SELECTION AND MAINTENANCE OF VIEWS IN A DATA WAREHOUSE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Himanshu Gupta
September 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey Ullman (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Inderpal Mumick

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

Approved for the University Committee on Graduate Studies:

Abstract

A data warehouse is a repository (database) that integrates information extracted from various remote sources, with the purpose of efficiently supporting decision support queries. The information stored at the warehouse is in the form of database tables, referred to as materialized views, derived from the data in the sources. In order to keep a materialized view consistent with the data at sources, the view needs to be incrementally maintained. The two important issues that arise in the design of a data warehouse are selection of views to materialize and incremental maintenance of materialized views. This doctoral thesis looks at these two design issues and presents comprehensive solutions to both problems.

Selection of views to materialize: We develop a theoretical framework for the general problem of selection of views in a data warehouse. Given a set of queries to be supported, the view selection problem is to select a set of views to materialize minimizing the query response time given some resource constraint. For different resource constraints and settings, we have designed approximation algorithms that provably return a set of views having a query benefit within a constant factor of the optimal.

Incremental maintenance of general view expressions: Traditional maintenance algorithms maintain view expressions in response to changes at the base relations by computing and propagating insertions and deletions through intermediate subexpressions. In this thesis, we have developed a change-table technique, that computes and propagates "change-tables" through subexpressions, for incremental maintenance of general view expressions involving aggregate and outerjoin operators. We show that the presented change-table technique outperforms the previously proposed techniques by orders of magnitude.

Acknowledgments

I would like to thank a number of people, each one of whom has made my four years of doctoral research at Stanford seem wonderful.

First and foremost, I would like to thank my advisor Professor Jeffrey Ullman. I would always be grateful to him for all the encouragement and support he has given. In effect, I learnt from him that frustration is only a small part of research, and perseverance and focussed thinking eventually yields fruitful research.

I am also grateful to Dr. Inderpal Mumick, with whom I did a great part of my research. It was indeed a great pleasure working with him, and I have learnt a great deal from him.

I would also like to thank Professor Hector Garcia-Molina and Professor Jennifer Widom for their contribution and support in making the WHIPS (Warehousing Information Project at Stanford) project very successful. My impulsive decision to join WHIPS has yielded great results, as most of my doctoral research has come out of the project.

Finally, I would like to thank all my co-authors for their ideas and collaboration. I have learnt a great deal from each one of them.

A note of thanks also to the whole database group and, in particular, the data warehouse (WHIPS) team.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In a typical organization, there is information stored in multiple, independent, and heterogenous data sources. Business analysts want to run applications that ask complex queries over these information sources to detect trends in businesses. Functioning as a "data library," a data warehouse makes information readily available for querying and analysis. In essence, a *data warehouse* extracts, integrates, and stores "relevant" information from independent information sources into a central database. The information is stored at the warehouse in *advance* of the queries. In such a system, user queries can be answered using the information stored at the warehouse and need not be translated and shipped to the original source(s) for execution. Also, warehouse data is available for queries even when the original information source(s) are inaccessible due to real-time operations or updates.

Figure 1.1 illustrates the architecture of a typical data warehouse. The bottom of the figure depicts the multiple *information sources* of interest. Near the top of the figure is the data warehouse, where data that is relevant to the queries to be supported is derived or copied and integrated. Between the sources and the warehouse lie the *source monitors* and the *integrator*. The monitors are responsible for automatically detecting changes in the source data, and reporting them to the integrator. The integrator is responsible for bringing source data into the warehouse, propagating changes in the source relations to the warehouse, and maintaining the tables at the warehouse. Widom in [Wid95] gives a nice overview of the technical issues that arise in the different components of a data warehouse.

The information stored at the warehouse is in the form of derived views of data from the sources. These views stored at the warehouse are often referred to as *materialized views*. Materialized views can speed up the execution of many queries. Any query whose execution

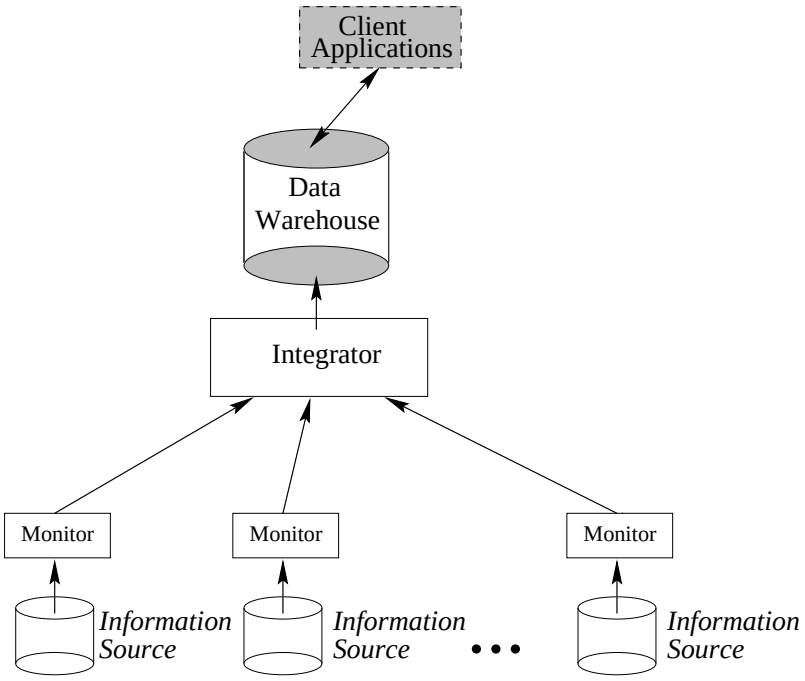

Figure 1.1: A typical data warehouse architecture

plan can be rewritten to use a materialized view is subject to speed-up. For complex queries involving large volumes of data, the speed-up possible using materialized views is dramatic: from hours or days down to seconds or minutes. In fact, materialized views are regarded as one of the primary means for managing performance in a data warehouse [Kim96].

There are several architectural issues concerned with the design of a data warehouse. One of the most important design problems is to select an appropriate set of materialized views to store at the data warehouse. Also, in order to keep a materialized view up to date with the information sources, the view has to be maintained in response to changes at the sources. As recomputing views is very expensive, we wish to maintain the view *incrementally* by calculating the effects of the changes at the sources. This thesis addresses these design issues, and presents comprehensive solutions to the design problems.

Below, we briefly discuss the problems addressed in this thesis, and present the contributions the thesis makes towards solving them.

1.1 Selection of Views to Materialize in a Data Warehouse

We are required to support a set of complex queries at the warehouse. To aid answering the queries efficiently, we materialize a set of views that are "closely-related" to the queries at the warehouse. We cannot materialize all possible views, as we are constrained by some resource like disk space, computation time, or maintenance cost. Hence, we need to select an appropriate set of views to materialize under some resource constraint. The *view-selection* problem is defined as selection of views to materialize to minimize the query response time under some resource constraint.

1.1.1 Contributions and Related Work

In the initial research done on the view-selection problem, Harinarayan, Rajaraman and Ullman [HRU96] presented algorithms for the view-selection problem in data cubes under a disk-space constraint. A data cube is a special purpose data warehouse, where there are only queries with aggregates over the base relation.

In Chapter 2, we present a theoretical framework for the general problem of selection of views in a data warehouse. We present approximation algorithms for the view-selection problem arising in various special cases of a general data warehouse, viz. (i) OR view graphs, where any view can be computed from *any one* of its related views, e.g., data cubes, and (ii) AND view graphs, where each query/view has a unique evaluation. We extend the algorithms to the case when there is a set of indexes associated with each view. Finally, we extend our heuristic to the most general case of AND-OR view graphs.

The work presented in [HRU96] and in Chapter 2 of this thesis has developed approximation algorithms to select a set of structures that minimizes the total query response time under a given *space* constraint; the constraint represents the maximum amount of disk space that can be used to store the materialized views. In practice, the real constraining factor may be the total maintenance cost incurred by the materialized views due to updates at the information sources. Thus, in Chapter 3 we address the problem of selecting views to materialize under the constraint of total maintenance cost. For the special case of "OR view graphs," we present an inverted-tree greedy algorithm that provably delivers a near-optimal solution. For the general case of AND-OR view graphs, we present an A^* heuristic that delivers an optimal solution. We present a performance study of the developed algorithms, which shows the inverted-tree greedy almost always returns an optimal solution

and outperforms the A^* algorithm by orders of magnitude. The resource constraint of total materialization or computation time of the materialized views is handled in the same way as the maintenance cost constraint.

Apart from [HRU96], other works on the view-selection problem have been as follows. Ross, Srivastava, and Sudarshan in [RSS96], Yang, Karlapalem, and Li in [YKL97], Baralis, Paraboschi, and Teniente in [BPT97], and Theodoratos and Sellis in [TS97] provide various frameworks and heuristics for selection of views in order to optimize the sum of query response time and view maintenance time without any resource constraint. The heuristics presented in these works are either exhaustive searches or do not have any performance guarantees on the quality of the solution delivered. In contrast, we have designed approximation algorithms that deliver a provably good solution with a query benefit within a constant factor of the optimal query benefit.

1.2 Incremental Maintenance of General View Expressions

In order to keep the views in the data warehouse upto date, it is necessary to maintain the materialized views in response to the changes at the information sources. The views can be either recomputed from scratch, or *incrementally maintained* by propagating the base data changes onto the view. As recomputing the views can be prohibitively expensive, the incremental maintenance of views is of significant value.

The problem of finding such changes at the views based on changes to the base relations has come to be known as the *view maintenance problem* and several algorithms have been proposed over the recent years for incremental maintenance of view expressions. However, none of the proposed algorithms handle the general case view expressions involving aggregate and outerjoin operators efficiently.

In Chapter 4, we develop a new *change-table technique* for incremental maintenance of general view expressions. All the previously proposed algorithms on incremental maintenance work by computing and propagating insertions and deletions at intermediate subexpressions. In contrast, the change-table technique computes and propagates "change-tables." We show that the presented change-table technique outperforms the previously proposed techniques by orders of magnitude.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In the next chapter, we address the problem of selecting views to materialize in a data warehouse under the constraint of disk space. Chapter 3 looks at the view-selection problem under a maintenance-cost constraint. In Chapter 4, we address the problem of incremental maintenance of general view expressions and develop the framework for the change-table technique. We end with future directions and conclusions in Chapter 5.

Chapter 2

Selection of Views to Materialize

2.1 Introduction

Decision support systems are rapidly becoming a key to gaining competitive advantage for businesses. Business analysts running the decision support applications want to detect business trends by mining the data stored in the information sources. Typically, the information sources in data warehouses maintain historical information, and hence the databases tend to be very large and grow over time. Also, decision support applications are typically interested in identifying trends rather than looking at individual records in isolation. Thus, decision-support queries make heavy use of aggregations and are very complex.

The size of the information source databases and the complexity of queries can cause queries to take very long to complete. This delay is unacceptable in most decision support environments, as it severely limits productivity. The usual requirement is query execution times of a few seconds or a few minutes at the most. There are many ways to achieve such performance goals. Query optimizers and query evaluation techniques can be enhanced to handle aggregations better [CS94, GHQ95, YL95], and one can also use different indexing strategies like bit-mapped indexes and join indexes [OG95].

A commonly used technique in data warehouses is to materialize (precompute) frequently asked queries. The data warehouse at the Mervyn's department-store chain, for instance, has a total of 2400 precomputed tables to improve query performance. Picking the right set of queries to materialize is a nontrivial task, since by materializing a query we may be able to answer other queries quickly. For example, we may want to materialize a query that is relatively infrequently asked if it helps us answer many other queries quickly.

Moreover, we could also materializes other tables that are not queries but nevertheless help in answering the queries efficiently. In this chapter, we present a framework and algorithms that enable us to pick a good set of tables to materialize.

The problem of selecting an appropriate set of views to materialize is one of the most important design decisions in designing a data warehouse. Given some resource constraint, the problem is to select a set of derived views to minimize total query response time and the cost of maintaining the selected views. We refer to this problem as the *view-selection problem*. In this chapter, we concentrate only on disk-space as resource constraint. We address the view-selection problem under other resource constraints in the next chapter.

In this chapter, we develop a theoretical framework for the general problem of selecting views to materialize in a data warehouse. We present competitive polynomial-time heuristics for selection of views to optimize total query response time, for some important special cases of the general data warehouse scenario, viz.: (i) an OR view graph, in which any view can be computed from *any one* of its related views, e.g., data cubes, and (ii) an AND view graph, where each query/view has a unique evaluation. We extend the algorithms to the case when there is a set of indexes associated with each view. Finally, we extend our heuristic to the most general case of AND-OR view graphs. The work presented in this chapter appears in [Gup97] and [GHRU97].

The rest of the chapter is organized as follows. In the next section, we develop a theoretical framework for the view-selection problem. The problem can be easily shown to be NP-complete even for some very simple cases. In the following two sections, we present and analyze heuristics for two special cases, viz: i) OR view graphs in which any view can be computed from any one of its related views, and ii) AND view graphs, where each view has a unique evaluation. For each of these cases, we extend the algorithms to a more general case when there are index structures associated with the views. In Section 2.5, we present an algorithm for the general view-selection problem in a data warehouse. Finally, we end with concluding remarks in Section 2.6.

2.2 View-Selection Problem Formulation

2.2.1 AND-OR View Graphs

In this subsection, we develop a notion of an AND-OR view graph, which is one of the inputs to the view-selection problem. We start by defining the notions of expression DAGs

for queries or views.

Definition 1 (**Expression AND-DAG**) An *expression AND-DAG* for a query or a view V is a directed acyclic graph having the base relations as "sinks" (no outgoing edges) and the node V as a "source" (no incoming edges). If a node/view u has outgoing edges to nodes $v_1, v_2, \ldots, v_k$, then *all* of the views $v_1, v_2, \ldots, v_k$ are required to compute u. This dependence is indicated by drawing a semicircle, called an *AND arc*, through the edges $(u, v_1), (u, v_2), \ldots, (u, v_k)$. Such an AND arc has an operator[1] and a cost associated with it, which is the cost incurred during the computation of u from $v_1, v_2, \ldots, v_k$. □

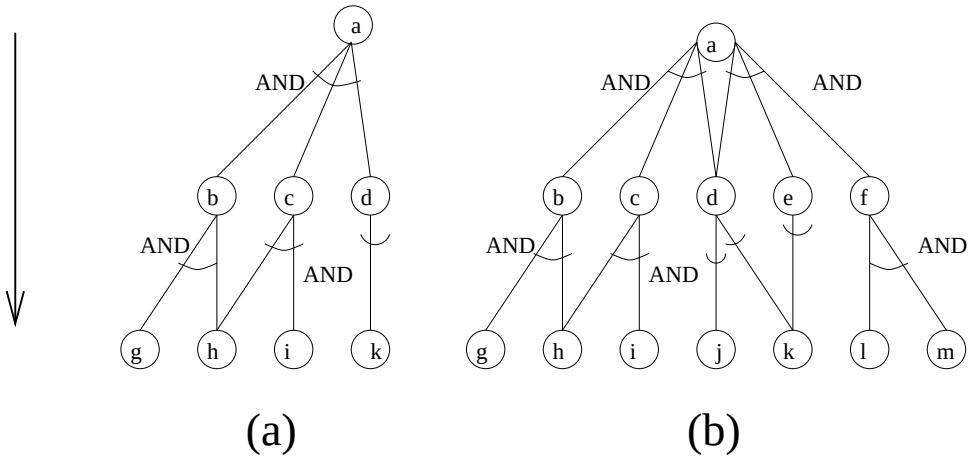

Figure 2.1: a) An expression AND-DAG, b) An expression ANDOR-DAG

An example of an expression AND-DAG is shown in Figure 2.1(a). Expression AND-DAGs are more commonly referred to as "expression trees." One inherent drawback of expression AND-DAGs is that they do not depict alternative ways of evaluating a view. The expression ANDOR-DAG, defined next, is a more general notion, which overcomes this shortcoming. An expression ANDOR-DAG may have more than one AND arc at each node, making it an AND/OR expression DAG.

Definition 2 (**Expression ANDOR-DAG**) An *expression ANDOR-DAG* for a view or a query V is a directed acyclic graph with V as a source and the base relations as sinks. Each nonsink node has associated with it one or more AND arcs, each binding a *subset* of

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

its outgoing edges. As in the previous definition, each AND arc has an operator and a cost associated with it. More than one AND arc at a node depicts multiple ways of computing that node. □

Figure 2.1 shows an example of an expression AND-DAG as well as an expression ANDOR-DAG. In Figure 2.1 (b), the node a can be computed either from the set of views $\{b, c, d\}$ or $\{d, e, f\}$. The view a can also be computed from the set $\{j, k, f\}$, as d can be computed from j or k and e can be computed from k.

Definition 3 (**AND-OR View Graph**) A directed acyclic graph G having the base relations as the sinks is called an *AND-OR view graph* for the views (or queries) $V_1, V_2, \ldots, V_k$ if for each V_i, there is a subgraph[2] G_i in G that is an expression ANDOR-DAG for V_i. Each node v in an AND-OR view graph has the following parameters associated with it: space S_v, query-frequency f_v (frequency of the queries on v), update-frequency g_v (frequency of updates on v), and reading-cost R_v (cost incurred in reading the materialized view v). □

Note that in an AND-OR view graph, if a view v can be computed from $v_1, v_2, \ldots, v_l$, and a view u can be computed from the views $v, u_1, u_2, \ldots, u_k$, then the view u can also be computed from $u_1, u_2, \ldots, u_k$, $v_1, v_2, \ldots, v_l$.

Definition 4 (**Evaluation Cost**) The *evaluation cost* of an AND-DAG H embedded in an AND-OR view graph G is the sum of the costs associated with the AND arcs in H, plus the sum of the reading costs associated with the sinks/leaves of H. □

2.2.2 Constructing an AND-OR View Graph

Given a set of queries $Q_1, Q_2, \ldots, Q_k$ to be supported at a warehouse, we construct an AND-OR view graph for the queries as follows. We first construct an expression ANDOR-DAG D_i for each query Q_i in the set. An AND-OR view graph G for the set of queries can then be constructed by "merging" the expression ANDOR-DAGs $D_1, D_2, \ldots, D_k$. Each node in the AND-OR view graph G will represent a view that could be selected for materialization,

[2] An AND-OR view graph H is called a subgraph of an AND-OR view graph G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$, and each edge e_1 in H is bound with the same set of edges through an AND-arc as it is bound through an AND-arc in G. That is, if $e_1, e_2 \in E(G)$, $e_1 \in E(H)$, and e_1 and e_2 are bound by an AND-arc (which may bind other edges too) in G, then $e_2 \in E(H)$, and e_1 and e_2 are bound with the same AND-arc in H. For example, Figure 2.1 (a) is a subgraph of Figure 2.1 (b), but Figure 2.1 (a) without the edge (c, h) is not.

and these are the only views considered for materialization. The space parameter S_v for a view v can be determined by computing the expected number of tuples in the view. The query and update frequencies are computed based on the query and update requirements of the data warehouse.

For a query Q_i we construct its expression ANDOR-DAG D_i to consist of alternative "useful" ways of evaluating Q_i from the given base relations, in the presence of other queries/views. Roussopoulos in [Rou82a] considers exactly this problem. The objective of his analysis is to identify all possible (useful) ways to produce the result of a view, given other view definitions and base relations.

2.2.3 The View-Selection Problem

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

More formally, let $Q(v, M)$ denote the cost of answering a query v (also a node of G) using the set M of materialized views in the given view graph G, and $UC(v, M)$[3] be the maintenance cost (due to updates to base tables) for the view v in the presence of the set of materialized views M. We will always assume that the set of sinks L is also available for querying and maintenance purposes. Then, given an AND-OR view graph G for queries $Q_1, \ldots, Q_k$ and a quantity S, the view-selection problem is to select a set of views/nodes $M = \{V_1, V_2, \ldots, V_m\}$, that minimizes $\tau(G, M)$, where

$$\tau(G, M) = \sum_{i=1}^{k} f_{Q_i} Q(Q_i, M) + \sum_{i=1}^{m} g_{V_i} UC(V_i, M),$$

under the constraint that $\sum_{v \in M} S_v \leq S$. Recall that S_v is the space occupied by the view v.

The view-selection problem is NP-hard even for the special case of an AND-OR graph where each AND arc binds exactly one edge, and the update frequencies are zero. There is a straightforward reduction from minimum `set cover`.

Computing Q(v,M) The cost of answering a query v in presence of a set of (materialized) views M, $Q(v, M)$, in an AND-OR view graph G is actually the evaluation cost of the

[3]The function symbol UC denotes *update cost*.

cheapest AND-DAG H_v for v, such that H_v is a subgraph of G and the sinks of H_v belong to the set $M \cup L$, where L is the set of sinks in G. Again, we have assumed that L, the set of sinks in G, is available for computation as it represent the set of base tables at the source(s). Thus, the value $Q(v, \phi)$ is the cost of answering a query on v directly from the source(s). In the special case of OR view graphs, $Q(v, M)$ is the minimum query-length of a path from v to some $u \in (M \cup L)$, where the *query-length* of a path from v to u is defined as R_u, the reading cost of u, plus the sum of the query-costs associated with the edges on the path.

In this chapter, we have ignored maintenance costs except in the Section 2.4.3. Hence, we defer the discussion on maintenance cost models and how to compute $UC(v, M)$ until the next chapter. The computation of $Q(v, M)$ and $UC(v, M)$ in OR view graphs is illustrated further in Example 3.

2.2.4 Benefit of a Set of Selected Views

In this subsection, we define the notion of a "benefit" function, which is central to the development of algorithms presented in this chapter. In the following two sections, we will present approximation algorithms for some special cases of the general view-selection problem.

Let C be an arbitrary set of views in a view graph G. The *benefit* of C with respect to M, an already selected set of views, is denoted by $B(C, M)$ and is defined as $\tau(G, M) - \tau(G, M \cup C)$, where τ is the function defined above. The benefit of C per unit space with respect to M is $B(C, M)/S(C)$, where $S(C)$ is the space occupied by the views in C. Also, $B(C, \phi)$ is called the *absolute benefit* of the set C.

Monotonicity Property

The benefit function B is said to satisfy the *monotonicity property* for M with respect to sets (of views) $O_1, O_2, \ldots, O_m$ if $B(O_1 \cup O_2 \ldots \cup O_m, M) \leq \sum_{i=1}^{i=m} B(O_i, M)$.[4]

The monotonicity property of the benefit function is important for the greedy heuristics to deliver competitive (within a constant factor of optimal) solutions. For a given instance of AND-OR view graph, if the optimal solution O can be partitioned into disjoint subsets of views $O_1, O_2, \ldots, O_m$ such that the benefit function satisfies the monotonicity property

[4]Considering $m = 2$ is sufficient, but we state it for general m so that its application is direct.

w.r.t. $O_1, O_2, \ldots, O_m$, then we guide the greedy heuristic to select, at each stage, an optimal set (of views) of type that includes O_i for all $i \leq m$. Such a greedy heuristic is guaranteed to deliver a solution whose benefit is at least 63% of the optimal benefit, as we show later.

2.3 OR View Graphs

In this section, we consider a special case of the view-selection problem for AND-OR view graphs. We restrict our attention to those AND-OR view graphs in which each AND arc binds exactly one edge. For such restricted AND-OR view graphs, we can remove AND arcs altogether, and associate the costs and the operators with the corresponding edges in the graph. We call such a AND-OR view graph G an *OR view graph*, where a node can be computed from any one of its children.

2.3.1 Motivation

The OR view graphs arise in many useful practical applications when computation of a view depends on only one other view. A simple application is when all the views and queries involved are aggregate queries over the base data. Data cubes is another example of OR view graphs.

Data Cubes In a data cube users can view the data as multidimensional data. *Data cubes* are databases where a critical value, e.g., `sales`, is organized by several dimensions, for example, sales of automobiles organized by model, color, day of sale, place of sale, age of purchaser and so on. The metric of interest is called the *measure attribute*, which is `sales` in the above example. Queries in such a system are of the OLAP (On line Analytic Processing) type, usually asking for a breakdown of `sales` by some of the dimensions. Therefore, we can associate an aggregate view, called a *cube*, V_α with each subset α of the dimensions. A view V_α is essentially a result of a "Select α, Sum(`sales`); group by α" SQL query over the base table. Hence, an aggregate view V_α can be computed from a view V_β iff $\alpha \subseteq \beta$.

In a data cube, the AND-OR view graph is an OR view graph, as for each view there are zero or more ways to construct it from other views, but each way involves only one other view. Data cubes being a special case of OR view graphs, all the results developed in this section apply to data cubes. As OLAP databases have very few or no updates at the base table, we assume that there are no maintenance costs at the materialized views throughout

this section.

2.3.2 Selection of Views in an OR View Graph

In this subsection, we present heuristics for solving the view-selection problem in OR view graphs without maintenance costs.

Problem: Given an OR view graph G without updates and a quantity S , find a set of views M that minimizes the quantity $\tau(G, M)$, under the constraint that the total space occupied by the views in M is at most S.

Greedy Algorithm

We present a simple greedy heuristic for selecting views. At each stage, we select a view which has the maximum benefit per unit space at that stage. The greedy heuristic is presented below as Algorithm 1.

Algorithm 1 Greedy Algorithm

Given: G, an AND-OR view graph, and S, the space constraint.
BEGIN
  $M = \phi$; /* M = set of structures selected so far. */
  while $(S(M) < S)$
    Let C be the view which has the maximum benefit per unit space
      with respect to M.
    $M = M \cup C$;
  end while;
  return M;
END. ◇

The running time of the greedy algorithm is $O(kn^2)$, where n is the number of nodes in the graph and k is the number of stages used by the algorithm.

Observation 1 *In an OR view graph without updates, the benefit function B satisfies the monotonicity property for any M with respect to arbitrary set of views* $O_1, O_2, \ldots, O_m$.

Theorem 1 *For an OR view graph G without updates and a quantity S, the greedy algorithm produces a solution M that uses at most $S + r$ units of space, where r is the size of the largest view in G. Also, the absolute benefit of M is at least $(1 - 1/e)$ times the optimal benefit achievable using as much space as that used by M.*

Proof: It is easy to see that the space used by the greedy algorithm solution, $S(M)$, is at most $S + r$ units. Let $k = S(M)$. Let the optimal solution using k units of space be O and the absolute benefit of O be B.

Consider a stage at which the greedy algorithm has already chosen a set G_l occupying l units of space with "incremental" benefits $a_1, a_2, \ldots, a_l$. Incremental benefit a_i is defined as the increase in benefit of M, when the i^{th} unit of space is added to M. Thus, the absolute benefit of G_l is $\sum_{i=1}^{l} a_i$. Surely the absolute benefit of the set $O \cup G_l$ is at least B. Therefore, the benefit of the set O with respect to G_l, $B(O, G_l)$, is at least $B - \sum_{i=1}^{l} a_i$.

Let $O = \{O_1, O_2, \ldots, O_m\}$. By the monotonicity property of the benefit function for the views O_i's, $B(O, G_l) \leq \sum_{i=1}^{m} B(O_i, G_l)$. Now, we show by contradiction that there exists a view O_i in O such that $B(O_i, G_l)/|O_i| \geq B(O, G_l)/k$. Let us assume that there is no such view O_i in O. Then, $B(O_j, G_l) < (B(O, G_l)/k) * |O_j|$ for every view $O_j \in O$. Thus, $\sum_{O_j \in O} B(O_j, G_l) < (B(O, G_l)/k) * \sum_{O_j \in O} |O_i| = B(O, G_l)$, which violates the monotonicity property of the benefit function for the views $O_j \in O$. Therefore, there exists a view O_i in O such that $B(O_i, G_l)/|O_i| \geq B(O, G_l)/k \geq (B - \sum_{i=1}^{l} a_i)/k$.

The benefit per unit space with respect to G_l of the view C selected by the algorithm is at least that of O_i, which is at least $B(O, G_l)/k = (B - \sum_{i=1}^{l} a_i)/k$, as shown above. Distributing the benefit of C over each of its unit spaces equally (for the purpose of analysis), we get $a_{l+j} \geq (B - \sum_{i=1}^{l} a_i)/k$, for $0 < j \leq S(C)$. As the above analysis is true for each view C selected at any stage, we have

$$B \leq k a_j + \sum_{i=1}^{j-1} a_i \qquad \text{for } \; 0 < j \leq k.$$

Multiplying the j^{th} equation by $(\frac{k-1}{k})^{k-j}$ and adding all the equations, we get

$A/B \geq 1 - (\frac{k-1}{k})^k \geq 1 - 1/e$, where $A = \sum_{i=1}^{k} a_i$ is the absolute benefit of M. ∎

Feige in [Fei96] showed that the `minimum set-cover` problem cannot be approximated within a factor of $(1 - o(1)) \ln n$, where n is the number of elements, using a polynomial time algorithm unless $P = NP$. There is a very natural reduction of the `minimum set-cover` problem to our problem of view selection in OR view graphs. The reduction shows that no

polynomial time algorithm for the view-selection problem in OR view graphs can guarantee a solution of better than 63% for all inputs unless $P = NP$ [Che96].

Greedy-Interchange Algorithm

We present another heuristic called the "greedy-interchange" algorithm, which starts with the solution produced by the greedy algorithm (Algorithm 1) and then improves the solution by interchanging a view already selected with some view not selected.[5] It iteratively performs such interchanging until the solution cannot be improved any further by an interchange. We present the Greedy Interchange Algorithm below as Algorithm 2.

Algorithm 2 **Greedy-Interchange Algorithm**

Given: G, an AND-OR view graph, and S, the space constraint.
Assume that all views occupy the same amount of space.
BEGIN
Run the greedy algorithm and let M be the solution returned.
repeat
Let (C_1, C_2) be a pair of views such that $C_1 \in M$ and $C_2 \notin M$ and the absolute benefit of $(M - C_1) \cup C_2$ is greater than that of M.
$M = (M - C_1) \cup C_2$;
until (no such pair (C_1, C_2) exists);
return M;
END. ◇

Unfortunately, not much can be proved about the competitiveness of the solution produced by the greedy-interchange algorithm except that it is obviously at least as good as the greedy algorithm. Moreover, the running time of the greedy-interchange algorithm is unbounded. We believe that the greedy-interchange algorithm in practice would perform much better than the greedy algorithm.

Cornuejols et al. in [CFN77] show for their similar facility location problem through extensive experiments that in most cases the running time of greedy-interchange is a little less than 1.5 times the running time of the greedy algorithm, and that it returns a much better solution than that returned by the greedy algorithm.

[5] When views occupy different amounts of space, more than one view may have to be added/removed.

2.3.3 OR View Graph with Indexes

In this section, we generalize the view-selection problem in an OR view graph by introducing indexes for each node/view. As in the original OR view graph, a node can be computed from any one of its children, but in the presence of indexes the cost of computation depends upon the index being used to execute the operation. As indexes are built upon their corresponding views, an index can be materialized only if its corresponding view has already been materialized. Thus, selecting an index without its view does not have any benefit, and the benefit of an index actually increases with the materialization of its view. Hence, the benefit function may not satisfy the monotonicity property for arbitrary sets of views and indexes. We use the term *structure* to denote a view or an index. We assume that if an index is not materialized, then it is never "computed" while answering user queries.

In most commercial systems today, the views that are to be precomputed are selected first, followed by the selection of the appropriate indexes on them. A trial-and-error approach is used to divide the space available between the summary tables and the indexes. This two-step process can perform very poorly. Since both views and indexes consume the same resource - space - their selection should be done together for the most efficient use of resouces. In this section, we present a family of algorithms of increasing time complexities, and prove strong performance bounds for them.

We need to introduce a slightly different cost model for the OR view graphs with indexes. In an OR view graph with indexes, there may be multiple edges from a node u to v, possibly one for each index of v. Instead of associating a cost with the edges, we associate a label (i, t_i) with each edge from u to v. The label $t_i (i > 0)$ can be thought of as the cost incurred in computing u from v using its i^{th} index. When $i = 0, t_0$ is the cost in computing u from v without any of its indexes.

Problem: Given a quantity S and an OR view graph G with indexes. Associated with each edge is a label $(i, t_i), i \geq 0$ as described above. Assume that there are no updates.

Find a set of structures M that minimizes the quantity $\tau(G, M)$, under the constraint that the total space occupied by the structures in M is at most S.

The r-Greedy Algorithm

The r-greedy algorithm executes in a number of stages. Lets consider a stage when the set of structures M has already been selected for materialization. At this stage, the r-greedy

algorithm considers for selection all sets of at most r structures that consist either of

- A view and some of its indexes, or
- A single index whose view has already been selected in one of the previous stages.

Among all the considered sets, the set that has the maximum benefit per unit space with respect to M is finally selected for materialization at this stage. The r-greedy algorithm has a good performance guarantee when each structure occupies the same amount of space. Algorithm 3 shows the r-greedy algorithm.

Algorithm 3 r-Greedy Algorithm

Given: G, an AND-OR view graph, and S, the space.
BEGIN
$M = \phi$; /* M = set of structures selected so far. */
while $(S(M) < S)$
Look at all sets of one of the following forms:
- $\{v_i, I_{ij_1}, I_{ij_2}, \ldots, I_{ij_p}\}$, such that $v_i \notin M$, $I_{ij_l} \notin M$ for $1 \leq l \leq p$, and $0 \leq p < r$,

or
- $\{I_{ij}\}$, such that v_i is in M, and $I_{ij} \notin M$.

Among these sets, let C be the set that has the maximum benefit per unit space with respect to M.
$M = M \cup C$;
end while;
return M;
END. ◇

Suppose there are v views and each view has at most i indexes. Then at each stage, the r-greedy algorithm must consider and calculate the benefit of at most $vi + v\binom{i}{r-1}$ possible sets. Hence an upper bound on the running time of the algorithm is $O(km^r)$, where m is the number of structures in the given AND-OR view graph and k is the number of structures selected by the algorithm, which is S in the worst case.

EXAMPLE 1 We illustrate the working of the r-greedy algorithm through a simple example.

Consider the OR view graph shown in Figure 2.2. For illustration, we assume that the only candidate views for materialization are the nodes at the bottom, and the only queries

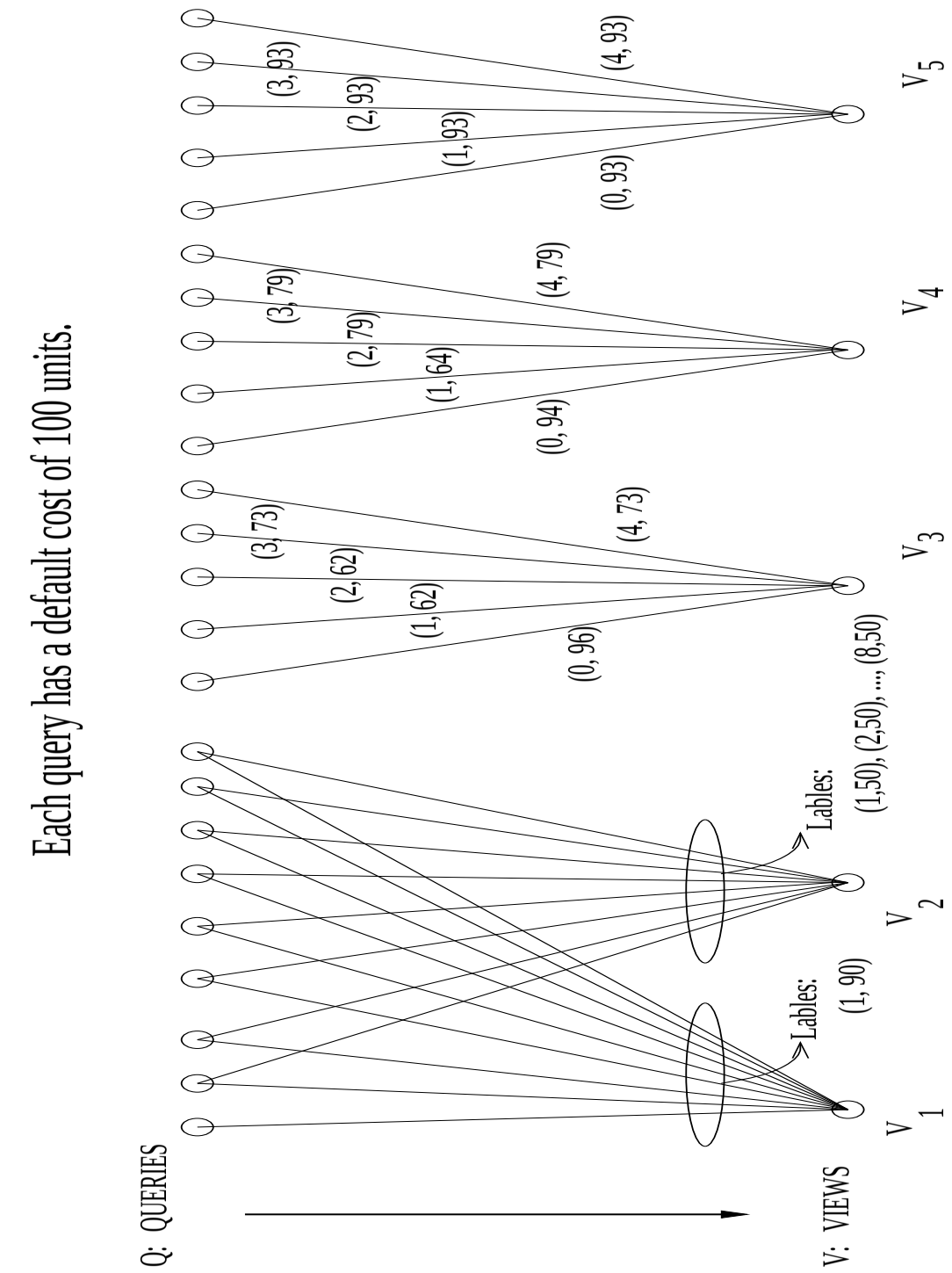

Figure 2.2: An OR view graph

to be supported are the nodes on the top. The queries have uniform query frequencies. We assume that the default cost of answering a query is 100 units, using the base relations (not shown in the figure). We have assigned a space cost of 1 unit to each of the indexes and views. Let the value of S be 7 units.

Let us see how the r-greedy algorithm works on the example for different values of r.

1. **1-greedy:** Initially, the absolute benefit of every index is zero. Absolute benefits of the views in order of their subscripts are viz. 0, 0, 4, 6, and 7. Hence, at the first stage the 1-greedy algorithm selects V_5. Except for the indexes of V_5, the benefits of all views and indexes with respect to $M = \{V_5\}$ remain the same as their absolute benefits. The benefit of each index of V_5 with respect to M becomes 7. Hence, the 1-greedy algorithm choses one by one all the indexes of V_5 in the later stages, followed by V_3 and V_4. Thus, the solution returned by 1-greedy is $\{V_5, I_{5,1}, I_{5,2}, I_{5,3}, I_{5,4}, V_3, V_4\}$, with an absolute benefit of 45.

2. **2-greedy:** In the first stage, the 2-greedy algorithm selects $C = \{V_1, I_{1,1}\}$ which has an absolute benefit of $10 \times 9 = 90$. The benefit of $\{V_2, I_{2,i}\}$ for any $i \leq 8$ with respect to C is 40 (i.e., 20 per unit space). Hence, $\{V_4, I_{4,1}\}$, whose benefit with respect to C is 42 (i.e., 21 per unit space), gets selected in the second stage. In the later stages, the other indexes of V_4 get selected one by one. Thus, the solution returned by 2-greedy is $\{V_1, I_{1,1}, V_4, I_{4,1}, I_{4,2}, I_{4,3}, I_{4,4}\}$ with an absolute benefit of 195.

3. **3-greedy:** As in the 2-greedy case, the first stage of the 3-greedy algorithm selects $C = \{V_1, I_{1,1}\}$, with the absolute benefit of 90. The second stage may select $\{V_3, I_{3,1}, I_{3,2}\}$, having a benefit of 80 with respect to C (i.e., 26.7 per unit space), as the benefit of V_2 with any two of its indexes is also 80 with respect to C (i.e., 26.7 per unit space). The structures selected in the later stages are $I_{3,3}$, and $I_{3,4}$. Thus, the solution returned by 3-greedy is $\{V_1, I_{1,1}, V_3, I_{3,1}, I_{3,2}, I_{3,3}, I_{3,4}\}$, which has an absolute benefit of 224.

4. **Optimal Solution:** It is not difficult to see that the optimal solution for the given example is $\{V_2, I_{2,1}, I_{2,2}, I_{2,3}, I_{2,4}, I_{2,5}, I_{2,6}\}$, having an absolute benefit of 300.

□

Theorem 2 *In the case when each structure occupies a unit of space, the r-greedy algorithm produces a solution M that uses at most $S + r - 1$ units of space. Also, the absolute benefit*

of M is at least $(1 - 1/e^{(r-1)/r})$ times the optimal benefit achievable using as much space as that used by M.

Proof: It is easy to see that the solution M produced by the r-greedy algorithm has at most $S + r - 1$ structures. Let $k = |M|$. Let the optimal solution containing k structures be O and the absolute benefit of O be B.

Consider a stage at which the r-greedy algorithm has already chosen a set G_l having l structures with incremental benefits $a_1, a_2, a_3, \ldots, a_l$. The absolute benefit of G_l is thus $\sum_{i=1}^{l} a_i$. Surely the absolute benefit of the set $O \cup G_l$ is at least B. Therefore, the benefit of the set O with respect to G_l, $B(O, G_l)$, is at least $B - \sum_{i=1}^{l} a_i$.

Without loss of generality, we can assume that the optimal set O doesn't contain any index whose corresponding view is not in O. Hence, if O contains m views, it can be split into m disjoint sets $O_1, O_2, \ldots, O_m$, such that each O_i consists of a view and its indexes in O. Then, by the monotonicity property of the benefit function for the sets O_i's, $B(O, G_l) \leq \sum_{i=1}^{m} B(O_i, G_l)$. As in Theorem 1, it is easy to show by contradiction that there exists an O_i such that $B(O_i, G_l)/|O_i| \geq B(O, G_l)/k$ (else $B(O, G_l) > \sum_{i=1}^{m} B(O_i, G_l)$).

Now, consider the best r-subset (a set having at most r structures) O_c of such an O_i. Its benefit per unit space with respect to G_l is at least $(\frac{r-1}{r})(\frac{k}{k-1})(B(O_i, G_l)/|O_i|)$, which happens when $|O_i| = k$, the benefit of the view in O_i is zero, and the rest of the benefit is equally divided among the $k - 1$ indexes. Let, $k' = (\frac{r-1}{r})(\frac{k}{k-1})$. As O_c (or its best subset) is also considered for selection at this stage of the r-greedy algorithm, the benefit per unit space with respect to G_l of the set C selected by the algorithm is at least $k'B(O_i, G_l)/|O_i| \geq k'(B(O, G_l)/k) \geq k'(B - \sum_{i=1}^{l} a_i)/k$. Note that O_c may contain some structures from G_l, but the argument still holds. Distributing the benefit of C over each of its structures equally (for the purpose of analysis), we get $a_{l+j} \geq k'(B - \sum_{i=1}^{l} a_i)/k$, for $0 < j \leq |C|$. As the above analysis is true for each set C selected at any stage, we have

$$B \leq \frac{k}{k'} a_j + \sum_{i=1}^{j-1} a_i \qquad \text{for } 0 < j \leq k.$$

Let $k'' = k/k'$. Multiplying the j^{th} equation by $(\frac{k''-1}{k''})^{k-j}$ and adding all the equations, we get $A/B \geq 1 - (\frac{k''-1}{k''})^k$, where $A = \sum_{i=1}^{k} a_i$ is the absolute benefit of M. This implies $A/B \geq 1 - (\frac{k''-1}{k''})^{k''k'} \geq 1 - 1/e^{k'} \geq 1 - 1/e^{(r-1)/r}$. ∎

We have come up with instances of the problem for which the r-greedy algorithm performs as bad as the worst case bound of $1 - 1/e^{(r-1)/r}$.

Inner-Level Greedy Algorithm

The inner-level greedy algorithm works in stages. At each stage, it selects a subset C, which consists of either a view and some of its indexes selected in a greedy manner, or a single index whose view has already been selected in one of the previous stages.

Each stage can be thought of as consisting of two phases. In the first phase, for each view v_i we construct a set IG_i which initially contains only the view. Then, one by one its indexes are added to IG_i in the order of their incremental benefits until the benefit per unit space of IG_i with respect to M, the set of structures selected till this stage, reaches its maximum. That IG_i having the maximum benefit per unit space with respect to M is chosen as C. In the second phase, an index whose benefit per unit space is the maximum with respect to M is selected. The benefit per unit space of the selected index is compared with that of C, and the better one is selected for addition to M. See Algorithm 4.

Algorithm 4 **Inner-Level Greedy Algorithm**

Given: G, a view graph with indexes, and S, the space constraint.
BEGIN
  $M = \phi$; /* M = Set of structures selected so far */
  while $(S(M) < S)$
    $C = \phi$; /* Set of structures to be selected */
    for each view v_i in M
      $IG = \{v_i\}$;
      /* IG = Set of v_i and some of its indexes selected in a greedy manner. */
      while $(S(IG) < S)$ /* Construct IG */
        Let I_{ic} be the index of v_i whose benefit per unit space w.r.t. $(M \cup IG)$ is maximum.
        $IG = IG \cup I_{ic}$;
      end while;
      if $(B(IG, M)/S(IG) > B(C, M)/|C|)$ **or** $C = \phi$
        $C = IG$;
    end for;
    for each index I_{ij} such that its view $v_i \in M$
      if $B(I_{ij}, M)/S(I_{ij}) > B(C, M)/S(C)$

$C = \{I_{ij}\}$;
end for;
$M = M \cup C$;
end while;
return M;
END. ◇

The running time of the inner-level greedy algorithm is $O(k^2m^2)$, where m is the total number of structures in the given OR view graph and k is the maximum number of structures that can fit in S units of space, which in the worst case is S.

EXAMPLE 2 We illustrate the working of the inner-level greedy algorithm for the example in Figure 2.2.

As the absolute benefit per unit space of V_2 with at most six of its indexes is 300/7, less than 43, the algorithm selects $\{V_1, I_{1,1}\}$, whose absolute benefit is 90 (i.e., 45 per unit space) in the first stage. In the next stage, the algorithm selects V_2 and six of its indexes with an "incremental" benefit of 240 (i.e., 34.3 per unit space). Thus, the solution returned by the inner-level greedy is $\{V_1, I_{1,1}, V_2, I_{2,1}, I_{2,2}, I_{2,3}, I_{2,4}, I_{2,5}, I_{2,6}\}$ with an absolute benefit of 330. Note that the size of the solution returned is 9 units, slightly more than the given space limit. The optimal solution using 9 units of space is V_2 with its eight indexes, having an optimal benefit of 400. □

Observation 2 *In an OR view graph with indexes and without updates, the benefit function B satisfies the monotonicity property for any M with respect to arbitrary sets of structures $O_1, O_2, \ldots, O_m$, where each O_i consists of a view and some of its indexes.*

Theorem 3 *For an OR view graph with indexes and a given quantity S, the inner-level greedy algorithm (Algorithm 4) produces a solution M that uses at most $2S$ units of space. Also, the absolute benefit of M is at least $(1 - 1/e^{0.63}) = 0.467$ of the optimal benefit achievable using as much space as that used by M, assuming that no structure occupies more than S units of space.*

Proof: It is easy to see that $S(M) \leq 2S$. Let $k = |M|$. Let the optimal solution be O, such that $S(O) = k$ and the absolute benefit of O be B.

Consider a stage at which the inner-level greedy algorithm has already chosen a set G_l occupying l units of space with incremental benefits $a_1, a_2, a_3 \ldots\ldots a_l$. The absolute benefit of

the set $O \cup G_l$ is at least B. Therefore, the benefit of the set O with respect to G_l, $B(O, G_l)$, is at least $B - \sum_{i=1}^{l} a_i$.

If O contains m views, it can be split into m disjoint sets $O_1, O_2, \ldots, O_m$, such that each O_i consists of a view V_i and its indexes in O. By the monotonicity property of the benefit function w.r.t. the sets $O_1, \ldots, O_m$, $B(O, G_l) \leq \sum_{i=1}^{m} B(O_i, G_l)$. Now, it is easy to show by contradiction that there exists at least one O_i such that $B(O_i, G_l)/S(O_i) \geq B(O, G_l)/k$ (else $B(O, G_l) > \sum_{i=1}^{m} B(O_i, G_l)$).

In this paragraph, we show that the benefit per unit space of the set C, selected by the inner-level greedy algorithm at this stage, is at least 0.63 times $B(O_i, G_l)/S(O_i)$. Without loss of generality, we assume that the view V_i in O_i has not been selected.[6] Consider the greedy solution G of the indexes of V_i of space $S(O_i) - S(V_i)$, when $G_l \cup V_i$ has already been selected. The benefit of G is at least 63% of the optimal, from the result of Theorem 1. Hence,

$$B(G, G_l \cup \{V_i\}) \geq 0.63 B(O_i - \{V_i\}, G_l \cup \{V_i\}),$$

as $O_i - \{V_i\}$ is also a solution (possibly non-optimal). Now,

$$\begin{aligned} B(G \cup \{V_i\}, G_l) &= B(V_i, G_l) + B(G, G_l \cup \{V_i\}) \\ &\geq B(V_i, G_l) + 0.63 B(O_i - \{V_i\}, G_l \cup \{V_i\}) \\ &\geq 0.63 B(O_i, G_l) \end{aligned}$$

As the inner-level greedy algorithm, while selecting indexes greedily, stops when the benefit per unit space of C becomes maximum, the benefit per unit space of C is at least that of $G \cup \{V_i\}$. Therefore,

$$\begin{aligned} B(C, G_l)/|C| &\geq B(G \cup \{V_i\}, G_l)/S(O_i) \\ &\geq 0.63 B(O_i, G_l)/S(O_i) \\ &\geq 0.63 B(O, G_l)/k \\ &\geq 0.63 (B - \sum_{i=1}^{l} a_i)/k. \end{aligned}$$

Let $k' = 0.63$. Distributing the benefit of C over each of its unit spaces equally (for the purposes of analysis), we get $a_{l+j} \geq k'(B - \sum_{i=1}^{l} a_i)/k$, for $0 \leq j < S(C)$. As the above analysis is true for each set C selected at any stage, we have

[6] If the view $V_i \in O_i$ has already been selected, then C is at least as good as O_i's best index not yet selected. In that case, the benefit per unit space of C is obviously at least $B(O_i, G_l)/S(O_i)$.

$$B \leq \frac{k}{k'} a_j + \sum_{i=1}^{j-1} a_i \qquad \text{for } 0 < j \leq k.$$

Let $k'' = k/k'$. Multiplying the j^{th} equation by $(\frac{k''-1}{k''})^{k-j}$ and adding all the equations, we get $A/B \geq 1-(\frac{k''-1}{k''})^k \geq 1-(\frac{k''-1}{k''})^{k''k'} \geq 1-1/e^{0.63}$, where $A = \sum_{i=1}^{k} a_i$ is the absolute benefit of M. ∎

2.4 AND View Graph

In this section, we consider another special case of the view-selection problem in AND-OR view graphs. Here, we assume that each AND arc binds all the outgoing edges from a node. This case depicts the simplied scenario where each view has a unique way of being computed. We call such a graph G an *AND view graph*, where a node is computed from *all* of its children. As before, each AND arc has an operator and a cost associated with it. An AND view graph for a set of queries is just a "merging" of the expression AND-DAGs of the queries. The proofs of the theorems in this section are similar to that of the corresponding theorems in Section 2.3.

2.4.1 Motivation

The general view-selection problem can be approximated by this simplified problem of selecting views in an AND view graph. Given a set of queries supported at the warehouse, instead of constructing an AND-OR view graph as in Section 2.2.2, we could run a multiple query optimizer [Sel88, CM82] to generate a global plan, which is essentially an AND view graph for the queries. Such a global plan takes advantage of the common subexpressions among the queries. Figure 2.4.1 shows an example of an AND view graph, a global plan, for the queries $R \bowtie S \bowtie T$ and $R \bowtie S \bowtie U$.

2.4.2 View Selection in an AND View Graph

In this subsection, we show that the greedy algorithm can also be applied to solve the view-selection problem in AND view graphs without maintenance costs. In the later subsection, we extend it to a special case of AND view graphs with maintenance costs.

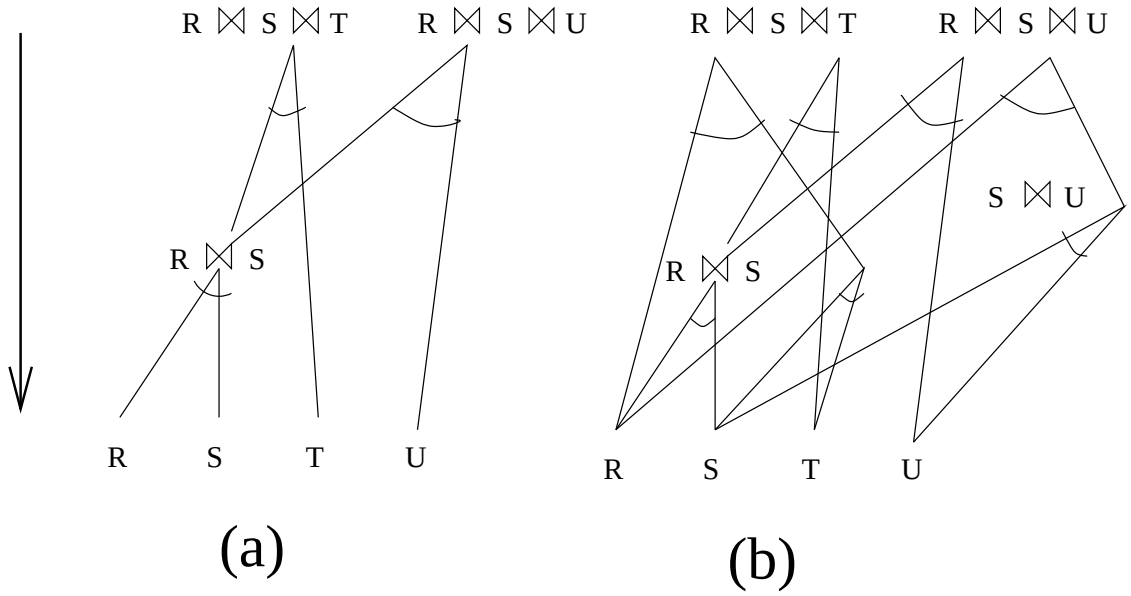

Figure 2.3: (a) An AND view graph, (b) An AND-OR view graph, for the queries $R \bowtie S \bowtie T$ and $R \bowtie S \bowtie U$

Problem: Given an AND view graph G and a quantity S, find a set of views M that minimizes the quantity $\tau(G, M)$, under the constraint that the total space occupied by the views in M is at most S. Assume that there are no updates.

Observation 3 *In an AND view graph without updates, the benefit function B satisfies the monotonicity property for any M with respect to arbitrary sets of views $O_1, O_2, \ldots, O_m$.*

Using the above observation, the proof of the following theorem is same as that of Theorem 1.

Theorem 4 *For an AND view graph G without updates and a given quantity S, the greedy algorithm produces a solution M that uses at most $S+r$ units of space, where r is the size of the largest view in G. Also, the absolute benefit of M is at least $(1-1/e)$ times the optimal benefit achievable using as much space as that used by M.* ∎

2.4.3 Incorporating Maintenance Costs

Unfortunately, the benefit function may not satisfy the monotonicity property when there are maintenance costs. To illustrate the nonmonotonicity of the benefit function, consider

a view C_1 that *helps* in maintaining another view C_2. Then, the benefit of $C_1 \cup C_2$ might be more than the sum of their benefits individually.

We show that when the update frequency at any node/view is less than its query frequency, i.e., when the number of times a view is updated (number of batch updates) is less than than the number of times it is queried, the benefit function does satisfy the monotonicity property with respect to single views. Thus, for this special case of AND view graph, the solution returned by the greedy algorithm is guaranteed to have a benefit of at least 63% of the optimum benefit.

Lemma 1 *In an AND view graph, $B(v, \phi) \geq B(v, M)$ for any view v and a set of views M, if the update frequency g_x at any view x is less than its query frequency f_x.*

Proof: Let A be the set of ancestors of v, including v, in the AND view graph G. Let $M_A = M \cap A$. Let A_D be the set of those ancestors of v which do not have any descendants in the set M_A. For any $x \in A$, we have $Q(x, \phi) - Q(x, v) = Q(v, \phi)$. Therefore,

$$\begin{aligned} B(v, \phi) &= \sum_{x \in A} f_x(Q(x, \phi) - Q(x, v)) - g_v UC(v, v) \\ &= \sum_{x \in A} f_x Q(v, \phi) - g_v UC(v, v) \qquad \text{as } x \in A. \end{aligned}$$

Now consider $B(v, M)$. When M has already been materialized, v reduces then query costs of only the nodes in A_D. Also, materialization of v also helps in reducing the maintenance costs of nodes in M_A. Therefore,

$$\begin{aligned} B(v, M) &= \sum_{x \in A_D} f_x(Q(x, M) - Q(x, M \cup \{v\})) - g_v UC(v, M \cup \{v\}) \\ &\quad + \sum_{x \in M_A} g_x(UC(x, M) - UC(x, M \cup \{v\})) \end{aligned}$$

Now $\quad Q(v, \phi) \geq Q(v, M) \geq UC(x, M) - UC(x, M \cup \{v\}) \qquad$ for any $x \in M_A$,

and $\quad Q(v, M) = Q(x, M) - Q(x, M \cup \{v\}) \qquad$ for any $x \in A_D$.

$$\begin{aligned} \text{Thus,} \quad B(v, M) &\leq \sum_{x \in A_D} f_x Q(v, M) - g_v UC(v, M) + \sum_{x \in M_A} g_x Q(v, \phi) \\ &\leq \sum_{x \in A_D} f_x Q(v, M) - g_v UC(v, M) + \sum_{x \in M_A} f_x Q(v, \phi) \qquad \text{as } g_x \leq f_x. \end{aligned}$$

$$
\begin{aligned}
B(v,\phi) - B(v,M) &\geq \sum_{x \in A} f_x Q(v,\phi) - g_v UC(v,v) \\
&\quad - \sum_{x \in A_D} f_x Q(v,M) + g_v UC(v,M) - \sum_{x \in M_A} f_x Q(v,\phi) \\
&\geq \sum_{x \in A} f_x Q(v,\phi) - \sum_{x \in A_D} f_x Q(v,M) - \sum_{x \in M_A} f_x Q(v,\phi) \\
&\quad - g_v(UC(v,v) - UC(v,M)) \\
&\geq \sum_{x \in A_D} f_x Q(v,\phi) + \sum_{x \in M_A} f_x Q(v,\phi) - \sum_{x \in A_D} f_x Q(v,M) \\
&\quad - \sum_{x \in M_A} f_x Q(v,\phi) - g_v(UC(v,v) - UC(v,M)), \text{ as } A_D \cap M_A = \phi. \\
&\geq \sum_{x \in A_D} f_x (Q(v,\phi) - Q(v,M)) - g_v(UC(v,v) - UC(v,M)) \\
&\geq f_v(Q(v,\phi) - Q(v,M)) - f_v(UC(v,v) - UC(v,M))
\end{aligned}
$$

Now, let C_{MD} be the cost of materialization all descendants of v that are in M. Then,

$$Q(v,\phi) - Q(v,M) = C_{MD} \geq UC(v,v) - UC(v,M).$$

Therefore, we get $B(v,\phi) - B(v,M) \geq 0$. ∎

Lemma 2 *In an AND view graph, the benefit function B satisfies the monotonicity property for any M with respect to sets consisting of single views, if the update frequency g_v at any view v is less than its query frequency f_v.*

Proof: Consider views $V_1, V_2, \ldots, V_m$ and a set of views M. Also, for simplicity, let $M_i = M \cup \{V_1, V_2, \ldots, V_i\}$ for $1 \leq i \leq m$. Note that Lemma 1 implies that $B(v,L) \geq B(v, L \cup M)$ for any view v and sets of views L and M. Therefore, we have

$$B(V_i, M) \geq B(V_i, M_i) \text{ for } 1 \leq i \leq m.$$

Also, by definition of the benefit function, we have

$$B(\{V_1, V_2, \ldots, V_m\}, M) = B(\{V_1\}, M) + B(\{V_2\}, M_1) + B(\{V_3\}, M_3) + \ldots + B(\{V_m\}, M_m).$$

Using the above two equations, we get

$$B(\{V_1, V_2, \ldots, V_m\}, M) \leq B(\{V_1\}, M) + B(\{V_2\}, M) + B(\{V_3\}, M) + \ldots + B(\{V_m\}, M),$$

which proves the monotonicity of the benefit function for an arbitrary M with respect to arbitrary views $V_1, V_2, \ldots, V_m$. ∎

Using the above Lemma 2, we can prove the following theorem. The proof is similar to that of Theorem 1.

Theorem 5 *Consider an AND view graph G with updates, where for any view the update frequency is less than its query frequency. For such a graph G, the greedy algorithm produces a solution M whose absolute benefit is at least $(1 - 1/e)$ times the optimal benefit achievable using as much space as that used by M.* ∎

2.4.4 AND View Graph With Indexes

As in the case of OR view graphs, we generalize the view-selection problem in AND view graphs by introducing indexes for each node/view. As in the original AND view graph, a node can be computed from all of its children, but in the presence of indexes the cost of computation depends upon the indexes being used to execute the operation.

We need to introduce a slightly different cost model for the AND view graphs with indexes. In an AND view graph with indexes, instead of associating costs with the arcs, we associate a label (i, t_i) with each edge from u to v. The cost t_i $(i > 0)$[7] can be thought of as the cost incurred in accessing the relation (as many times as required to compute u) at v using its i^{th} index. In addition, we have a k-ary monotonically increasing cost *function* associated with every arc that binds k edges.

Consider a node u that has k outgoing edges to nodes $v_1, v_2, \ldots, v_k$ and let the k-ary cost function associated with the arc binding all these outgoing edges be f. Then, the cost of computing u from all its children $v_1, v_2, \ldots, v_k$ using their $i_1, i_2, \ldots, i_k^{th}$ indexes respectively is $f(t_{i_1}, t_{i_2}, \ldots, t_{i_k})$, where there is an edge from u to v_j, for $0 < j \leq k$, with a label (i_j, t_{i_j}). We omit the proofs of the theorems in this subsection, as they are similar to the proofs in Section 2.3.3.

Problem: Given a quantity S and an AND view graph G with indexes, find a set of structures M that minimizes the quantity $\tau(G, M)$, under the constraint that the total space occupied by the structures (views and indexes) in M is at most S. Assume that there are no updates.

[7] When $i = 0$, t_0 is the cost in accessing v without any of its indexes.

Observation 4 *In an AND view graph with indexes and without updates, the benefit function B satisfies the monotonicity property for any M with respect to disjoint sets of structures $O_1, \ldots, O_m$, where each O_i consists of a view and some of its indexes.*

Theorem 6 *For an AND graph, when each structure occupies unit space, the r-greedy algorithm produces a solution M that uses at most $S + r - 1$ units of space. Also, the absolute benefit of M is at least $(1 - 1/e^{(r-1)/r})$ times the optimal benefit achievable using as much space as that used by M.* ∎

Theorem 7 *For an AND graph, the inner-level greedy algorithm produces a solution M that uses at most $2S$ units of space. Also, the absolute benefit of M is at least $(1-1/e^{0.63}) = 0.467$ of the optimal benefit achievable using as much space as that used by M, assuming that no structure occupies more than S units of space.* ∎

2.5 View Selection in AND-OR View Graphs

In this section, we try to generalize our results developed in the previous sections to the view-selection problem in general AND-OR view graphs. Unfortunately, we couldn't devise a polynomial time algorithm for the general AND-OR view graphs that delivers a competitive solution. Instead, we present here an AO-greedy algorithm (a modification of the greedy heuristic) that could take exponential time in the worst case, but has a performance guarantee of 63%. We show that the AO-greedy algorithm developed here runs in polynomial time when the view graph is an OR view graph. We also present a multi-level greedy algorithm which is a generalization of the inner-level greedy algrorithm (Algorithm 4).

We give a different formulation of the view-selection problem in AND-OR graphs, for the sake of simplifying the presentation. First, we define the notion of query-view graphs.

Definition 5 (**Query-View Graph**) A query-view graph G is a bipartite graph $(Q \cup \zeta, E)$, where Q is the set of queries to be supported at the warehouse and ζ is a subset of the power set of V, the set of views. An edge (q, σ) is in E iff the query q can be answered using the views in the set σ, and the cost associated with the edge is the cost incurred in answering q using σ.[8] There is also a frequency f_q associated with each query $q \in Q$. We

[8] A query-view graph can be looked upon as an OR graph, as a query $q \in Q$ can be computed by any of the *set* of views σ where $(q, \sigma) \in E$.

assume that there is a set $\rho \in \zeta$ (the set of base tables) such that $(q, \rho) \in E$ for all $q \in Q$. □

Note that an arbitrary AND-OR view graph can be converted into an equivalent query-view graph. We now formulate the view-selection problem in a query-view graph.

Problem (View Selection in Query-View Graphs): Given a quantity S and a query-view graph $G = (\zeta \cup Q, E)$, select a set of views $M \subseteq V$ that minimizes the total query response time,[9] under the constraint that the total space occupied by the views in M is at most S.

2.5.1 AO-Greedy Algorithm for Query-View Graphs

We define an *intersection graph* F_ζ of ζ as a graph having ζ and D as its set of vertices and edges respectively, such that an edge $(\alpha, \beta) \in D$ if and only if the sets of views α and β intersect.

The AO-greedy algorithm works in stages as follows. At each stage, the algorithm picks a connected subgraph H of F_ζ whose corresponding set of views V_H (union of the sets of views corresponding to the vertices of H) offers the maximum benefit per unit space at that stage. The set of views V_H is then added to M, the set of views already selected in previous stages. The algorithm halts and returns M when the space occupied by M exceeds S.

To improve the running time, after the selection at each stage, we can change the set ζ by removing the selected views V_H from each element (a set of views) in ζ. Graph F_ζ, for the next stage, is then reconstructed from the new ζ.

Lemma 3 *An optimal solution O of the view-selection problem in query-view graph $G = (\zeta \cup Q, E)$ can be partitioned into (*disjoint*) sets of views $O_1, O_2, \ldots, O_m$, such that each O_i corresponds to a connected subgraph in F_ζ, as defined above, and $B(O, M) \leq \sum_{i=1}^{m} B(O_i, M)$.*

Proof: We start by showing that there exists a subset Γ of ζ such that $O = \bigcup_{\sigma \in \Gamma} \sigma$, if O is an optimal set.

Let Γ be a maximal subset of ζ such that for every $\sigma \in \Gamma$, $\sigma \subseteq O$. Consider an arbitrary view $v \in O$. As O is optimal, v helps answer some query, else it could be removed from O. Thus, for some $\sigma_v \subseteq O$, $v \in \sigma_v \in \zeta$, implying that $\sigma_v \in \Gamma$. Thus, $v \in O$ implies $v \in \bigcup_{\sigma \in \Gamma} \sigma$

[9] Though we ignore maintenance costs, it can be incorporated by adding additional nodes in ζ.

for an arbitrary v. Therefore, $O \subseteq \bigcup_{\sigma \in \Gamma} \sigma$. Also, by definition of Γ, it is obvious that $\bigcup_{\sigma \in \Gamma} \sigma \subseteq O$. Hence, $O = \bigcup_{\sigma \in \Gamma} \sigma$.

Now, consider the intersection graph F_Γ of Γ. The intersection graph F_Γ is only an induced subgraph of the intersection graph F_ζ on the nodes in Γ. Consider the connected components of F_Γ which partition the set of nodes Γ into $\Gamma_1, \Gamma_2, \ldots, \Gamma_m$. Let $O_i = \bigcup_{\sigma \in \Gamma_i} \sigma$. Now O_i's also form a partition of O, because there are no edges in F_ζ between the nodes of Γ_i and Γ_j for any i and j. It is easy to see that for the above O_i's, $B(O, M) \leq \sum_{i=1}^{m} B(O_i, M)$, because exactly one materialized node in ζ is used to answer any query q in Q. ∎

Theorem 8 *For a query-view graph without updates and a quantity S, the AO-greedy algorithm produces a solution M that uses at most $2S$ units of space. Also, the absolute benefit of M is at least $(1 - 1/e)$ times the optimal benefit achievable using as much space as that used by M.* ∎

Proof: It is easy to see that the space used by the greedy algorithm solution, $S(M)$, is at most $2S$ units. Let $k = S(M)$. Let the optimal solution using k units of space be O and the absolute benefit of O be B.

Consider a stage at which the greedy algorithm has already chosen a set G_l occupying l units of space with incremental benefits $a_1, a_2, \ldots, a_l$. Thus, the absolute benefit of G_l is $\sum_{i=1}^{l} a_i$. Surely the absolute benefit of the set $O \cup G_l$ is at least B. Therefore, the benefit of the set O with respect to $G_l, B(O, G_l)$, is at least $B - \sum_{i=1}^{l} a_i$. Due to Lemma 3, the optimal set O can be partitioned into disjoint sets $O_1, O_2, \ldots, O_m$, such that $B(O, G_l) \leq \sum_{i=1}^{m} B(O_i, G_l)$. Now, as O_i's are disjoint, it is easy to show by contradiction that there exists at least one O_i such that $B(O_i, G_l)/S(O_i) \geq B(O, G_l)/k$ (else $B(O, G_l) > \sum_{i=1}^{m} B(O_i, G_l)$).

The benefit per unit space with respect to G_l of the set C selected at this stage by the AO-Greedy algorithm is at least that of O_i, which is at least $B(O, G_l)/k = (B - \sum_{i=1}^{l} a_i)/k$, as shown above. Distributing the benefit of C over each of its unit spaces equally (for the purpose of analysis), we get $a_{l+j} \geq (B - \sum_{i=1}^{l} a_i)/k$, for $0 < j \leq S(C)$. As the above analysis is true for each set C selected at any stage, we have

$$B \leq k a_j + \sum_{i=1}^{j-1} a_i \qquad \text{for } 0 < j \leq k.$$

Multiplying the j^{th} equation by $(\frac{k-1}{k})^{k-j}$ and adding all the equations, we get

$A/B \geq 1 - (\frac{k-1}{k})^k \geq 1 - 1/e$, where $A = \sum_{i=1}^{k} a_i$ is the absolute benefit of M. ∎

The equivalent query-view graph $G = (\zeta \cup Q, E)$ of an OR view graph is such that each element $\sigma \in \zeta$ consists of exactly one view and hence F_ζ has zero edges. For such a graph G, the AO-greedy algorithm behaves exactly as the greedy algorithm (Algorithm 1), taking polynomial time for OR view graphs.

2.5.2 Multi-Level Greedy Algorithm

In this section, we generalize the inner-level greedy algorithm (Algorithm 4) to multiple levels of greedy selection in query-view graphs. We try to modify the AO-greedy algorithm for query-view graphs in an attempt to improve its running time at the expense of its performance guarantee.

Consider a query-view graph $G = (Q \cup \zeta, E)$ and the intersection graph F_ζ of ζ. Let F_ζ have $l > 1$ connected components and the let $G_1, G_2, \ldots, G_l$ where $G_i = (Q \cup \zeta_i, E_i)$ be the corresponding query-view subgraphs of G. The multi-level inner greedy algorithm works in stages. At each stage, it searches for a set of views W_i in each G_i, such that the benefit per unit space of W_i is maximum. Each set W_i is computed by using the recursive function `InnerGreedy` on ζ_i. Among all W_i's, the set W_i that has the maximum benefit per unit space is added to the solution M being maintained. The solution M is returned when the total space constraint has been consumed.

The recursive function `InnerGreedy` works as follows. Let the input be the set of nodes Γ. Let us assume that there is a view v where $v \in \sigma$ for each node σ in Γ. If no such v exists, then the InnerGreedy function does an exhaustive search (or run the AO-greedy algorithm) and return a set of views that has the optimal benefit per unit space. If v exists, let $\Gamma_1, \Gamma_2, \ldots, \Gamma_m$ be the sets corresponding to the connected components of the resulting intersection graph. The set of views U, that has to be returned by the InnerGreedy function, is selected in the following greedy manner. Initially the set U contains only v. Then, at each stage, we search recursively in each Γ_i for a set of views J_i that has the maximum benefit per unit space. The set J_i that has the maximum benefit per unit space is added to the set U being maintained. We continue adding views to U until the total benefit per unit space of U cannot be further improved. At that point, the set U is returned.

The multi-level (r-level) greedy algorithm and the InnerGreedy function is shown below as Algorithm 5.

Algorithm 5 **Multi-level (r-level) Greedy Algorithm**

Given: A query-view graph $G = (Q \cup \zeta, E)$ and the space constraint S.

BEGIN

$M = \phi$; /* M = set of structures selected so far. */

while $(S(M) < S)$

Let $G_1, G_2, \ldots, G_l$ be the connected components of the intersection graph F_ζ and let $\zeta_1, \ldots, \zeta_m$ be the corresponding subsets of ζ.

For each $i \leq m$, W_i = InnerGreedy(r, ζ_i, M);

Let W be the W_i that has the maximum benefit per unit space;

$M = M \cup W$;

Reduce ζ by removing the views in W from each of its elements;

end while;

return M;

END.

Function **InnerGreedy**(r, Γ, M) /* Returns a set of views U that has the best benefit per unit space. The main inputs G and S are globally defined. */

BEGIN

If $r = 0$, pick U by doing exhaustive search;

Let F_Γ be the intersection graph of Γ.

Let v be such that for all $\sigma \in \Gamma$, $v \in \sigma$.

If no such v exists, pick U by doing exhaustive search.

Let $\Gamma_1, \ldots, \Gamma_m$ be the corresponding subsets of Γ

obtained after removing the view v from each element.

$P = 0$; $U = \{v\}$;

while $(S(U) < S)$

For each i, let J_i = InnerGreedy$(r-1, \Gamma_i, (M \cup U))$;

Let J be the J_i with the maximum benefit per unit space.

if $(B(U \cup J, M)/S(U \cup J) \leq B(U, M)/B(U))$

return U;

$U = U \cup J$;

end while;

return U.

END. ◇

Lemma 4 *The InnerGreedy function with the first parameter value equal to r delivers a solution U whose benefit per unit space is at least $g(r)$ of the optimal benefit per unit space achievable. The function $g(r)$ is defined recursively as $g(r) = 1 - 1/e^{g(r-1)}$, and $g(0) = 1$.*

Proof: We prove this lemma by induction. The base case for $r = 0$ is obvious. Assume that the value of the first parameter to the InnerGreedy function is r.

Without loss of generality, we assume that the input parameter M to InnerGreedy is ϕ. Consider a set of views O' in Γ that has the optimal benefit per unit space. It is obvious that O' contains the view v that is in all elements of Γ. Let $O' - \{v\} = O$.

Consider a stage at which the InnerGreedy algorithm has already chosen a set G_l (apart from v) occupying l units of space with incremental benefits $a_1, a_2, a_3.....a_l$ with respect to v. Let $G_l^v = G_l \cup \{v\}$, also the value of U (see Algorithm 5) at this stage. The benefit of the set $O \cup G_l$ with respect to $\{v\}$ is at least that of O with respect to v, i.e., $B(O \cup G_l, \{v\}) \geq B(O, \{v\})$. Also, $B(O \cup G_l, \{v\}) = B(O, G_l^v) + \sum_{i=1}^{l} a_i$. Therefore, the benefit of the set O with respect to G_l^v, $B(O, G_l^v)$, is at least $B(O, \{v\}) - \sum_{i=1}^{l} a_i$.

As Γ consists of m connected components $\Gamma_1, \ldots, \Gamma_m$ after deleting v, the set O can be split into m disjoint sets $O_1, O_2, \ldots, O_m$, such that each O_i belongs to Γ_i. By the monotonicity property of the benefit function w.r.t. the sets $O_1, \ldots, O_m$, $B(O, G_l^v) \leq \sum_{i=1}^{m} B(O_i, G_l^v)$. Now, it is easy to show by contradiction that there exists at least one O_i such that $B(O_i, G_l^v)/S(O_i) \geq B(O, G_l^v)/S(O)$ (else $B(O, G_l^v) > \sum_{i=1}^{m} B(O_i, G_l^v)$).

Now, by inductive hypothesis, the benefit per unit space of the set J, selected by the InnerGreedy algorithm at this stage, is at least $g(r-1)$ times $B(O_i, G_l^v)/S(O_i)$, as the InnerGreedy function when called with the first parameter equal to $r-1$ returns a solution that is within $g(r-1)$ of the optimal. Thus, $B(J, G_l^v) \geq g(r-1)B(O_i, G_l^v)/S(O_i) \geq g(r-1)B(O, G_l^v)/S(O) \geq g(r-1)(B(O, \{v\}) - \sum_{i=1}^{l} a_i)/S(O)$.

Let us assume, that the InnerGreedy Algorithm continues to select views (apart from v) till it has exhausted S(O) space, and the final set of views selected is G. Using techniques similar to the proof of Theorem 3, it is easy to show that $B(G, \{v\}) = (1 - 1/e^{g(r-1)})B(O, \{v\}) = g(r)B(O, \{v\})$, as $k' = g(r-1)$ here. Thus, we have

$$\begin{aligned} B(G \cup \{v\}, \phi) &= B(v, \phi) + B(G, \{v\}) \\ &\geq B(v, \phi) + g(r)B(O, \{v\}) \\ &\geq g(r)B(O', \phi) \end{aligned}$$

Thus, the benefit per unit space of $G \cup \{v\}$ is at least $g(r)B(O', \phi)/S(O')$, as G and O occupy the same space. But, the InnerGreedy algorithm actually stops when the benefit of U per unit space reaches the maximum. Therefore, the benefit per unit space of U is at least equal to the benefit per unit space of G, which is $g(r)$ times the optimal. ∎

Theorem 9 *For a query-view graph G and a given quantity S, the r-level greedy algorithm delivers a solution M that uses at most $2S$ units of space. Also, the benefit of M is at least $g(r+1)$ times the optimal benefit achievable using as much space as that used by M, assuming that no view occupies more than S units of space.*

Proof: It is easy to see that $S(M) \leq 2S$. Let $k = |M|$. Let the optimal solution be O, such that $S(O) = k$ and the absolute benefit of O be B.

Consider a stage at which the multi-level greedy algorithm has already chosen a set G_l occupying l units of space with incremental benefits $a_1, a_2, a_3.....a_l$. The absolute benefit of the set $O \cup G_l$ is at least B. Therefore, the benefit of the set O with respect to G_l, $B(O, G_l)$, is at least $B - \sum_{i=1}^{l} a_i$.

As ζ consists of m connected components $\zeta_1, \ldots, \zeta_m$, the set O can be split into m disjoint sets $O_1, O_2, \ldots, O_m$, such that each O_i belongs to ζ_i. By the monotonicity property of the benefit function w.r.t. the sets $O_1, \ldots, O_m$, $B(O, G_l) \leq \sum_{i=1}^{m} B(O_i, G_l)$. Now, it is easy to show by contradiction that there exists at least one O_i such that $B(O_i, G_l)/S(O_i) \geq B(O, G_l)/k$ (else $B(O, G_l) > \sum_{i=1}^{m} B(O_i, G_l)$).

Using Lemma 4, we know that the benefit per unit space of the set W_i is at least $g(r)$ times $B(O_i, G_l)/S(O_i)$. Thus, the benefit per unit space of the set W, selected by the inner-level greedy algorithm at this stage, is at least $g(r)$ times $B(O_i, G_l)/S(O_i) \geq B(O, G_l)/k \geq (B - \sum_{i=1}^{l} a_i)/k$. Let $k' = g(r)$. Distributing the benefit of W over each of its unit spaces equally (for the purposes of analysis), we get $a_{l+j} \geq k'(B - \sum_{i=1}^{l} a_i)/k$, for $0 \leq j < S(W)$. As the above analysis is true for each set W selected at any stage, we have

$$B \leq \frac{k}{k'} a_j + \sum_{i=1}^{j-1} a_i \qquad \text{for } \ 0 < j \leq k.$$

Let $k'' = k/k'$. Multiplying the j^{th} equation by $(\frac{k''-1}{k''})^{k-j}$ and adding all the equations, we get $A/B \geq 1 - (\frac{k''-1}{k''})^k \geq 1 - (\frac{k''-1}{k''})^{k''k'} \geq 1 - 1/e^{g(r)} = g(r+1)$, where $A = \sum_{i=1}^{k} a_i$ is the absolute benefit of M. ∎

For a given instance one could estimate the value of r such that at the r^{th} level the graphs F_i are small constant-size graphs. The last level would then take only a constant

amount of time. The r-level greedy algorithm takes $O((kn)^{2r})$ time, excluding the time taken at the final level, where k is the maximum number of views that can fit in S units of space. Also, the values of the function $g(r)$ for increasing values of r are 1, 0.63, 0.46, 0.37, 0.31, 0.26, 0.23 and so on.

The equivalent query-view graph $G = (\zeta \cup Q, E)$ of an OR view graph with indexes is such that each element $\sigma \in \zeta$ consists of a single view and one of its indexes. For such a query-view graph G, the 1-level greedy algorithm behaves exactly the same as the inner-level greedy algorithm (Algorithm 4)on OR view graphs with indexes. The 2-level greedy algorithm is very well suited for the case of OR graph with index, where even the indexes are indexed. So, $r - level$ greedy algorithm is well suited for OR or AND view graphs with $r - level$ indexing schemes.

2.6 Concluding Remarks

In this chapter, we have developed a theoretical framework for the general problem of selection of views in a data warehouse. We have presented competitive polynomial-time heuristics that deliver a solution within a 0.63 factor of the optimal for some important special cases of the general problem in a data warehouse viz. OR view graphs and AND view graphs. For both these special cases, we have extended the results to graphs with indexes associated with each view. Finally, we extended our heuristic to the most general case of AND-OR view graphs.

In this chapter, we looked at various cases of the view-selection problem under the disk space constraint. In the next chapter, we address the view-selection problem in a data warehouse when the constraint is that of the total maintenance cost of the materialized views.

Chapter 3

Selection of Views Under a Maintenance Cost Constraint

3.1 Introduction

As described in the previous chapters, a data warehouse stores materialized views derived from one or more sources for the purpose of efficiently implementing decision-support or OLAP queries. The selection of which views to materialize is one of the most important decisions in the design of a data warehouse. Chapter 2 presented a theoretical formulation of the general "view-selection" problem in a data warehouse. Given some resource constraint and a query load to be supported at the warehouse, the *view-selection* problem, as defined in Chapter 2, is to select a set of derived views to materialize, under a given resource constraint, that will minimize the sum of total query response time and maintenance time of the selected views. We have presented near-optimal polynomial-time greedy algorithms for some special cases of the general problem where the resource constraint is disk space.

In this chapter, we consider the view-selection problem of selecting views to materialize in order to optimize the total query response time, under the constraint that the selected set of views incur less than a given amount of total maintenance time. Hereafter, we will refer to this problem as the *maintenance-cost view-selection* problem. The maintenance-cost view-selection problem is much more difficult than the view-selection problem with a disk-space constraint, because the maintenance cost of a view v depends on the set of other materialized views. For the special case of "OR view graphs," we present a competitive greedy algorithm that provably delivers a near-optimal solution. We prove that the query

benefit of the solution delivered by the proposed greedy heuristic is within 63% of that of the optimal solution. The OR view graphs, which are view graphs where exactly one view is used to derive another view, arise in many important practical applications. A very important application is that of OLAP warehouses called data cubes, where the candidate views for precomputation (materialization) form an "OR boolean lattice." For the general maintenance-cost view-selection problem that arises in a data warehouse, i.e., for the general case of AND-OR view graphs, we present an A^* heuristic that delivers an optimal solution. We implemented our algorithms and a performance study of the algorithms shows that the proposed greedy algorithm for OR view graphs almost always delivers an optimal solution.

The rest of the chapter is organized as follows. The rest of this section gives a brief summary of the related work. In Section 3.2, we present the motivation for the maintenance-cost view-selection problem and the main contributions of this chapter. We define the maintenance-cost view-selection problem formally in Section 3.3. In Section 3.4, we present an approximation greedy algorithm for the maintenance-cost view-selection problem in OR view graphs. Section 6 presents an A^* heuristic that delivers an optimal set of views for the maintenance-cost view-selection problem in general AND-OR view graphs. We present our experimental results in Section 3.6. Finally, we give some concluding remarks in Section 3.7.

3.1.1 Related Work

Recently, there has been a lot of interest on the problem of selecting views to materialize in a data warehouse. Harinarayan, Rajaraman and Ullman [HRU96] provide algorithms to select views to materialize in order to minimize the total query response time, for the case of data cubes. The view graphs that arise in data cubes are special cases of OR view graphs. In the previous chapter, we developed a theoretical formulation of the general view-selection problem in a data warehouse and generalized the results in [HRU96] to (i) OR view graphs, (ii) AND view graphs, (iii) OR view graphs with indexes, (iv) AND view graphs with indexes, and some other special cases of AND-OR view graphs. All of the above mentioned work presents approximation algorithms to select a set of structures that minimizes the total query response time under a given *space* constraint; the constraint represents the maximum amount of disk space that can be used to store the materialized views.

Ours is the first work to address the problem of selecting views to materialize in a data warehouse under the constraint of a given amount of total view maintenance time. We present a nonexhaustive approximation algorithm that provably returns a near-optimal

solution for the special case of OR view graphs.

3.2 Motivation and Contributions

In this section, we briefly present the motivation behind the maintenance-cost view-selection problem and the contributions made by the thesis. Most of the previous work done ([HRU96], Chapter 2) on designing approximation algorithms for the view-selection problem suffers from one drawback. The designed algorithms apply only to the case of a disk-space constraint.

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

In this chapter, we consider the maintenance-cost view-selection problem which is to select a set of views to materialize in order to minimize the query response time under a constraint of maintenance time. We do not make any assumptions about the query or the maintenance cost models. It is easy to see that the view-selection problem under a disk-space constraint is only a special case of the maintenance-time view-selection problem, when maintenance cost of each view remains a constant, i.e., the cost of maintaining a view is independent of the set of other materialized views. Thus, the maintenance-cost view-selection problem is trivially NP-hard, since the space-constraint view-selection problem is NP-hard, as mentioned in Section 2.2.3.

Now, we explain the main differences between the view-selection problem under the space constraint and the maintenance-cost view-selection problem, which make the maintenance-cost view-selection optimization problem more difficult. In the case of the view-selection problem with space constraint, as the query benefit of a view never increases with materialization of other views, the query-benefit per unit space of a nonselected view always decreases monotonically with the selection of other views. This property is formally defined

in the previous chapter as the monotonicity property of the benefit function and is stated here in a slight different form for sake of clarity.

Definition 6 (**(Monotonicity Property)**) A benefit function B, which is used to prioritize views for selection, is said to satisfy the *monotonicity property* for a set of views M with respect to distinct views V_1 and V_2 if $B(\{V_1, V_2\}, M)$ is less than (or equal to) either $B(\{V_1\}, M)$ or $B(\{V_2\}, M)$. □

In the case of the view-selection problem under space constraint, the query-benefit per unit space function satisfies the above defined monotonicity property for all sets M and views V_1 and V_2. However, for the maintenance-cost view-selection problem, the maintenance cost of a view can decrease with the selection of other views for materialization, and hence the query-benefit per unit of maintenance-cost of a view can actually increase. Thus, the total maintenance cost of two "dependent" views may be much less than the sum of the maintenance costs of the individual views, causing the query-benefit per unit maintenance-cost of two dependent views to be sometimes much greater than the query-benefit per unit maintenance-cost of either of the individual views. The above described nonmonotonic behavior of the query-benefit function makes the maintenance-problem view-selection problem intractable. The nonmonotonic behavior of the query-benefit per unit maintenance-cost function is illustrated in Example 4 in Section 3.4, where it is shown that the simple greedy approaches presented in previous works for the space-constraint view-selection problem could deliver an arbitrarily bad solution when applied to the maintenance-cost view-selection problem.

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

graphs. Performance studies indicate that the proposed greedy heuristic almost always returns an optimal solution for OR view graphs. The maintenance-cost view-selection was one of the open problems mentioned in [Gup97]. By designing an approximate algorithm for the problem, this chapter essentially answers one of the open questions raised in [Gup97].

3.3 The Maintenance-Cost View-Selection Problem

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

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

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

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

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

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

The view-selection problem under a disk-space constraint can be easily shown to be `NP`-hard, as there is a straightforward reduction [Gup97] from the `minimum set cover` problem. Thus, the maintenance-cost view-selection problem, which is a more general problem as discussed in Section 3.2, is trivially `NP`-hard.

3.3.1 Incorporating Maintenance Costs in View Graphs

We need to incorporate maintenance costs in the definition of AND-OR view graphs, so that the input to the view-selection maintenance-cost problem is complete. So, with each AND-OR graph defined there is a maintenance-cost function UC associated with it. The function UC is such that for a view v and a set of views M, $UC(v, M)$ gives the cost of maintaining v in presence of the set M of materialized views. We assume that the set L of base relations in G is always available. In this thesis, we do not discuss various maintenance cost models possible for an AND-OR view graph, and hence we assume that the function UC is given as part of an input with the AND-OR graph.

Maintenance Costs in OR View Graphs In case of OR view graphs, instead of the maintenance cost function UC for the graph, there is a maintenance-cost value associated with each edge (u, v), which is the maintenance cost incurred in maintaining u using the materialized view v. Figure 3.1 shows an example of an OR view graph $\mathcal{G}$ with the associated maintenance-costs. For the special case of OR view graphs, $UC(v, M)$ is computed from the maintenance costs associated with the edges in the graph as follows. The quantity $UC(v, M)$ is defined as the minimum maintenance-length of a path from v to some $u \in (M \cup L) - \{v\}$, where the *maintenance-length* of a path is defined as the sum of the maintenance-costs associated with the edges on the path.[1] The above characterization of $UC(v, M)$ in OR view graphs is without any loss of generality of a maintenance-cost model, because in OR view graphs a view u uses at most one view to help maintain itself.

In the following example, we illustrate how to compute $Q(v, M)$ and $UC(v, M)$ on OR view graphs.

EXAMPLE 3 Consider the OR view graph $\mathcal{G}$ of Figure 3.1. In the given OR view graph $\mathcal{G}$, the maintenance-costs and query-costs associated with each edge is zero, except for the maintenance-cost of 4 associated with the edges (V_1, B) and (V_2, B). Also, all query and update frequencies are uniformly 1. The label associated with each of the nodes in $\mathcal{G}$ is the reading-cost of the node. Also, the set of sinks $L = \{B\}$.

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

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

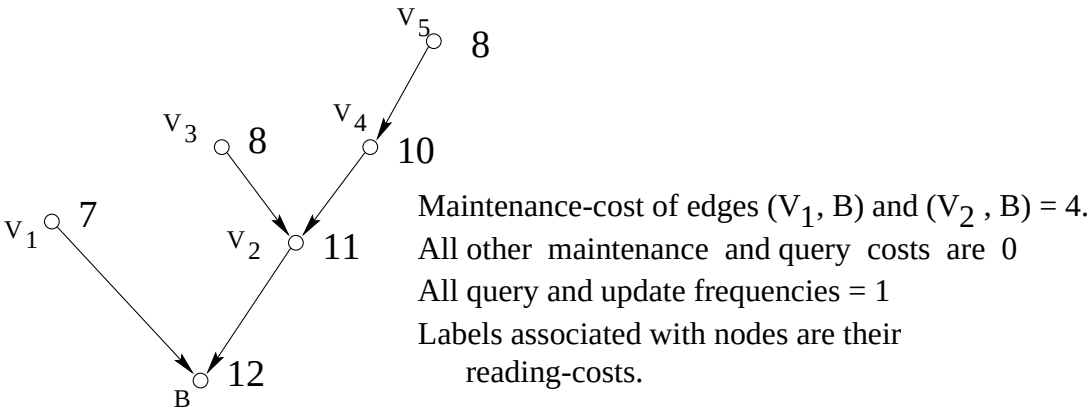

Figure 3.1: $\mathcal{G}$: An OR view graph

that $Q(B, \phi) = 12$. Also, as the minimum maintenance-length of a path from a view V_i to B is 4, $UC(V_i, \phi) = 4$ for all $i \leq 5$. □

Knapsack Effect We simplify the view-selection problem as in Chapter 2 by allowing that a solution may consume "slightly" more than the given amount of constraint. This assumption is made to ignore the `knapsack` component of the view-selection problem. However, when proving performance guarantee of a given algorithm, we compare the solution delivered by the algorithm with an optimal solution that consumes the same amount of resource as that consumed by the delivered solution.

3.4 Inverted-Tree Greedy Algorithm

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

In the context of the view-selection problem, a greedy algorithm was originally proposed in [HRU96] for selection of views in data cubes under a disk-space constraint. In Chapter 2, we generalized the results to some special cases of AND-OR view graphs, but still for the constraint of disk space. The greedy algorithms proposed in the context of view-selection work in stages. At each stage, the algorithm picks the "most beneficial" view. The algorithm continues to pick views until the set of selected views take up the given resource constraint. One of the key notions required in designing a greedy algorithm for selection of views is the

notion of the "most beneficial" view.

In the greedy heuristics proposed in Chapter 2 for selection of views to materialize under a space constraint, views are selected in order of their "query benefits" per unit space consumed. We now define a similar notion of benefit for the maintenance-cost view-selection problem addressed in this chapter.

Most Beneficial View

Consider an OR view graph G. At a stage, when a set of views M has already been selected for materialization, the *query benefit* $B(C, M)$ associated with a set of views C with respect to M is defined as $\tau(G, M) - \tau(G, M \cup C)$. We define the *effective maintenance-cost* $EU(C, M)$ of C with respect to M as $U(M \cup C) - U(M)$.[2] Based on these two notions, we define the view that has the most query-benefit per unit effective maintenance-cost with respect to M as the *most beneficial view* for greedy selection at the stage when the set M has already been selected for materialization.

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

EXAMPLE 4 Consider the OR view graph $\mathcal{G}$ shown in Figure 3.1. We assume that the base relation B is materialized and we consider the case when the maintenance-cost constraint is 4 units.

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

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

It is easy to see that the optimal solution is $\{V_2, V_3, V_4, V_5\}$ with a query benefit of 11 and a total maintenance time of 4. To demonstrate the **nonmonotonic** behavior of the benefit function, observe that the query-benefits per unit maintenance-cost of sets

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

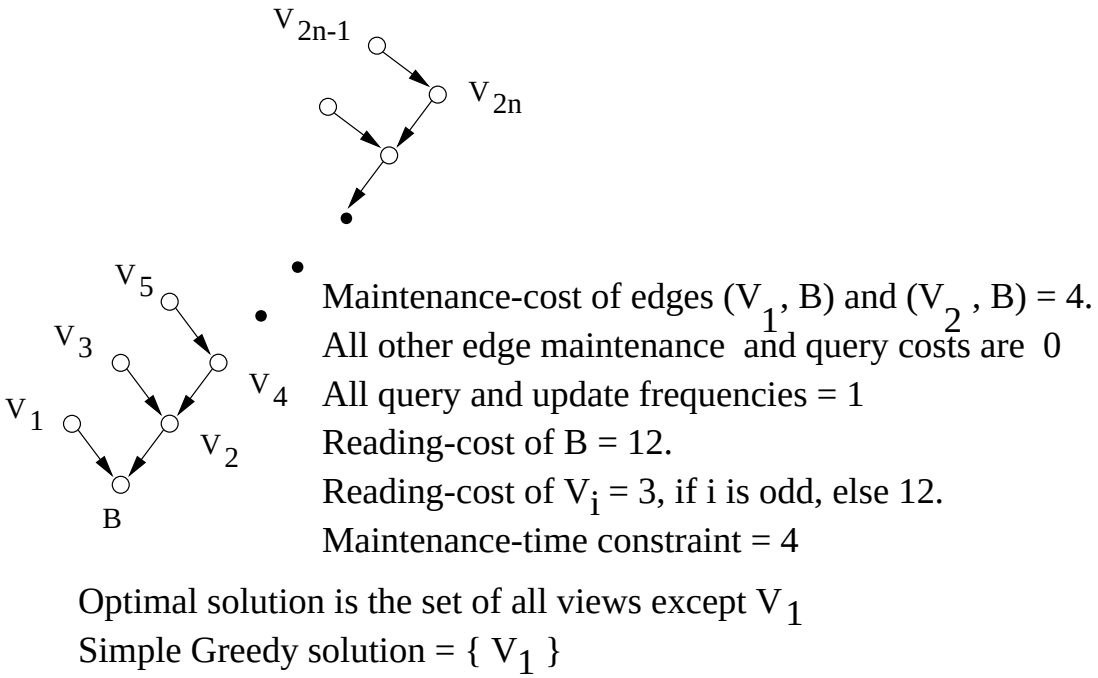

Figure 3.2: An OR view graph, $\mathcal{H}$, for which simple greedy performs arbitrarily bad

$\{V_2\}, \{V_3\}, \{V_2, V_3\}$ are 1, 1, and 7/4 respectively. This nonmonotonic behavior is the reason why the simple greedy algorithm that selects views on the basis of their query-benefits per unit maintenance-cost can deliver an arbitrarily bad solution.

Figure 3.2 shows an extended example where the optimal solution can be made to have an arbitrarily high query benefit, while keeping the simple greedy solution unchanged. Initially, the query benefit of any odd numbered V_i is 12 -3 = 9, while the query benefit of any even numbered V_i is 0. Also, the maintenance cost of any V_i is 4. Hence, the simple greedy algorithm starts by selecting $\{V_1\}$. As the maintenance cost constraint is 4 units, the solution returned by the simple greedy algorithm is $\{V_1\}$. The optimal solution is $\{V_2, V_3, \ldots, V_{2n}\}$ with a huge query benefit, but a maintenance cost of only 4. □

The above example shows that an appropriate definition of benefit is not enough to guarantee a good solution, and that selecting the most beneficial view at each stage can lead to a very bad greedy strategy.

Note that the nodes in the OR view graphs $\mathcal{G}$ and $\mathcal{H}$, presented in Figure 3.1 and Figure 3.2 respectively, can be easily mapped into *real* queries involving aggregations over the base data B. The query-costs associated with the edges in $\mathcal{G}$ and $\mathcal{H}$ depict the *linear*

cost model, where the cost of answering a query on v using its descendant u is directly proportional to the size of the view u, which in our model of OR view graphs is represented by the reading-cost of u. Notice that the minimum query-length of a path from u to v in $\mathcal{G}$ or $\mathcal{H}$ is R_v, the reading-cost of v. As zero maintenance-costs in the OR view graphs $\mathcal{G}$ and $\mathcal{H}$ can be replaced by extremely small quantities, the OR view graphs $\mathcal{G}$ and $\mathcal{H}$ depict the plausible scenario when the cost of maintaining a view u from a materialized view v is negligible in comparison to the maintenance cost incurred in maintaining a view u directly from the base data B.

Definition 7 (**Inverted Tree Set**) A set of nodes R is defined to be an inverted tree set in a directed graph G if there is a subgraph (not necessarily induced) T_R in the transitive closure of G such that the set of vertices of T_R is R, and the inverse graph[3] of T_R is a tree.[4]

In the OR view graph $\mathcal{G}$ of Figure 3.1, any subset of $\{V_2, V_3, V_4, V_5\}$ that includes V_2 forms an inverted tree set. Note that $\{V_4, V_5\}$ is forms an inverted tree set. The T_R graph corresponding to the inverted tree set $R = \{V_2, V_3, V_5\}$ has the edges (V_2, V_5) and (V_2, V_3) only. □

The motivation for the inverted tree set comes from the following observation, which we prove in Lemma 5. In an OR view graph, an arbitrary set O (in particular an optimal solution O), can be partitioned into inverted tree sets such that the effective maintenance-cost of O with respect to an already materialized set M is greater than the sum of effective-costs of inverted tree sets with respect to M.

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

Algorithm 6 **Inverted-Tree Greedy Algorithm**

Given: An OR view graph (G), and a total view maintenance time constraint S
BEGIN
$M = \phi$; $B_C = 0$;
repeat

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

for each inverted tree set of views T in G such that $T \cap M = \phi$
if $(EU(T,M) \leq S)$ **and** $(B(T,M)/EU(T,M) > \mathrm{B_C})$
$B_C = B(T,M)/EU(T,M)$;
$C = T$;
end if;
end for;
$M = M \cup C$;
until ($U(M) \geq S$);
return M;
END. ◇

We prove in Theorem 10 that the Inverted-tree greedy algorithm is guaranteed to deliver a near-optimal solution. In Section 3.6, we present experimental results that indicate that in practice, the Inverted-tree greedy algorithm almost always returns an optimal solution. We now define a notion of update graphs which is used to prove Lemma 5.

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

It is easy to see that an update graph is an embedded forest in G. An update graph of O is useful in determining the flow of changes when maintaining the set of views O. An edge (v,u) in an update graph U_O signifies that the view u uses the view v (or tables computed for maintenance of v) to incrementally maintain itself, when the set O is materialized. Figure 3.3 shows the update graph of $\{V_1, V_2, V_5\}$ in the OR view graph $\mathcal{G}$ of our running example in Figure 3.1.

Lemma 5 *For a given set of views M, a set of views O in an OR view graph G can be partitioned into inverted tree sets $O_1, O_2, \ldots, O_m$, such that $\sum_{i=1}^{m} EU(O_i, M) \leq EU(O, M)$.*

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

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

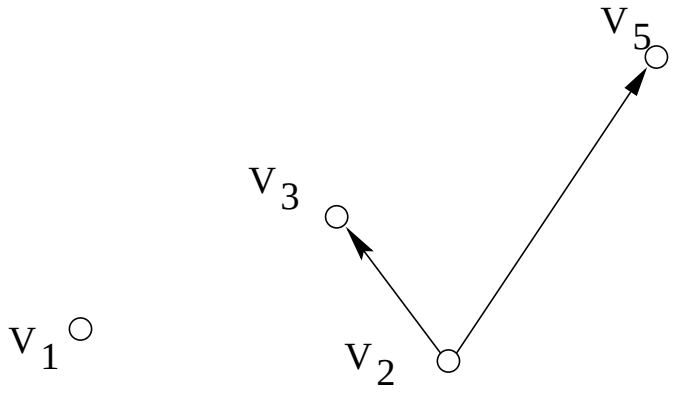

Figure 3.3: The update-graph for $\{V_1, V_2, V_3, V_5\}$ in G

tree in the transitive closure of the inverse graph of G. Hence, the set of vertices O_i is an inverted tree set in G.

For a set of views C, we use $UC(C, M)$ to denote the maintenance cost of the set C w.r.t. M, i.e., $UC(C, M) = \sum_{v \in C} g_v UC(v, M \cup C)$, where $UC(v, M)$ for a view v is the maintenance cost of v in presence of M as defined in Section 2.2.3. Also, let $Rd(M, C) = U(M) - UC(M, C)$, i.e., the reduction in the maintenance time of M due to the set of views C. Now, the effective maintenance-cost of a set O_i with respect to a set M, $EU(O_i, M)$, can be written as

$$\begin{aligned} EU(O_i, M) &= (UC(O_i, M) + UC(M, O_i)) - U(M) \\ &= UC(O_i, M) - (U(M) - UC(M, O_i)) \\ &= UC(O_i, M) - Rd(M, O_i) \end{aligned}$$

As no view in a set O_i uses a view in a different set O_j for its maintenance,

$$UC(O, M) = \sum_{i=1}^{m} UC(O_i, M).$$

Also, as any view uses at most one view to help maintain itself, the reduction in the maintenance cost of M due to the set O is less than the sum of the reductions due to the sets $O_1, \ldots, O_m$, i.e.,

$$Rd(M, O) \leq \sum_{i=1}^{m} Rd(M, O_i).$$

Therefore, we have

$$EU(O, M) = UC(O, M) - Rd(M, O)$$

$$\begin{aligned} &\geq \sum_{i=1}^{m} UC(O_i, M) - \sum_{i=1}^{m} Rd(M, O_i) \\ &\geq \sum_{i=1}^{m} (EU(O_i, M) \end{aligned}$$

■

Theorem 10 *Given an OR view graph G and a total maintenance-time constraint S. The Inverted-tree greedy algorithm (Algorithm 6) returns a solution M such that $U(M) \leq 2S$ and M has a query benefit of at least $(1 - 1/e) = 63\%$ of that of an optimal solution that has a maintenance cost of at most $U(M)$, under the assumption that the optimal solution doesn't have an inverted tree set O_i such that $U(O_i) > S$.*

Proof: It is easy to see that the maintenance cost of the solution returned by the Inverted-tree greedy algorithm is at most $2S$ units. Let O be a solution having $U(M) = k$ units of total maintenance time, with an optimal query benefit of B.

Consider a stage when the Inverted-tree greedy algorithm has already chosen a set M having a total maintenance time of l units with incremental per unit query benefits of $a_1, a_2, \ldots, a_l$. Thus, the absolute query benefit of M, $B(M, \phi)$, is $\sum_{i=1}^{l} a_i$. Trivially, the query benefit of the set $O \cup M$ is at least B. Therefore, the query benefit $B(O, M)$ of the set O with respect to M is at least $B - \sum_{i=1}^{l} a_i$.

By Lemma 5, the set O can be partitioned into inverted tree sets $O_1, O_2, \ldots, O_m$ such that $\sum_{i=1}^{m} EU(O_i, M) \leq EU(O, M)$. Also, by monotonicity of the query benefit function, $B(O, M) \leq \sum_{i=1}^{m} B(O_i, M)$. Now, it is easy to show by contradiction that there is an inverted tree set view O_i such that $B(O_i, M)/EU(O_i, M) \geq B(O, M)/EU(O, M)$, i.e., the query-benefit per unit of effective maintenance-cost of O_i is at least that of O at this stage (else $B(O, M) > \sum_{i=1}^{m} B(O_i, M)$).

As $EU(O_i, M) \leq U(O_i) \leq S$, O_i is also considered for selection by the Inverted-tree greedy at this stage. Thus, the query benefit per unit of effective maintenance-cost of the set C selected by the Inverted-tree algorithm is at least the query-benefit per unit effective maintenance-cost of O_i at this stage. Now as $EU(O, M) \leq k$, we have

$$\begin{aligned} B(C, M)/EU(C, M) &\geq B(O_i, M)/EU(O_i, M) \\ &\geq B(O, M)/EU(O, M) \\ &\geq B(O, M)/k \end{aligned}$$

$$\geq \quad (B - \sum_{i=1}^{l} a_i)/k$$

Distributing the benefit of C over each of its unit spaces equally (for the purpose of analysis), we get $a_{l+j} \geq (B - \sum_{i=1}^{l} a_i)/k$, for $0 < j \leq EU(C, M)$. As the above analysis is true for each set C selected at any stage, we have

$$B \leq ka_j + \sum_{i=1}^{j-1} a_i, \qquad \text{for } \ 0 < j \leq k.$$

Multiplying the j^{th} equation by $(\frac{k-1}{k})^{k-j}$ and adding all the equations, we get $A/B \geq 1 - (\frac{k-1}{k})^k \geq 1 - 1/e$, where $A = \sum_{i=1}^{k} a_i = B(M, \phi)$, the (absolute) query benefit of M. ∎

The simplifying assumption made in the above algorithm is almost always true, because $U(M)$ is not expected to be much higher than S. The following theorem proves a similar performance guarantee on the solution returned by the Inverted-tree greedy algorithm without the assumption used in Theorem 10.

Theorem 11 *Given an OR view graph G and a total maintenance-time constraint S. The Inverted-tree greedy algorithm (Algorithm 6) returns a solution M such that $U(M) \leq 2S$ and $B(M, \phi)/U(M) \geq 0.5B(O, \phi)/S$, where O is an optimal solution such that $U(O) \leq S$.*

Proof: Let $k = U(O) = S$ and $k' = U(M) \geq k$. We use the same notations as in Theorem 10. Now, using the arguments similar to that in Theorem 10, we get

$$B \quad \leq \quad ka_j + \sum_{i=1}^{j-1} a_i, \qquad \text{for } \ 0 < j \leq k, \text{ and}$$

$$B \quad \leq \quad ka_j + \sum_{i=1}^{k} a_i, \qquad \text{for } \ k < j \leq k'.$$

Multiplying the j^{th} equation by $(\frac{k-1}{k})^{k-j}$ and adding the first k equations, we get

$$(\sum_{i=1}^{k} a_i)/B \geq 1 - (\frac{k-1}{k})^k \geq 1 - 1/e.$$

Substituting the above in the rest of the equations, we get $a_j \geq (1/e)B/k$ for $k \leq j \leq k'$. Thus, $B(M, \phi) = \sum_{i=1}^{k'} a_j \geq (k'/k)(B/2)$, as $k' \leq 2k$. ∎

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

In Section 3.6, we present experimental results that indicate that the Inverted-tree greedy algorithm almost always returns an optimal solution. We end this section with time complexity analysis of the Inverted-tree greedy heuristic.

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

We note that for the special case of an OR view graph being a *balanced binary tree*, each stage of the Inverted-tree greedy algorithm runs in polynomial time $O(n^2)$, where n is the number of nodes in the graph. The number of inverted tree sets, $T(h)$, in a general balanced tree of height h can be computed as follows. Let $T(h)$ be the total number of inverted trees in a balanced tree of height h with a branching factor of $r > 1$. To derive a recursive function for $T(h)$, let us consider a tree G of height h. Let R be the root of G. Now, the number of inverted trees in G not containing the node R is $rT(h-1)$, as any such inverted tree in G is an inverted tree in *one* of the r subtrees of R. Also, the number of inverted trees in G containing R is also $rT(h-1)$, as any such inverted tree in G is formed by attaching R to an inverted tree in *one* of R's subtrees. There is one more inverted tree in G that contains the node R only. Thus, $T(h) = rT(h-1) + rT(h-1) + 1$ is the recursive function for number of inverted trees in a balanced tree of branching factor r. The recursive function for $r > 1$ gives $T(h) = ((2r)^h - 1)/(r-1) = O(n^2)$.

The worst-case time complexity of the Inverted-tree greedy algorithm for general OR view graphs is exponential in the size of a given graph. However, as discussed in Section 3.6, our experiments show that the Inverted-tree greedy approach takes substantially less time

than the A^* algorithm presented in the next section, especially for sparse graphs. Also, the space requirements of the Inverted-tree greedy algorithm is polynomial in the size of graph while that of the A^* heuristic is exponential in the size of the input graph.

3.5 A^* Heuristic

In this section, we present an A^* heuristic that, given an AND-OR view graph and a quantity S, deliver a set of views M that has an optimal query response time such that the total maintenance cost of M is less than S. Recollect that an A^* algorithm [Nil80] searches for an optimal solution in a search graph where each node represents a candidate solution. Roussopoulos in [Rou82b] also demonstrated the use of A^* heuristics for selection of indexes in relational databases.

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

The Algorithm 7 shown below depicts the A^* heuristic for the maintenance-cost view-selection problem in general AND-OR graphs.

We define two functions[5] $g : V(T_G) \mapsto \mathcal{R}$, and $h : V(T_G) \mapsto \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers. For a node $x \in V(T_G)$, with a label $<N_x, M_x>$, the value $g(x)$ is the total query cost of the queries on N_x using the selected views in M_x. That is,

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

The number $h(x)$ is an estimated lower bound on $h^*(x)$ which is defined as the remaining query cost of an optimal solution corresponding to some descendant of x in T_G. In other

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

words, $h(x)$ is a lower bound estimation of $h^*(x) = \tau(G, M_y) - g(x)$, where M_y is an optimal solution corresponding to some descendant y of x in T_G.

Algorithm 7 A^* **Heuristic**

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

We now show how to compute the value $h(x)$, a lower bound for $h^*(x)$, for a node x in the binary tree T_G. Let $N = V(G)$ be the set of all views/nodes in G. Given a node x, we need to estimate the optimal query cost of the remaining queries in $N - N_x$. Let $s(v) = g_v UC(v, N)$, the minimum maintenance time a view v can have in presence of other materialized views. Also, if a node $v \in V(G)$ is not selected for materialization, queries on v have a minimum query cost of $p(v) = f_v Q(v, N - \{v\})$. Hence, for each view v that is not selected in an optimal solution M_y containing M_x, the remaining query cost accrues by at least $p(v)$. Thus, we fill up the remaining maintenance time available $S - U(M_x)$ with views in $N - N_x$ in the order of their $p(v)/s(v)$ values. The sum of the $f_v Q(v, N - \{v\})$ values for the views left out will give a lower bound on $h^*(x)$, the optimal query cost of the remaining queries. In order to nullify the knapsack effect as mentioned in Section 3.3, we start by leaving out the view w that has the highest $f_v Q(v, N - p\{v\})$ value. The algorithm is presented in Algorithm 8. We prove in Theorem 12 that $h(x)$, as computed by

the Algorithm 8, is indeed a lower bound of $h^*(x)$.

Algorithm 8 **Computing h**

Input: An AND-OR view graph (G), a maintenance-time constraint S, and
a node x with a label $<N_x, M_x>$ in the search tree T_G.
$N = \{v_1, v_2, \ldots, v_n\}$ is the set of all views/nodes in G.
Output: The value $h(x)$.
BEGIN
Let $N_x = \{v_1, v_2, \ldots, v_d\}$ and $N'_x = N - N_x = \{v_{d+1}, \ldots, v_n\}$.
For each view $v \in N'_x$, define a profit $p(v) = f_v Q(v, N - \{v\})$ and space $s(v)$
is $g_v U(v, N)$, the minimum possible maintenance cost of v.
$S_x = 0$;
Let w be the view that has the highest profit in N'_x.
$P_x = p(w)$; $N'_x = N'_x - \{w\}$;
repeat
Let v be the view with the highest value of $p(v)/s(v)$ in N'_x.
$S_x = S_x + s(v)$;
$P_x = P_x + p(v)$;
$N'_x = N'_x - \{v\}$;
until ($S_x \geq S - U(M_x)$);
$h(x) = 0$;
for $v \in N'_x$
$h(x) = h(x) + p(v)$;
return $h(x)$;
END. ◇

Theorem 12 *The A^* algorithm (Algorithm 7) returns an optimal solution.*

Proof: If an A^* heuristic expands nodes in the increasing order of their $g(x) + h(x)$ values, it is known ([Nil80]) that the first leaf node found by the algorithm corresponds to an optimal solution. Thus, we only need to show that $h(x)$ is indeed a lower bound of $h^*(x)$, i.e., $h(x) \leq h^*(x)$ for all $x \in V(T_G)$.

Consider an optimal feasible solution M_y corresponding to a node y that is a descendant of x in T_G. Each view $v \notin M_y$ adds at least $p(v) = f_v Q(v, N - \{v\})$ units to the remaining

query cost. So, $h^*(x)$, the remaining query cost of the optimal solution M_y, is at least $\sum_{v \in (N-(M_y \cup N_x))} p(v) = \sum_{v \in (N-N_x)} p(v) - \sum_{v \in (M_y - N_x)} p(v) = P_x + h(x) - \sum_{v \in (M_y - N_x)} p(v)$, where $h(x) = \sum_{v \in (N-N_x)} p(v) - P_x$, as computed by Algorithm 8. We will show that $P_x \geq \sum_{v \in (M_y - N_x)} p(v)$, which will imply that $h^*(x)$ is at least $h(x)$.

To prove the above claim, note that $U(M_y) \leq S + UC(v, M_y)$ for some $v \in (M_y - M_x)$. As, $U(M_y) \geq U(M_x) + \sum_{u \in (M_y - M_x)}(s(u))$, where $s(u)$ is the minimum possible maintenance cost of u, we get $\sum_{u \in (M_y - M_x)}(s(u)) \leq S + UC(v, M_y) - U(M_x)$. As $v \in M_y - M_x$ and $UC(v, M_y) \geq s(v)$, we have $\sum_{u \in ((M_y - M_x) - \{v\})}(s(u)) \leq S - U(M_x)$. Note that P_x, as computed by the Algorithm 8, is such that $P_x - p(w)$ is more than the maximum profit that can be fit in the knapsack of size $S - U(M_x)$. Thus, $P_x - p(w) \geq \sum_{u \in ((M_y - N_x) - \{v\})} p(u)$, which implies that $P_x \geq \sum_{u \in (M_y - N_x)} p(u)$. ∎

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

3.6 Experimental Results

We ran some experiments to determine the quality of the solution delivered and the time taken in practice by the Inverted-tree Greedy algorithm for OR view graphs. We implemented both the algorithms, Inverted-tree Greedy and A^* heuristic, and ran them on random instances of OR view graphs that are balanced trees and directed acyclic graphs with varying edge-densities. A random directed acyclic is generated by tossing a biased coin to decide whether an edge exists between a pair of nodes. The random bias gives the edge-density of the generated graph. For each randomly generated instance of a view graph, we labeled the nodes with random query and update frequencies. We ran experiments on random OR view graphs of size upto 25 nodes, as it was impossible to run A^* heuristic on any larger graphs because of memory space constraints.

We start by describing the cost model we used for the purposes of our experiments.

Cost Model For the purposes of experimentation, we assumed that each view is an aggregate view defined over the base data. Hence, we assume a linear query cost model,

wherein the cost of answering a query u from its descendant v in a view graph is proportional to $|v|$, the size of v. The experimental results are independent of the proportionality factor(s). The linear cost model is a very reasonable assumption, as shown in [HRU96, GHRU97], when each view in the OR graph is an aggregate view.

For the purposes of computing maintenance costs, we assume the following model for cost of maintaining a view u in presence of a descendant v. The changes to u, Δu, can be computed from changes to v, Δv, in time proportional to $|\Delta v|$, and the view u can be refreshed using Δu in time proportional to $|\Delta u|$. Thus, the total maintenance time incurred in maintaining u using its materialized descendant v is proportional to $(|\Delta u| + |\Delta v|)$. For sake of simplicity, we further assume that $(|\Delta u| + |\Delta v|)$ is proportional to $(|u| + |v|)$ as is likely to be the case when updates are insertion generating (as defined in [MQM97]), or when the updates are update generating but uniformly spread across the domain.

Also, as we are considering aggregate views, we assigned random sizes to each view/node in the view graph in such a way that the size of a view u was less than the size of each of its descendants.

Observations We made the following observations. The Inverted-tree Greedy Algorithm (Algorithm 6) returned an optimal solution as computed by the A^* heuristic for almost all (96%) view graph instances. In other cases, the solution returned by the Inverted-tree greedy algorithm had a query benefit of around 95% of the optimal query benefit, as shown in Figure 3.4.

For balanced trees and sparse graphs having edge density less than 40%, the Inverted-tree greedy took substantially less time (a factor of 10 to 500) than that taken by the A^* heuristic. With the increase in the edge density, the benefit of Inverted-tree greedy over the A^* heuristic reduces and for very dense graphs, A^* may actually perform marginally better than the Inverted-tree greedy. One should observe that OR view graphs that are expected to arise in practice would be very sparse. For example, the the OR view graph corresponding to a data cube having n dimensions has $\sum_{i=1}^{n}(\binom{n}{i}2^i) = 3^n$ edges and 2^n vertices. Thus, the edge density is approximately $(0.75)^n$, for a given n.

The comparison of the time taken by the Inverted-tree greedy and the A^* heuristic is briefly presented in Figures 3.5-3.6. In all the plots shown in Figures 3.5-3.6, the different view graph instance s of the maintenance-cost view-selection problem are plotted on the x-axis. A view graph instance G is represented in terms of N, the number of nodes in G,

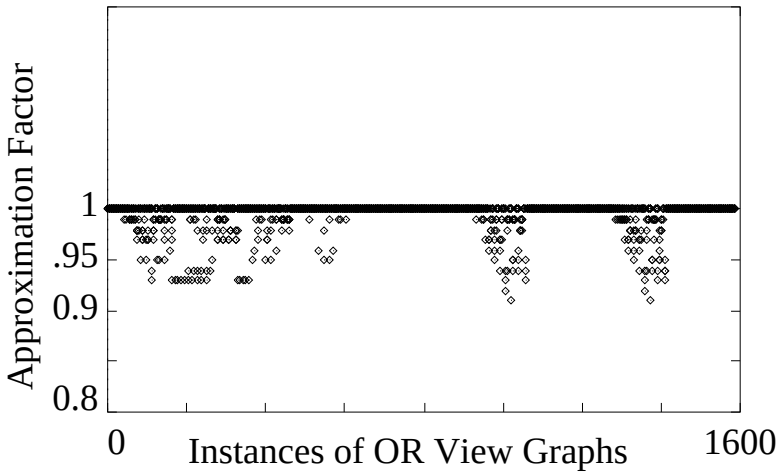

Figure 3.4: Quality of the Inverted-tree greedy solutions

and S, the maintenance-time constraint. The view graph instances are arranged in the lexicographic order of (N, S), i.e., all the view graphs with smallest N are listed first, in order of their constraint S values. In all the graph plots, the number N varies from 10 to 25, and S varies from the time required to maintain the smallest view to the time required to maintain all views in a given view graph. The breaks in the graph plots depict a change in the value of N.

In Figures 3.5-3.6, for the case of balanced trees and view graphs with an edge-density of 10%, we have plotted times taken by the Inverted-tree greedy as well as the A^* heuristic. One can see that the time taken by A^* heuristic is 100 to 500 times the time taken by the Inverted-tree greedy. For other graphs instances of edge densities 15%, 25%, and 40%, we have plotted the performance ratio (the ratio of the time taken by the A^* heuristic to the time taken by the Inverted-tree Greedy.) We also ran experiments on random graphs with number of edges linear in the number of nodes. Figure 3.6 shows the performance ratio obtained for random graphs having $2n$ edges, where n is the number of nodes in the graph.

For a particular value of N, the time taken by the A^* heuristic first increases and then decreases, with increase in S, the maintenance-time constraint. The initial increase is due to the increase in the number of feasible solutions, and hence the number of nodes in the solution-tree T_G. With increase in S, the value $h(x)$ gets closer and closer to $h^*(x)$. This offsets the previous effect of increase in the number of feasible solutions, after a certain value of S, causing a sudden decrease in the time taken by the A^* heuristic. In the extreme

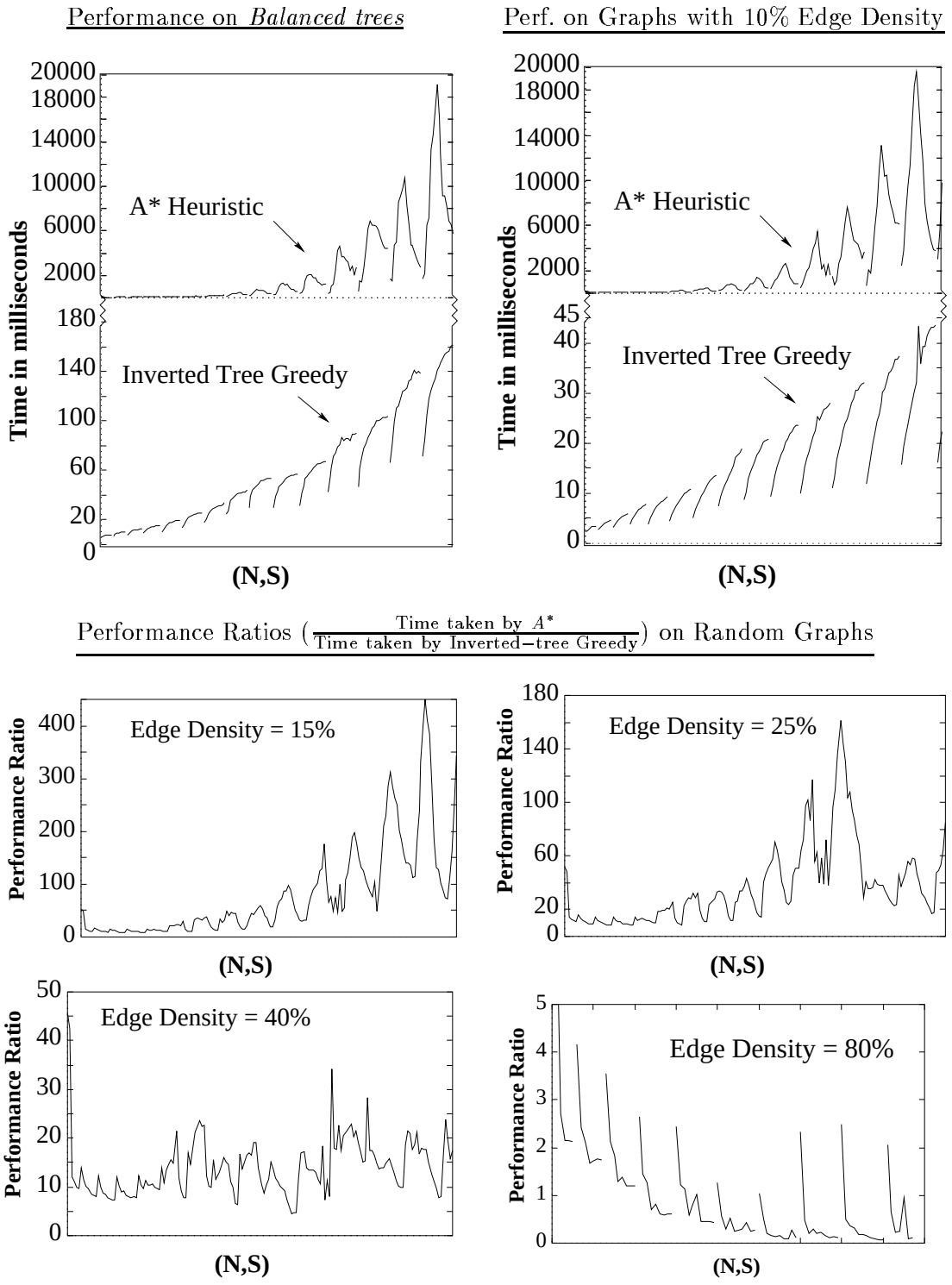

Figure 3.5: Experimental results. The x-axis shows the view graph instances in lexicographic order of their (N, S) values, where N is the number of nodes in the graph and S is the maintenance-time constraint.

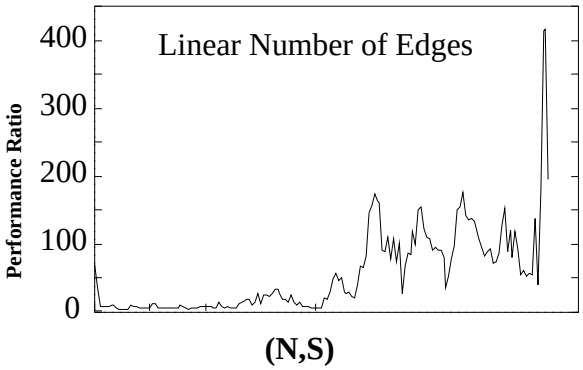

Figure 3.6: Performance ratios on graphs with linear number of edges

case, when almost every view is eventually selected for materialization, one can see that $h(x)$ is is very close to 0, yielding an efficient heuristic.

As the space requirement of an A^* heuristic grows exponentially in the size of the input graph, we could not run A^* heuristic for N larger than around 25 because of memory-space limitations. Note that, in contrast, the Inverted-Tree Greedy heuristic takes only quadratic $O(n^2)$ space, where n is the number of views in an OR view graph.

3.7 Concluding Remarks

The view-selection problem in a data warehouse is to select a set of views to materialize so as to optimize the total query response time, under some resource constraint such as total space and/or the total maintenance time of the materialized views. In the previous chapter, we addressed the view selection problem under a disk-space constraint. In practice, the real constraining factor is the total maintenance time. So, in this chapter, we have considered the maintenance-cost view-selection problem where the constraint is of total maintenance time.

We designed an Inverted-tree Greedy approximation for the special case of OR view graphs. We also designed an A^* heuristic that delivers an optimal solution. We carried out some preliminary experiments to measure the performance of the two algorithms. The results were very encouraging for the Inverted-tree greedy algorithm.

Chapter 4

Incremental Maintenance of Views

4.1 Introduction

In a data warehouse, views are computed and stored in the database to allow efficient querying and analysis of the data. These views stored at the data warehouse are known as *materialized views*. In order to keep the views in the data warehouse upto date, it is necessary to maintain the materialized views in response to the changes at the sources. The view can be either recomputed from scratch, or *incrementally maintained* by propagating the base data changes onto the view so that the view reflects the changes. Incrementally maintaining a view can be significantly cheaper than recomputing the view from scratch, especially if the size of the view is large compared to the size of the changes [BM90, MQM97, CKL+97].

The problem of finding such changes at the views based on changes to the base relations has come to be known as the *view maintenance problem* and has been studied extensively. Several algorithms have been proposed over the recent years [BLT86, BCL89, CW91, QW91, GMS93, GL95, Qua97, MQM97, GJM97] for incremental maintenance of view expressions. The previously proposed algorithms on incremental maintenance suffer from the following shortcomings:

- None of the earlier work handles the case of general view expressions involving aggregate and outerjoin operators. Quass in [Qua97] is the only work that attempts to maintain general view expressions involving aggregate operators, but the expressions obtained are very inefficient and complicated. Gupta et al. in [GJM97] show how to

maintain a simple outerjoin view, but do not address general expressions involving outerjoin operators. In recent work done concurrently with ours, Griffin and Kumar [GK98] derives expressions for propagating insertions and deletions through outerjoin operators.

- To date, most of the incremental maintenance approaches compute and propagate insertions and deletions at each node in a view expression tree, which could be very inefficient in cases involving aggregations or outerjoins.

In this chapter, we develop a change-table technique for incrementally maintaining general view expressions involving relational and aggregate operators. Instead of computing insertions and deletions, the change-table technique developed here computes and propagates "change-tables." We show that the developed change-table technique outperforms the previously proposed techniques by orders of magnitude. The developed framework easily extends to efficiently maintaining view expressions containing outerjoin operators. Moreover, the framework also yields an approach to allow propagation of certain kinds of deletions and updates directly in a very efficient manner.

Chapter Organization In the rest of this section, we present some basic notation used throughout this chapter. Section 4.2 presents a motivating example that illustrates the idea behind this chapter and contrasts previous techniques with the change-table technique presented in this chapter. In Section 4.3, we briefly describe how our work fits in the previous frameworks of incremental view maintenance algorithms. In Section 4.4, we define the refresh operator used to apply the changes represented in a change table and briefly outline its implementation. In the following section, we discuss propagation of change tables that originate at an aggregate operator. In Section 4.6, we discuss propagation of change tables that originate at an outerjoin node. In Section 4.6, we extend our techniques to propagating certain kinds of deletions and updates directly and more efficiently. We discuss the optimality of our techniques under some reasonable cost model in Section 4.8. A brief survey of related work is presented in Section 4.9. Finally, we present our concluding remarks in Section 4.10.

4.1.1 Notation

We consider only *bag* semantics in this chapter, i.e., all the relational operators used are duplicate-preserving. We use $\uplus$ to denote bag union, $\dot{-}$ to denote bag minus, ∇E to denote deletions from a bag-algebra expression E, $\triangle E$ to denote insertions into E, σ_q to denote selection on condition q, Π_A to denote duplicate-preserving projection on a set of attributes A, π to denote the generalized projection operator (note that we use slightly different symbols for duplicate-preserving projection (Π) and for generalized projection (π) operators), $\times$ to denote cross-product, $\bowtie$ to denote natural join, and $\bowtie_J$ and $\overset{fo}{\bowtie}_J$ to denote join and full outerjoin operations with the join condition J. The symbols $\overset{lo}{\bowtie}$ and $\overset{ro}{\bowtie}$ are used for left and right outerjoin respectively. Also, $Attrs(J)$ denotes the set of attributes used in a predicate J or a relation J.

The only operators that may require explanation are the outerjoin and generalized projection operators. The (full) *outerjoin* differs from an ordinary join by including in the result any "dangling"[1] tuple of either relation after "padding" it with `NULL`'s in those attributes that belong to the other relation. For example, $R(A,B) \overset{fo}{\bowtie}_{R.B=S.B} S(B,C)$ will include a tuple $(a, b, \texttt{NULL}, \texttt{NULL})$, if $(a,b) \in R$ and $(b,c) \notin S$ for any c. One variant of the outerjoin operator is a *left* (*right*) outerjoin, where the dangling tuples of only the left (right) operand relation are padded with `NULL`'s and included in the result. Hence, in the above example, $(a, b, \texttt{NULL}, \texttt{NULL})$ would be included in $R \overset{lo}{\bowtie}_J S$, but not in $R \overset{ro}{\bowtie}_J S$. The *generalized projection* operator introduced in [GHQ95] is used to represent the groupby operation of SQL algebraically. For example, we could use the following expression to define the `SISales` view (V_1) of Example 5.

$$V_1 = \pi_{storeID, itemID, SumSISales=sum(price), NumSISales=count(*)}(\sigma_{date>1/1/95}(\texttt{sales}))$$

For a set of attributes G, we use the notation $\equiv_G$ to represent the predicate

$$\bigwedge_{g \in G}(LHS.g = RHS.g)$$

in a join condition, where LHS and RHS are the left and right operand relations of the join operator. For example, when J is $(\equiv_G \ \wedge p)$, the expression $R \bowtie_J S$ denotes a join operation with the join condition $(\bigwedge_{g \in G}(R.g = S.g) \wedge p)$, for a predicate p and a set of attributes G in R and S.

[1] Dangling tuples are the ones that fail to join with any tuple from the other relation.

4.2 Motivating Example and Previous Approaches

EXAMPLE 5 Consider the classic example of a warehouse containing information about stores, items, and day-to-day sales. The warehouse stores the information in three base relations viz. `stores`, `items`, and `sales` defined by their schemas as follows.

`stores`(storeID, city, state)

`InfoStores`(state, area, population)

`items`(itemID, category, cost)

`sales`(storeID, itemID, date, price)

For each store location, the relation `stores` contains the storeID, the city, and the state in which the store is located. The relation `InfoStores` contains more information about each state. For each item, the relation `items` contains its itemID, its category, and its purchase price (cost). An item has a unique cost, but can belong to multiple categories, e.g., a *whitening toothpaste* could belong to *dental care*, *cosmetics*, and *hygiene* categories.[2] The relation `sales` contains detailed information about sales transactions. For each item sold, the relation `sales` contains a tuple storing the storeID of the selling store, itemID of the item sold, date of sale, and the sale price.

Let us consider the following views `SISales`, `CitySales`, `CategorySales`, `SSInfo`, and `SSFullInfo` defined over the base relations. The view `SISales` computes for each storeID and itemID the total price of items sold after 1/1/95. The view `SISales` is an intermediate table used to define the views `CitySales` and `CategorySales`. The view `CitySales` stores, for each city, the total number and dollar value of sales of all the stores in the city. The view `CategorySales` stores the total sale for each category of items. All the above described views consider only those sales that occur after 1/1/95. The view `SSInfo` stores the full outerjoin of the base relations `sales` and `stores`, retaining stores that have had no sales due to some reasons and also retaining those sales whose corresponding storeID is missing from the table `stores`, because, maybe, the table `stores` has not been updated yet. We define another view `SSFullInfo` as the outerjoin view of `SSInfo` and `InfoStores`. The views `CitySales`, `CategorySales`, and `SSFullInfo` are the only views that are stored (materialized) at the data warehouse and this is represented below by the keyword "MATERIALIZED"[3] in the SQL definitions of the views. We wish to maintain these materialized views in response to changes at the base relations.

[2]Note that `items` is denormalized with a functional dependency from *itemID* to *cost*.

[3]The keyword "MATERIALIZED" is not supported by SQL, but has been introduced in this chapter.

```
CREATE VIEW SISales AS
SELECT storeID, itemID, sum(price) AS SumSISales, count(*) AS NumSISales
FROM   sales
WHERE date > 1/1/95
GROUP BY storeID, itemID;

CREATE MATERIALIZED VIEW CitySales AS
SELECT city, sum(SumSISales) AS SumCiSales, sum(NumSISales) AS NumCiSales
FROM   SISales, stores
WHERE SISales.storeID = stores.storeID
GROUP BY city;

CREATE MATERIALIZED VIEW CategorySales AS
SELECT category, sum(SumSISales) AS SumCaSales, sum(NumSISales) AS NumCaSales
FROM   SISales, items
WHERE SISales.itemID = items.itemID
GROUP BY category;

CREATE VIEW SSInfo AS
SELECT *
FROM   sales FULL OUTERJOIN stores
WHERE sales.storeID = stores.storeID;

CREATE MATERIALIZED VIEW SSFullInfo AS
SELECT *
FROM   SSInfo FULL OUTERJOIN InfoStores
WHERE SSInfo.state = InfoStores.state;
```

Consider the database sizes shown in Table 4.1. We assume that the base relation `sales` has one billion sales transactions, and the base relations `stores`, `InfoStores`, and `items` have 1,000, 100, and 10,000 tuples respectively. We illustrate the various maintenance approaches for the case when 10,000 tuples are inserted into the base relation `sales`. Table 4.1 shows the number of tuples changed in the views, as a result of the insertion of 10,000 tuples into `sales`. The table also shows the number of tuple accesses (reads and writes) incurred by different maintenance techniques to update the materialized views.

Summary Table	Number of Tuples	Changes (No. of Tuples)	Tuple Reads and Writes	
			Previous Work	Our Work
`sales`	1,000,000,000	10,000		
`stores`	1,000	-		
`InfoStores`	100	-		
`items`	10,000	-		
V_1 = `SISales`	1,000,000	600	610,000 ([Qua97])	10,000
V_2 = `CitySales`	100	10	1,020 ([Qua97])	1,020
V_3 = `CategorySales`	1,000	1,000	12,000 ([Qua97])	12,000
Total for V_1, V_2, V_3			**623,020** [Qua97]	**23,020**
V_4 = `SSInfo`	1,000,000,010	10,000	2,000,000,020 [GJM97] 11,000 [GK98]	11,000
V_5 = `SSFullInfo`	1,000,000,020	10,000	10,100 [GJM97]/[GL95] 1,000,020,110 [GK98]	20,100
Total for V_4 and V_5			**1,000,031,110** [GK98]	**31,100**

Table 4.1: Benefits of propagating change tables (Materialized views are V_2, V_3, and V_5).

Cost Model We have used the simple model of counting tuple accesses for sake of convenience as orders of magnitude improvement in number of tuples computed and accessed translates directly into significant improvement in number of disk accesses. Also, the model of counting only the tuple reads and writes is implicitly assuming that all the required indexes are available to both techniques thereby a tuple read/write taking only a unit amount of time. In Section 4.8, we show that the change-table technique developed in this chapter is superior than previous techniques under a data warehouse cost model.

We use the names V_1, V_2, V_3, V_4, V_5 for the views `SISales`, `CitySales`, `CategorySales`, `SSInfo`, and `SSFullInfo` respectively. We assume that the relations `stores`, `InfoStores`, and `items` are small enough to fit in main-memory.

Previous Techniques

Of the previous approaches, only [Qua97] provides techniques to maintain general view expressions involving aggregate operators. Prior works in [GMS93], [GL95], and [MQM97] consider aggregates, but in a very limited fashion. Gupta et al. in [GMS93] consider view expressions involving aggregates, but their maintenance expressions assume that the intermediate aggregate subexpressions are materialized. Modifying the techniques presented in [GMS93] to accommodate nonmaterialized aggregate subexpressions yields an approach that is conceptually similar to that of [Qua97]. The works of [GL95] and [MQM97] are restricted to views that have at most one aggregate operator as the last operator in the

view expression.[4] Also, group-by attributes are not allowed in [GL95], and [MQM97] maintains views on star schemas only. Thus, for the case of general view expressions involving aggregate operators, we can compare our techniques with that of [Qua97] only.

Quass in [Qua97] extends the techniques in [GL95] by including aggregate operators. [Qua97] computes maintenance expressions for general views by recursively computing insertions and deletions for each of the subexpressions in the view expression in response to changes at the base relations. In our example, the insertions to `sales`, $\triangle$`sales`, result in insertions ($\triangle V_1$) and deletions (∇V_1) to the view $V_1 =$ `SISales`, which is an aggregate view over the base relation `sales`. The expressions that compute $\triangle V_1$ and ∇V_1, as derived in [Qua97], are quite complex and are shown in Figure 4.1). As V_1 is not materialized, the maintenance expressions for V_1 essentially recompute the aggregate values of the affected tuples in V_1 from the base relation `sales`. Using the propagation equations from [Qua97], one can propagate $\triangle V_1$ and ∇V_1 upwards to obtain expressions for $\nabla V_2, \triangle V_2, \nabla V_3$, and $\triangle V_3$ as shown in Figure 4.1).

For our purposes, its not important to understand the maintenance expressions given in Figure 4.1, and they are given here primarily to show their complexity. We have used $\bowtie$ to denote a natural join operation, $\bowtie_G$ to denote an equi-join operation on a set of attributes G, $\overline{\ltimes}$ to denote an anti-semijoin operation, and π_A to denote the generalized projection symbol, which represents the groupby operation of SQL as described in Section 4.1. Also, $l.a$ and $r.a$ refer to the attribute a of the left and right operands respectively.

Computing Tuple Accesses As V_1 is not materialized, computing tuples of $\triangle V_1$ requires that for each tuple in $\pi_{A_{1,\delta}}(\delta$`sales`$)$, we must look up all tuples of `sales` that have the same *storeID* and *itemID* values. Given the database sizes of Table 4.1, assume that each tuple of V_1 is derived from 1,000 tuples of `sales` on an average. Thus, computing 600 tuples of $\triangle V_1$ requires 600,000 tuple accesses. We need 11,000 accesses to read the base relations `stores` and `items` into main-memory. Even assuming that rest of the computation can be done in main memory, the total number of tuples accesses to refresh the views V_2 and V_3 is $10,000 + 600,000 + 11,000 + 2,020$, where 2,020 tuple accesses are due to the final tuple updates in V_2 and V_3. Note that each update requires a read and a write access.

[4] In some cases, view expressions can be rewritten so that aggregation is the last operator, but the rewritten query has worse query performance.

$$
\begin{aligned}
\text{Let } A_1 &= \{\text{storeID}, \text{itemID}, \text{SumSISales}, \text{NumSISales}\} \\
\text{Let } A_{1,ins} &= \{\text{storeID}, \text{itemID}, \text{price}, \_\text{count} = 1\} \\
\text{Let } A_{1,del} &= \{\text{storeID}, \text{itemID}, \text{price} = -\text{price}, \_\text{count} = -1\} \\
\text{Let } A_{1,\delta} &= \{\text{storeID}, \text{itemID}, \text{SumSISales} = \text{sum}(\text{price}), \text{NumSISales} = \text{sum}(\_\text{count})\} \\
\delta\texttt{sales} &= \Pi_{A_{1,ins}}(\sigma_{(date>1/1/95)}(\triangle\texttt{sales})) \uplus \Pi_{A_{1,del}}(\sigma_{(date>1/1/95)}(\triangledown\texttt{sales})) \\
\triangledown(V_1) &= \Pi_{r.a|a\in A_1}(\pi_{A_{1,\delta}}(\delta\texttt{sales}) \bowtie_{storeID,itemID} V_1) \\
\triangle(V_1) &= \\
&\quad \Pi_{\{(r.a+l.a)|a\in A_1\}}(\sigma_{r.NumSISales+l.NumSISales>0}(\pi_{A_{1,\delta}}(\delta\texttt{sales}) \bowtie_{storeID,itemID} V_1)) \\
&\qquad \uplus \sigma_{l.NumSISales>0}(\pi_{A_{1,\delta}}(\delta\texttt{sales}) \overline{\rhd\!\!<}_{storeID,itemID} V_1) \\
\text{Let } A_2 &= \{\text{city}, \text{SumCiSales}, \text{NumCiSales}\} \\
\text{Let } A_{2,ins} &= \{\text{city}, \text{SumSISales}, \text{NumSISales}\} \\
\text{Let } A_{2,del} &= \{\text{city}, \text{SumSISales} = -\text{SumSISales}, \text{NumSISales} = -\text{NumSISales}\} \\
\text{Let } A_{2,\delta} &= \{\text{city}, \text{SumCiSales} = \text{sum}(\text{SumSISales}), \text{NumCiSales} = \text{sum}(\text{NumSISales})\} \\
\delta_2 &= \Pi_{A_{2,ins}}(\triangle V_1 \bowtie \texttt{stores}) \uplus \Pi_{A_{2,del}}(\triangledown V_1 \bowtie \texttt{stores}) \\
\triangledown(V_2) &= \Pi_{r.a|a\in A_2}(\pi_{A_{2,\delta}}(\delta_2) \bowtie_{storeID,itemID} V_2) \\
\triangle(V_2) &= \Pi_{\{(r.a+l.a)|a\in A_2\}}(\sigma_{r.NumCiSales+l.NumCiSales>0}(\pi_{A_{2,\delta}}(\delta_2) \bowtie_{city} V_2)) \\
&\qquad \uplus \sigma_{l.NumCiSales>0}(\pi_{A_{2,\delta}}(\delta_2) \overline{\rhd\!\!<}_{city} V_2) \\
\text{Let } A_3 &= \{\text{category}, \text{SumCaSales}, \text{NumCaSales}\} \\
\text{Let } A_{3,ins} &= \{\text{category}, \text{SumSISales}, \text{NumSISales}\} \\
\text{Let } A_{3,del} &= \{\text{category}, \text{SumSISales} = -\text{SumSISales}, \text{NumSISales} = -\text{NumSISales}\} \\
\text{Let } A_{3,\delta} &= \{\text{category}, \text{SumCaSales} = \text{sum}(\text{SumSISales}), \text{NumCaSales} = \text{sum}(\text{NumSISales})\} \\
\delta_3 &= \Pi_{A_{3,ins}}(\triangle V_1 \bowtie \texttt{items}) \uplus \Pi_{A_{3,del}}(\triangledown V_1 \bowtie \texttt{items}) \\
\triangledown(V_3) &= \Pi_{3.a|a\in A_3}(\pi_{A_{3,\delta}}(\delta_3) \bowtie_{storeID,itemID} V_3) \\
\triangle(V_3) &= \Pi_{\{(r.a+l.a)|a\in A_3\}}(\sigma_{r.NumCaSales+l.NumCaSales>0}(\pi_{A_{3,\delta}}(\delta_3) \bowtie_{category} V_3)) \\
&\qquad \uplus \sigma_{l.NumCaSales>0}(\pi_{A_{3,\delta}}(\delta_3) \overline{\rhd\!\!<}_{category} V_3)
\end{aligned}
$$

Figure 4.1: Maintenance expressions derived using techniques in [Qua97].

Our Techniques

The approach proposed in this chapter is the following. Instead of computing and propagating insertions and deletions beyond an aggregate node V_1, we compute and propagate a change table for V_1. A *change table* is a general form of summary-delta tables introduced in [MQM97]. We show that propagation of change tables yields very efficient and simple maintenance expressions for general view expressions involving aggregate and outerjoin operators. The change table cannot be simply inserted or deleted from the materialized view. Rather, the change table must be applied to the materialized view using a special "refresh" operator. In Section 4.4, we define a `refresh` operator that is used to refresh a view using its change table. The symbol $\sqcup_{\theta}^{U}$ is used to denote the refresh operator, where θ and U are its parameters specifying join conditions and update functions respectively. Also, the change table of a view V is denoted by $\square V$.

For our example, we start with computing the change table $\square V_1$ that summarizes the net changes to V_1. For this first level of aggregates, the expression that computes $\square V_1$ is similar to that derived in [MQM97]. The change table $\square V_1$ is computed from the insertions and deletions into `sales` by using the same generalized projection (aggregation) as that used for defining V_1. More precisely, $\square V_1 =$

$$\pi_{storeID,itemID,SumSISales=sum(price),NumSISales=sum(\_count)}(\Pi_{storeID,price,\_count=1}(\sigma_q(\triangle\texttt{sales})) \uplus \Pi_{storeID,\ price=\ -price,\ \_count=\ -1}(\sigma_q(\triangledown\texttt{sales}))),$$ where q is $(date > 1/1/95)$

Figure 4.2 presents an instance of the base relation `sales` and the table $\triangle$`sales`, which is the set of insertions into `sales`. For the given tables, Figure 4.2 also shows the computed table $\square V_1$. Next we propagate the change table $\square V_1$ upwards to derive expressions for the change tables $\square V_2$ and $\square V_3$.

$$\square V_2 = \pi_{city,SumCiSales=sum(SumSISales),NumCiSales=sum(NumSISales)}(\square V_1 \bowtie \texttt{stores})$$

$$\square V_3 = \pi_{category,SumCaSales=sum(SumSISales),NumCaSales=sum(NumSISales)}(\square V_1 \bowtie \texttt{items})$$

Figure 4.2 shows the change table $\square V_3$ for the given instance of the base table `items`. The change table $\square V_2$ can be computed similarly. The new propagated change tables are then used to refresh their respective materialized views V_2 and V_3 using the refresh equations below (disregard θ and U for now). The details of the refresh equations are given in Example 7. Note that, as V_1 is not materialized, it does not need to be refreshed.

$$V_2 = V_2 \sqcup_{\theta_2}^{U_2} \square V_2, \text{ and } V_3 = V_3 \sqcup_{\theta_3}^{U_3} \square V_3$$

`sales` Table

storeID	itemID	date	price
NY1	I1	1/3/95	110
NY2	I2	7/3/95	200
NY2	I2	3/1/95	280
SJ1	I1	1/1/95	100

Insertions into `sales` ($\triangle$`sales`)

storeID	itemID	date	price
NY1	I1	1/4/95	70
NY1	I1	1/5/95	50
NY2	I2	7/3/95	50
NY3	I3	1/1/92	100
MA1	I4	7/3/95	50

`stores` Table

storeID	city	state
NY1	New York	NY
SJ1	San Jose	CA
NY3	Buffalo	NY
MA1	Boston	MA

`items` Table

itemID	category	cost
I1	C1	50
I1	C2	50
I2	C3	80
I2	C1	80

`SISales` Change Table ($\square V_1$)

storeID	itemID	SumSISales	NumSISales
NY1	I1	120	2
NY2	I2	50	1

`CategorySales` (V_3) View

category	SumCaSales	NumCaSales
C1	690	4
C2	210	2
C3	480	2

$\sqcup_{\theta}^{U}$

`CategorySales` Change Table ($\square V_3$)

category	SumCaSales	NumCaSales
C1	170	3
C2	120	2
C3	50	1

$=$

Refreshed `CategorySales` (V_3) View

category	SumCaSales	NumCaSales
C1	860	7
C2	330	4
C3	530	3

Figure 4.2: Computing change tables and refreshing the `CategorySales` (V_3) View

`InfoStores` View

state	Area (in thousand sq. miles)	Population (in millions)
NY	100	25
CA	150	35

`SSFullInfo` (V_5) View

storeID	itemID	date	price	st.storeID	st.city	st.state	IS.state	IS.area	IS.pop
NY1	I1	1/3/95	110	NY1	New York	NY	NY	100	25
NY2	I2	7/3/95	200	NULL	NULL	NULL	NULL	NULL	NULL
NY2	I2	3/1/95	280	NULL	NULL	NULL	NULL	NULL	NULL
SJ1	I1	1/1/95	100	SJ1	San Jose	CA	CA	150	35
NULL	NULL	NULL	NULL	NY3	Buffalo	NY	NY	100	25
NULL	NULL	NULL	NULL	MA1	Boston	MA	NULL	NULL	NULL

`SSFullInfo` Change Table ($\square V_5 = (\nabla$`sales` $\overset{lo}{\bowtie}$ `stores`$) \overset{lo}{\bowtie}$ `InfoStores`$)$

storeID	itemID	date	price	st.storeID	st.city	st.state	IS.state	IS.area	IS.pop
NY1	I1	1/4/95	70	NY1	New York	NY	NY	100	25
NY1	I1	1/5/95	50	NY1	New York	NY	NY	100	25
NY2	I2	7/3/95	50	NULL	NULL	NULL	NULL	NULL	NULL
NY3	I3	1/1/92	100	NY3	Buffalo	NY	NY	100	25
MA1	I4	7/3/95	50	MA1	Boston	MA	NULL	NULL	NULL

Refreshed `SSFullInfo` (V_5)

storeID	itemID	date	price	st.storeID	st.city	st.state	IS.state	IS.area	IS.pop
NY1	I1	1/3/95	110	NY1	New York	NY	NY	100	25
NY2	I2	7/3/95	200	NULL	NULL	NULL	NULL	NULL	NULL
NY2	I2	3/1/95	280	NULL	NULL	NULL	NULL	NULL	NULL
SJ1	I1	1/1/95	100	SJ1	San Jose	CA	CA	150	35
NY3	I3	1/1/92	100	NY3	Buffalo	NY	NY	100	25
NY1	I1	1/4/95	70	NY1	New York	NY	NY	100	25
NY1	I1	1/5/95	50	NY1	New York	NY	NY	100	25
NY2	I2	7/3/95	50	NULL	NULL	NULL	NULL	NULL	NULL
MA1	I4	7/3/95	50	MA1	Boston	MA	NULL	NULL	NULL

Figure 4.3: Refreshing the outerjoin view expression `SSFullInfo` (V_5)

We illustrate the refresh operation by showing how the `CategorySales` view (V_3) is refreshed. Figure 4.2 shows the materialized table V_3 = `CategorySales` for the given instance of base tables. For each tuple $\square v_3$ in $\square V_3$, we look for a match in V_3 using the join condition V_3.category = $\square V_3$.category (specified in θ_3). The tuple $\square v_3$ = <C1,170,3> in $\square V_3$ matches with the tuple v_3 = <C1,690,4> of V_3. The tuple <C1,170,3> in $\square V_3$ means that three more sales totaling \$170 have occurred for category C1. The total number of sales for C1 is now 7 for a total amount of \$860. To reflect the change, the tuple v is updated to <C1,860,7> by adding together the corresponding aggregated attributes (specified in the U_3 parameter of the refresh operator).

To compute the number of tuple accesses in Table 4.1, note that most of the computation is done in computing $\square V_1$, which requires 10,000 tuple accesses to read $\triangle$`sales`. Given the small sizes of $\square V_1$, `items`, and `stores`, the rest of the computation can be done in main memory, and hence the total number of tuples accesses is 10,000 (to read $\triangle$`sales`) + 11,000 (to read `items` and `sales`) + 2,020 (to refresh V_2 and V_3) = 23,020, showing that our technique is very efficient in comparison to previous approaches.

Outerjoins

The change-table technique can also be used for maintenance of view expressions involving outerjoin operators. Outerjoin views are supported by SQL and are commonly used in practice, such as for data integration [GJM96]. We illustrate our technique for maintaining outerjoin views by deriving maintenance expressions for outerjoin views V_4 and V_5. The net changes to V_4, in response to insertions $\triangle$`sales` into `sales`, can be succinctly summarized in a change table $\square V_4$. The change table $\square V_4$ is computed and then propagated up as follows.

$$\begin{aligned} \square V_4 &= \triangle\texttt{sales} \overset{lo}{\bowtie}_{\texttt{sales}.storeID=\texttt{stores}.storeID} \texttt{stores} \\ \square V_5 &= \square V_4 \overset{lo}{\bowtie}_{\texttt{stores}.state=\texttt{InfoStores}.state} \texttt{InfoStores} \\ V_5 &= V_5 \sqcup^{U_5}_{\theta_5} \square V_5 \end{aligned}$$

Recall that $\overset{lo}{\bowtie}$ denotes the left-outerjoin operator. Figure 4.3 shows the materialized view V_5 and the change tables $\square V_4$ and $\square V_5$ for the database instance of Figure 4.2. The refresh of V_5 proceeds as follows. Each tuple in $\square V_5$ is matched with tuples in V_5 that have the same `stores` and `InfoStores` attributes, but have all `NULL`'s in the attributes of `sales`. This join condition used for matching is specified in θ_5. In our example of Figure 4.3, one of the matching pairs is $(\square v_5, v_5)$, where $\square v_5$ = <NY3, I3, 1/1/92, 100, NY3, Buffalo, NY,

NY, 100, 25> $\in \square V_5$ and v_5 = <NULL, NULL, NULL, NULL, NY3, Buffalo, NY, NY, 100, 25> $\in V_5$. The match results in an update of v_5 to $\square v_5$, according to the update specifications in U_5. Similarly, the other matching pair <NULL, NULL, NULL,NULL, MA1, Boston, MA, NULL, NULL, NULL> $\in V_5$, and <MA1, I4, 7/3/95, 50, MA1, Boston, MA, NULL, NULL, NULL> $\in \square V_5$ is handled. The remaining unmatched tuples in $\square V_5$ are inserted into the view V_5. The refreshed view V_5 is shown in Figure 4.3.

Given the small sizes of `stores`, `InfoStores`, ∇`sales`, and $\square V_4$, the number of tuple accesses required to compute the change tables and refresh V_5 is 10,000 (to read $\triangle$`sales`) + 1,000 (to read `stores`) + 100 (to read `InfoStores`) + 20,000 (to refresh V_5).

Ours is the first work to address maintenance of general view expressions involving outerjoin operators. Recently, in work done concurrently with ours, [GK98] also reports an algorithm to handle view expressions involving outerjoins, extending previous work on maintenance of outerjoin views in [GJM97]. [GK98] uses insertion and deletion sets to propagate changes through outerjoin operators. Thus, insertions in `sales` result in insertions and deletions at `SSInfo`, which in turn result in insertions and deletions at `SSFullInfo`. However, according to the change propagation equations in [GK98], in order to compute the insertions and deletions at `SSFullInfo`, we have to compute the intermediate view `SSInfo`, thereby incurring more than a billion tuple accesses. □

4.3 The Change-Table Technique for View Maintenance

In this section, we explain the framework developed in [QW91, GL95] for deriving incremental view maintenance expressions and relate it to the change-table technique developed in this chapter.

Let a database contain a set of relations $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. A *change transaction* t is defined to contain for each relation R_i the expression $R_i \leftarrow (\ R_i \ \dot{-}\ \nabla R_i) \uplus \triangle R_i$, where ∇R_i is the set of tuples to be deleted from R_i, and $\triangle R_i$ is the set of tuples to be inserted into R_i. Let V be a bag-algebra expression defined on a subset of the relations in $\mathcal{R}$. The *refresh-expression* $New(V,t)$[5] is used to compute the new value of V. Griffin and Libkin in [GL95] define the expression New(V,t) to be:

$$\text{New}(V,t) = (V \ \dot{-}\ \nabla(V,t)) \uplus \triangle(V,t).$$

[5] [GL95] uses the notation pre(t,V) instead.

So, deriving view maintenance expressions for a view V entails deriving two functions $\nabla(V,t)$ and $\triangle(V,t)$ such that for any transaction t, the view V can be maintained by evaluating

$$(V \dot{-} \nabla(V,t)) \uplus \triangle(V,t).$$

In order to derive $\nabla(V,t)$ and $\triangle(V,t)$, [GL95] gives *change propagation equations* that show how deletions and insertions are propagated up through each of the relational operators. The work of [GL95] was extended to include aggregate operators by Quass in [Qua97].

The change-table technique presented in this chapter can be thought of as introducing a new definition for New(V,t). We define the expression New(V,t) for view expressions involving aggregates and outerjoins as

$$\text{New}(V,t) = (V \sqcup_{\theta}^{U} \square(V,t)),$$

where $\square(V,t)$ is called the *change table*, $\sqcup_{\theta}^{U}$ is the `refresh` operator used to apply the net changes in a change table to its view, and (θ, U) are the parameters of the `refresh` operator. The parameter θ specifies the join conditions on the basis of which the tuples from the change table and the view are matched, and U specifies the functions that are used to update the matched tuples in V.

The new definition of New(V,t) is motivated from the following observation. In the case of general view expressions involving aggregate operators, it is usually more efficient to propagate the change tables beyond an aggregate operator, instead of propagating insertions and deletions. Propagation of a change table is particularly efficient when the change table depends only on the changes to the base relation (self-maintainability [GJM96]), while the insertions and deletions depend on the old value of the view. As we showed in the motivating example, if the aggregate node is not materialized, the computation of insertions and deletions could be very expensive.

The new definition of New(V,t) means that we need to define a general refresh operator, and derive change propagation equations for propagating change tables through various relational, aggregate and outerjoin operators, to obtain a complete technique for efficient incremental maintenance of general view expressions. In the following section, we present a formal definition of the `refresh` operator. In later sections, we derive change propagation equations for general view expressions involving aggregate and outerjoin operators.

4.4 The `refresh` Operator

In this section, we give a formal treatment of the refresh operator. Given a materialized table V and its change table $\square V$, `refresh` applies the changes represented in a change table. The binary `refresh` operator is a generalization of the refresh algorithm used in [MQM97] and can be implemented using the `modify` operation discussed in [QM97].

We denote the `refresh` operator by $\sqcup_{\theta}^{U}$, where θ is a pair of two mutually exclusive join conditions and U is a list of update function specifications. The `refresh` operator takes two operands, a view V to be updated, and a corresponding change table (denoted by $\square V$).

Let V and $\square V$ be views with the same attribute names. The subscript θ associated with the operator is a pair of join conditions $\mathcal{J}_1$ and $\mathcal{J}_2$. The update list U is a specification of how the attributes are updated. In an expression $V \sqcup_{\theta}^{U} \square V$, each tuple $\square v$ of $\square V$ is checked for possible matches (due to $\mathcal{J}_1$ or $\mathcal{J}_2$) with tuples in V. If a match is found due to the join condition $\mathcal{J}_1$, then the corresponding matching tuple v of V is changed using the specifications in the update list U (as described in the next paragraph). If the match is due to $\mathcal{J}_2$, the tuple v of V is deleted. The unmatched tuples in $\square V$ are inserted into V. The matching done is *one-to-one* in the sense that a tuple $\square v \in \square V$ matches with at most one tuple in V and vice-versa. If $\square v$ finds more than one match in V, then an arbitrary matching tuple from V is picked.

The tuple v of V matching with the tuple $\square v$ of $\square V$ due to join condition $\mathcal{J}_1$ is updated as follows. Let $U = <(B_1, f_1), (B_2, f_2), \ldots, (B_k, f_k)>$, where $B_1, \ldots, B_k$ are attributes of V and $f_1, \ldots, f_k$ are binary functions. For *each* pair (B_j, f_j) in U, the B_j attribute of v is changed to $f_j(v(B_j), \square v(B_j))$, where $v(X)$ and $\square v(X)$ denote the values of the X attribute of v and $\square v$ respectively.

EXAMPLE 6 Consider the view `CategorySales` (V_3) defined in Example 5. The table `CategorySales` as defined in Example 5 computes the total sales for each category. In this example, we illustrate the refresh operation by applying the changes summarized in a change table $\square$`CategorySales` to its view `CategorySales` using the `refresh` operator.

For this example, we consider the instance of the base table shown in Figure 4.2. The figure also shows the materialized table `CategorySales` for the given instance. In response to the insertion of the table $\triangle$`sales` into the base table `sales`, the change table $\square$`CategorySales` can be computed and is shown in the figure. The view `CategorySales`

is refreshed using the expression `CategorySales` $\sqcup_{\theta_3}^{U_3}$ `□CategorySales`, where the parameters, $\theta_3 = (\mathcal{J}_1, \mathcal{J}_2)$ and U_3, of the `refresh` operator are defined as follows.

- $\mathcal{J}_1$ is
  ($\equiv_{category}$ $\wedge$ ((`CategorySales`.NumCaSales +(`□CategorySales`).NumCaSales) $\neq$ 0))
- $\mathcal{J}_2$ is
  ($\equiv_{category}$ $\wedge$ ((`CategorySales`.NumCaSales +(`□CategorySales`).NumCaSales) = 0))
- U_3 = <(SumCaSales, f), (NumCaSales, f)>, where $f(x, y) = x + y$ for any x, y.

Note that $\equiv_{category}$ represents the predicate

$$(\texttt{CategorySales}.category = (\texttt{□CategorySales}).category)$$

here. Now, we try to run the refresh operation on the view `CategorySales` and its change table `□CategorySales` of Figure 4.2.

The first tuple $\square v_3$ = <C1,170,3> of `□CategorySales` matches with the tuple v_3 = <C1,690,4> in `CategorySales` using the join condition $\mathcal{J}_1$. The match results in update of the tuple <C1,690,4> in `CategorySales` according to the specifications in the update list U_3. The attribute *SumCaSales* of the tuple v_3 is changed to $\square v_3(SumCaSales) + v_3(SumCaSales) = 170 + 690 = 860$ and the attribute *NumCaSales* is changed to $4 + 3 = 7$. Similarly, the tuple <C2,210,2> $\in$ `CategorySales` matches with the tuple <C2,120,2> of `□CategorySales` and is updated to <C2,330,4>. The tuple <C3,480,2> of `CategorySales` is also updated to <C3,530,3> accordingly.

To illustrate deletion from the view `CategorySales`, let us assume that the change table `□CategorySales` contains a tuple b = <C2,-210,-2> as a result of a deletion of a couple of tuples from the `sales` table. The tuple b = <C2,-210,-2> will match with the tuple v = <C2,210,2> in V_3 due to the join condition $\mathcal{J}_2$, and the match will result in deletion of v from `CategorySales`. □

Implementation of the `refresh` operator One simple way to implement the `refresh` operator is to use a nested loop algorithm, with the change table as the outer table, and the materialized view table as the inner table. The algorithm is shown in Algorithm 9. The nested loop algorithm is just one possible way to implement the `refresh` operator. In fact, Quass and Mumick [QM97] show that the refresh operation can be implemented more efficiently by using existing outerjoin methods inside the DBMS. [QM97] defines a `modify` operator using a pair of join conditions and a series of condition-action lists. [QM97]

shows that the `modify` operator can be easily implemented in a DBMS through slight modifications of existing physical query operators for outerjoins, such as hash outerjoin, sort-merge outerjoin, and nested-loop outerjoin. Cost-based optimization can be used to select the method most suitable for a given evaluation. [QM97] shows that doing the refresh operation using the `modify` operator is significantly faster than iterative algorithms. [QM97] also gives a performance analysis of several different physical implementations of the `modify` operator. The implementation techniques of the `modify` operator can also be used for the `refresh` operator used in this chapter for applying change tables.

Algorithm 9 **Refresh Algorithm**

Input

Table $V(A_1, A_2, \ldots, A_n)$. %

Change Table $\Box V(A_1, A_2, \ldots, A_n)$. %

$\theta = (\mathcal{J}_1, \mathcal{J}_2), U = <(A_{i_1}, f_1), (A_{i_2}, f_2), \ldots, (A_{i_k}, f_k)>$

Output

Refreshed table $V = V \sqcup_{\theta}^{U} \Box V$.

Method

DECLARE cursor_box CURSOR FOR
    SELECT $A_1, A_2, \ldots, A_n$ FROM $\Box V$;
OPEN cursor_box;
LOOP
    FETCH cursor_box INTO $:\Box a_1, :\Box a_2, \ldots, :\Box a_n$;
UNTIL *not-found*
    DECLARE cursor_view CURSOR FOR
        SELECT $A_1, A_2, \ldots, A_n$ FROM V
        WHERE ($\mathcal{J}_1(A_1, A_2, \ldots, A_n, :\Box a_1, :\Box a_2, \ldots, :\Box a_n)$ OR
                $\mathcal{J}_2(A_1, \ldots, A_n, :\Box a_1, \ldots, :\Box a_n)$) AND (NOT `changed`);
    OPEN cursor_view;
    LOOP
        FETCH cursor_view INTO $:a_1, :a_2, \ldots, :a_n$;
    UNTIL *not-found*
    IF *not-found*
        INSERT INTO V VALUES $(:\Box a_1, :\Box a_2, \ldots, :\Box a_n)$;
    ELSE-IF $\mathcal{J}_2(:a_1, :a_2, \ldots, :a_n, :\Box a_1, :\Box a_2, \ldots, :\Box a_n)$

```
            DELETE FROM V WHERE CURRENT OF cursor_view;
            break;
        ELSE
            UPDATE V SET update attributes =
                    function of update attributes from cursor_box and cursor_view;
            Mark the tuple at the CURRENT OF cursor_view as changed.
            break;
        ENDIF
        ENDLOOP
        CLOSE cursor_view;
    ENDLOOP
    CLOSE cursor_box;                                                    ◇
```

4.5 Propagating Change Tables Generated at Aggregate Nodes

In this section, we show how to generate a change table at an aggregate node, and derive change-propagation equations used to propagate these change tables through various operators. We start with a few definitions.

Definition 9 (**Aggregate-change table**) A change table for a view expression involving aggregates is defined as an *aggregate-change table*[6] if the change table either originated at an aggregate operator or is a result of propagation of a change table that originated at an aggregate node, using the propagation equations we will present in Table 4.2. □

For example, the change tables, □`SISales`, □`CitySales`, and □`CategorySales`, computed for the views `SISales`, `CitySales`, and `CategorySales` respectively in Example 5 are aggregate-change tables.

Definition 10 (**Distributive functions**) An (aggregate) function is defined as *distributive* if it can be computed by partitioning the input parameters into disjoint sets of parameters, aggregating each set individually, then further aggregating the (partial) results from each

[6]The notions of aggregate-change table and outerjoin-change table (introduced later in Section 4.6) have been defined only for simplying the presentation of the material in this chapter.

set into the final result. For eg., `sum`, `max`, and `min` are distributive aggregate functions, while `average` is not. □

Definition 11 (**Self-maintainable aggregates**) An aggregate function is defined to be *self-maintainable* with respect to a change transaction t if the new value of the function can be computed solely from the old value of the function and the change transaction t to the base data. For eg., `sum` is self-maintainable with respect to insertions as well as deletions. The aggregate functions `min`, `max` are self-maintainable with respect to insertions but not with respect to deletions. □

The notions of aggregate-change table and outerjoin-change table (introduced later in Section 4.6) have been defined only for simplying the presentation of the material in this chapter. We will show that a restricted definition of the general `refresh` operator is sufficient to refresh a view using its aggregate-change table. The restricted form yields very simple change-propagation equations for propagation of aggregate-change tables through various operators.

We use the notation $Attrs(\varphi)$ to represent the set of attributes referenced in φ. Thus, Attrs(U) refers to the set of attributes specified in the update list U and Attrs(θ) represents Attrs($\mathcal{J}_1$) ∪ Attrs($\mathcal{J}_2$), where $\mathcal{J}_1$ and $\mathcal{J}_2$ are the join conditions in $\theta = (\mathcal{J}_1, \mathcal{J}_2)$.

4.5.1 Generating the Aggregate-Change Table at an Aggregate Node

Consider a view V defined as an aggregation over a select-project-join (SPJ) expression. In this section, we give a brief description of how an aggregate-change table is generated at V in response to the insertions and deletions at the base tables. The method is similar to that of generating a "summary-delta table" for a "summary table" [MQM97].

For the case when a view V is defined as an aggregation over an SPJ expression, the insertions and deletions to the base tables can be propagated to V as a single aggregate-change table, which we denote by $\Box V$. Without loss of generality, let us assume V to be $\pi_{G,B=f(A)}(R)$, where R is the SPJ subview, G is the set of group-by attributes, f is an aggregate function, and A is an attribute of R. In the case of self-maintainable aggregate functions, the aggregate-change table $\Box V$ can be computed from the insertions and deletions into R by using the same generalized projection as that used for defining the view V. More precisely, $\Box V$ can be computed as

$$\Box V = \pi_{G,f(A),Count=sum(\_count)}(\Pi_{G,A,\_count=1}(\triangle R) \uplus \Pi_{G,A=N(A),\_count=-1}(\triangledown R)),$$

where the function N is suitably defined depending on the aggregate function f. For example, in the case of a sum aggregate and the count aggregate, the function N negates the attribute value passed.

If the aggregate function f is self-maintainable (i.e., the new value can be computed from the old value of the function and the changes to the base data), then the special form of `refresh` operator defined in the next subsection is sufficient to apply the change table $\square V$ to the view V. For the case of aggregate functions that are not self-maintainable, we need more complex functions in the update list U of the `refresh` operator. For example, to handle deletions from a subview of a simple MIN/MAX aggregate view, the update function needs to compute the new MIN/MAX value, if needed, by accessing the base relations (see [MQM97] for details). The change propagation equations and other formalisms presented in this chapter are independent of the complexity and input parameters of update functions used in the update list U of `refresh`.

In a general view expression tree V, an aggregate-change table is generated at the first aggregate-node, and then propagated upwards through various operators to V. In the next subsections, we define a restricted form of `refresh` operator used to apply aggregate-change tables, and then derive change propagation equations for propagating aggregate-change tables.

4.5.2 `Refresh` Operator for Applying Aggregate-Change Tables

In this section, we define the characteristics of the "aggregate-refresh" operator that is used to apply an aggregate-change table to its view. The "aggregate-refresh" operator is a special case of the generic `refresh` operator defined in Section 4.4. In the next subsection, we derive simple change propagation equations using these special characteristics.

Recall that the expression used to refresh a view V using its change table $\square V$ is: $V = V \ \sqcup_{\theta}^{U} \ \square V$, where $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ and U is the update list. In the case of an `aggregate-refresh` operator used to apply aggregate-change tables, $\mathcal{J}_1$ is $(\equiv_G \ \wedge \neg p)$ and $\mathcal{J}_2$ is $(\equiv_G \ \wedge \ p)$, for some predicate p and a set of attributes G common to both V and $\square V$. As defined before, the notation $\equiv_G$ here represents the predicate $\bigwedge_{g \in G}(V.g = \square V.g)$, as V and $\square V$ are the left and right operands of the join operator. Also, the set of attributes G is disjoint from the set of attributes, Attrs(U), that are being updated. The predicate p specifies when the matching tuple in V is to be deleted, i.e., when the value of the attribute that stores the number of deriving base tuples becomes zero. The above characteristics

are summarized in the definition of an `aggregate-refresh` operator below.

Definition 12 (`Aggregate-refresh` **Operator**) A `refresh` operator $\sqcup_{\theta}^{U}$, where $\theta = (\mathcal{J}_1, \mathcal{J}_2)$, is said to be an `aggregate-refresh` operator if, for some predicate p and a set of attributes G common to both the view and its aggregate-change table,

(1) The join conditions $\mathcal{J}_1$ and $\mathcal{J}_2$ can be represented as:

- $\mathcal{J}_1$: $(\equiv_G \ \wedge \neg p)$
- $\mathcal{J}_2$: $(\equiv_G \ \wedge \ p)$, and

(2) $G \ \cap \text{Attrs}(U) = \phi$. □

The characteristics of the `aggregate-refresh` operator are used to derive simple change propagation equations for propagating aggregate-change tables.

4.5.3 Propagating Aggregate-Change Tables

For the purposes of change propagation equations shown in Table 4.2, we assume that an aggregate-change table has been *generated* (as shown in Section 4.5.1) at the first aggregate operator in a view expression, in response to insertions and/or deletions at a base relation.[7] Table 4.2 gives change propagation equations for propagating (already generated) aggregate-change tables through relational, aggregate and outerjoin operators. In Theorem 13, we will prove the correctness of the change propagation equations of Table 4.2.

Each row in Table 4.2 considers propagation of an aggregate-change table through a relational, aggregate, or outerjoin operator. The first column gives the equation number used for later reference in examples. The second column in the table gives the outermost operator used in the definition of V. Consider a change transaction t consisting of the following change: $E_1 \leftarrow (E_1 \sqcup_{\theta}^{U} \square E_1)$, where E_1 is a subexpression, $\square E_1$ is an aggregate-change table and $\sqcup_{\theta}^{U}$ is an `aggregate-refresh` operator. We assume θ to be $(\mathcal{J}_1, \mathcal{J}_2)$, where $\mathcal{J}_1$ is $\equiv_G \ \wedge \neg p$ and $\mathcal{J}_2$ is $\equiv_G \ \wedge \ p$, for some predicate p and a set of attributes G in E_1. The third column expresses the new expression for V due to the change t, by replacing E_1 in V by $(E_1 \sqcup_{\theta}^{U} \square E_1)$.[8] The fourth column gives the refresh equation $New(V, t)$ that is used to refresh the view V. Theorem 13 proves that the refresh equations ($New(V, t)$ of fourth column) are correct, by showing that they are equivalent to the expressions in

[7] When two or more base relations are updated simultaneously (or a relation appears more than once in a view expression), we handle the updates in a sequence.

[8] The case of changes occurring at *both* the subexpressions E_1 and E_2 can be handled by first propagating changes due to E_1, followed by propagating changes due to E_2.

No.	V	New V $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ $\mathcal{J}_1$ is $\equiv_G \wedge \neg p$ $\mathcal{J}_2$ is $\equiv_G \wedge p$	Refresh Equation	$\square V$	Conditions
1	$\sigma_q(E_1)$	$\sigma_q(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \sigma_q(\square E_1)$	$\sigma_q(\square E_1)$	$\text{Attrs}(p) \subseteq G$
2	$\Pi_A(E_1)$	$\Pi_A(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \Pi_A(\square E_1)$	$\Pi_A(\square E_1)$	$\text{Attrs}(\theta) \subseteq A$
3	$E_1 \times E_2$	$(E_1 \sqcup_\theta^U \square E_1) \times E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \times E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \times E_2$	
4	$E_1 \bowtie_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \bowtie_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \bowtie_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \bowtie_J E_2$	$\text{Attrs}(J) \subseteq G$
5	$E_1 \uplus E_2$	$(E_1 \sqcup_\theta^U \square E_1) \uplus E_2$	$((V \dot{-} E_2) \sqcup_\theta^U \square E_1) \uplus E_2)$	$\square E_1$	
6	$\pi_{G',F}(E_1)$ $F = f_1(A_1),$ $\ldots, f_k(A_k).$	$\pi_{G',F}(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_{\theta_1}^{U_1} \pi_{G',f(A)}(\square E_1)$ $\theta_1 = (\equiv_{G'} \wedge \neg p_1, \equiv_{G'} \wedge p_1)$ p_1 is $(V.Cnt + \square V.Cnt) \neq 0$ $U_1 = <(A_1, f_1), \ldots, (A_k, f_k)>$	$\pi_{G',F}(\square E_1)$	$A_i \in \text{Attrs}(U)$, $G' \subseteq G$, f_i's are distributive, and the function in U for A_i is f_i.
7	$E_1 \overset{fo}{\bowtie}_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \overset{fo}{\bowtie}_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \overset{lo}{\bowtie}_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_1)$	$(\square E_1 \overset{lo}{\bowtie}_J E_2)$	$\text{Attrs}(J) \subseteq G$

Table 4.2: Change propagation equations for propagating aggregate-change tables

the third column. Theorem 13 also proves that the `refresh` operator used in the refresh equation of fourth column is an `aggregate-refresh` operator. The fifth column gives the expression for the propagated aggregate-change table $\Box V$, which can be derived from the refresh equation of the fourth column. Finally, the last column of the table states the conditions under which the equivalence of the fourth column and third column expressions holds, i.e., conditions under which the change propagation can be done. If the condition is not satisfied, then the refresh equation cannot be used to propagate the aggregate-change table. We will show later how to handle changes at an operator node when the conditions in the sixth column are not satisfied.

The *first row* of the table depicts the case of a selection view $V = \sigma_q(E_1)$, where E_1 is a subexpression. In this case, the expression for the propagated aggregate-change table is: $\Box V = \sigma_q(\Box E_1)$, as shown in the fifth column. For the case of selection view, the condition required for the change propagation is $\text{Attrs}(p) \subseteq G$. In other words, an aggregate-change table can be propagated through a selection operator if the selection condition is defined over the G attributes, which are the attributes that are not being updated by the update functions of the `refresh` operator.

The *second row* depicts the case of projection on a set of attributes A. The condition implies that an aggregate-change table can be propagated through a duplicate-preserving projection only if all the attributes used in the join conditions of θ are included in the set A.

The *third row* considers the case of a cross product operation. In this case, the parameter θ of the refresh operator is changed also to include the condition $\equiv_{Attrs(E_2)}$ in the join conditions $\mathcal{J}_1$ and $\mathcal{J}_2$. No conditions are specified in the sixth column, meaning that an aggregate-change table can always be propagated through a cross product operator.

The *fourth row*, depicting the case of a join operation, follows from the combination of previous cases of selection and cross product.

The *fifth row* considers propagation of an aggregate-change table through the bag union operator. The expression for $\Box V$ in the fifth column gives only the change table ($\Box V = \Box E_1$). As the refresh equation in the fourth column shows, the refresh in this case is more complex than simply applying the change table. We need to first apply a set of deletions ($\nabla V = E_2$), followed by refreshing the result with the change table ($\Box V = \Box E_1$), followed by inserting the set ($\triangle V = E_2$) into the result. One can derive a very efficient refresh equation for the case of bag union operator, if the tuples in V are "tagged" L or R depending on

whether they come from the left operand E_1 or the right operand E_2. We omit the trivial details here.

The *sixth row* depicts the case of propagation through a generalized projection (aggregate) operator. For the purposes of aggregate-change tables, we assume that any subexpression involving an aggregate operator stores with each tuple a count of the number of deriving base tuples. This count is stored in a general attribute which we will call the *count* attribute. For example, NumCiSales and NumCaSales are count attributes in the views `CitySales` and `CategorySales` of Example 5. After propagation through the aggregate operator, the join conditions and the update specifications of the `aggregate-refresh` operator change as shown in the fourth column. The attribute named *Cnt*, used in defining p_1 in the join conditions of θ_1 represents the count attribute of V. The condition in the sixth column says that each aggregate function f_i is distributive, the set of group-by attributes G' is a subset of G, and the update list U used to change E_1 contains (A_i, f_i) for all $1 \leq i \leq k$. Note that the duplicate-elimination operation is a special case of generalized projection, and is covered by the sixth row.

The *seventh row* gives the refresh equation for the case of propagating an aggregate-change table through a full outerjoin operation. In this case, the `refresh` operator specifications change as in the case of cross product. For simplicity, we have assumed that the aggregate-change table $\square E_1$ doesn't result in any deletions from E_1. Otherwise, a more extended refresh operator is required as discussed in Section 4.6.3.

Note that we do not give any change propagation equation for the case of monus because monus is a singularity point (see paragraph below).

Singularity Points We refer to the operator nodes in a view expression tree, where none of the refresh equations in Table 4.2 apply, as *singularity* points. Aggregate-change tables cannot be propagated through singularity points. For example, a selection on the result of an aggregate function is a singularity point, because it will not satisfy the condition $Attr(p) \subseteq G$ given in the first row of Table 4.2.

Consider a view V and a singularity point V_1, which is a subexpression of V, in the expression tree of V. As the changes to V_1 cannot be summarized into a change table, we instead compute insertions ($\triangle V_1$) and deletions ($\triangledown V_1$) into V_1 and propagate the insertions and deletions beyond V_1. The tables $\triangle V_1$ and $\triangledown V_1$ can be easily computed

from the change table of its descendant in the expression tree. If V_1 is not materialized, we use the base relations to compute (relevant parts of) V_1 whenever required by the refresh algorithm. The insertions ($\triangle V_1$) and deletions ($\triangledown V_1$) to V_1 are then propagated through the singularity point using the change propagation equations of [GL95, Qua97]. The computed insertions and deletions at a singularity point can then be propagated further upwards in the expression tree using techniques presented in this chapter (as they may result in change tables further on). Hence, the presence of singularity points in an expression tree doesn't preclude application of our change table techniques for incremental maintenance.

Theorem 13 *Assume that the* `refresh` *operator used in the expression of the third column in Table 4.2 is an* `aggregate-refresh` *operator. Then,*

(1) the change propagation equations given in Table 4.2 for propagation of aggregate-change tables are correct, i.e., for each row, the expression in the third column is equivalent to the refresh equation in the fourth column, and

(2) the `refresh` *operator derived in the refresh equation (column 4) is an* `aggregate-refresh` *operator as well.*

Proof: We refer to the expression $E_1 \sqcup_\theta^U \square E_1$ as the *change equation* throughout this proof. As shown in the Table 4.2, the expression in the fourth column is referred to as the refresh equation.

• Selection: $V = \sigma_q(E_1)$. It is easy to see that if a tuple $v \in V$ is deleted or updated during the refresh equation of the fourth column, then $v \in E_1$ is also deleted or updated in the same manner by the change equation. And as updates do not affect any attributes in Attrs(q), the updated v is retained in $\sigma_q(E_1 \sqcup_\theta^U \square E_1)$.

In this paragraph, we show that the refresh equation indeed captures *all* the updates or deletions required. First, note that in the change equation $E_1 \sqcup_\theta^U \square E_1$ if a tuple $\square e_1 \in \square E_1$ deletes/updates a tuple $v \in \sigma_q(E_1) \subseteq E_1$, then $\square e_1 \in \sigma_q(\square E_1)$. The above is true because the matched pair $(v, \square e_1)$ should have the same G attributes, and as Attrs(p) $\subseteq G$, if v satisfies q then $\square e_1$ must also satisfies q. Thus, such a tuple $\square e_1$ in $\sigma_q(\square E_1) = \square V$ would update/delete the corresponding tuple $v \in V = \sigma_p(E_1)$ in the refresh equation. Therefore, all the updates or deletions that happen to tuples in $\sigma_q(E_1)$ due to the change equation are also captured by the refresh equation. The update to a tuple $\bar{v} \notin \sigma_q(E_1)$ due to the change equation is irrelevant, as neither $\bar{v}$ nor its updated form will satisfy the predicate q.

Let I be the set of tuples that is inserted into E_1 due to the change equation. Now, we show that $\sigma_q(I)$ is inserted into V by the refresh equation, implying that the refresh equation doesn't miss any legitimate insertions into V. Note that I is the set of tuples in $\square E_1$ that do not find a match in E_1. As $\sigma_q(I) \subseteq \sigma_q(\square E_1) = \square V$, no tuple in $\sigma_q(I)$ will find a match in $V \subseteq E_1$. Hence, $\sigma_q(I)$ will be inserted into V due to the refresh equation.

Finally, we show that all the insertions into V due to the refresh equation are legitimate. A tuple $\square v$ matches with a tuple v only if they have the same G attributes. Hence, if a tuple $\square v \in \sigma_q(\square E_1)$ is inserted into $V = \sigma_q(E_1)$ in the refresh equation (due to a lack of match in V), then $\square v$ will not find a match in E_1 also. Hence, $\square v \in \square E_1$ will be inserted into E_1 by the change equation, and as $\square v$ satisfies q, it will also be retained in $\sigma_q(E_1 \sqcup_\theta^U \square E_1)$.

• Projection: $V = \Pi_A(E_1)$. If $\text{Attrs}(\theta) \subseteq A$, then the information needed to decide the effect of a tuple $\square e_1 \in \square E_1$ on a tuple $e_1 \in E_1$, if any, is available in $\Pi_A(e_1)$. Hence, $\Pi_A(\square E_1)$ can be applied directly to $V = \Pi_A(E_1)$. Also, note that the resulting `refresh` operator is also an `aggregate-refresh` operator.

• Cross Product: $V = E_1 \times E_2$. Consider $(E_1 \times E_2) \sqcup_{\theta_1}^U (\square E_1 \times E_2)$, the refresh equation. Let us partition the tables $(E_1 \times E_2)$ and $(\square E_1 \times E_2)$ by the tuple values of E_2. As $\mathcal{J}_1$ and $\mathcal{J}_2$ in θ_1 include $\equiv_{\text{Attrs}(E_2)}$, each of the partitions is refreshed independently by the refresh equation. It is easy to see that a tuple $<\square e_1, e_2> \in \square V$ will match with a tuple $<e_1, e_2> \in V$ due to θ_1 in the refresh equation, if and only if the tuple $\square e_1 \in \square E_1$ matches with a tuple $e_1 \in E_1$ due to the parameter θ in the change equation. The matches will result in same update to the E_1 attributes or deletion in both the expressions. Also, a tuple $\square e_1 \in \square E_1$ doesn't find a match in E_1 if and only if the tuples in $(\square e_1 \times E_2) \subseteq \square V$ do not find a match in V.

It is easy to see that the refresh operator in the fourth column with the new specifications is also an `aggregate-refresh` operator.

• Join: $V = E_1 \bowtie_J E_2$. Follows from the previous cases, but stated in the table for convenience.

• Union: $V = E_1 \uplus E_2$. As $((E_1 \uplus E_2) \dot{-} E_2) = E_1$, the equivalence of the expressions is obvious.

• Aggregation: $V = \pi_{G', f(A)}(E_1)$. Without loss of generality, we prove this case when $k = 1$. Let $f_1 = f$ and $A_1 = A$, the aggregated attribute. We assume that $(G' \subseteq G), A \in \text{Attrs}(U), U = <(A, f)>$, and that f is a distributive function. Consider tuples $e_1, e_2, \ldots, e_n$ in E_1 such that they have the same G' values and their attribute A values are $a_1, a_2, \ldots, a_n$.

Also, assume that the tuple e_i matches with some tuple $\square e_i \in \square E_1$ due to $\mathcal{J}_1$ and that the aggregated attribute A value of $\square e_i$ is $\square a_i$.[9] If e_i doesn't find a match in $\square E_1$, then assume that $\square a_i$ is such that $f(a, \square a_i) = a$ for simplicity of the proof. Note that $\square e_i$'s have the same G' values too. The attribute value a_i of e_i is updated to $f(a_i, \square a_i)$ due to U in the change equation. Thus, due to the change equation the tuple $e_1, e_2, \ldots, e_n$ will result in an aggregated value of $f(f(a_1, \square a_1), f(a_2, \square a_2), \ldots, f(a_n, \square a_n))$ in the equation of the third column. Thus, we need to show that the aggregated A value of the tuple $v \in V = \pi_{G', f(A)}(E_1)$ that is derived from $e_1, \ldots, e_k$ changes from $f(a_1, a_2, \ldots, a_n)$ to $f(f(a_1, \square a_1), f(a_2, \square a_2), \ldots, f(a_n, \square a_n))$ in the refresh equation. By the definition of $\square V$, the tuples $\square e_1, \ldots, \square e_n \in \square E_1$ will be grouped to yield the aggregated attribute value $f(\square a_1, \ldots, \square a_n))$, and the refresh equation of V will change the aggregated value of the grouped value of e_is from $f(a_1, a_2, \ldots, a_n)$ to $f(f(a_1, a_2, \ldots, a_n), f(\square a_1, \ldots, \square a_n))$. As f is a distributive function, we have $f(f(a_1, \square a_1), f(a_2, \square a_2), \ldots, f(a_n, \square a_n)) = f(f(a_1, a_2, \ldots, a_n), f(\square a_1, \ldots, \square a_n))$, hence the refresh equation of V correctly updates the aggregated attribute value of the tuple v in V.

All insertions into E_1 due to $\square E_1$ in the change equation will be converted to appropriate aggregated insertions or updates into V by the refresh equation. The deletions from E_1 need not necessarily result in any deletions from V. A tuple is deleted from V only if its aggregated attributes become zero, which is independent of the deletions of the deriving tuples from E_1.

It is easy to see that the refresh operator in the fourth column with the new specifications is also an `aggregate-refresh` operator.

• Outerjoin: $V = E_1 \overset{fo}{\bowtie}_J E_2$. Suppose $\square E_1$ induces a set of insertions I into the relation E_1. Each tuple $i \in I$ results in a set of tuples $E_2^i = i \overset{lo}{\bowtie}_J E_2$ in $\square V$. No tuple $e_2^i \in E_2^i$ finds a match in V due to the predicate $(\equiv_G \wedge \equiv_\tau)$, because if it did, i would have found a match in E_1 due to $\equiv_G$. Therefore, the refresh equation results in E_2^i being inserted into V for each $i \in I$.

Let us assume that $M(\subseteq \square E_1)$ is a set of tuples that find a match in E_1 due to $\equiv_G$, and thus result in update of a tuple in E_1. Each tuple $m \in M$ results in a set of tuples $E_2^m = m \overset{lo}{\bowtie}_J E_2$ in $\square V$. Note that, E_2^m may consist of just one tuple $<m, \texttt{NULL}>$. If $m \in M$ matches with a tuple $e_1 \in E_1$ due to both having the same G attributes, then each tuple

[9] If the match is due to $\mathcal{J}_2$, then we need to show that the pair of values a_i and $\square a_i$ is such that $f(a_i, \square a_i, b) = f(b)$ for any b. Once shown, the observation can be used to easily make the rest of the argument go through.

$<m, e_2> \in E_2^m$ would match with the corresponding tuple $<e_1, e_2> \in E_2^{e_1} = e_1 \overset{lo}{\bowtie}_J E_2$. The tuple $<e_1, e_2>$ exists in $E_1 \overset{ro}{\bowtie}_J e_2$, because the pair (m, e_2) satisfies J and Attrs(J) $\subseteq G$. Note that if E_2^m consists of only $(m, \texttt{NULL})$, then $E_2^{e_1}$ consists of $<e_1, \texttt{NULL}>$ only. Also, as $E_1^{e_2} \subseteq V$, the refresh equation of V affects the updates correctly. As noted before, if $\Box E_1$ results in deletions from E_1, then the refresh equation derived here would need to be modified using a more extended refresh operator as illustrated in Section 4.6.3. ∎

EXAMPLE 7 In this example, we illustrate the techniques developed in this section on the views of the motivating Example 5. Recall from Example 5 the definitions of V_1 (`SISales`), V_2 (`CitySales`), and V_3 (`CategorySales`). We have

$$
\begin{aligned}
V_1 &= \pi_{storeID,itemID,SumSISales=sum(price),NumSISales=count(*)}(\sigma_{date>1/1/95}\texttt{sales}) \\
V_2' &= V_1 \bowtie \texttt{stores} \\
V_2 &= \pi_{city,SumCiSales=sum(SumSISales),NumCiSales=sum(NumSISales)}(V_2') \\
V_3' &= V_1 \bowtie \texttt{items} \\
V_3 &= \pi_{category,SumCaSales=sum(SumSISales),NumCaSales=sum(NumSISales)}(V_3')
\end{aligned}
$$

where the (virtual) views V_2' and V_3' have been added for better illustration of how the aggregate-change tables propagate. We use the change propagation equations of Table 4.2 to derive the maintenance expressions for V_2 and V_3, in response to changes in `sales`, as follows.

$\Box V_1 =$

$\pi_{storeID,itemID,SumSISales=sum(price),NumSISales=sum(\_count)}(\Pi_{storeID,price,\_count=1}(\sigma_q \triangle \texttt{sales})$
$\uplus\ \Pi_{storeID,\ price=\ -price,\ \_count=\ -1}(\sigma_q \triangledown \texttt{sales}))$, where q is $date > 1/1/95$.

[From Section 4.5.1]

$$
\begin{aligned}
V_1 &= V_1 \sqcup_{\theta_1}^{U} (\Box V_1), \text{ where } \theta_1 \text{ is } (\equiv_{\{storeID,itemID\}} \wedge \neg p,\ \equiv_{\{storeID,itemID\}} \wedge\ p) \\
\Box V_2' &= \Box V_1 \bowtie \texttt{stores} \\
V_2' &= V_2' \sqcup_{\theta_{12}}^{U} \Box V_2', \qquad \text{[From (4) in Table 4.2]}
\end{aligned}
$$

where θ_{12} is $(\equiv_{\{storeID,itemID\} \cup Attrs(\texttt{stores})} \wedge \neg p,\ \equiv_{\{storeID,itemID\} \cup Attrs(\texttt{stores})} \wedge\ p)$

$$
\begin{aligned}
\Box V_2 &= \pi_{city,SumCiSales=sum(SumSISales),NumCiSales=sum(NumSISales)}(\Box V_2') \\
V_2 &= V_2 \sqcup_{\theta_2}^{U} \Box V_2, \text{ where } \theta_2 \text{ is } (\equiv_{city} \wedge \neg p,\ \equiv_{city} \wedge\ p) \qquad \text{[From (6) in Table 4.2]} \\
\Box V_3' &= \Box V_1 \bowtie \texttt{items}
\end{aligned}
$$

$V_3' = V_3' \sqcup^U_{\theta_{13}} \square V_3'$, [From (4) in Table 4.2]

where θ_{13} is ($\equiv_{\{storeID,itemID\} \cup Attrs(\texttt{items})} \wedge \neg p$, $\equiv_{\{storeID,itemID\} \cup Attrs(\texttt{items})} \wedge p$)

$\square V_3 = \pi_{category,SumCaSales=sum(SumSISales),NumCaSales=sum(NumSISales)}(\square V_3')$

$V_3 = V_3 \sqcup^U_{\theta_3} \square V_3$, where θ_3 is ($\equiv_{category} \wedge \neg p$, $\equiv_{category} \wedge p$) [From (6) in Table 4.2]

In all the above equations, U is of the form $<(\texttt{SUM}, f), (\texttt{COUNT}, f)>$ and p is of the form $((LHS.\texttt{COUNT} + RHS.\texttt{COUNT}) = 0)$,[10] where SUM is the aggregated attribute (*SumSISales, SumCiSales, or SumCaSales*) in the corresponding view, COUNT is the count attribute (*NumSISales, NumCiSales, or NumCaSales*) depending on the view, and $f(x, y) = x + y$ for all x, y.

As shown in Example 5, the above derived maintenance expressions for V_2 and V_3 are very efficient compared to the expressions derived by previous approaches. □

4.6 Propagating Change Tables Generated at Outerjoin Nodes

In this section, we show how our change-table technique can be used to derive efficient and simple algebraic expressions for maintenance of view expressions involving outerjoin operators. Outerjoin is supported in SQL. Further, outerjoins have recently gained importance because data from multiple distributed databases can be integrated by means of outerjoin views [GJM96, GM95, RU96]. Outerjoins are also extensively used in object-relational systems [BW89, BW90, BPP+93].

Definition 13 (**Outerjoin-change table**) A change table for a view involving outerjoin operators is defined as an *outerjoin-change table* if the change table was either generated at an outerjoin operator or is a result of propagation of an outerjoin-change table, using the propagation equations we will derive for propagating outerjoin-change tables. □

For example, the change tables, □SSInfo and □SSFullInfo, computed for the views SSInfo and SSFullInfo respectively in Example 5 are outerjoin-change tables.

We start by showing how the insertions and deletions into an outerjoin view ($R \overset{f}{\bowtie}^o_J S$), in response to insertions into the base table R, can be summarized into an outerjoin-change table. Computation of an outerjoin-change table at an outerjoin view in response to deletions from a base table requires a more general refresh operator and is briefly discussed in Section 4.6.3.

[10] Recall that LHS and RHS refer to the left and right operands of the join operation where p occurs.

4.6.1 Generating the Outerjoin-Change Table at an Outerjoin Node

Given tables $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$, consider an outerjoin view

$$V(A_1, \ldots, A_n, B_1, \ldots, B_m) = R \overset{fo}{\bowtie}_J S,$$

where J is an equi-join condition. Insertions into R, represented by $\triangle R$, can result in some insertions and deletions into view V. We summarize the set of insertions and deletions to V in an outerjoin-change table $\square V$ defined as $\square V = \triangle R \overset{lo}{\bowtie}_J S$. Note that the tables $\square V$ and V have the same schema and attribute names. We show that with the following specification of the `refresh` operator, the net changes in the outerjoin-change table $\square V$ can be applied to the view V to obtain the correctly refreshed V. The specifications, $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ and U, of the `refresh` operator used to apply $\square V$ to V are defined as follows.

- $\mathcal{J}_1$ is $(\equiv_G \ \wedge p)$, where $G =$ Attrs(S) and $p = (\bigwedge_{1 \le j \le n}(V.A_j = \texttt{NULL}))$.
- $\mathcal{J}_2$ is `FALSE`.
- The update list U is $<(A_1, g), (A_2, g), \ldots, (A_n, g)>$, where $g(x, y) = y$ for all x, y.

Theorem 14 *Consider the view $V = R \overset{fo}{\bowtie}_J S$ and the outerjoin-change table $\square V = \triangle R \overset{lo}{\bowtie}_J S$. For the above definition of the* `refresh` *operator specifications of $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ and U, the following holds:*

$$(R \uplus \triangle R) \overset{fo}{\bowtie}_J S = (R \overset{fo}{\bowtie}_J S) \sqcup_\theta^U (\square V)$$

Proof: Due to insertion of $\triangle R$ into R, the view V should change as follows. First, the set of tuples $\triangle R \overset{lo}{\bowtie}_J S$ should be inserted to V. Then, if there is a tuple $\square v = <r_1, r_2, \ldots, r_n, s_1, s_2, \ldots, s_m>$ in $(\triangle R \overset{lo}{\bowtie}_J S)$, i.e., if $\square v$ is being inserted into V, then the tuple $<\texttt{NULL}, \ldots, \texttt{NULL}, s_1, s_2, \ldots, s_m> \in V$ should be deleted from V.

The above effect can be achieved by updating a tuple $v = <\texttt{NULL}, \ldots, \texttt{NULL}, s_1, s_2, \ldots, s_m>$ in V to $\square v = <r_1, r_2, \ldots, r_n, s_1, s_2, \ldots, s_m>$ if such a tuple $\square v$ exists in $\square V = \triangle R \overset{lo}{\bowtie}_J S$. The refresh of V would be complete if the tuples $\square v$ in $\square V$ for which no such match occurs are inserted into V. By the definition of the `refresh` operator and its specification, one can see that this is exactly what is achieved by the refresh expression $V \sqcup_\theta^U \square V$. ∎

4.6.2 Propagation of Outerjoin-Change Tables

Only a special form of the generic `refresh` operator, which we call an `outerjoin-refresh` operator, is required to refresh a view using its outerjoin-change table.

Definition 14 (`Outerjoin-refresh` **operator**) Let $\{A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m\}$ be the set of attributes in V and its outerjoin-change table $\Box V$. A `refresh` operator $\sqcup_\theta^U$ used to apply the outerjoin-change table $\Box V$ to its view V is said to be an `outerjoin-refresh` operator if:

- The join condition $\mathcal{J}_1$ is $(\equiv_G \ \wedge p)$, where $G = \{B_1, B_2 \ldots, B_m\}$ and p is a predicate on the attributes $(LHS.A_1, LHS.A_2, \ldots, LHS.A_n)$.
- The join condition $\mathcal{J}_2$ is `FALSE`, and
- The update list U is $<(A_1, g), (A_2, g), \ldots, (A_n, g)>$, where $g(x, y) = y$ for all x, y.

Note that $(\text{Attrs}(U) \cap G) = \phi$ and $(\text{Attrs}(U) \cup G) = \text{Attrs}(V)$. □

The refresh equations given in Table 4.2 correctly propagate an outerjoin-change table as well, except for the case of propagation through the outerjoin operator, for which we derive a different equation below.

Consider a view $V = E_1 \overset{fo}{\bowtie}_J E_2$, where E_1 and E_2 are general view expressions. Suppose that the expression E_1 changes to $E_1 \sqcup_\theta^U \Box E_1$ using its outerjoin-change table $\Box E_1$, where the refresh operator $\sqcup_\theta^U$ is an `outerjoin-refresh` operator. Let θ be $(\equiv_G \ \wedge p, \texttt{FALSE})$, where p is a predicate, and G is a set of attributes common to E_1 and $\Box E_1$. The following row, which replaces the row (7) in Table 4.2, shows how to propagate an outerjoin-change table $\Box E_1$ through the outerjoin operator.

7b	$E_1 \overset{fo}{\bowtie}_J E_2$	$(E_1 \sqcup_\theta^U \Box E_1) \overset{fo}{\bowtie}_J E_2$	$V \sqcup_{\theta_1}^{U_1} (\Box E_1 \overset{lo}{\bowtie}_J E_2)$	$(\Box E_1 \overset{lo}{\bowtie}_J E_2)$	$\text{Attrs}(J) \subseteq G$

As already mentioned, θ is $(\equiv_G \ \wedge p, \texttt{FALSE})$ in the row above. Also,

- $\theta_1 = (\mathcal{J}_1, \mathcal{J}_2)$, where $\mathcal{J}_1$ is $\equiv_{Attrs(E_2)} \ \wedge ((p \wedge \ \equiv_G) \vee (\bigwedge_{e_1 \in Attrs(E_1)} LHS.e_1 = \texttt{NULL}))$, $\mathcal{J}_2$ is `FALSE`, and
- $U_1 = <(\mathcal{A}_1, g), (\mathcal{A}_2, g), \ldots, (\mathcal{A}_k, g)>$, where $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\} = \text{Attrs}(E_1)$ and $g(x, y) = y$ for all x, y.

Theorem 15 *Assume that the* `refresh` *operator used in the expression of the third column in Table 4.2 is an* `outerjoin-refresh` *operator.*

(1) The change propagation equations given in Table 4.2, with the following two changes, correctly propagate outerjoin-change tables.

- *Disregard the condition in column 6 of the first row (selection view)*
- *Replace the seventh row by row (7b) given above*

(2) The `refresh` *operator derived in each of the refresh equations (column 4) is also an* `outerjoin-refresh` *operator.*

Proof: It is easy to see that the refresh operator in the fourth column with its new specifications is also an `outerjoin-refresh` operator. As before, we refer to the expression $E_1 \sqcup_\theta^U \square E_1$ as the *change equation* throughout this proof.

• $\underline{V = \sigma_q(E_1)}$. We use the characteristics of the `outerjoin-refresh` operator to show that $\sigma_q(E_1 \sqcup_\theta^U \square E_1) = \sigma_q(E_1) \sqcup_\theta^U \sigma_q(\square E_1)$.

First, we show that the refresh equation of V doesn't miss any legitimate insertions into V affected by the change equation $E_1 \sqcup_\theta^U \square E_1$. Let I be the set of tuples that is inserted into E_1 due to the change equation. We will show that $\sigma_q(I)$ is inserted into V by the refresh equation, implying the result. Note that I is the set of tuples in $\square E_1$ that do not find a match in E_1. As $\sigma_q(I) \subseteq \sigma_q(\square E_1) = \square V$, no tuple in $\sigma_q(I)$ will find a match in $V \subseteq E_1$. Hence, $\sigma_q(I)$ will be inserted into V using the refresh equation.

Now, we show that all insertions into V due to the refresh equation are legitimate. The refresh equation may induce an insertion of the tuple $\square v \in \square V$ into V if $\square v$ doesn't find a match in V. If $\square v \in I$, where I is the set of tuples in $\square E_1$ that don't find a match in E_1, then the insertion is obviously legitimate. Suppose, $\square v \notin I$. This implies that $\square v \in \square E_1$ found a match with a tuple $e_1 \in E_1$. Because of the specifications of the `outerjoin-refresh` operator, the match results in the tuple e_1 in E_1 being updated to $\square v$ by the change equation $E_1 \sqcup_\theta^U \square E_1$. Now note that as $\square v \in \square V$, it satisfies the selection condition q, and hence the tuple $\square v$ (updated from e_1 in E_1) will be included in the expression $\sigma_q(E_1 \sqcup_\theta^U \square E_1)$ of the third column. Thus, the insertion of $\square v$ into V by the refresh equation is correct. As the match between tuples is one-to-one, the tuple $\square v$ results in only one update in the table E_1 due to the change equation, which as shown above corresponds to the insertion of $\square v$ into V due to the refresh equation.

In the refresh equation, if a tuple $v \in V$ is updated by a tuple $\square v \in \square V = \sigma_q(\square E_1)$, then $v \in E_1$ would have been updated by $\square v \in \square E_1$ by the change equation too. According to the update characteristics of the `outerjoin-refresh` operator, the tuple v is updated to $\square v$ by $\square v$. Now, $\square v$ satisfies the predicate p, hence the updated tuple $\square v$ is correctly retained in V by the refresh equation. This shows that the updates to V in the refresh equation are legitimate.

The only updates the refresh equation might miss are of the kind where a tuple $\boxdot e_1 \in \boxdot E_1$ matches with a tuple $e' \in \sigma_{\overline{q}}(E)$ in the change equation. The match results in the tuple e' being updated to $\boxdot e_1$. If $\boxdot e_1$ satisfies the condition p, then it is included in V by the change equation. In the refresh equation, as $\boxdot e_1 \in \boxdot V$ doesn't find a match in V, it is inserted into V. Hence, the desired effect is achieved.

• Projection: $V = \Pi_A(E_1)$. If $\text{Attrs}(\theta) \subseteq A$, then the information needed to decide the effect of a tuple $\boxdot e_1 \in \boxdot E_1$ on a tuple $e_1 \in E_1$, if any, is available in $\Pi_A(e_1)$. Hence, $\Pi_A(\boxdot E_1)$ can be applied directly to $V = \Pi_A(E_1)$.

• Cross Product: $V = E_1 \times E_2$. Consider $(E_1 \times E_2) \sqcup^U_{\theta_1} (\boxdot E_1 \times E_2)$, the refresh equation. Let us partition the tables $(E_1 \times E_2)$ and $(\boxdot E_1 \times E_2)$ by the tuple values of E_2. As $\mathcal{J}_1$ and $\mathcal{J}_2$ in θ_1 include $\text{Attrs}(E_2)$, each of the partitions is refreshed independently by the refresh equation. It is easy to see that a tuple $<\boxdot e_1, e_2> \in \boxdot V$ will result in a match with a tuple $<e_1, e_2> \in V$ due to θ_1 in the refresh equation, if and only if the tuple $\boxdot e_1 \in \boxdot E_1$ matches with a tuple $e_1 \in E_1$ due to the parameter θ in the change equation. The match will result in the same updates to the E_1 attributes in both the expressions. Also, a tuple $\boxdot e_1 \in \boxdot E_1$ doesn't find a match in E_1 if and only if the tuples in $(\boxdot e_1 \times E_2) \subseteq \boxdot V$ do not find a match in V.

• Join: $V = E_1 \bowtie_J E_2$. Follows from the previous cases.

• Union: $V = E_1 \uplus E_2$. As $((E_1 \uplus E_2) \dot{-} E_2) = E_1$, the equivalence of the expressions is obvious.

• Aggregation: $V = \pi_{G', f(A)}(E_1)$. Without loss of generality, we prove this case when $k = 1$. Let $f_1 = f$ and $A_1 = A$, the aggregated attribute. We assume that $(G' \subseteq G)$, $A \in \text{Attrs}(U)$, $U = <(A, f)>$, and that f is a distributive function. Consider tuples $e_1, e_2, \ldots, e_n$ in E_1 such that they have the same G' values, and let their attribute A values be $a_1, a_2, \ldots, a_n$. Assume that the tuples $e_1, \ldots, e_l$ find matches with tuples $\boxdot e_1, \ldots, \boxdot e_l$ in $\boxdot E_1$ due to the join condition $\mathcal{J}_1$. Thus, $a_i =$ `NULL` and $\boxdot e_i$'s have the same G' values as e_i's for $1 \leq i \leq l$. Also, the attribute A value of e_i is updated to $\boxdot a_i$, for $1 \leq i \leq l$, due to the update parameter U in the change equation. Hence, due to the change equation the tuples $e_1, e_2, \ldots, e_n$ will result in an aggregated value of $f(\boxdot a_1, \boxdot a_2, \ldots, \boxdot a_l, a_{l+1}, a_{l+2}, \ldots, a_n)$ in the expression of the third column. Thus, we need to show that the aggregated A value of the tuple v that is derived from $e_1, \ldots, e_n$ in the view $V = \pi_{G', f(A)}(E_1)$ changes from $f(a_1, a_2, \ldots, a_n)$ to $f(\boxdot a_1, \boxdot a_2, \ldots, \boxdot a_l, a_{l+1}, a_{l+2}, \ldots, a_n)$ in the refresh equation. Now, by the definition of $\boxdot V$, the tuples $\boxdot e_1, \ldots, \boxdot e_l \in \boxdot E_1$ will be

grouped to yield the aggregated attribute value $f(\square a_1, \ldots, \square a_l)$ in a tuple in $\square V$. The refresh equation of V then updates the aggregated value of the grouped value of e_is from $f(a_1, a_2, \ldots, a_k)$ to $f(f(a_1, a_2, \ldots, a_n), f(\square a_1, \ldots, \square a_l)) = f(a_{l+1}, \ldots, a_n, \square a_1, \ldots, \square a_l)$, as a_1 to a_l are NULL. Hence the refresh equation of V correctly updates the aggregated attribute value of the tuple v in V.

All insertions into E_1 due to $\square E_1$ will be converted to aggregated insertions into V by the refresh equation.

• Outerjoin: $V = E_1 \overset{fo}{\bowtie}_J E_2$. Let $M(\subseteq \square E_1)$ be the set of tuples in $\square E_1$ that find a match in E_1 in the change equation due to $\mathcal{J}_1 = (\equiv_G \wedge p)$. Note that G is a set of attributes in E_1 and p is a predicate over the rest of the attributes in E_1. Consider $m \in M$ and that m finds a match with a tuple e_1 in E_1 due to $\mathcal{J}_1$. Each m results in a set of tuples $E_2^m = m \overset{lo}{\bowtie}_J E_2$ in $\square V = \square E_1 \overset{fo}{\bowtie}_J E_2$, where E_2^m may consist of just one tuple $<m, \texttt{NULL}>$. Each tuple $e_2^m = <m, e_2> \in E_2^m$ will find a match with the corresponding tuple $v = <e_1, e_2> \in V$ (note that e_2 may be NULL) due to the join condition $\mathcal{J}_1 \wedge \equiv_{Attrs(E_2)}$. As Attrs$(J) \subseteq G$ and $<m, e_2> \in E_2^m$, the tuple v exists in V (even if $e_2 =$ NULL). The tuple $<e_1, e_2>$ gets updated to $<m, e_2>$ due to the update list U_1 in the refresh equation of V. This update affected in V by the refresh equation is correct because $m \in \square E_1$ also updates $e_1 \in E_1$ to m in the change equation, as Attrs(G) $\cup$ Attrs(U) = Attrs(E_1), and m and e_1 have the same G attributes. Thus, all the updates of tuples in E_1 due to the change equation are correctly applied to V by $\square V$ in the refresh equation.

Lets consider the other set of tuples I in $\square E_1$, that are inserted into E_1 (because they didn't find a match in E_1). Let $i \in I$. The tuple i results in the set of tuples $E_2^i = i \overset{lo}{\bowtie}_J E_2$ in $\square V$. Tuple $<i, e_2> \in E_2^i$ may find a match v in V due to the condition $\equiv_{Attrs(E_2)} \wedge (\bigwedge_{e \in Attrs(E_1)} LHS.e =$ NULL) in $\mathcal{J}_1$. In that case the tuple v is correctly updated to $<i, e_2>$. All the other unmatched tuples in $\square V$ are correctly inserted into V. ∎

EXAMPLE 8 Consider a view

$$V = \pi_{A,B,F=sum(D),H=sum(E),Num=Count(*)}(\sigma_{A<5}((R \overset{fo}{\bowtie}_{C=D} S) \bowtie T)),$$

where $R(A, B, C), S(D, E)$, and $T(A, B, L)$ are base relations, and $\bowtie$ is the natural join operation, i.e., a join with the join condition $(\equiv_{\{A,B\}})$. Recall that for computing the SUM aggregates, the attribute value of NULL is taken as 0, provided at least one tuple has a non-NULL value.

For clarity of presentation, let us assume $V_1 = R \stackrel{fo}{\bowtie}_{C=D} S$, $V_2 = V_1 \bowtie T$, $V_3 = \sigma_{A>5}(V_2)$. Let us define a predicate p as $((LHS.D = \texttt{NULL}) \wedge (LHS.E = \texttt{NULL}))$, a predicate q as $((LHS.Num + RHS.Num) = 0)$, an update list U_1 as $<(F,g),(H,g)>$, and U as $<(D,h),(E,h),(Num,h)>$. Here, $g(x,y) = y$ for all x, y, $h(x,y) = x+y$ for all $x, y \neq \texttt{NULL}$, and $h(\texttt{NULL}, y) = y$. We illustrate our techniques of maintaining views involving outerjoin operators by deriving maintenance expressions for V in response to insertions, $\triangle S$, into S. The equation used for computing $\square V_1$ is similar to that derived in Theorem 14. In this case, we have insertions into S, and hence we use a right outerjoin operation instead.

$$
\begin{aligned}
\square V_1 &= (R \stackrel{ro}{\bowtie}_{C=D} \triangle S) \\
V_1 &= V_1 \sqcup_{\theta_1}^{U_1} \square V_1, \text{ where } \theta_1 \text{ is } (\equiv_{Attrs(R)} \wedge p, \texttt{FALSE}) && \text{[From Theorem 14]} \\
\square V_2 &= \square V_1 \bowtie T \\
V_2 &= V_2 \sqcup_{\theta_2}^{U_1} \square V_2, \text{where } \theta_2 \text{ is } (\equiv_{Attrs(R) \cup Attrs(T)} \wedge p, \texttt{FALSE}) && \text{[From (4) in Table 4.2]} \\
\square V_3 &= \sigma_{A>5}(\square V_2) \\
V_3 &= V_3 \sqcup_{\theta_2}^{U_1} \square V_3 && \text{[From (1) in Table 4.2]} \\
\square V &= \pi_{A,B,F=Sum(D),H=Sum(E),Num=Count(*)}(\square V_3) \\
V &= V \sqcup_{\theta}^{U} \square V, \text{where } \theta \text{ is } (\equiv_{\{A,B\}} \wedge \neg q, \equiv_{\{A,B\}} \wedge q) && \text{[From (4) in Table 4.2]} \quad \square
\end{aligned}
$$

4.6.3 Propagation of Deletions through Outerjoin Operators

The changes in an outerjoin $V = R \stackrel{fo}{\bowtie}_J S$ due to deletions from a base relation R cannot be summarized in an outerjoin-change table within our restricted definition of the `refresh` operator. In this section, we show that by using an extended definition of the `refresh` operator, we can apply changes summarized in an appropriately defined change table to V in response to deletions from a base table.

Consider a simple outerjoin view $V = R \stackrel{fo}{\bowtie}_J S$. Let ∇R be the set of deletions from R, and let $S^{set} = \pi_{Attrs(S),Num=Count(*)}(S)$. We define the *OJDeletion-change table* $\square V$ that succinctly represents changes to V in response to ∇R as

$$\square V = \nabla R \stackrel{lo}{\bowtie}_J S^{set}.$$

The view V is refreshed using the refresh equation $V \sqcup_{\theta}^{U} \square V$, where $\sqcup_{\theta}^{U}$ is the `OJDeletion-refresh` operator as defined in Algorithm 10. Here, $\theta = (\equiv_{Attrs(V)}, (\equiv_{Attrs(S)} \wedge \text{Attrs}(S) \neq \texttt{NULL}))$ and $U = <(A_1, f), \ldots, (A_n, f)>$, where $f(x,y) = \texttt{NULL}$ for all x, y and $\{A_1, \ldots, A_k\} = \text{Attrs}(R)$.

The OJDeletion-refresh algorithm (Algorithm 10) used to refresh the view V works as follows. A tuple (r, s, l) in $\square V$ comes from l copies of s in S and a tuple r in $\triangledown R$. Thus, the tuple r belonged to R before deletions and V has at least l copies of (r, s). Now, if there is another tuple $v' = (r_1, s) \in V$, then the l copies of (r, s) can be deleted from V as a result of deletion of r from R. But, if no such v' exists in V, then each of the copies of (r, s) in V should be changed to $(\texttt{NULL}, s)$, as the tuple $s \in S$ now becomes a dangling tuple after deletions from R. Thus, the l copies of (r, s) are updated accordingly in Algorithm 10. One can see that it is essential to store in $\square V$ the number of copies l of s from S. Also, note that the OJDeletion-refresh algorithm doesn't to query any sources, and hence can be executed very efficiently.

Algorithm 10 **OJDeletion-Refresh Algorithm**

Used to apply an OJDeletion-change table to its view

Input

View $V(A_1, \ldots, A_n, B_1, \ldots, B_m)$

OJDeletion-change Table $\square V(A_1, \ldots, A_n, B_1, \ldots, B_m, Num)$

Characteristics of the OJDeletion-refresh Parameters

$\theta = (\mathcal{J}_1, \mathcal{J}_2).\mathcal{J}_1$ is $\equiv_{Attrs(V)}$ and $\mathcal{J}_2$ is $\equiv_G$, where $G = \{B_1, B_2, \ldots, B_m\}$

$U = <(A_1, f_1), (A_2, f_2), \ldots, (A_n, f_n)>$.

Output

Refreshed table V, i.e., $V \sqcup_\theta^U \square V$.

BEGIN

for each tuple $\square v = (r, s, l)$ in $\square V$ /* r is the value of U attributes, */
/* s is the value of the G attributes, and l is an integer. */

Let $\{v_1, \ldots, v_l\}$ be the tuples in V that match $\square v$ due to the join condition $\mathcal{J}_1$.
/* Note that there are at least l tuples in V that will match $\square v$ due to $\mathcal{J}_1$. */

if there is a tuple $v' \in V$ such that $v' \notin \{v_1, \ldots, v_l\}$ and
v' matches with $\square v$ due to the join condition $\mathcal{J}_2$

then Delete tuples $v_1, v_2, \ldots, v_l$ from V;

else Update each tuple $v_1, v_2, \ldots, v_l$ in V using the specifications in U;

end if;

end for;

return V;

END. $\diamond$

No.	V	New V $\theta = (\mathcal{J}_1, \mathcal{J}_2)$ $\mathcal{J}_1$ is $\equiv_V$ $\mathcal{J}_2$ is $\equiv_G$	Refresh Equation	$\square V$	Conditions
1	$\sigma_q(E_1)$	$\sigma_q(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \sigma_q(\square E_1)$	$\sigma_q(\square E_1)$	$\text{Attrs}(q) \subseteq G$
2	$\Pi_A(E_1)$	$\Pi_A(E_1 \sqcup_\theta^U \square E_1)$	$V \sqcup_\theta^U \Pi_A(\square E_1)$	$\Pi_A(\square E_1)$	$G \subseteq A$
3	$E_1 \times E_2$	$(E_1 \sqcup_\theta^U \square E_1) \times E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \times E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \times E_2$	
4	$E_1 \bowtie_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \bowtie_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \bowtie_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_2)$	$\square E_1 \bowtie_J E_2$	$\text{Attrs}(J) \subseteq G$
5	$E_1 \uplus E_2$	$(E_1 \sqcup_\theta^U \square E_1) \uplus E_2$	$((V \dot{-} E_2) \sqcup_\theta^U \square E_1) \uplus E_2)$	$\square E_1$	
6	$E_1 \overset{fo}{\bowtie}_J E_2$	$(E_1 \sqcup_\theta^U \square E_1) \overset{fo}{\bowtie}_J E_2$	$V \sqcup_{\theta_1}^U (\square E_1 \overset{lo}{\bowtie}_J E_2)$ $\theta_1 = (\mathcal{J}_1 \wedge \equiv_\tau, \mathcal{J}_2 \wedge \equiv_\tau)$ $\tau = \text{Attrs}(E_1)$	$(\square E_1 \overset{lo}{\bowtie}_J E_2)$	$\text{Attrs}(J) \subseteq G$

Table 4.3: Change propagation equations for propagating OJDeletion-change tables

Propagating OJDeletion-change Table

In this section, we present change propagation equations that are used to propagate an OJDeletion-change table through various operators.

Theorem 16 *Assume that the* `refresh` *operator used in the expression of the third column in Table 4.3 is an* `OJDeletion-refresh` *operator. Then,*

(1) the change propagation equations given in Table 4.3 for propagation of OJDeletion-change tables are correct, i.e., for each row, the expression in the third column is equivalent to the refresh equation in the fourth column, and

(2) the `refresh` *operator derived in the refresh equation (column 4) is an* `OJDeletion-refresh` *operator as well.*

Proof: It is easy to see that the refresh operator in the fourth column with its new specifications is also an `OJDeletion-refresh` operator. As before, we will refer to the expression $E_1 \sqcup_\theta^U \square E_1$ as the change equation throughout this proof.

• Selection: $V = \sigma_q(E_1)$. As, $Attrs(q) \cap Attrs(U) = \phi$, we know that q attributes are not updated by any refresh operation. Thus, tuples in E_1 that do not pass the q condition end up being deleted (possibly after being updated first) from both the expressions in third column and the fourth column.

Now, consider a tuple $<r, s>$ in E_1 that passes the q condition, where s is the value of the G attributes. As $Attrs(q) \subseteq Attrs(\mathcal{J}_2) = G$, any tuple of the form (r_1, s) that belongs to E_1 will also pass the condition q and is retained in $V = \sigma_q(\Box E_1)$. Similarly, any tuple of the form $(r_1, s, l_1) \in \Box E_1$ will also be retained in $\sigma_q(\Box E_1)$. Therefore, if the tuple (r, s) is deleted in the change equation (the refresh operation of E_1), then it is also deleted in the refresh equation. Similarly, if (r, s) is updated to $(\texttt{NULL}, s)$ by the change equation, then it is also updated to $(\texttt{NULL}, s)$ by the refresh equation.

• Projection: $V = \Pi_A(E_1)$. As $G \subseteq A$, all the attributes that govern the refresh operation are retained in the projected table. Hence, the refresh operation is not affected.

• Cross Product: $V = E_1 \times E_2$. Consider $(E_1 \times E_2) \sqcup_{\theta_1}^{U} (\Box E_1 \times E_2)$, the refresh equation. Let us partition the tables $(E_1 \times E_2)$ and $(\Box E_1 \times E_2)$ by the tuple values of E_2. As $\mathcal{J}_1$ and $\mathcal{J}_2$ in θ_1 include Attrs(E_2), each of the partitions is refreshed independently by the refresh equation. A tuple (r, s, e_2) is deleted or updated to $(\texttt{NULL}, s, e_2)$ if and only if the corresponding tuple (r, s) is deleted or updated to $(\texttt{NULL}, s)$ by the operation $E_1 \sqcup_{\theta}^{U} \Box E_1$ in the change equation.

• Join: $V = E_1 \bowtie_J E_2$. Follows from the previous cases.

• Outerjoin: $E_1 \overset{fo}{\bowtie}_J E_2$. Consider the refresh equation $(E_1 \overset{fo}{\bowtie}_J E_2) \sqcup_{\theta_1}^{U} (\Box E_1 \overset{lo}{\bowtie}_J E_2)$, where $Attrs(J) \subseteq G$. Let $v = <r, s, e_2>$ be a tuple in $V = (E_1 \overset{fo}{\bowtie}_J E_2)$, where $<r, s> \in E_1$ and $e_2 \in E_2$ and the pair $(<r, s>, e_2)$ passes the join condition J. The tuple v is affected by the refresh equation in the fourth column only if there is a tuple $<r, s, l, e_2>$ in $(\Box E_1 \overset{lo}{\bowtie}_J E_2)$. That is, v is updated to $<\texttt{NULL}, s, e_2>$ if there is a tuple $<r, s, l>$ in $\Box E_1$, a tuple (e_2) in E_2, and a tuple $<r_1, s, e_2>$ in V for some $r_1 \neq r$. This implies that the tuple $<r, s>$ is also updated to $<\texttt{NULL}, s>$ in the refresh operation $E_1 \sqcup_{\theta}^{U} \Box E_1$, as $<r_1, s>$ belongs to E_1. If there is no such tuple $<r_1, s, e_2>$ in V, then v is deleted from V in the refresh equation of V, and similarly $<r, s>$ is deleted from E_1 in the change equation.

Similarly, it can be argued that a tuple $(r, s, \texttt{NULL})$ is deleted or updated in V if and only if (r, s) is deleted or updated in E_1. Also, it is easy to see that tuples that are of the type $(\texttt{NULL}, \texttt{NULL}, e_2)$ appear in both the tables, represented by the third and fourth column expressions, consistently. ∎

4.7 Handling Deletions and Updates Directly and Efficiently

In this section, we present an approach for efficiently handling certain kinds of deletions using our change-table technique. The same techniques can be used to also propagate some updates directly instead of propagating them as insertions and deletions as in traditional approaches. As the techniques for handling deletions and updates in this context are very similar, we discuss handling of deletions in detail and illustrate how updates can be handled through an example.

4.7.1 The `Deletion-Refresh` Operator

For the case of propagating deletions efficiently, we define a `deletion-refresh` operator that is slightly different than the general `refresh` operator (Algorithm 9) defined in Section 4.4. First, the `deletion-refresh` operation has only the $\theta = (\mathcal{J}_1, \mathcal{J}_2)$, parameter associated with it. Also, $\mathcal{J}_1$ is `FALSE` and $\mathcal{J}_2$ is an arbitrary join condition. Now, consider a refresh equation $V \sqcup_\theta \square V$, where $\sqcup_\theta$ is a `deletion-refresh` operator. The *deletion-refresh algorithm* is defined as follows. If a pair of tuples $\square v \in \square V$ and $v \in V$ match due to the join condition $\mathcal{J}_2$, then the tuple v from V is deleted. However, if one tuple in $\square V$ matches with many tuples in V, then all the matching tuples in V must be deleted (difference with the Algorithm 9), causing more than one deletion. This is the key idea in the derivation of efficient expressions for propagating deletions. Also, the unmatched tuples in $\square R$ are ignored (instead of being inserted as in Algorithm 9). For the case of applying a change table using a `deletion-refresh` operator, the view and its change table can have very different schemas. We summarize the characteristics of the `deletion-refresh operator` below.

Characteristics of the Deletion-Refresh Operator

- The parameter $\theta =$ (`FALSE`, $\mathcal{J}_2$), where $\mathcal{J}_2$ is some join condition. As $\mathcal{J}_1 =$ `FALSE`, U is not needed.
- A tuple of the change table can match with more than one tuple of its view.
- Matched tuples in V are deleted, and unmatched tuples in the change table are ignored.

Definition 15 (**Deletion-Change Table**) A change table that is applied to its view using the above defined `deletion-refresh` operator is referred to as a *deletion-change table.* □

Generation of Deletion-Change Tables

In this subsection, we discuss generation of deletion-change tables from a set of deletions into a view. We will see that only certain kinds of deletions can be converted to deletion-change tables.

The techniques presented in this section can be used to propagate any deletion $\bigtriangledown R$ that can be represented as a pair $(\mathcal{J}_2, \square R)$, such that $R \dot{-} \bigtriangledown R = R \ \sqcup_\theta \ \square R$, where $\sqcup_\theta$ is a `deletion-refresh` operator. For example, any SQL statement that deletes all tuples in R satisfying a given predicate can be represented as such a pair. Below, we define a notion of "duplicate-intensive" deletions, which can also be converted to deletion-change tables, and propagated efficiently using the change-table technique.

Definition 16 (**Duplicate-intensive deletions**) A set of tuples $\bigtriangledown R$ is called a duplicate-intensive deletion (D-ID) with respect to a table R if for every tuple $t \in \bigtriangledown R$, the number of duplicates of t in R is no more than the number of duplicates of t in $\bigtriangledown R$.

A duplicate-intensive deletion signifies that if a tuple t is deleted from R then all its duplicates are also deleted. Any deletion from a set R is trivially a duplicate-intensive deletion. □

The change-table technique can be extended to efficiently propagate any duplicate-intensive deletion. If $\bigtriangledown R$ is a D-ID w.r.t. R, then for $\mathcal{J}_2 = (\equiv_{Attrs(R)})$ and $\square R = \bigtriangledown R$, the following equation holds: $R \dot{-} \bigtriangledown R = R \ \sqcup_\theta \ \square R$, where $\sqcup_\theta$ is a `deletion-refresh` operator. See Theorem 17 for a proof.

Propagating Deletion-Change Tables

In this subsection, we present change propagation equations for propagating deletion-change tables through various operators. The change propagation equations are given in Table 4.4.

Theorem 17 *Assume that the* `refresh` *operator used in the expression of the third column in Table 4.4 is a* `deletion-refresh` *operator. Then,*

(1) the change propagation equations given in Table 4.4 for propagation of deletion-change tables are correct, i.e., for each row, the expression in the third column is equivalent to the refresh equation in the fourth column, and

(2) the `refresh` *operator derived in the refresh equation (column 4) is a* `deletion-refresh` *operator as well.*

No.	V	New V	Refresh Equation	$\square V$	Conditions
1	R	$(R \dot{-} \nabla R)$	$(R \sqcup_\theta \square R)$ $\theta = (\texttt{FALSE}, \equiv_{Attrs(R)})$	$\square R = \nabla R$	∇R is D-ID w.r.t. R
2	$\sigma_q(E)$	$\sigma_q(E \sqcup_\theta \square E)$	$\sigma_q(E) \sqcup_\theta \square E$	$\square E$	
3	$\Pi_A(E)$	$\Pi_A(E \sqcup_\theta \square E)$	$\Pi_A(E) \sqcup_\theta \Pi_A(\square E)$	$\Pi_A(\square E)$	$\text{Attrs}(\theta) \subseteq A$
4	$E_1 \times E_2$	$(E_1 \sqcup_\theta \square E_1) \times E_2$	$(E_1 \times E_2) \sqcup_\theta \square E_1$	$\square E_1$	
5	$E_1 \bowtie_J E_2$	$(E_1 \sqcup_\theta \square E_1) \bowtie_J E_2$	$(E_1 \bowtie_J E_2) \sqcup_\theta \square E_1$	$\square E_1$	
6	$E_1 \uplus E_2$	$(E_1 \sqcup_\theta \square E_1) \uplus E_2$	$((V \dot{-} E_2) \sqcup_\theta \square E_1) \uplus E_2$	$\square E_1$	
7	$\pi_{G,f(A)}(E)$	$\pi_{G,f(A)}(E \sqcup_\theta \square E)$	$\pi_{G,f(A)}(E) \sqcup_\theta \square E$	$\square E$	$\text{Attrs}(\theta) \subseteq G$

Table 4.4: Change propagation equations for efficiently propagating deletions

Proof: It is easy to see that the refresh operator in the fourth column with its new specifications is also a `deletion-refresh` operator.

As the `deletion-refresh` operation deletes all the duplicates of any tuple deleted from the view, it suffices to show that the refresh equation in the fourth column is *set*-equivalent to the change equation in the third column.

• Generation: $V = R$. We need to show that if ∇R is a D-ID w.r.t. R, then for $\mathcal{J}_2 = (\equiv_{Attrs(R)})$ and $\square R = \nabla R$, the following equation holds: $R \dot{-} \nabla R = R \sqcup_\theta \square R$, where $\sqcup_\theta$ is a `deletion-refresh` operator. Consider a tuple $r \in (R \dot{-} \nabla R)$, which implies $r \in R$ and $r \notin \nabla R$. Hence, $r \notin \square R = \nabla R$. Thus, $r \in R \sqcup_\theta \square R$. Now, consider $r \in (R \sqcup_\theta \square R)$, which implies r is in R, but not in $\square R = \nabla R$. Hence, r belongs to $R \dot{-} \nabla R$.

• Selection: $V = \sigma_q(E)$. If a tuple e is in $\sigma_q(E \sqcup_\theta \square E)$, then $q(e)$ is true, $e \in E$, and there is no tuple $v \in \square E$ such that $\mathcal{J}_2(e, v)$ is true. Thus, $e \in \sigma_q(E)$ and also, no such v

exists in $\square E$. Hence, $e \in (\sigma_q(E) \ \sqcup_\theta \ \square E)$.

Now, if a tuple $e \in (\sigma_q(E) \ \sqcup_\theta \ \square E)$, then $e \in \sigma_q(E)$, and there is no $v \in \square E$ such that $\mathcal{J}_2(e, v)$ is true. Hence, $e \in E$, $p(e)$ is true, and no such v is in $\square E$. Thus, e is in $\sigma_q(E \ \sqcup_\theta \ \square E)$.

• Projection: $V = \Pi_A(E)$. If a tuple e_a is in $\Pi_A(E \ \sqcup_\theta \ \square E)$, then there exists some tuple $e \in E$ such that $\Pi_A(e) = e_a$. Also, there is no tuple v in $\square E$ such that $\mathcal{J}_2(e, v)$ is true. Let $v_a = \Pi_A(v)$. As, $e_a \in \Pi_A(E)$ and $\text{Attrs}(\theta) \subseteq A$, we can show by contradiction that there is no tuple v_a in $\Pi_A(\square E)$ such that $\mathcal{J}_2(e_a, v_a)$ is true (else, there will be a v in $\square E$ such that $\mathcal{J}_2(e, v)$ is true). Hence, e_a also belongs to $(\Pi_A(E) \ \sqcup_\theta \ \Pi_A(\square E))$.

Now, if a tuple e_a is in $\Pi_A(E) \ \sqcup_\theta \ \Pi_A(\square E)$, then there exists some tuple e such that $\Pi_A(e) = e_a$. Also, $e \in E$ and there is no tuple $v \in \square E$ such that $v_a = \Pi_A(v)$ and $\mathcal{J}_2(e_a, v_a) = \mathcal{J}_2(e, v)$ is true. Therefore, e_a is in $\Pi_A(E \ \sqcup_\theta \ \square E)$.

• Cross Product: $V = E_1 \times E_2$. Let us assume there is tuple $\square e$ in $\square E$ that matches with a tuple $e \in E$ due to $\mathcal{J}_2$. Such a tuple $\square e$ will also match with a tuple $(e, e_2) \in E_1 \times E_2$ for any tuple $e_2 \in E_2$. Hence, the refresh equation of the fourth column is true.

• Join: $V = E_1 \bowtie_J E_2$. Follows from the previous cases of selection and cross product.

• Union: $V = E_1 \cup E_2$. This case is obvious, as $V \dot{-} E_2 = E_1$.

• Aggregation: $V = \pi_{G,f(A)}$. As $\text{Attrs}(\theta) \subseteq G$, this case is also obvious. ∎

EXAMPLE 9 Consider a view $V = \pi_{A,B,E=Sum(D)}(R \bowtie S)$, where $R(A, B, C)$ and $S(B, C, D)$ are base relations. We will evaluate the maintenance expressions of the view V in response to deletions in R. Let us assume that $T = R \bowtie S$ and that R is a set without duplicates.

The approach of [GL95, Qua97] yields very complicated expressions for $\triangle V$ and $\triangledown V$ in terms of $\triangledown T = \triangledown R \bowtie S$. If we were to use the aggregate-change table techniques presented earlier, then we would compute the aggregate-change table $\square V$ as

$$\square V = \pi_{A,B,E=\ -sum(D),cnt=-count(*)}(\triangledown R \bowtie S).$$

The `aggregate-refresh` operator is then used to refresh the view V using its aggregate-change table $\square V$.

Using the techniques presented in this section, we derive the maintenance expressions in response to deletions in R as follows. As R is a set, the set of deletions $\triangledown R$ from R is such that $R \dot{-} \triangledown R = R \sqcup_\theta \square R$, where $\square R(A, B, C) = \triangledown R$ and $\theta = (\texttt{FALSE}, \equiv_{\{A,B,C\}})$.

So, we have

$$\begin{aligned} R \dot{-} \bigtriangledown R &= R \sqcup_\theta \square R \\ \square V &= \square T = \square R \qquad \text{[From (5, 7) in Table 4.4]} \\ V &= V \sqcup_\theta (\square V) \end{aligned}$$

The expression obtained above is more efficient than the maintenance expressions obtained by the aggregate-change table techniques or [MQM97, GL95, Qua97], as in all these approaches computation of the changes to V will require evaluating an expensive join operation $\bigtriangledown R \bowtie S$. □

EXAMPLE 10 In this example, we briefly show how updates can be propagated efficiently using the techniques presented in this section.

Consider a view $V = \pi_{A,B,D=Sum(C)}(R \bowtie S)$, where $R(A, B)$ and $S(B, C)$ are the base relations and B is the primary key of R.

A set of updates to the table R of the kind $(a, b) \rightarrow (c, b)$ can be represented in a change table $\square R$, which has the same schema as that of R. A tuple $\square r = (c, b) \in \square R$ signifies that any tuple $r = (a, b) \in R$ is updated to $\square r$. This refresh operation can be algebraically represented as $R = R \sqcup_\theta^U \square R$, where $U = <(A, f)>$ and $f(x, y) = y$ for any x, y. Also, $\theta = (\equiv_B, \texttt{FALSE})$ and the matches may be many-to-one as in the case of deletions.

Using the change propagation equations from Table 4.4, we get the maintenance expression of V as follows. We assume that $V_1(A, B, C) = R \bowtie S$.

$$\begin{aligned} R &= R \sqcup_\theta^U \square R \\ \square V &= \square V_1 = \square R \\ V &= V \sqcup_\theta^U (\square V) \end{aligned}$$

□

4.8 Optimality Issues

In this section, we discuss the optimality of the incremental maintenance algorithm based on our change-table techniques. For simplicity and concreteness, we consider the simplistic cost model wherein the cost incurred by a maintenance algorithm in maintaining a view in response to some changes at the base relations is the number of sources (base relations)

queried. The above cost model is reasonable in a data warehouse model where the dominant cost is the cost incurred in querying the sources.

Let us consider change-table techniques presented in this chapter for incremental maintenance of general view expressions along with the following two minor optimizations:

- In computing a view, we tag the tuples in the result of a union operator with L or R depending on whether the tuple comes from the left operand or the right operand. The above improvement makes propagation of an aggregate and outerjoin change table through a union operator very efficient. In essence, the refresh equation of the fifth row in Table 4.2 will not involve E_1 or E_2.

- When computing the refresh equation, the view contents are used, whenever possible, to compute a required subexpression using the minimum number of source queries. For e.g., consider $V = R \times (S \times T)$. In response to insertions into R, to compute changes at V, we need to query $(S \times T)$ according to the propagation equations in Table 4.2. Instead, we query R and compute $(S \times T)$ using the value of V, minimizing the number of source queries.

We incorporate the above two improvements into the change-table technique and prove the following result.

Theorem 18 *Given an expression tree of a view V, the change-table technique queries the minimum number of sources required in order to compute changes at* each *node in the expression tree.*

Proof: We are given an expressions tree for a view V, which is materialized, and are required to compute changes at each node in the expression tree. We need to show that the incremental maintenance algorithm based on our change-table technique achieves that by querying minimum number of sources.

The change-table technique works by computing a change table and `refresh` operator specifications at each operator/node of the given expression tree of V. As a change table along with the refresh operator specifications is sufficient to refresh the corresponding subexpression, the change-table technique in effect computes changes at each node in the expression tree.

We now show by induction on the height of the expression tree that the change-table technique queries the minimum number of sources required to compute changes at each

node of an expression tree. The claim is trivially true for $V = R$, where R is a base relation. Let us assume the claim to be true for any expression tree of height less than h.

Consider a view expression $V = f(E_1, E_2)$ of height h, where the view expressions E_1 and E_2 are of heights less than h. By the inductive hypothesis, we know that the change-table technique has computed a change table for E_1 and all its subexpressions using minimum number of source queries. We assume that the subexpression E_2 doesn't change. A tuple $(g, u) \in E_1$ changes to (g, u_1) due to changes at E_1, where u is the value of the update attributes (Attrs(U)).

We consider various cases.

Selection: $V = \sigma_q(E_1)$. Unless V is a singularity point, the change-table technique doesn't query any sources to compute the change table $\Box V$ at V. Now, V is a singularity point iff $\Box E_1$ is an aggregate change-table and an attribute of the selection condition q belong to the set of aggregated attributes $Attrs(U)$. As the selection depends on the aggregated attribute, the tuple (g, u) may not be in V but (g, u_1) may pass the condition q. Hence, as u may not be available, in order to computer u_1, it is essential to query the expression E_1, which is what the change-table technique does.

Projection: $V = \Pi_A(E_1)$. In this case, V is a singularity point iff an attribute of the group set G, on the basis of which matching of tuples is done, does not belong to the set of projected attributes A. In E_1, the tuple (g, u) changes to (g, u_1). In absence of some of the G attributes in V, it may be impossible to refresh V based on just the changes to E_1. Hence, it may be necessary to query E_1, when V is a singularity point.

Cross Product: $V = E_1 \times E_2$. Given changes into the expression E_1, it is obvious that one would need to query E_2 in order to compute changes at V.

Union: $V = E_1 \uplus E_2$. After incorporating the improvements stated before this theorem, the change-table technique does not need to query any sources to compute $\Box V$ from $\Box E_1$.

Monus: $V = E_1 \dot{-} E_2$. Given changes into the expression E_1 (E_2), it is easy to see that one would need to query E_2 (E_1 and E_2) in order to compute changes at V.

Monus: $V = E_1 \overset{fo}{\bowtie}_J E_2$. Given changes into the expression E_1, it is easy to see that one would need to query E_2 in order to compute changes at V. ∎

We define an algorithm to be a *change-propagating algorithm* if it computes changes (in some form) at each node in the given expression tree of the view. The above theorem shows that our change-table technique is better than any change-propagating maintenance

algorithm for a given expression tree (even in the presence of singularity points). Though its not necessary for an incremental maintenance algorithm to be change-propagating, all the previously proposed maintenance algorithms fall in that category.

An interesting open problem is to design a provably optimal incremental maintenance algorithm under the above cost model for maintenance of general view expressions (without restricting ourselves to the class of change-propagating algorithms). We have focussed our recent work on adapting the maintenance algorithm based on our change-table techniques to obtain an optimal approach. We conjecture that the change-table technique with some minor improvements/optimizations can be translated into an optimal incremental maintenance algorithm under the above cost model.

4.9 Related Work

A large body of work exists describing different algorithms for incrementally maintaining materialized views [BLT86, RK86, Han87, BCL89, CW91, QW91, GMS93, GLT94, GL95, CGL$^+$96, GJM97, Qua97, GK98]. Each work applies to different classes of views and has various advantages and disadvantages. As mentioned in Section 4.1, none of the above algorithms can deal with general view expressions involving aggregations and outerjoins. Further, these algorithms work with sets (or bags) of insertions and deletions, rather than with change tables. Below, we discuss some of the above algorithms.

Qian and Wiederhold in [QW91] present a technique (later corrected in [GLT94]) to propagate sets of insertions and deletions through relational algebra operators without duplicates. The techniques in [QW91] were extended to bag algebra by Griffin and Libkin in [GL95]. While [QW91] did not deal with aggregates at all, [GL95] did consider aggregates when they are applied as the last operator in an expression, and when there are no groupby columns. The work of [GL95] was further extended by Quass [Qua97] to include general expressions involving aggregation. However, as illustrated in Section 4.1, the technique of [Qua97] works with bags of insertions and deletions, and is thereby less efficient, and more complex, than the change table technique presented in this chapter. None of [QW91, GL95, Qua97] can deal with outerjoins.

Gupta et al. [GMS93] also present algorithms for incrementally maintaining views with duplicates. Aggregation is considered only for the case where a view is materialized at each aggregation node. Outerjoins are not considered in [GMS93]. Mumick et al. in [MQM97]

give an algorithm for efficiently maintaining a set of summary tables. A summary table is the result of applying a single aggregation over an SPJ expression over star schema tables in a data warehouse. [MQM97] does not consider outerjoins, or general view expressions involving aggregate operators.

Gupta et al. in [GJM97] present maintenance and self-maintenance algorithms to compute the incremental changes to a materialized outerjoin view $R \overset{o}{\bowtie} S$, where R and S are base tables. The algorithms presented in [GJM97] are procedural rather than algebraic, and do not apply to general view expressions containing outerjoin operators. In recent work done concurrently with ours, Griffin and Kumar in [GK98] extend the techniques of [GJM97] by deriving propagation equations through outerjoin operators. As [GK98] propagates changes in the form of insertions and deletions, their incremental algorithm is less efficient than our change-table techniques for general view expressions, as briefly illustrated in our motivating example.

The problem of incremental view maintenance is closely related to the self-maintainability problem of views [GJM94, QGMW96, GJM97]. A view is defined as self-maintainable with respect to certain kinds of changes if the view can be updated using the old view value and the changes, without accessing any base relations. The change-table technique presented in this chapter, in most cases, derives maintenance expressions that do not refer to the base tables, even when such self-maintenance expressions were not possible using only insertions and deletions as types of updates. Hence, the techniques presented in this chapter help in deriving efficient self-maintenance expressions.

4.10 Concluding Remarks

In this chapter, we have developed a change-table technique for incremental maintenance of general view expressions involving aggregate and outerjoin operators. Traditional maintenance techniques [QW91, GL95, Qua97] propagate insertions and deletions from the base relations to the view through each of its operators. In contrast, we compute change tables at an aggregate or outerjoin operator, and use change propagation equations to propagate the change tables through various operators. The resulting maintenance expressions for general view expressions are simple and very efficient compared to previous traditional techniques.

Chapter 5

Conclusions

A data warehouse is built for the purposes of information integration and/or decision support and analysis. A data warehouse extracts, integrates and stores relelvant information from a distributed set of the data sources.

In this thesis, we have addressed the core issues that arise in design of a data warehouse viz., selection of views to materialize and efficient incremental maintenance of materialized views. This thesis has made a significant contribution in solving these problems comprehensively.

5.1 Selection of Views to Materialize

In this thesis, we have developed a theoretical framework for the general problem of selection of views in a data warehouse. The view-selection problem in a data warehouse is to select a set of views to materialize so as to optimize the total query response time, under some resource constraint such as total space and/or the total maintenance time of the materialized views. For the simpler constraint of disk space, we have presented competitive polynomial-time heuristics that deliver a solution within a 0.63 factor of the optimal for some important special cases of the problem that occur in practice viz. AND view graphs and OR view graphs. For both these special cases, we have extended the results to graphs with indexes associated with each view. Finally, we have also developed a greedy heuristic (AO-greedy) for general AND-OR view graphs.

All the above described work done on the view-selection problem considered only a disk-space constraint. In practice, the more realistic constraining factor is the total maintenance

time. Thus, we also considered the maintenance-cost view-selection problem where the constraint is the total maintenance time of the materialized views. As the maintenance-cost view-selection problem is intractable, we designed an Inverted-tree Greedy algorithm for the special case of OR view graphs, that provably delivers a solution within a constant factor of the optimal. We also designed an A^* heuristic that delivers an optimal solution for general AND-OR graphs. Our preliminary experiment results are very encouraging for the Inverted-tree Greedy algorithm. The results show that the Inverted-tree greedy almost always delivers an optimal solution and the time taken by the Inverted-tree greedy algorithm for OR view graphs is orders of magnitude less than that taken by the A^* heuristic.

There are still a lot of questions which remain unanswered and need considerable attention. Noteworthy among them are:

1. Does there exist a polynomial time algorithm for the view-selection problem for the special case of AND view graphs?

2. Are there competitive polynomial-time heuristics for other special cases (even without updates)? For e.g., for AND-OR *trees* or binary AND-OR view trees?

3. Are there view-selection approximation algorithms even for optimizing just maintenance costs in any of the special cases?

4. Can we prove any negative results about the approximability of the view-selection problem?

'We believe that the techniques developed in this thesis will offer significant insights into the greedy heuristic and the nature of the view-selection problem in a data warehouse.

5.2 Incremental Maintenance of General View Expressions

In this thesis, we have developed a change-table technique for the problem of incremental maintenance of general view expressions. Ours is the first research work presenting algebraic expressions for maintaining general views involving aggregate and outerjoin operators.

Traditional maintenance techniques [QW91, GL95, Qua97] propagate insertions and deletions from the base relations to the view through each of its operators. In contrast, we compute change tables at an aggregate or outerjoin operator, and use change propagation equations to propagate the change tables through the relational, aggregate and outerjoin

operators. We show that the changes represented in change tables can be applied to its corresponding materialized view using an appropriately defined refresh operator. The resulting maintenance expressions for general view expressions are simple and very efficient compared to previous techniques.

The change-table technique presents a new paradigm for view maintenance using change tables. The maintenance expressions derived by the change-table techniques are usually self-maintenance expressions, as they usually refer only to the view and the changes to the base tables, minimizing the number of queries to the base tables. Such a paradigm is likely to encourage research into developing more efficient maintenance and self-maintenance expressions than are possible using the insertion/deletion paradigm. For example, the change-table technique can be used to (1) efficiently propagate certain kinds of deletions, and (2) for propagating certain kinds of updates directly, without querying the sources.

Our future work focuses on translating our techniques into a *provably* optimal incremental maintenance approach under some reasonable cost model.

Bibliography

[BCL89] J. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In Carlo Zaniolo, editor, *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, pages 61–71, Washington, D.C., May 28-30 1986.

[BM90] Jose A. Blakeley and Nancy L. Martin. Join index, materialized view, and hybrid hash join: A performance analysis. In *Proceedings of the Sixth IEEE International Conference on Data Engineering*, pages 256–263, Los Angeles, CA, February 5-9 1990.

[BPP+93] B.Mitschang, H. Pirahesh, P. Pistor, B. Lindsay, and N. Sudkamp. Sql/xnf - processing composite objects as abstractions over relational data. In *Proceedings of the Ninth IEEE International Conference on Data Engineering*, Vienna, Austria, April 1993.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, Aug 25-29 1997.

[BW89] T. Barsalou and G. Wiederhold. Knowledge based mapping of relations into objects. In *Proceeding of Computer Aided Design*, 1989.

[BW90] B.S.Lee and G. Wiederhold. Outer joins and filters for instantiating objects from relationals databases through views. Technical report, CIFE - Stanford University, 1990.

[CFN77] G. Cornuejols, M. L. Fisher, and G. L. Nemhauser. Location of bank accounts to optimize float: An analytic study of exact and approximate algorithm. *Management Science*, 23(8):789–810, 1977.

[CGL+96] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, pages 469–480, 1996.

[Che96] Chandra Chekuri. Personal Communication, 1996.

[CKL+97] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting multiple view maintenance policies. In *Proceedings of ACM SIGMOD 1997 International Conference on Management of Data*, pages 405–416, 1997.

[CM82] U. S. Chakravarthy and J. Minker. Processing multiple queries in database systems. *Database Engineering*, 5(3):38–44, September 1982.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including groupby in query optimization. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20^{th} International Conference on Very Large Databases*, pages 354–366, Santiago, Chile, September 12-15 1994.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 108–119, Barcelona, Spain, September 3-6 1991.

[Fei96] U. Feige. A threshold of ln n for approximating set cover. In *Proceedings of the Twenty-Eighth annual ACM Symposium on the Theory of Computing*, pages 314–318, Philadelphia, Pennsylvania, May 1996.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections: A powerful approach to aggregation. In *Proceedings of the 21^{st} International Conference on Very Large Databases*, Zurich, Switzerland, September 11-15 1995.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection in OLAP. In *Proceedings of the International Conference on Data Engineering*, Burmingham, U.K., April 1997.

[GJM94] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. Technical Memorandum 113880-941101-32, AT&T Bell Laboratories, November 1994.

[GJM96] A. Gupta, H. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proceedings of the Fifth International Conference on Extending Database Technology*, pages 140–144, Avignon, France, March 1996. Industrial Session.

[GJM97] A. Gupta, H.V. Jagadish, and I.S. Mumick. Maintenance and self-maintenance of outerjoin views. In *Proceedings of the NGITS*, Tel Aviv, Israel, June 1997.

[GK98] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3), September 1998.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, May 1995.

[GLT94] Timothy Griffin, Leonid Libkin, and Howard Trickey. A correction to "incremental recomputation of active relational expressions" by Qian and Wiederhold. Technical report, AT&T Bell Laboratories, Murray Hill NJ, 1994.

[GM95] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applica tions. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin, 18(2):3–19, June 1995.

[GMS93] 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.

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

[Han87] E. Hanson. A performance analysis of view materialization strategies. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of ACM SIGMOD 1987 International Conference on Management of Data*, pages 440–453, San Francisco, CA, May 27-29 1987.

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

[Kim96] R. Kimball. *The Data Warehouse Toolkit.* John Wiley & Sons, Inc., 1996.

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

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

[OG95] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[QGMW96] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fifth International Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[QM97] D. Quass and I. Mumick. Optimizing the refresh of materialized views. Unpublished Manuscript, 1997.

[Qua97] D. Quass. *Materialized Views in Data Warehouses.* PhD thesis, Stanford University, Department of Computer Science, 1997. Chapter 4. Preliminary version appears as *Maintenance Expressions for Views with Aggregation* in the *ACM Workshop on Materialized Views*, 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, pages 337–341, 1991.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS±. *IEEE Computer*, pages 19–25, December 1986.

[Rou82a] N. Roussopoulos. The logical access path schema of a database. *IEEE Transaction in Software Engineering*, SE-8(6):563–573, November 1982.

[Rou82b] N. Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

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

[RU96] A. Rajaraman and J. Ullman. Integrating information by outerjoins and full disjunctions. In *Proceedings of the Fifteenth Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, June 1996.

[Sel88] Timos K. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[TS97] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, Aug 25-29 1997.

[Wid95] J. Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, November 1995.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, Aug 25-29 1997.

[YL95] W. Yan and P. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the Twenty-First International Conference on Very Large Databases (VLDB)*, pages 345–357, 1995.