. (1999b): Selection of views to materialize under a maintenance cost constraint. In *Proceedings of the 7th International Conference on Database Theory*, 453–470.

GUPTA, H. (1997): Selection of views to materialize in a data warehouse. In *Proceedings of the 6th International Conference on Database Theory*, 98–112.

HARINARAYAN, V., RAJARAMAN, A. and ULLMAN, J.D. (1996): Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 205–216.

KIMBALL, R. (1996): *The Data Warehouse Toolkit*. John Wiley & Sons.

LIANG, W., WANG, H. and ORLOWSKA, M.E. (2001): Materialized view selection under the maintenance time constraint. In *Proc. of DKE01*.

MUMICK, I.S., QUASS, D. and MUMICK, B.S. (1997): Maintenance of data cubes and summary tables in a warehouse. In *Proc. of SIGMOD '97*.

SHUKLA, A., DESHPANDE, P. and NAUGHTON, J.F. (1998): Materialized view selection for multidimensional datasets. In *Proceedings of 24th International Conference on Very Large Data Bases*, 488–499.

BIOGRAPHICAL NOTES


Jeffrey Xu Yu

Jeffrey Xu Yu received his BE, ME and PhD in computer science from the University of Tsukuba, Japan, in 1985, 1987 and 1990, respectively. He was a research fellow and an Assistant Professor in the Institute of Information Sciences and Electronics, University of Tsukuba, a Lecturer in the Department of Computer Science, Australian National University. Currently he is an Associate Professor in the Department of Systems Engineering and Engineering Management, at the Chinese University of Hong Kong. His major research interests include wireless information retrieval, data warehouse, on-line analytical processing, query processing and optimisation, and design and implementation of database management systems. He is a member of ACM, and a society affiliate of IEEE Computer Society.


Chi-Hon Choi

Chi-Hon Choi is currently a Master of Philosophy student in Systems Engineering and Engineering Management at the Chinese University of Hong Kong where she also received her BEng degree. Her current research interests include design and analysis of data warehousing and online analytical processing, design and implementation of database management systems, query processing and query optimisation.


Gang Gou

Gang Gou received his BSc degree from the Department of Computer Science and Technology at NanKai University, China, in 2000. He is currently studying in the Department of Systems Engineering and Engineering Management at the Chinese University of Hong Kong, as a second-year Master of Philosophy student. His recent research focuses on data warehouse, OLAP queries, and materialised view selection. He also has interests in approximate query processing, data streams processing and data mining.


Hongjun Lu

Hongjun Lu received his BSc from Tsinghua University, China, and MSc and PhD from the Department of Computer Science, University of Wisconsin-Madison. Before joining Hong Kong University of Science and Technology, he worked as an engineer in the Chinese Academy of Space Technology, and as a principal research scientist in the Computer Science Centre of Honeywell Inc., Minnesota, USA, and as a professor at the School of Computing of the National University of Singapore. His research interests are in data/knowledge base management systems with emphasis on query processing and optimisation, physical database design, and database performance. His recent research work includes data quality, data warehousing and data mining, and management of XML data. He is also interested in development of Internet-based database applications and electronic business systems.

Hongjun Lu is currently a trustee of the VLDB Endowment, an associate editor of IEEE Transactions on Knowledge and Data Engineering (TKDE) and a member of the review board of Journal of Database Management. He is the Chair of the steering committee of the International Conference on Web-Age Information Management (WAIM), and the Chair of the steering committee of Pacific-Asia Conference of Knowledge Discovery and Data Mining (PAKDD). He served as a member of the ACM SIGMOD Advisory Board in 1998–2002.