Physical Database Design for Data Warehouses [*]

Wilburt Juan Labio,Dallan Quass, Brad Adelberg
Department of Computer Science
Stanford University
e-mail: {wilburt,quass,adelberg}@cs.stanford.edu

Abstract

Data warehouses collect copies of information from remote sources into a single database. Since the remote data is cached at the warehouse, it appears as local relations to the users of the warehouse. To improve query response time, the warehouse administrator will often materialize views defined on the local relations to support common or complicated queries. Unfortunately, the requirement to keep the views consistent with the local relations creates additional overhead when the remote sources change. The warehouse is often kept only loosely consistent with the sources: it is periodically refreshed with changes sent from the source. When this happens, the warehouse is taken off-line until the local relations and materialized views can be updated. Clearly, the users would prefer as little down time as possible. Often the down time can be reduced by adding *carefully selected materialized views or indexes to the physical schema. This paper studies how to select the sets of supporting views and of indexes to materialize to minimize the down time. We call this the view index selection (VIS) problem. We present an A* search based solution to the problem as well as rules of thumb. We also perform additional experiments to understand the space-time tradeoff as it applies to data warehouses.*

1. Introduction

Data warehouses collect information from many sources into a single database. This allows users to pose queries within a single environment and without concern for schema integration. Figure 1 shows a typical warehousing system. Relations R_{src}, S_{src}, and T_{src}, referred to as *source relations*, from sources 1, 2, and 3 respectively, are replicated at the warehouse as R, S, and T in order to answer user queries posed at the warehouse such as $R \bowtie S \bowtie T$. We refer to the relations R, S, and T as *warehouse relations*. Changes to the source relations are queued and periodically shipped and applied to the warehouse relations. We call these changes *deltas*.

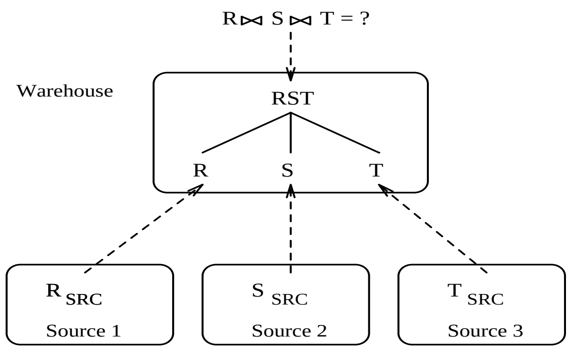

Figure 1. Warehouse with primary view.

Queries posed at a data warehouse are often complex—involving joins of multiple relations as well as aggregation. Due to the complexity of these queries, *views* are usually defined — a view is a derived relation expressed in terms of the warehouse relations. Because the views are defined in terms of the warehouse relations, we refer to the warehouse relations also as *base relations*. For example, referring again to Figure 1, RST represents a view that is the expression $R \bowtie S \bowtie T$. Warehouses can store large amounts of data, and so in order to improve the performance of queries written in terms of the views, the views are often *materialized* by storing the result of the view at the warehouse. Unmaterialized views are called *virtual* views. Queries written in terms of materialized views are usually significantly faster than queries written in terms of virtual views because the view tuples are stored rather than having to be computed.

[*]This work was supported by Hewlett Packard, by Philips, and by Rome Laboratories under Air Force Contract F30602-94-C-0237. This work was also supported by an equipment grant from Digital Equipment Corporation.

Since materialized views are computed once and stored, they become inconsistent as the deltas from the sources are applied to the base relations. In order to make a materialized view consistent again with the base relations from which it is derived, the view may be recomputed from scratch, or *incrementally maintained* [6] by calculating just the effects of the deltas on the view. These effects are captured in *view maintenance expressions* [5]. For example, if view RST in Figure 1 is materialized, the maintenance expression calculating the tuples to insert into RST due to insertions into R is $\triangle R \bowtie S \bowtie T$, where $\triangle R$ denotes the insertions into R.

Since the sizes of the views at a warehouse are usually so large and the changes small in comparison, it is often much cheaper to incrementally maintain the view than to recompute it from scratch. Incrementally maintaining a number of materialized views at a warehouse, even though cheaper than recomputing the views from scratch, may still involve a significant processing effort. To avoid impacting clients querying the warehouse views, view maintenance is usually performed at night during which time the warehouse is made unavailable for answering queries. A major concern for warehouses using this approach is that the views be maintained in time to be available for querying again the next morning. An important problem for data warehousing is thus: Given a set of materialized views that need to be maintained due to a set of deltas shipped from the data sources, how is it possible to reduce the total maintenance time?

Our approach to the problem of minimizing the time spent maintaining a set of views may seem counter-intuitive at first: add additional views and/or indexes. In this paper we will approximate maintenance time by the number of I/O's required and then endeavor to minimize the number of I/O's. We start with the number of I/O's required for maintaining the materialized views and the base relations at the warehouse. We then add a set of additional views and indexes that themselves must be maintained, but whose *benefit* (reduction in I/O's) outweighs the *cost* (increase of I/O's) of maintaining them.

As an example, let us return to Figure 1. Suppose that in addition to materializing the *primary view*, RST, another view, ST, is also materialized. By materializing view ST, the total cost of maintaining both RST and ST can be less than the cost of maintaining RST alone. For example, suppose that there are insertions to R but no changes whatsoever to S and T. To propagate the insertions to R onto RST, we must evaluate the maintenance expression that calculates the tuples to insert into RST due to insertions into R, which is $\triangle R \bowtie S \bowtie T$. With ST materialized, it is almost certain that this expression can be evaluated more efficiently as $\triangle R \bowtie ST$, joining the insertions to R with ST, instead of with S and T individually. Even if there are changes to S and T, the benefit of materializing ST may still outweigh the extra cost involved in maintaining it. Since the view ST is materialized to assist in the maintenance of the primary view RST, we call the view ST a *supporting view*.

In addition to materializing supporting views, it may also be beneficial to materialize indexes. Indexes may be built on the base relations, primary views, and on the supporting views. The general problem, then, is to choose a set of supporting views and a set of indexes to materialize such that the total maintenance cost for the warehouse is minimized. We call this the *View Index Selection (VIS)* problem and it is the focus of this paper.

Below we list the primary contributions of this paper.

- We propose and implement an optimal algorithm based on A* that prunes as much as 99% of the possible supporting view and index sets to solve the VIS problem.

- We develop rules of thumb that can help a warehouse administrator (WHA) find a reasonable set of supporting views and indexes to materialize in order to reduce the total maintenance cost.

- We compare the benefit of materializing supporting views as opposed to indexes, and discuss which should be chosen when the total storage space at the warehouse is constrained.

- We perform experiments to determine how sensitive the choice of supporting view and index sets are to the input parameters of the optimizer.

The rest of the paper proceeds as follows. Section 2 describes the VIS problem in detail. Section 3 presents the scope of our results and our approach to view maintenance. We describe our A*-based algorithm in Section 4. Section 5 presents rules of thumb for choosing a set of supporting views and indexes to materialize. In Section 6, we report on additional experiments such as comparing the importance of indexes and supporting views when space is constrained. Next, in Section 7, we discuss how this paper relates to previous work in the area. Finally, we present our conclusions in Section 8.

2. General Problem

Having introduced the VIS problem, in this section we describe it fully and present an exhaustive search algorithm to solve it. We also show the worst case complexity of the VIS problem. Lastly, we present an example schema to illustrate the concepts introduced.

2.1. The Optimization Problem

An optimal algorithm must minimize the total cost of maintaining the warehouse. The total cost that we attempt to minimize is the sum of the costs of: (1) applying the deltas to the base relations, (2) evaluating the maintenance expressions for the materialized views, and (3) modifying affected indexes. The cost of maintaining one view differs depending upon what other views are available. It is therefore incorrect to calculate the cost of maintaining the original view and each of the additional views in isolation. Moreover, in order to derive the total cost it is necessary to consider the view selection and index selection together.

To find the optimal solution, then, we must solve the optimization problem globally. One approach, proposed in Ross et al. [13] (although this work does not consider indices), is to exhaustively search the solution space. Although exhaustive search is impractical for large problems, it illustrates the complexity of the problem and provides a basis of comparison for other solutions. The exhaustive algorithm works as follows (each stage is described below):

```
for each subset of supporting views
  for each subset of indexes
    compute total cost and keep track of
    the supporting views and indexes that
    obtain the minimum cost
```

2.1.1 Choosing the views

In the first step we consider all possible subsets of the set of candidate views $\mathcal{C}$. As proposed in [13], we consider as candidate views all distinct nodes that appear in a query plan for the primary view. Since the primary view is already materialized, it is not included in the candidate view set. For example, given a view $V = R \bowtie S \bowtie T$, $\mathcal{C} = \{RS, RT, ST\}$. In general, for a view joining n relations there are roughly $\mathcal{O}(2^n)$ different nodes that appear in some query plan for the view, one joining each possible subset of the base relations. Thus, to consider all possible subsets of $\mathcal{C}$, we need to evaluate roughly $\mathcal{O}(2^{2^n})$ different view states.

2.1.2 Choosing the indexes

Now we must consider all possible subsets of the set of candidate indexes, $\mathcal{I}$. Candidate indexes, as defined in Finkelstein et. al. [3], are indexes on the following types of attributes:

- attributes with selection or join predicates on them.
- key attributes for base relations where changes to the base relation include deletions or updates.
- attributes in GROUP BY or ORDER BY clauses.

Additional attributes can be candidates depending on the query optimizer being used.

Since each materialized view will usually have candidate indexes, $\mathcal{I}$ must be recomputed at the beginning of every inner loop. The cardinality of $\mathcal{I}$ for a particular view state is proportional to the number of materialized views and base relations in that state. Further, a particular state contains between n and $\mathcal{O}(2^n)$ materialized views and base relations, so there can be as many as $\mathcal{O}(2^n)$ candidate indexes to consider. Since we must evaluate possible subsets of candidate indexes, the number of possible index states for a view state can be up to $\mathcal{O}(2^{2^n})$. (See Section 7 for an explanation of why standard approaches for index selection are not appropriate.)

2.1.3 Computing the total update cost

Once a particular view and index state are chosen, obtaining the total cost is a query optimization problem in itself since it involves finding the most efficient query plan for each of the view maintenance expressions. Thus, the VIS problem for a single primary view joining n base relations contains roughly $\mathcal{O}(2^{2^n})$ query optimization problems in the most general case.

The query optimization itself is complicated by the presence of materialized views since the optimizer must also determine if it can use another materialized view in the query plan evaluating a maintenance expression.

This problem is known as "answering queries using views" [10].

To complicate matters, one batch of changes can generate multiple maintenance expressions that need to be evaluated. This happens due to different types of changes to the base relations. The maintenance expressions must be optimized as a group because of possible common subexpressions [13]. This problem is known as the "multiple-query optimization" problem [19].

2.2 Example

Consider the following base relations and view.

```
R(R0,R1),  S(S0,S1),  T(T0,T1)

create view V(R0,R1,S0,S1,T0,T1) as
select *
from   R, S, T
where  R.R1 = S.S1 and S.S0 = T.T0
       and T.T1 <= 10
```

Figure 2 shows an *expression dag* [13] that includes all the nodes that could appear in a query plan for V, assuming the selection on $T.T1$ is pushed down. The view T' is the result of applying the selection condition to T. Under each view is the set of operations that could be used to derive the view. For example, the view RST could be derived as the result of $R \bowtie S$ joined with T', or the result of $R \bowtie S$ joined with the result of $S \bowtie T'$, and so on. Each of the intermediate results could be materialized as a supporting view. Following the definition in Section 2.1.1, the set of candidate supporting views, $\mathcal{C}$, is $\{RS, ST', RT', T'\}$. Assuming V is materialized at a data warehouse (as well as the base relations), any possible subset of $\mathcal{C}$ might also be materialized as supporting views at the warehouse in order to minimize the total maintenance cost. In addition, indexes on V, the base relations, and the supporting views need to be considered.

It is useful to think of the expression dag in Figure 2 when considering the different *update paths* [13] changes to base relations can take as they are propagated to the view. An update path corresponds to a specific query plan for evaluating a view maintenance expression. For example, the maintenance expression for propagating insertions to R onto V is to insert the result of $\triangle R \bowtie S \bowtie T'$ into V. The graph depicts seven possible update paths for this expression, two of which are shown in Figure 2:

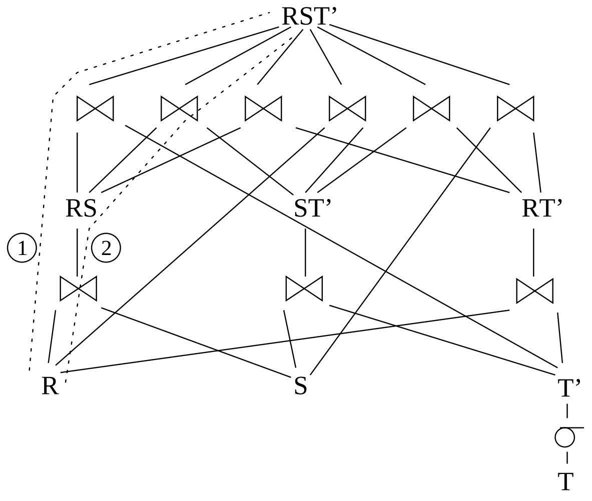

Figure 2. Example Schema.

(1) $(\triangle R \bowtie S) \bowtie T'$, (2) $(\triangle R \bowtie S) \bowtie (S \bowtie T')$. Notice that the choice of update path can affect which indexes get materialized. If update path (1) is chosen, an index may be built on the join attribute of T' to help compute the maintenance expression. If path (2) is chosen however and view ST' is materialized, an index may be built on the join attribute of ST'. The implication of using different indexes depending upon which update path is chosen is discussed in Section 7.

Changes to base relations need to be propagated both to the primary view as well as to the supporting views that have been materialized. When propagating changes to several base relations onto several materialized views there are opportunities for multiple-query optimization. Results of maintenance expressions for one view can be reused when evaluating maintenance expressions for another view. For example, suppose view $RS = R \bowtie S$ is materialized. The result of propagating insertions to R onto RS, $\triangle R \bowtie S$, can be reused when propagating insertions to R onto V, $\triangle R \bowtie S \bowtie T'$, so that only the join with T' need be performed. In addition, common subexpressions can be detected between several maintenance expressions.

3 Problem Studied

As discussed in the previous section, the VIS problem is very complex. While the algorithm we present is quite general, we have made simplifying assumptions to allow us to study the most important effects without getting lost in irrelevant detail. The resulting problem is still doubly-exponential and we feel that the insights we have gained from this study can lead to more general solutions. Furthermore, our assumptions are very similar to those made previously in the literature.

3.1 Database Model

We limit our consideration to maintaining a single select-join (SJ) view. Any combination of selections and joins in a view definition can be represented in this form. We assume that the view does not involve self-joins and that all base relations have keys (to simplify the cost model). In addition, we assume that all indexes are stored as B+-trees, that indexes are built on single attributes only, and indexes are built on relations and views stored as heaps. We consider the two most common physical join operators: nested-block joins and index joins.

We assume that the base relations from the source are replicated at the warehouse. In addition, we assume that selection conditions are always "pushed down" onto the base relations. When considering what additional data structures to materialize, we restrict ourselves to data structures that are themselves easily maintainable through SQL update statements. To this end we consider materializing supporting views and/or indexes. The indexes are on attributes in the base relations, primary view, and supporting views that are keys or involved in selection/join conditions.

A materialized supporting view V could thus be the result of applying selection conditions to a base relation, or joining two or more base relations, each having selection conditions pushed down.

3.2 Change Propagation Model

We consider three types of deltas: insertions, deletions, and updates. We distinguish between two types of updates: Updates that alter the values of key attributes or attributes involved in selection/join conditions are called *exposed updates*; all other updates are called *protected updates*. Exposed updates can result in tuples being deleted from or inserted into the view. For this reason, we propagate exposed updates as deletions followed by insertions. Henceforth, all references to 'updates' should be interpreted to mean 'protected update'. Protected updates can be applied directly to the view since they only change attribute values of tuples in the view, and never insert or remove tuples from the view.

We assume for the purposes of determining the cost of maintaining a view that each type of change to each base relation is propagated to the view and relevant supporting views separately. Therefore, the cost of maintaining a view or supporting view V is the sum of the costs of propagating each type of change to each of the base relations involved in V. For example:

- **Insertions** The cost of propagating insertions to R onto $V = R \bowtie S \bowtie T$ is the cost of evaluating $\triangle R \bowtie S \bowtie T$, inserting the result into V, and updating the indexes of V. When propagating insertions we consider reusing the results of evaluating insertions for one view in evaluating insertions for another. For instance, the result of $\triangle R \bowtie S \bowtie T$ can be used in evaluating $\triangle R \bowtie S \bowtie T \bowtie U$.

- **Deletions** The cost of propagating deletions to R ($\triangledown R$) onto V is the cost of evaluating $V \ltimes \triangledown R$ ($\ltimes$ denotes semijoin), removing those tuples from V, and updating the indexes of V.

- **Updates** The cost of propagating updates to R (μR) onto V is the cost of evaluating $V \ltimes \mu R$ and updating those tuples in V. Assuming protected updates and that indexes are on attributes that are keys or involved in selection/join conditions, we do not have to update the indexes of V.

4 Optimal Solution using A* algorithm

In this section we describe an optimal algorithm to solve the VIS problem and then show through experimental results that it vastly reduces the number of candidate solutions that must be considered.

4.1 Algorithm description

The A* algorithm ([12]) is an improvement over exhaustive search because it attempts to prune the parts of the search space that cannot contain the optimal solution. In this section we describe how we have used the A* algorithm to solve the VIS problem.

The algorithm takes as input the set of all possible views and indexes to materialize, $\mathcal{M}$. $\mathcal{M}$ does not include the base relations ($\mathcal{B}$) nor the primary view V but includes indexes that can be defined on them. (V and $\mathcal{B}$ are constrained to be materialized.) The goal of the algorithm is to choose a subset $\mathcal{M}'$ of $\mathcal{M}$ to materialize such that the total cost, $\mathcal{C}$, is minimized. The total cost given a particular subset of views and indexes $\mathcal{M}'$ can be expressed as

$$C(\mathcal{M}') = \sum_{m \in (\mathcal{M}' \cup \mathcal{B} \cup \{V\})} \mathit{maint\_cost}(m, \mathcal{M}')$$

Function $\mathit{maint\_cost}(m, \mathcal{M}')$ returns the cost of propagating all changes to view or index m assuming only the views and indexes in $\mathcal{M}'$ are materialized (in addition to $\mathcal{B}$ and V).

Instead of directly searching the power set of $\mathcal{M}$, we set up the A* search to build the solution incrementally. It begins with an empty materialization set ($\mathcal{M}' = \phi$) and then considers adding single views or indexes. The algorithm terminates when a solution is found that has considered every view and index and is guaranteed to have the minimum total cost. We will call the intermediate steps reached in the algorithm *partial states*. Each partial state is described by the tuple $(\mathcal{M}_\mathcal{C}, \mathcal{M}')$ where $\mathcal{M}_\mathcal{C}$ is the set of features from $\mathcal{M}$ that have been considered and $\mathcal{M}'$ is the set of features from $\mathcal{M}_\mathcal{C}$ that have been chosen to be materialized. For convenience, we will also refer to the set of unconsidered features, $\mathcal{M}_\mathcal{U}$, which is $\mathcal{M} - \mathcal{M}_\mathcal{C}$.

Presented with a set of partial states from which to incrementally search, A* attempts to choose the most promising. It does so by *estimating* the cost of the best solution $\mathcal{M}' \cup \mathcal{M}'_\mathcal{U}$ ($\mathcal{M}'_\mathcal{U}$ is the unconsidered features that would be chosen) that can be achieved from each state.

The *exact* cost of the best solution given a partial state can be decomposed as

$$\mathcal{C} = g + h$$

where g is the maintenance cost for the features chosen so far ($\mathcal{M}'$) and h is the maintenance cost for the features in $\mathcal{M}'_\mathcal{U}$. In general, g also needs $\mathcal{M}'_\mathcal{U}$ for its computation; that is, it is necessary to know which unconsidered features will be chosen in order to compute the maintenance cost of features in $\mathcal{M}'$. Fortunately, we can compute g using only $\mathcal{M}'$ so long as we impose a partial ordering on the features in $\mathcal{M}$ so that we only consider a feature when a decision has been made on every feature that affects its cost. Formally, a partial order $\prec$ is imposed upon $\mathcal{M}$ such that if a feature m_1 can be used in a query plan for propagating insertions to view m_2, then $m_1 \prec m_2$. Also, for an index m_1 on a view m_2, $m_2 \prec m_1$.

```
Input:      $\mathcal{M}$, $\prec$
Output:     Optimal $\mathcal{M}'$

Let state set $S = \{s\}$, where $s$ is a partial state having
    $\mathcal{M}_\mathcal{C}(s) = \mathcal{M}'(s) = \phi$, and $\mathcal{M}_\mathcal{U}(s) = \mathcal{M}$
    (base relations and $V$ are materialized)
Loop
    Select the partial state $s \in S$ with the minimum value of $\hat{\mathcal{C}}$
    If $\mathcal{M}_\mathcal{C}(s) \equiv \mathcal{M}$, return $\mathcal{M}'(s)$
    Let $S = S - \{s\}$
    For each view or index $m \in \mathcal{M}_\mathcal{U}(s)$ such that
    for all $m' \prec m$: $m' \in \mathcal{M}_\mathcal{C}(s)$
        Construct partial state $s'$ such that
            $\mathcal{M}_\mathcal{C}(s') = \mathcal{M}_\mathcal{C}(s) \cup \{m\}$, $\mathcal{M}_\mathcal{U}(s') = \mathcal{M}_\mathcal{U}(s) - \{m\}$,
            $\mathcal{M}'(s') = \mathcal{M}_\mathcal{C}(s) \cup \{m\}$
        Construct partial state $s''$ such that
            $\mathcal{M}_\mathcal{C}(s'') = \mathcal{M}_\mathcal{C}(s) \cup \{m\}$, $\mathcal{M}_\mathcal{U}(s'') = \mathcal{M}_\mathcal{U}(s) - \{m\}$,
            $\mathcal{M}'(s'') = \mathcal{M}_\mathcal{C}(s)$
        Let $S = S \cup \{s'\} \cup \{s''\}$
    Endfor
Endloop
```

Table 1. A* Algorithm

The exact formula for h is

$$\min_{\mathcal{M}'_\mathcal{U} \subseteq \mathcal{M}_\mathcal{U}} \Big(\sum_{m \in \mathcal{M}'_\mathcal{U}} \mathit{maint\_cost}(m, \mathcal{M}' \cup \mathcal{M}'_\mathcal{U}) \Big)$$

Unfortunately, this formula requires an exhaustive search to find the $\mathcal{M}'_\mathcal{U}$ that minimizes the equation.

Instead of performing this exhaustive search, we calculate a lower bound on h denoted $\hat{h}$. Using $\hat{h}$, the A* algorithm can prune some of the partial states while still guaranteeing an optimal solution. Using $\hat{h}$, for any partial state we can compute a lower bound on $\mathcal{C}$ as

$$\hat{\mathcal{C}} = g + \hat{h}$$

Note that if $\mathcal{M}_\mathcal{C} \equiv \mathcal{M}$ then $\hat{\mathcal{C}} = \mathcal{C}$. We will develop an expression for $\hat{h}$ below but first we present the A* algorithm for the VIS problem.

The algorithm appears in Table 1. The state set S contains all active partial states. It initially contains only the partial state where none of the views and indexes have been considered. Each time through the loop the algorithm selects the partial state with the minimum lower bound on the cost. If the selected state has $\mathcal{M}_\mathcal{C} \equiv \mathcal{M}$, it is guaranteed to be the optimal choice. If the selected state is not a complete state, it is removed from the set of active states and for each view or index that can be added to the set of considered views and indexes without violating the partial order, two states are added to the set of active states: one with the view or index added to the chosen set ($\mathcal{M}'$), and one without.

The formula for $\hat{h}$ computes the cost of maintaining views and indexes in $\mathcal{M}_{\mathcal{U}}$ minus the upper bound of their benefit toward maintaining other views (including V).

$$\hat{h} = \sum_{m \in \mathcal{M}_{\mathcal{U}}} (\mathit{h\_maint\_cost}(m, \mathcal{M}') - \mathit{max\_benefit}(m, \mathcal{M}'))$$

We guarantee that any overestimation of the actual maintenance cost of m is more than compensated for by the overestimation of the benefit. Note that our function $\hat{h}$, although it achieves considerable pruning, can be improved.

The function $\mathit{h\_maint\_cost}(m, \mathcal{M}')$ differs depending on whether m is a view or an index. If m is an index, the function returns the cost of maintaining m for all insertions and deletions that will be propagated to the view that m is on. (The details of our cost model are found in [9].) If m is a view, the function returns the cost of propagating onto m insertions to each of the base relations referenced in m, plus the cost of propagating onto m deletions and updates to each of the base relations referenced in m assuming the appropriate index exists. Note that when m is a view, we might overestimate the cost for propagating insertions since we are assuming that all other views in $\mathcal{M}_{\mathcal{U}}$ are not materialized (this is compensated for in *max_benefit*).

The function $\mathit{max\_benefit}(m, \mathcal{M}')$ also differs depending on whether m is a view or an index. First we consider the case where m is an index.

1. If m is an index on a view v for the key attribute of a base relation R that is referenced in v, the function returns the cost of propagating deletions and updates from R to v without m minus the cost of propagating deletions and updates from R to v with m.

2. If m is an index on a view v for a join attribute that joins v to some relation R not referenced in v, the function sums for each view $v' \in \mathcal{M}_{\mathcal{U}}$ that includes R as well as all the relations in v and for every relation S in v' but not in v, the cost of scanning v (the maximum savings due to an index join using m when propagating insertions from s onto v').

3. If m is an index for both a key and a join attribute, the two benefits described are added.

Next we consider the case where m is a view. Intuitively, the maximum benefit of m is the cost of materializing m when propagating insertions to views for which m is a subview. The *max_benefit* function sums for each view $v' \in \mathcal{M}_{\mathcal{U}}$ that includes all the relations in m and for every relation S in v' but not in m, the cost of materializing m given the views and indexes in $\mathcal{M}_{\mathcal{C}}$.

# of relations	# of selections	# of states visited exhaustive	A*	% pruned
2	0	32	11	67.7
2	1	192	21	89.1
2	2	960	28	97.1
2	4	960	29	97.0
3	1	$2.1 * 10^6$	17735	99.2
3	2	$1.1 * 10^7$	22809	99.8

Table 2. Comparison of A* and exhaustive algorithms.

4.2 Experimental results

To test the A* algorithm described in the previous section, we coded both it and the exhaustive algorithm described in Section 2. We then ran both algorithms on a variety of sample schemas. A summary of the results are presented in Table 2. Clearly, the A* algorithm prunes the vast majority of the search space. As the problems gets larger, due to more views or selection predicates, its relative performance increases as well. While it may still be possible to derive a tighter lower bound on h, even the algorithm as presented is a vast improvement over previously proposed algorithms.

5 Rules of Thumb

The A* algorithm presented in the last section yields optimal solutions while achieving impressive pruning. Still, due to the doubly exponential nature of the VIS problem, views that are computed from many base relations may still be too large to handle. Fortunately, finding an optimal solution is not critical since there are often many solutions that are close to optimal ([9]). What is required, then, is to avoid poor view sets and then to pick a good index set.

In this section we propose rules of thumb that can help guide a WHA in choosing a reasonable set of supporting views and indexes without resorting to the full algorithm. The underlying theme of these rules of thumb is to materialize a supporting view or index if its benefit is greater than its cost. These rules of thumb function similarly to the rule "join small relations first" in query optimization. These are not hard and fast rules but we have found that the rules apply in general.

Due to space constraints, we just list the rules in this section. In [9], we present a full justification of each rule of thumb through analysis and also through experimentation.

Rule 5.1 *Materialize a supporting view V when its size (in terms of pages) is less than the sum of the sizes of the views and base relations on which it is derived.* ⊙

Rule 5.2 *Materialize a supporting view V having no deletions or updates.* ⊙

Rule 5.3 *In considering whether to materialize a supporting view, the ratio of its size to the size of the memory buffer does not matter.* ⊙

Rule 5.4 *Build an index on a supporting view V for an attribute $R.A$ that is the key of base relation R involved in V if (1) there are some deletions and updates to R, (2) the number of deletions and updates to all base relations involved in V do not exceed the number of pages in V, and (3) the number of insertions and deletions to V does not exceed the number of pages in V.* ⊙

Rule 5.5 *Build an index on a supporting view V for an attribute $R.A$ that is involved in a join condition $R.A = S.B$ in the primary view when (1) S is not involved in V, (2) the number of insertions to base relations not involved in V but involved in the primary view does not exceed the number of pages in V, and (3) the number of insertions and deletions to V does not exceed the number of pages in V.* ⊙

Rule 5.6 *Don't build an index on a base relation R for an attribute $R.A$ involved in a selection condition C unless (1) indexes on R for attributes involved in join conditions have not been built, (2) a view $R' = \sigma_C R$ has not been materialized, (3) C is very selective, and (4) the number of deletions and updates to R do not exceed the number of pages in R.* ⊙

Rule 5.7 *Build an index on a supporting view V for an attribute $R.A$ if for any of the above Rules 5.4, or 5.6, all but the final condition hold, and the index fits in memory.* ⊙

6 Experimental Results

In this section, we present results that were borne directly from experimentation. In particular, we attempt to answer the following questions:

- Are views or indexes better when space is constrained?
- How sensitive is the optimal solution to the WHA's estimates of system parameters?

Due to space constraints, we present the results of only one representative experiment for each question although many more were performed. In addition, in the full version of the paper [9], we also consider whether protected updates should be treated atomically or split into pairs of insertions and deletions. The experiments shown in this section were all run on a view

$$\sigma_{R1=S0 \wedge S1=T0 \wedge C}(R(\underline{R}0, R1) \bowtie S(\underline{S}0, S1) \bowtie T(\underline{T}0, T1)), \quad (1)$$

where C is a condition on T with a selectivity of 10 %. Also, the relative cardinalities of the relations is: $T(R) = 3 * T(S) = 9 * T(T)$. ($T(R)$ denotes the number of tuples in R.) Although this schema is composed of only 3 relations, we believe our results to be more general because we have explored a number of larger schema with heuristic search algorithms and the results so far support those reported here.

6.1 Are Views or Indexes Better When Space is Constrained?

In this paper, we have shown how to find the optimal set of supporting views and indexes to materialize in order to minimize the total maintenance time. Sometimes, however, the amount of additional storage required is prohibitive. In these cases, one may ask how much storage is required to attain most of the performance gains and which structures should be materialized. We considered these questions for the schema given in Equation (1) under two different update loads. Due to space constraints, we only discuss the results for the high-update load experiment but the results for the low-update load experiment were similar. In the experiment, we gradually increase the available storage from that required to materialize the primary view (RST) to that required by the optimal solution for the unconstrained problem. For generality, we measure the additional space as a fraction of the space required to store the base relations. At each point we find the best solution that fits in the available storage. The cost of this solution relative to the non-constrained optimum is plotted on the y-axis.

The result of the experiment are shown in Figure 3. As the graph indicates, the schemas evolve in discrete steps - only changing when enough storage becomes available to add a new index or materialized view. The number of steps in the progression is too large (52 to be exact) to show every schema change but the results are summarized in Figure 4. The numbers next to the features indicate in what order they are added as storage increases. The experiment starts with only the base relations and primary view materialized – they are numbered 0. The next features to be added are indexes on the keys of the base relations present in the view RST, starting with $T0$ and then adding $S0$ and $R0$. Next, the selection node T' is materialized and an index built on its attribute $T'0$. The reason that it takes 52 steps to add all 10 numbered feature sets is that a new feature is often added at the expense of an older one. For instance, when the view T' is materialized, the index on $R0$ in RST is dropped until enough space is available to add it again. The graph in Figure 3 is also annotated with the feature numbers to help indicate which features impact the update performance.

The first important point to note from this experiment is that a large portion of the total update savings can be achieved with a reasonably small amount of additional storage. Note the large drop in I/Os in the experiment that results from materializing view T' (feature 3) and then adding indexes on $T0$ and $S0$ again (they were dropped earlier to make space for T'). The next large drop occurs after enough space is found to materialize ST (feature 5). By the time point A (which corresponds to features 1,2 and 5) is reached, the update cost is within 5% of the optimal cost. This is encouraging for warehouses that have space constraints. It should be noted that even though the extra storage required for the views and indexes does not seem that large compared to the warehouse relation sizes (≈ 25%), there will typically be many views defined over the same relations so the total storage required by views and indexes can be larger than that of warehouse relation when the warehouse is considered in its entirety.

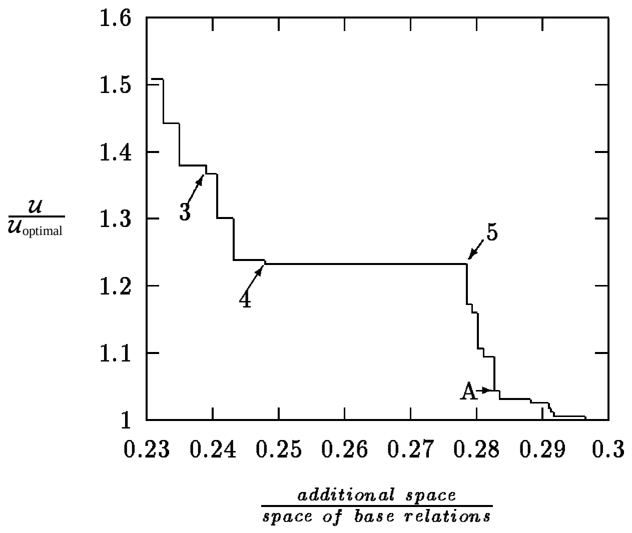

Figure 3. Effects of space on update cost.

It is interesting to see how Figure 4 is supported by our rules of thumb. Because RST is such a large relation, and there are deletions (but relatively few) to warehouse relations R, S, and T, by Rule 5.4, indexes should be built on RST for the keys of each of the warehouse relations. Also, because of the selection condition on T, the materialized view T' is much smaller than T. Therefore, by Rule 5.1 view T' should be materialized. Finally, note that view ST is not materialized until near the end. Even though the number of pages in ST is less than the sum of the pages in S and T and should be materialized by Rule 5.1, ST is a relatively large structure to materialize in comparison to the indexes. Therefore, we find that the maintenance cost is minimized overall in this case by materializing several small beneficial structures (*i.e.*, indexes) than by materializing one large one (*i.e.*, view ST). It isn't until the most useful indexes have already been materialized that view ST is chosen.

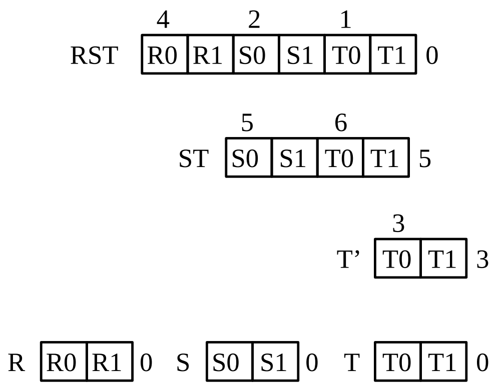

Figure 4. Evolution of the physical design.

6.2 Sensitivity Analysis

This paper has focused on finding an optimal solution to the VIS problem and also approximate solutions using rules of thumb. Just how well any solution works on the actual warehouse depends on how closely the input parameters, such as relation sizes and delta rates, match the real values of the system.[1] An important question for the WHA, then, is just how sensitive the optimizer is to the estimates of the input parameters. Clearly, one would hope that the optimal solution for the estimates is at least a good solution for systems with only slightly different parameters. In this section, we investigate just how badly optimal solutions decay at neighboring points. Due to space constraints, we consider only the estimate of insertion and deletion rates.

In this experiment, we varied the combined insertion and deletion rates to each base relation such that the ratio $\frac{I(R)+D(R)}{\|R\|} = \frac{I(S)+D(S)}{\|S\|} = \frac{I(T)+D(T)}{\|T\|}$ increased from 0.001 to 0.1 in five steps. At each step, we found the optimal solution and then plotted its performance over the entire range. The results, which are shown in Figure 5, suggest that except for a small region in the middle of the graph, the choice of optimum in not sensitive to the combined insertion-deletion rate. For instance, the optimal solution for an estimated ratio of 0.001 is still optimal even when the ratio grows to 0.01. The only area where the optimizer seems sensitive is in the range shown in the middle of the graph where an order of magnitude error in estimation can lead to a three-fold performance hit or worse. This sensitive region

[1]It also depends on how closely the VIS optimizer's cost model follows that of the dbms. This concept is discussed in [3].

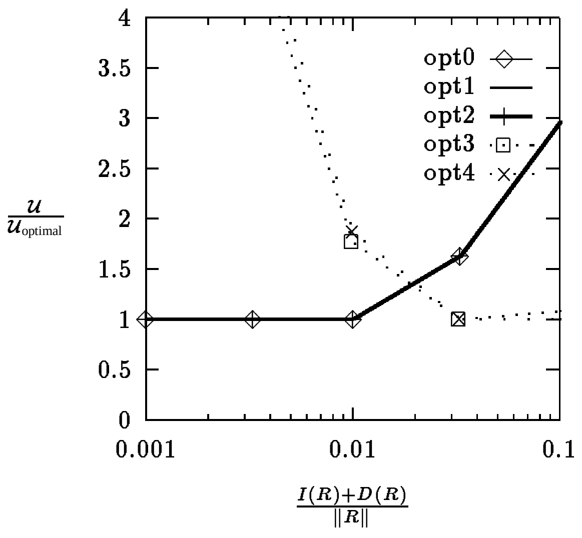

Figure 5. Sensitivity of Optimal Solutions to Insert/Delete Rates.

corresponds to the point when the insertion-deletion rate to the base relations becomes large enough that it is no longer worthwhile to build indexes on their attributes.

This experiment is typical of many sensitivity analyses that we have performed. The optimal solutions perform well across a wide range of parameter values except for a few small regions that correspond to major schema changes. This is reassuring. One must be careful, however, in over-generalizing this result. It is likely that in schemas with more relations there will be more frequent shifts in the optimal schema. Whether these shifts will result in large differences in the maintenance cost is a subject for future research.

7 Related Work

Previous work related to this paper falls into two categories, depending on the context in which it was written: physical database design and rule condition maintenance.

Physical Database Design Three costs must be balanced in physical database design for warehouses: (1) the cost of answering queries using warehouse relations and additional structures, (2) the cost of maintaining additional structures, and (3) the cost of secondary storage. We have assumed that the primary view is materialized, which minimizes the cost of (1), and focused on choosing supporting view and indices such that the cost of (2) is minimized. We have also considered constraining cost (3).

This problem was first studied by Roussopoloulos [14]. The additional structures considered for materialization are view indices, rather than the views themselves, to save on storage. A view index is similar to a materialized view except that instead of storing the tuples in the view directly, each tuple in the view index consists of pointers to the tuples in the base relations that derive the view tuple. In our paper we choose to maintain the actual views since the cost of secondary storage is now much lower.

The Roussopoloulos paper presents an elegant algorithm based on A* and the approximate knapsack problem to find an optimal solution to the view selection problem. The algorithm, however, works because of two simplifying assumptions. First, it uses a very simple cost model for updating a view: the cost is proportional to the size of the view. But we have shown in Section 2 that the cost of maintenance is a complex query optimization problem and cannot be estimated without knowing which subviews are materialized. Second, the Roussopoloulos algorithm does not consider index selection (other than view indices). We have shown in Section 6.1 that index selection has a significant impact on choosing which subviews to materialize. Relaxing either of the above two assumptions invalidates the use of the Roussopoloulos algorithm. Still, this is a very good first treatment of the subject.

More recently, Ross et al. [13] examines the same problem. They describe an exhaustive search algorithm to solve the VIS problem but without considering indexes. They also propose heuristics for pruning the space to search. We have extended their work by considering indexes, developing rules of thumb, and presenting an improved optimal algorithm. We have implemented our algorithm and used it to generate experimental results.

Other work has looked at the initial problem of choosing a set of primary views such that the cost of (1) is minimized, while ensuring that the costs of (2) and (3) are not too high. [17] considers this problem in the case of distributed views. [8] has investigated this problem for the case of aggregate views. Tsatalos et al. [20] consider materializing views in place of the base relations in order to improve query response time. Rozen et al. [15] look at this problem as adding a set of "features" to the database.

In particular, the index selection part of our VIS problem has been well-studied [3,1] in the context of physical database design. Choosing indexes for materialized views is a straightforward extension. What is troublesome, however, is that the previous algorithms require the queries (and their frequencies) on each base relation as inputs. In the VIS problem there are no user generated queries on the base relations or supporting views since they are all handled by the primary views: The only queries on base relations or supporting views are generated by maintenance expressions. Unfortunately, the set of generated queries depends on the update paths chosen for each type of delta. However, the choice between two update paths depends on the indexes chosen, which has not yet been determined. Thus one cannot determine the query set on each base relation and supporting view without knowing which indexes are present, which makes the algorithms proposed in previous work unusable here.

Rule Condition Maintenance Previous work on active database and production systems also relates to the VIS problem we have described. Many authors have considered how to evaluate trigger conditions for rules. This can be considered a view maintenance problem where a rule is triggered whenever the view that satisfies its condition becomes non-empty. Wang and Hanson [21] study how the production system algorithms Rete [4] and TREAT [11] perform in a database environment. An extension to TREAT called A-TREAT is considered in [7]. Fabret et al. [2] considered how to choose supporting views for the trigger condition view. Using our terminology, the rule of thumb they developed is to materialize a supporting view if it is *self-maintainable*; *i.e.,* when it can be maintained for the changes to the base relations by referencing the changes and the view itself, but without referencing any base relations. We have found that this is not true for our environment. In general, if there are insertions to the base relations, a join view is not self-maintainable and the Fabret approach does not materialize such a join view. However, even if there are insertions the join view may be beneficial (Rule 5.2) because the work for propagating insertions can be reused.

Segev et al. [16,18] consider a similar problem in expert systems. They also assume small deltas and ubiquitous indexes. They do not, however, consider maintaining subviews of the primary view, but instead describe *join pattern indexes*, which are specialized structures for maintaining materialized views. Join pattern indexes are an interesting approach, but require specialized algorithms to maintain.

A major difference between all of these studies and this one is that they consider a rule environment where changes in the underlying data are propagated immediately to the view. Hence, the size of the deltas sets are relatively small, which means that index joins will usually be much cheaper than nested-block joins. They therefore assume that indexes exist on all attributes involved in selection and join conditions. However, in the data warehousing environment studied here, a large number of changes are propagated at once, and the cost of maintaining the indexes often outweighs any benefit obtained by doing index joins.

8 Conclusions

This paper considered the VIS problem, which is one aspect of choosing good physical designs for relational databases used as data warehouses. We described and implemented an optimal algorithm based on A* that vastly prunes the search space compared to previously proposed algorithms [13]. Since even the A* algorithm is impractical for many real world problems, we developed rules of thumb for the for view and index selection.

By running experiments with the optimal algorithm, we studied how space can be best used when it is constrained: whether for materializing indexes or supporting views. Our results indicate that building indices on key attributes in the primary view lead to solid maintenance cost savings with modest storage requirements.

In the future we plan to develop and compare a number of heuristics for pruning the exhaustive search space so that good solutions can be found through limited search.

References

[1] S. Choenni, H. Blanken, and T. Chang. On the selection of secondary indices in relational databases. *Data and Knowledge Engineering*, 11:207–33, 1993.

[2] F. Fabret, M. Regnier, and E. Simon. An adaptive algorithm for incremental evaluation of production rules in database. In *Proceedings of International Conference on Very Large Data Bases*, pages 455–66, 1993.

[3] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.

[4] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[5] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In M. Carey and D. Schneider, editors, *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data*, pages 328–339, San Jose, CA, May 23-25 1995.

[6] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[7] E. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of 1992 ACM SIGMOD*, pages 49–58, 1992.

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

[9] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses - the vis problem. Technical report, Stanford University, 1996. Available by anonymous ftp from db.stanford.edu in /pub/labio/1996.

[10] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)*, pages 95–104, San Jose, CA, May 22-24 1995.

[11] D. P. Miranker. Treat: A better match algorithm for ai production systems. In *Proceedings of AAII 87 Conference on Artificial Intelligence*, pages 42–47, August 1987.

[12] N. Nilsson. *Problem solving methods in artificial intelligence*. McGraw-Hill, 1971.

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

[14] N. Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):258–90, 1982.

[15] S. Rozen and D. Shasha. A framework for automating physical database design. In *Proceedings of International Conference on Very Large Data Bases*, pages 401–11, 1991.

[16] A. Segev and W. Fang. Optimal update policies for distribtued materialized views. *Management Science*, 17(7):851–70, 1991.

[17] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[18] A. Segev and J. Zhao. Data management for large rule systems. In *Proceedings of International Conference on Very Large Data Bases*, pages 297–307, 1991.

[19] T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[20] O. Tsatalos, M. Solomon, and Y. Ioannidis. The gmap: A versatile tool for physical data independence. In *Proceedings of International Conference on Very Large Data Bases*, pages 367–78, 1994.

[21] Y. Wang and E. Hanson. A performance comparison of the rete and treat algorithms for testing database rule conditions. In *Proceedings of International Conference on Very Large Data Bases*, pages 88–97, 1992.