Metadata View Graphs: Maintaining Dynamic Semantic Rules for Query Optimization

Jeff Pittges
Leo Mark
Sham Navathe

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{pittges, leomark, sham}@cc.gatech.edu

October 1, 1995

Abstract

Although previous experimental results have shown that dynamic semantic rules may significantly reduce query execution time, the problem of how to maintain dynamic semantic rules has remained an open research problem. The Metadata View Graph is a metadatabase that maintains semantic, statistical, and structural metadata for views of a database. Capturing dynamic rules for relevant views increases the amount of information contained in the rules and improves the performance of discovering, retrieving, maintaining, and using the rules.

This paper presents a set of strategies, methods, and algorithms for efficiently maintaining dynamic rules in the Metadata View Graph Framework. The paper describes how dynamic rules are retrieved, maintained, and applied during query processing. The paper also presents a run-time cost model that determines when metadata maintenance is cost effective during query processing.

Keywords: Metadata View Graph, Metadata Maintenance, Dynamic Semantic Rules, Query Optimization.

1 Introduction

Previous experimental results have shown that semantic query optimization may significantly reduce query execution time by reformulating a query into a semantically equivalent query that is more efficient to execute [HK94, HK93, SSS92, SSD92, CGM90, SO89, Kin81, HZ80]. Semantic query optimization has traditionally derived transformation rules from *schema-based constraints* which are valid for all instances of a database. However, schema-based constraints are often difficult to define and are typically too general to be useful. Consequently, a number of researchers have developed methods for discovering *dynamic rules* which are only valid for particular instances of a database. Dynamic rules contain more information than schema-based constraints because they represent the current contents of the database. For example, consider an employee base relation with four attributes: (1) employee number, (2) salary, (3) department, and (4) position. As shown below, a schema-based constraint may assert that the salary of all employees is greater than or equal to zero, assuming that some of the employees may be volunteers. Alternatively, a set of dynamic rules can maintain the minimum and maximum salary for the current state of the database.

Schema-Based Constraint	**Dynamic Rules**
Salary ≥ 0	Salary $\geq$ \$22,000
	Salary $\leq$ \$85,000

Now consider the following query which selects all of the employees who earn \$90,000 or more. Schema-based constraints cannot define a useful upper bound on salary since the upper bound would have to apply to all instances of the database. Therefore, schema-based constraints are unable to provide any useful information for this query. Dynamic rules, however, can maintain an upper bound on salary since they are only valid for the current state of the database. In this example, the value of the maximum salary can be used to determine that none of the employees in the database satisfy the query. Therefore, the query can be answered without retrieving a single data page.

Relation	**Attributes**		
Employees	Enum, Salary, Dept, Pos		

Select	Enum
From	Employees
Where	Salary $\geq$ \$90,000

Although dynamic rules are more useful for semantic query optimization than schema-based constraints, dynamic rules must be maintained when changes are made to the database. Although several researchers have acknowledged the need to maintain dynamic rules [YS89, SSS92], no one has formally addressed the problem. This paper presents a set of strategies, methods, and algorithms for maintaining dynamic semantic rules in the Metadata View Graph Framework. The rest of the paper is organized as follows. Section 2 describes the Metadata View Graph, Section 3 illustrates how query execution plans are selected at run-time, Section 4 presents our maintenance algorithms, and the last section contains our concluding remarks.

2 Metadata View Graphs

The Metadata View Graph [PMN95] is a metadatabase capable of maintaining semantic, statistical, and structural metadata for views of a database. As shown in Figure 1, the Metadata View Graph consists of four components: (1) a lexicon, (2) a semantic network, (3) a view network, and (4) a QEP Network.

The lexicon maintains the system's vocabulary and contains information for each word or phrase that is recognized by the system. The semantic network captures domain knowledge and can be used to disambiguate queries. The lexicon and semantic network are used for query interpretation and will not be discussed further.

The View Network is an extension of Roussopoulos' Logical Access Path schema [Rou82]. The View Network stores semantic, statistical, and structural metadata that improves query optimization. The view nodes in the network ($V_1 - V_{11}$ in Figure 1) represent views of the database and store metadata specific to the particular data set. The links represent logical operations and semantic relationships. The View Network is essentially a collection of query graphs overlaid on top of each other and each view node represents an intermediate result.

The QEP Network stores query execution plans. The network contains a *query execution packet* for each query that is compiled by the system. Query execution packets store two types of query execution plans, *dynamic plans* and *semantic plans*. A dynamic plan [GW89, CG94] links several query execution plans together with *choose-plan operators* that allow a decision to be postponed until the run-time conditions are known. The choose-plan operators are linked to the view nodes that contain relevant statistics. Semantic plans contain pointers to the rules that they depend on. These rules must be updated and validated before a semantic plan can be executed.

3 Query Optimization

When a query is processed at compile-time, several query execution plans are generated and metadata is collected for each intermediate result. We assume that existing methods have been used to discover dynamic rules. These rules are used to generate semantic plans and the statistics that are collected are used to evaluate each query execution plan. The most efficient query execution plans are stored in the QEP Network.

When a query is processed at run-time, the query's execution packet is retrieved from the QEP Network along with any relevant view nodes from the View Network. The semantic plans are indexed such that the run-time bindings of the query can be used to select the semantic plans that match the conditions of the query.

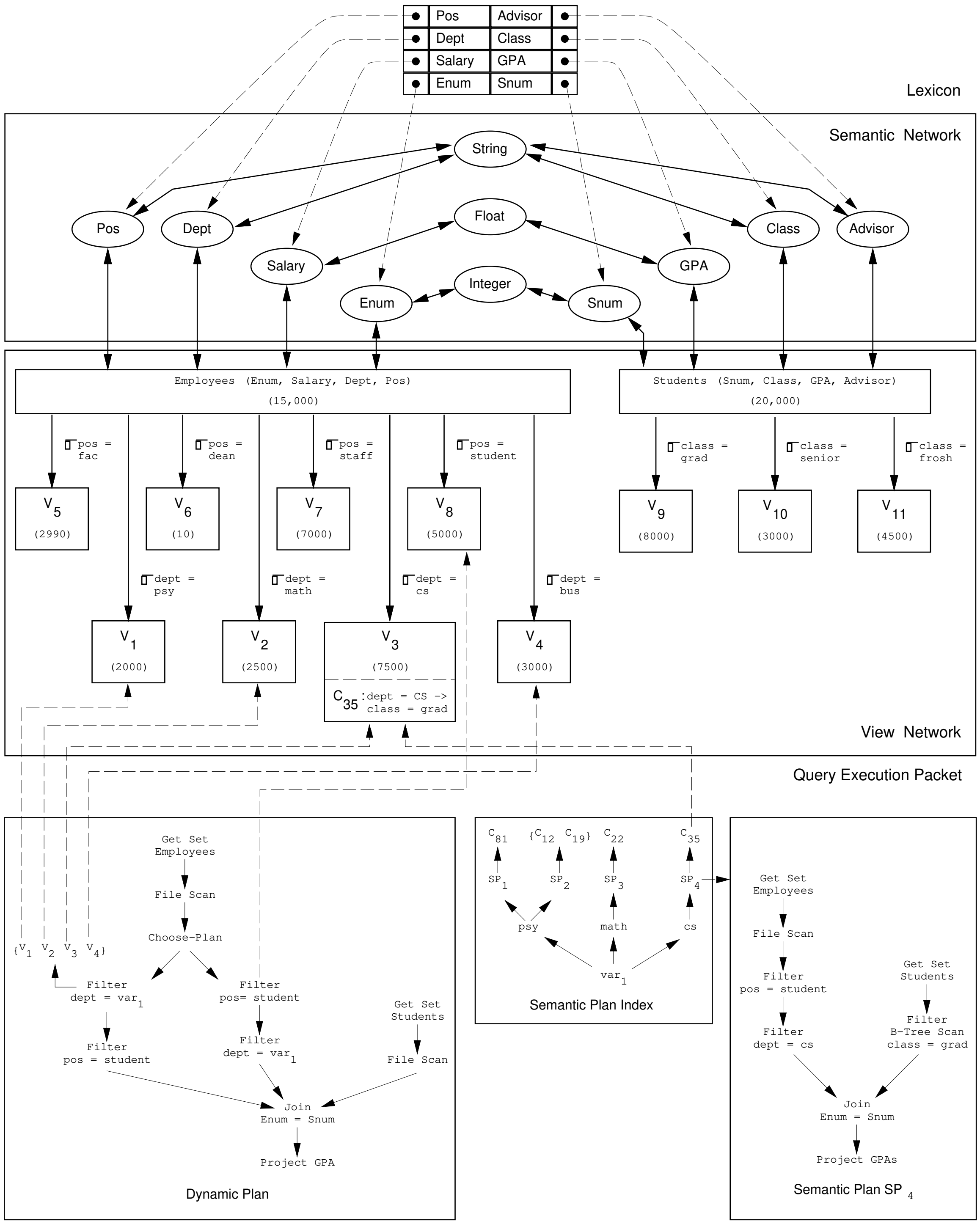

Figure 1: An example Lexicon, Semantic Network, View Network and Query Execution Packet.

3.1 Selecting a Query Execution Plan at Run-Time

Consider the two base relations and the *template query*, Q_1, shown below. A template query contains one or more variables. Instantiations of a template query are received at run-time with all of the variables bound.

Relation	**Attributes**
Students	snum, class, GPA, advisor
Employees	enum, salary, dept, pos

Q_1:	Select	GPA
	From	Employees, Students
	Where	pos = student AND dept = var_1
		AND enum = snum

Query Q_1 requests the grade point averages of the students in the var_1 department who are employeed. Figure 1 illustrates part of a View Network that supports this query along with the Query Execution Packet for Q_1. Since the two selections can be performed in either order, the dynamic plan contains a choose-plan operator that allows the order to be determined at run-time when the variable var_1 is bound. The first two filter operations point to the view nodes that contain statistics to determine which path to take. Tuple counts are shown in parentheses at each view node in Figure 1. Once var_1 is bound at run-time, the choice is obvious. If DEPT = CS, then POS = STUDENT produces a smaller intermediate result (5000 tuples) than DEPT = CS (7500). For the other three departments, however, selecting the department first produces the smaller intermediate result.

Semantic plans are indexed by the values of the variables. The packet contains four semantic plans, SP_1 – SP_4, corresponding to the variable bindings var_1 = PSY, var_1 = MATH, and var_1 = CS. Each plan points to the set of rules that the plan depends on.

Assume an instantiation of query Q_1 is received at run-time with var_1 bound to CS. The semantic plan index would return plan SP_4 which depends on rule C_{35} stored at node V_3. The semantic plan SP_4 is shown in Figure 1. Rule C_{35} states that if the department is CS then the class must be GRAD. In other words, only graduate students from the computer science department are employed.

Assuming there is an index on the CLASS attribute of the Student base relation, the semantic plan SP_4 can use that index to reduce the size of the join between the Student base relation and the view of employed computer science students. Since C_{35} is a dynamic rule, C_{35} must be refreshed and validated before SP_4 can be executed.

4 Maintaining Dynamic Semantic Rules

The semantic, statistical, and structural metadata stored in the View Network represents the current contents of the database. Consequently, the metadata must be maintained when changes are made to the database. Maintenance can occur at compile-time or run-time. Although the algorithms are essentially the same at both times, efficiency is a major factor at run-time. Therefore, this section primarily focuses on strategies, methods, and algorithms for efficiently maintaining metadata at run-time.

The section begins by describing the general maintenance problem. Several strategies for managing update logs are presented followed by strategies for managing view nodes. Finally, a run-time cost model is presented which is used to determine when metadata maintenance is cost effective during query processing.

4.1 The Maintenance Problem

When a database evolves from one database state x to another database state y due to an update (i.e., a tuple is inserted, deleted, or modified), the metadata in the View Network may become invalid. In general, two actions must occur before the metadata can be used: (1) the metadata that is affected by the update must be identified, and (2) if the metadata is no longer valid, the metadata must be modified or invalidated.

As described in Section 3.1, when a query is processed at run-time, the most efficient query execution plan is selected from the query's execution packet based on the metadata contained in the View Network. Typically, several plans will be considered. The cost of each candidate plan must be estimated in order to select the best plan. Cost estimates depend on the statistics stored at the view nodes that are indexed by the plans. If enough updates have been made to the database, the statistics may need to be refreshed. In addition, if a semantic plan is chosen, the rules that the plan depends on must be refreshed.

Rather than consider each update as it is received, it is more efficient to process a batch of updates for a particular set of metadata when the metadata is needed. Therefore, the system maintains an update log for each base relation. When an update is made to a base relation, the system writes the update to the base relation's update log. When the metadata at a view node needs to be refreshed, all of the updates for the base relations from which the view is derived are processed. However, only a subset of the updates will apply to a given view node based on the definition of the node (e.g., GPA $\geq$ 3.0). Therefore, the updates must be *filtered* to remove the *irrelevant updates* [BLT86]. An example is provided below.

Figure 2 illustrates the contents of a base relation and a view node for two states of an example database. The figure also illustrates an update log for the base relation which contains a number of tuples to be inserted into the base relation. State x represents the database before the updates are received, and state y represents the database after the updates have been processed and the metadata at the view node has been refreshed.

The view node shown in this example contains seven rules, a tuple count, a distribution profile, and a view cache pointer (since the pointer is nil, there is no view cache for this node). The view is defined for students with a GPA of 3.0 or greater. The rules and tuple count at the view node can be verified for states x and y by selecting the tuples from the base relation (as shown) that satisfy the definition of the node (i.e., GPA $\geq$ 3.0). An update contains a unique time-stamp, which indicates when the update was received, along with the tuple to be inserted, deleted, or modified. Updates to the database are maintained in a set of logs, one log per base relation. In order to refresh the metadata at a view node, the

State X

Students (Snum, Class, GPA, Advisor)

2	Grad	3.8	Smith
3	Frosh	3.2	Jones
6	Soph	2.7	Smith
8	Grad	3.6	Davis
9	Frosh	2.9	Davis
11	Grad	4.0	Davis
12	Senior	3.9	Jones
16	Junior	3.3	Smith
21	Soph	3.7	Jones

σ GPA >= 3.0

View Node

C_1	2 <= Sno <= 21	T3
C_2	Frosh <= Class <= Grad	T12
C_3	3.2 <= GPA <= 4.0	T3
C_4	Davis <= Advisor <= Smith	T9
C_5	Class = Grad -> GPA >= 3.6	T2
C_6	Advisor = Jones -> GPA >= 3.2	T5
C_7	GPA = 4.0 -> Sno = 11	T4
	Tuple Count: 15	T12
	distributions	T9
	View Cache: nil	

Updates

Base Relation Update Log

4	Junior	3.1	Smith	T3
22	Soph	2.8	Jones	T4
10	Grad	3.9	Jones	T5
1	Grad	4.0	Davis	T8
14	Frosh	3.3	Smith	T9
13	Senior	3.7	Smith	T11
5	Grad	3.9	Jones	T12
7	Soph	3.2	Davis	T15
18	Junior	1.8	Smith	T16
20	Grad	3.6	Davis	T17
23	Frosh	2.7	Davis	T19

Filter (GPA >= 3.0)

4	Junior	3.1	Smith	T3
10	Grad	3.9	Jones	T5
1	Grad	4.0	Davis	T8
14	Frosh	3.3	Smith	T9
13	Senior	3.7	Smith	T11
5	Grad	3.9	Jones	T12
7	Soph	3.2	Davis	T15
20	Grad	3.6	Davis	T17

State Y

Students (Snum, Class, GPA, Advisor)

1	Grad	4.0	Davis
2	Grad	3.8	Smith
3	Frosh	3.2	Jones
4	Junior	3.1	Smith
5	Grad	3.9	Jones
6	Soph	2.7	Smith
7	Soph	3.2	Davis
8	Grad	3.6	Davis
9	Frosh	2.9	Davis
10	Grad	3.9	Jones
11	Grad	4.0	Davis
12	Senior	3.9	Jones
13	Senior	3.7	Smith
14	Frosh	3.3	Smith
16	Junior	3.3	Smith
18	Junior	1.8	Smith
20	Grad	3.6	Davis
21	Soph	3.7	Jones
22	Soph	2.8	Jones
23	Frosh	2.7	Davis

σ GPA >= 3.0

View Node

C_1	1 <= Sno <= 23	T19
C_2	Frosh <= Class <= Grad	T19
C_3	3.1 <= GPA <= 4.0	T19
C_4	Davis <= Advisor <= Smith	T19
C_5	Class = Grad -> GPA >= 3.6	T19
C_6	Advisor = Jones -> GPA >= 3.2	T19
C_7	GPA = 4.0 -> Sno = 1 or 11	T19
	Tuple Count: 23	T19
	distributions	T9
	View Cache: nil	

Figure 2: The base relation update log contains a set of insertions that are applied to the view node as the database evolves from state x to state y.

updates in the base relation update log must be filtered to select the updates that satisfy the definition of the node. By propagating the updates through the View Network, the system can identify which metadata is affected by an update. When an update applies to a node, the metadata can be tested to determine which metadata is no longer valid, and the system can modify or invalidate the incorrect metadata.

Before the updates can be filtered and propagated through the View Network, the update logs and the view nodes to be refreshed must be retrieved from disk. The cost of these disk accesses dominates the cost of the maintenance process. Therefore, maintaining metadata can be divided into three subproblems: (1) managing the update logs, (2) managing the view nodes, and (3) refreshing the metadata. Section 4.2 discusses several strategies for managing update logs, section 4.3 discusses strategies for organizing view nodes on disk, and Section 4.4 reviews several methods for refreshing dynamic rules.

4.2 Managing Update Logs

The cost of metadata maintenance is dominated by disk I/O, primarily reading and writing the update logs. Two factors affect disk access time: (1) the number of logs that are read and written, and (2) the number of log pages that are read and written. Because the pages of a log are stored in contiguous memory blocks, disk I/O costs can be minimized by reading from and writing to a single log, thus taking advantage of prefetched pages and other benefits, rather than reading and writing multiple logs which are stored at different locations. However, the total number of pages accessed may be reduced by reading and writing several smaller logs that have been filtered.

When the metadata at a node is refreshed, the base relation update log is filtered to select the updates that apply to the node. If the selectivity of the filter is high enough (i.e., the filter eliminates enough updates), the cost of future updates may be reduced by saving the filtered updates in another log. This leads to a tradeoff between the number of logs and the size of each log. The number of logs should be increased if the additional logs reduce the average number of disk pages read and written when updating a set of view nodes.

Three log processing strategies will be considered. The first strategy, Universal Logs, maintains a single log at each base relation. This strategy meets our first objective (minimize the number of logs). The second strategy, View Logs, maintains a log at each view node. Although this strategy violates the first objective (minimize the number of logs), it reduces the total number of log pages read. The third strategy, 50-Percent Logs, attempts to optimize the number of logs and log pages that are processed.

4.2.1 Universal Logs

Our first objective is to minimize the number of logs. The Universal Logs strategy takes this to the extreme by maintaining a single log for each base relation. Updates are added

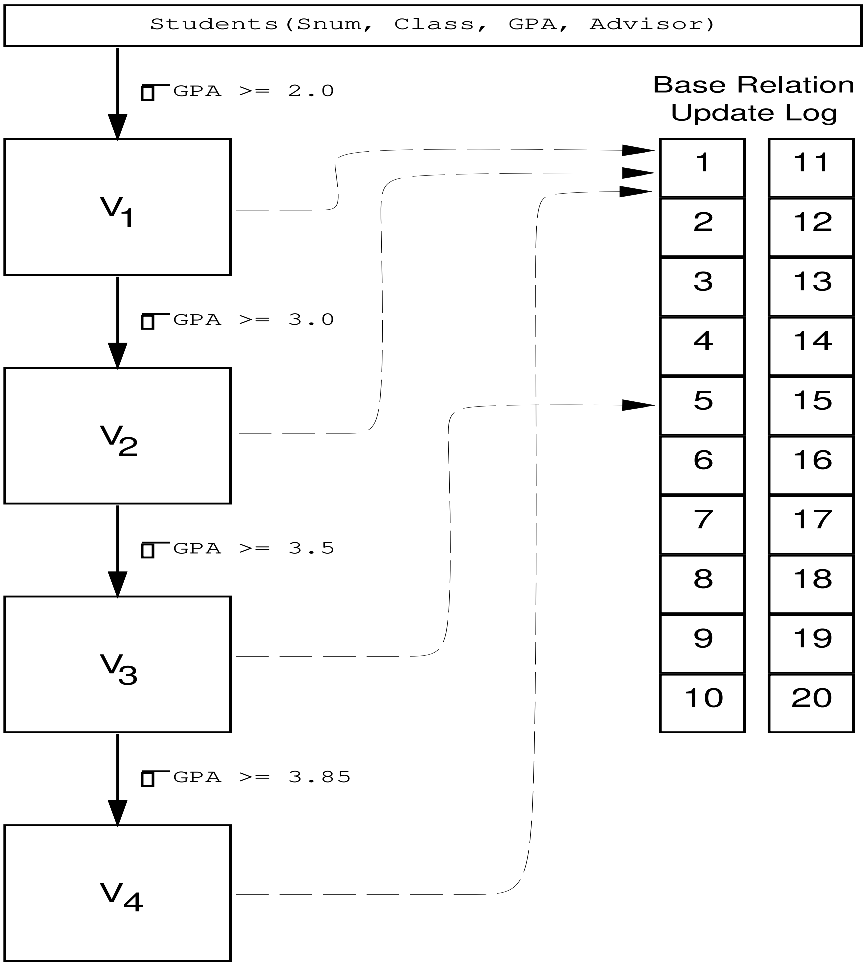

Figure 3: The Universal Logs Strategy.

to the base relation update log as they are received (i.e., time-stamp order). Each view node is refreshed independently of the other view nodes. In other words, a node in the View Network can be refreshed without refreshing any other node(s). Therefore, each view node maintains a pointer into the base relation update log to reference the last update that was applied to the node. To refresh a single node, the node's log pointer is used to retrieve the set of updates from the base relation update log that have not been processed for the node. The unprocessed updates are then filtered and applied to the view node. To refresh a set of nodes, the log pointer of the node that is most out of date (i.e., points to an update with the smallest (oldest) time-stamp) is used to retrieve the set of unprocessed updates from the base relation update log. Each update is filtered and applied to the nodes that have not yet processed the update. The algorithm is given below.

An Example

Consider the Student base relation node, base relation update log, and the four view nodes shown in Figure 3. Each numbered box in the base relation update log represents a memory page on disk. Each view node has a pointer into the base relation update log. Assume that node V_2 is updated at time T_1 and node V_4 is updated at time T_2. At T_1, V_2's log pointer is used to retrieve the first log page, and all 20 pages are read in order to update V_2. At time T_2, the same log pages must be read a second time in order to update node V_4. The total disk I/O for this example is 40 page reads and no writes.

Analysis

This strategy has several advantages because it only uses one log. First, prefetched disk pages are utilized most efficiently because all of the updates are stored on consecutive memory pages. Second, having only one log makes the retrieval algorithm fairly simple. Finally, each update is only stored once which makes this strategy the most space efficient.

The disadvantage is that the same log pages are read repeatedly. The maintenance cost is not amortized in any way. In the worst case, each log page will be read once for each view node. Saving the filtered updates (i.e., those updates that apply to the node being updated) to another log would amortize the maintenance cost and reduce the number of log pages retrieved for future updates. In addition, each time the base relation update log is filtered for V_4, it could also be filtered for V_1, V_2, and V_3. The next two strategies will attempt to reduce the number of log pages read and written by maintaining multiple update logs.

```
INPUT: LOG,                // the base relation update log to be processed
       NODE;               // the view node to be refreshed

universal_log_update(LOG, NODE)
/* retrieve the unprocessed updates */
read(LOG, NODE.log_ptr)
While (Not end of log)
    read(LOG, update)

    /* filter the update */
    If (update applies to NODE)

        /* refresh constraints */
        cons_ptr ← NODE.cons_list
        While (cons_ptr <> nil)
            refresh_constraint(cons_ptr, update)
            increment cons_ptr

        /* refresh statistics */
        refresh_stats(NODE.stats, update)
```

Algorithm 1: Universal Log Update

4.2.2 View Logs

Although the Universal Logs strategy minimized the number of logs, it did nothing to reduce the number of log pages to be read during an update. As shown in Figure 4, the View Logs strategy maintains an update log for each view node (i.e., a view log at each view node), except for leaf nodes. Each view node maintains a log pointer into its view log to indicate the last update that was processed for the view node. Each view log also maintains a pointer into the log above to indicate the last update that was processed for the view log.

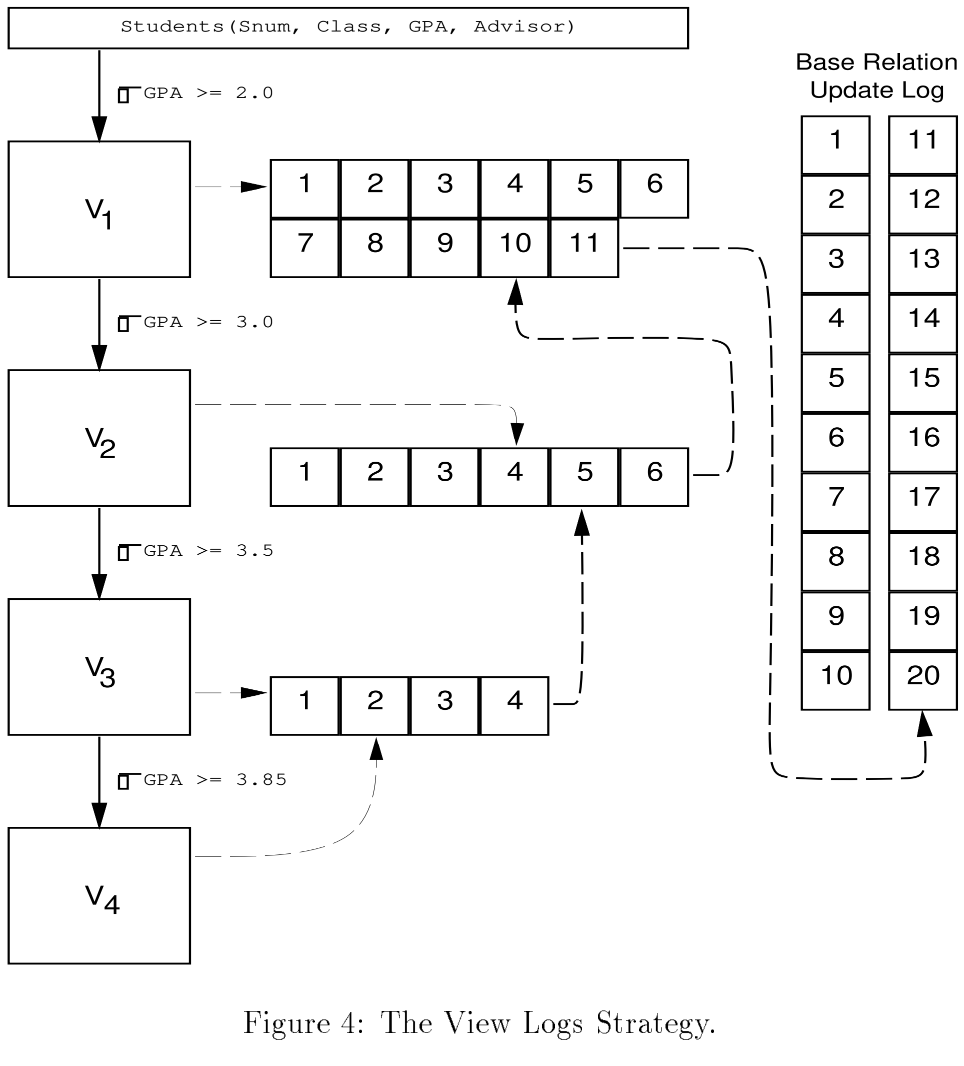

Figure 4: The View Logs Strategy.

View nodes are refreshed independently of other view nodes, just as they were in the Universal Logs strategy. In other words, a node in the View Network can be refreshed without refreshing any other node(s). The view logs are updated independently of the view nodes. If a view log is updated, the log's view node does not have to be refreshed. However, the view logs do depend on other logs. Given a view node to be refreshed, the node's *update path* is the path from the given view node to the base relation node. The logs along the update path form a *log chain*. To update a view log, all of the logs in the log chain are updated from top to bottom (i.e., from the base relation update log to the view node being updated). Each update in the base relation update log is filtered down through the log chain. When an update is eliminated, it will never be considered by any of the view logs below. Therefore, each view log filters the updates for the view logs below. This reduces the cost of filtering and reduces the number of pages in each log.

To refresh a node, the node's view log is updated (subsequently updating the entire log chain as described above) and then the unprocessed updates in the node's view log are applied to refresh the metadata at the node. The algorithm is given below.

An Example

Consider the base relation node, base relation update log, and four view nodes shown in Figure 4. Each view log has a pointer (shown in bold) into the view log above and each view node has a pointer into the node's view log. Since V_4 is a leaf node, its log pointer refers to V_3's view log. Assume that all of the view logs are empty at time T_1, and assume that V_1's log pointer references the first page of the base relation update log. Further assume that node V_2 is refreshed at time T_1 and node V_4 is refreshed at time T_2.

At T_1, node V_2 is refreshed by following the log chain to the base relation update log and processing the base relation update log and the view logs at V_1 and V_2. The updates in the base relation update log that have not been processed for V_1 are filtered. The updates that apply to V_1 are written to V_1's view log. Next, the updates in V_1's view log that have not been processed for V_2 are filtered. The updates that apply to V_2 are written to V_2's view log. Finally, the metadata at V_2 is refreshed by processing the updates in V_2's view log that have not yet been applied to the metadata.

At time T_2, V_3's view log is still empty and V_3's log pointer is pointing to the first page of V_2's view log. In order to update V_4, the log chain is followed to the base relation update log and all of the logs in the chain are processed. Since there are no new updates in the base relation update log and there are no new updates in the view logs at V_1 and V_2, nothing is added to the view logs at V_1 and V_2.

After V_2's view log is processed, V_2's view log is filtered for V_3. The updates that apply to V_3 are written to V_3's view log. Next, the updates in V_3's view log that have not been filtered for V_4 are filtered. The metadata at V_4 is refreshed by processing the updates in V_3's view log that have not yet been applied to the metadata. A total of 47 disk pages were accessed in this example (26 page reads and 21 page writes). Although the View Logs strategy read 14 less pages than the Universal Logs strategy (26 - 40), the View Logs strategy accessed seven more total pages (47 - 40). In other words, the View Logs strategy wrote more pages than it saved.

The nodes along the log chain do not have to be refreshed when the logs are processed (e.g., V_1 at T_1, and V_1, V_2, and V_3 at T_2). The nodes that are refreshed will depend on the node management strategy being used (see section 4.3). Also, view logs are not maintained for leaf nodes since there is no reason to save updates unless there are nodes below.

```
INPUT:  LOGS,            // list of update logs to be processed
        NODE;            // the view node to be refreshed

view_log_update(LOG, NODE)
/* retrieve the update logs */
next_log ← pop(LOGS)
While (LOGS <> nil)
    log ← next_log
    next_log ← pop(LOGS)
    read(log, next_log.log_ptr)
    While (Not end of log)
        read(log, update)

        /* filter the update */
        If (update applies to next_log)
            write(next_log, update)

/* refresh the node */
read(next_log, NODE.log_ptr)
While (Not end of next_log)
    read(next_log, update)

    /* refresh constraints */
    cons_ptr ← NODE.cons_list
    While (cons_ptr <> nil)
        refresh_constraint(cons_ptr, update)
        increment cons_ptr

    /* refresh statistics */
    refresh_stats(NODE.stats, update)
```

Algorithm 2: View Log Update

Analysis

The View Logs strategy takes advantage of each filter operation in the log chain. Therefore, rather than filtering the base relation update log each time a view node is refreshed, each view log processes the view log above. The updates that did not apply to the log above are never considered by a log below. This may require more work for an individual query, since all of the logs in the chain are processed, but over time the strategy reduces the average maintenance cost per query by amortizing the maintenance cost over a set of queries.

Although the View Logs strategy maximizes the number of logs, it minimizes the number of log pages that will be read during an update. However, the strategy also greatly increases the number of log pages that must be written to disk since an update must be written to each log that the update applies to. Therefore, in the average and worst case, the total disk I/O is greater for the View Logs strategy than the Universal Logs strategy. Writing an update to multiple logs also makes the View Logs strategy space inefficient (in the worst case, an update may be stored once in each view log).

4.2.3 50-Percent Logs

The Universal Logs strategy maintains a single log for each base relation which minimizes the number of logs to be read, but does nothing to reduce the number of pages that must be read when the metadata is refreshed. The View Logs strategy increases the number of logs to be read, but decreases the number of pages to be read by filtering the updates at each node. Therefore, as updates are eliminated at each node, the filtered updates can be written to a separate log with fewer pages. Unfortunately, the cost of writing the updates to the view logs makes the View Logs strategy more expensive than the Universal Logs strategy in the average and worst case. The 50-Percent Logs strategy offers a compromise solution that minimizes the total disk I/O.

The total disk I/O for the View Logs strategy was greater than the Universal Logs strategy because the View Logs strategy wrote more pages than it saved. In order to achieve a compromise between these two strategies, the number of log pages that are written must be balanced against the number of log pages that will be saved the next time the logs are read. The following equation represents the break-even point.

$$read_1 + written_1 + read_2 = 2 * read_1$$

$$written_1 + read_2 = read_1$$

$$2 * written_1 = read_1$$

$$written_1 = \frac{read_1}{2}$$

If the number of pages read the first time the logs are processed ($read_1$), plus the number of pages written the first time the logs are processed ($written_1$), plus the number of pages read the next time the logs are processed ($read_2$) is equal to the number of pages that would be read if the original log was read both times the logs were processed ($2*read_1$), then the total disk I/O is the same for both strategies. As the calculations above show, the break-even point occurs when the number of pages written is equal to half of the pages read the first time the logs are processed. If the number of pages written is greater than half the pages read, then the cost of the write plus the cost of the second read is more than the cost of reading the original log a second time. However, if the number of pages written is less than half of the pages read, then the cost of the write plus the cost of the second read is less than the cost of reading the original log a second time. This analysis leads to the 50-Percent Rule.

$$\boxed{\text{written} \le \text{read} * 0.5}$$

The 50-Percent Rule

The 50-Percent Logs strategy modifies the View Logs strategy such that the view logs are allocated based on the 50-Percent Rule. If the selectivity factor at a view node with respect to the nearest log on the node's update path is 50 percent or less (i.e., half, or less than half, of the updates in the log apply to the node), then a view log is created at the view node.

An Example

Consider the following example, shown in Figure 5. The View Network contains three view nodes ($V_1 - V_3$) and a base relation node. The base relation update log contains 200 updates (20 pages). 150 of the updates in the base relation update log apply to V_1, 80 apply to V_2, and 50 apply to V_3. Since the selectivity factor between the base relation update log and node V_1 is 75 percent, no log is created at V_1. However, since the selectivity factor between the base relation update log and node V_2 is less than 50 percent, a log is created at V_2. Since node V_3 is a leaf node, no log is created at V_3. In the example, the logs are processed three times: V_1 is refreshed at time T_1, V_2 is refreshed at time T_2, and V_3 is refreshed at time T_3.

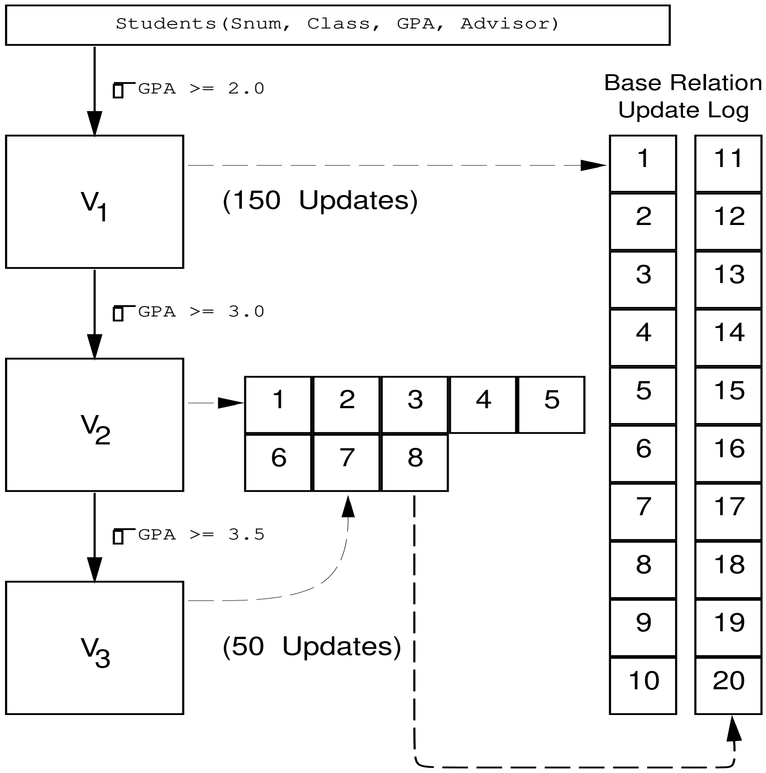

Figure 5: 50-Percent Logs.

At time T_1, 200 updates (20 pages) are read from the base relation update log. The updates are filtered and the metadata at V_1 is refreshed. At time T_2, 200 updates (20 pages) are read from the base relation update log. 80 of these updates apply to V_2, 80 updates (8 pages) are written to the view log at V_2 and the metadata at V_2 is refreshed. At time T_3, 80 updates (8 pages) are read from the log at V_2. The updates are filtered and the metadata at V_3 is refreshed. The total disk I/O is 20 + 20 + 8 + 8 = 56.

Analysis

This section compares all three strategies using the example given above. First, consider the Universal Logs strategy. Each time the logs are processed, all 200 updates (20 pages) will be read. Since the logs are processed three times, the total disk I/O is 60 pages.

The View Logs strategy will create a view log at each view node. At time T_1, 200 updates (20 pages) will be read. Since 150 updates (15 pages) apply to V_1, 15 pages will be written to the view log at V_1. At time T_2, 150 updates (15 pages) will be read from the log at V_1 and 80 updates (8 pages) will be written to the log at V_2. At time T_3, 80 updates will be read from the log at V_2. The total disk I/O is 66 pages.

The 50-Percent Logs strategy was described above. The results of this comparison are tabulated in Table 1. Based on the 50-Percent Rule, the 50-Percent Logs strategy never does worse than the Universal Logs strategy or the View Logs strategy.

	T_1		T_2		T_3		
	Reads	Writes	Reads	Writes	Reads	Writes	Total I/O
Universal Logs	20	0	20	0	20	0	60
View Logs	20	15	15	8	8	0	66
50-Percent Logs	20	0	20	8	8	0	56

Table 1: Comparison of all three log strategies.

Two approaches may be taken when considered processing time: (1) minimize the time required for an individual query (per query processing time), or (2) minimize the average time for a set of queries (average processing time per query). The Universal Logs strategy minimizes the processing time per query by only maintaining and processing a single update log. Therefore, whenever a view node is refreshed, none of the other nodes benefit from the maintenance cycle. The entire process is devoted to the node being refreshed in order to minimize the maintenance cost. The View Logs strategy and the 50-Percent Logs strategy minimize the average time per query by processing a set of logs, thus amortizing the cost of the maintenance cycle over a set of queries. When a view node is refreshed, the current maintenance cycle benefits from previous maintenance cycles and performs additional work to benefit future maintenance, thereby reducing the amount of duplicate effort and minimizing the average cost per query.

Since disk I/O dominates CPU processing time, we assume that all of the metadata at a node is refreshed at the same time in order to fully utilize the disk access required to retrieve the node. This means that a node will never consider an update more than once. Therefore, unless an update applies to a node below, the update does not have to be saved for future consideration once it has been applied to the metadata.

If processing time approaches disk access time, then it may become more efficient to only refresh some of the metadata at a view node when the node is refreshed. This decision will affect the analysis of the log strategies. Assuming that some of the metadata at a node is refreshed at different times, the same node may read the same log page several times until all of the updates on the page have been applied to all of the metadata at the node. This applies to all three strategies. Under this assumption, the View Log strategy becomes more attractive because a node will never consider an irrelevant update more than once. Only the updates that apply to a node are written to the node's view log, thus minimizing the total disk I/O since the minimum number of updates are read each time the node is refreshed. This is not true for the Universal Logs and 50-Percent Logs strategies since the view nodes that do not have view logs must repeatedly read the logs above which contain irrelevant updates that do not apply to the node being refreshed, thereby increasing the total disk I/O (i.e., increasing the number of pages that must be read).

4.2.4 Allocating Logs

The 50-Percent Logs strategy is optimal because it minimizes the number of logs in the network and minimizes the number of log pages that are read and written during each update. However, in order to allocate the view logs correctly, the 50-Percent Logs strategy requires the system to predict the distribution of updates. This is accomplished by using the tuple count at each view node. If the database is fairly stable, the current tuple distribution should provide an accurate approximation of the update distribution. Therefore, view logs can be allocated statically at compile-time according to the tuple counts at the view nodes. If the distribution of the database changes, the view logs can be reallocated during periodic recompilations. Recompilation could also be triggered by monitoring the distribution at run-time.

Consider the network shown in Figure 6. The tuple count for each view is shown in parentheses. To allocate logs for this network, the process beings at the base relation node. The algorithm will traverse the tree in depth-first order and assign view logs based on the predicted selectivity factor. Each time a view log is allocated, the process starts over for the subtrees subsumed by the new view log. The algorithm is given below.

The number of tuples at V_1 (900) is 45 percent (900/2000) of the tuples at the base relation node (2000). Therefore, the selectivity factor for V_1 with respect to the base relation is less than 50 percent, so a view log is created at V_1 as indicated by the shaded box. The algorithm proceeds to V_2 and compares the percentage of tuples at V_2 with respect to V_1 ($350/900 = 39\%$). Since the selectivity factor for V_2 with respect to V_1 is less than 50 percent, a view

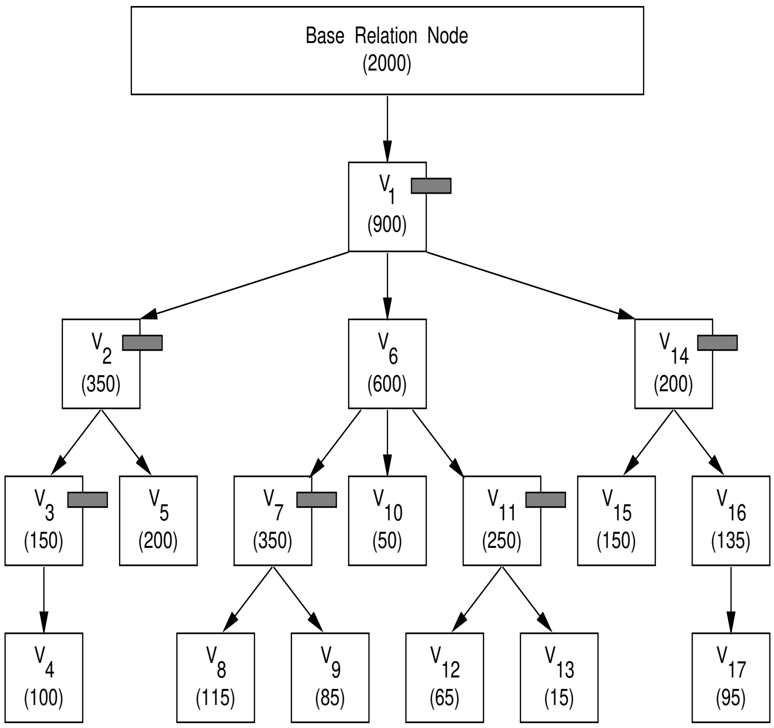

Figure 6: Allocating 50-Percent Logs.

log is created at V_2. Next, the algorithm considers V_3 with respect to V_2 (150/350 = 43%). Once again the selectivity factor is less than 50 percent, so a view log is created at V_3. The algorithm considers V_4 and V_5, but since these are both leaf nodes, no view logs are allocated.

```
INPUT:  TC,                   // tuple count for parent log
        ROOT;                 // root node for a tree of nodes

allocate_view_logs(TC, ROOT)
i = 0
While (ROOT.next[i] <> nil)
    If ((ROOT.next[i].tuple_count ≤ 0.5)

        /* check for leaf node */
        If (ROOT.next[i].next[0] = nil)
            create_view_log(ROOT.next[i])
            allocate_view_logs(ROOT.next[i], ROOT.next[i].tuple_count)
    Else
        allocate_view_logs(ROOT.next[i], TC)
    i = i + 1
```

Algorithm 3: Allocating View Logs

The algorithm then considers V_6 with respect to V_1 ($600/900 = 67\%$). Since the selectivity factor is greater than 50 percent, the algorithm proceeds to node V_7. The selectivity factor for V_7 with respect to V_1 ($350/900 = 39\%$) is less than 50 percent, so a view log is created at V_7. The algorithm continues to traverse the network in depth-first order (i.e., the order in which the nodes are numbered), and view logs are created as indicated by the shaded boxes.

4.2.5 Logs Maps

When a query is received at run-time, the system must estimate the cost of refreshing the relevant metadata in order to determine if metadata maintenance is cost effective. Since the cost of refreshing the metadata is dominated by the cost of reading and writing the update logs, the system must estimate the number of log pages that will be read and written. Therefore, given a view node to be refreshed, the system must trace the node's update path and determine how many pages will be read from and written to each log along the update path. Obviously, this must be accomplished without reading the logs or the purpose of estimating the processing cost will have been defeated. This section introduces a data structure, called a *log map*, which maintains the size of each log. The system will use the information contained in the log map to estimate the number of log pages that must be read and written in order to refresh a given view node.

Consider Figure 7 (a) which illustrates an example View Network. Since we are not concerned with the view nodes, only the update logs (represented as rectangles $L_1 - L_{16}$) have been shown. Each log in the View Network may participate in several update paths. For example, log L_1 participates in every update path shown in Figure 7 (a). When there is a direct parent-child relationship between two logs (e.g., between log L_1 and log L_2) we say that the parent log (L_1) *serves* the child log (L_2), and we say that the child log (L_2) *depends* on the parent log (L_1). In general, log L_1 serves all of the logs in Figure 7 (a) and all of the logs in the figure depend on log L_1.

Since update paths are processed at different times, the size of an update log depends on the update path. In other words, the number of unprocessed updates in a log depends on which dependent log is being served. This creates a problem since the system must estimate the size of each log relative to a given update path. For example, consider log L_1 along with its four dependent logs $L_2 - L_5$. Assume that all of the update logs are empty and assume that there are 100 updates in the base relation update log. If log L_2 is updated, 50 updates will be written to log L_1. All of these updates will have been processed for L_2, but none of them will have been processed for L_3, L_4, or L_5.

This example has shown that for each log in the View Network, the system must maintain the size of the log relative to the logs that directly depend on it. A data structure, called a *log map*, accomplishes this by maintaining a *log record* for each log in the View Network. A log record contains an entry for each dependent log. An entry maintains the number of updates that have not been processed for the dependent log.

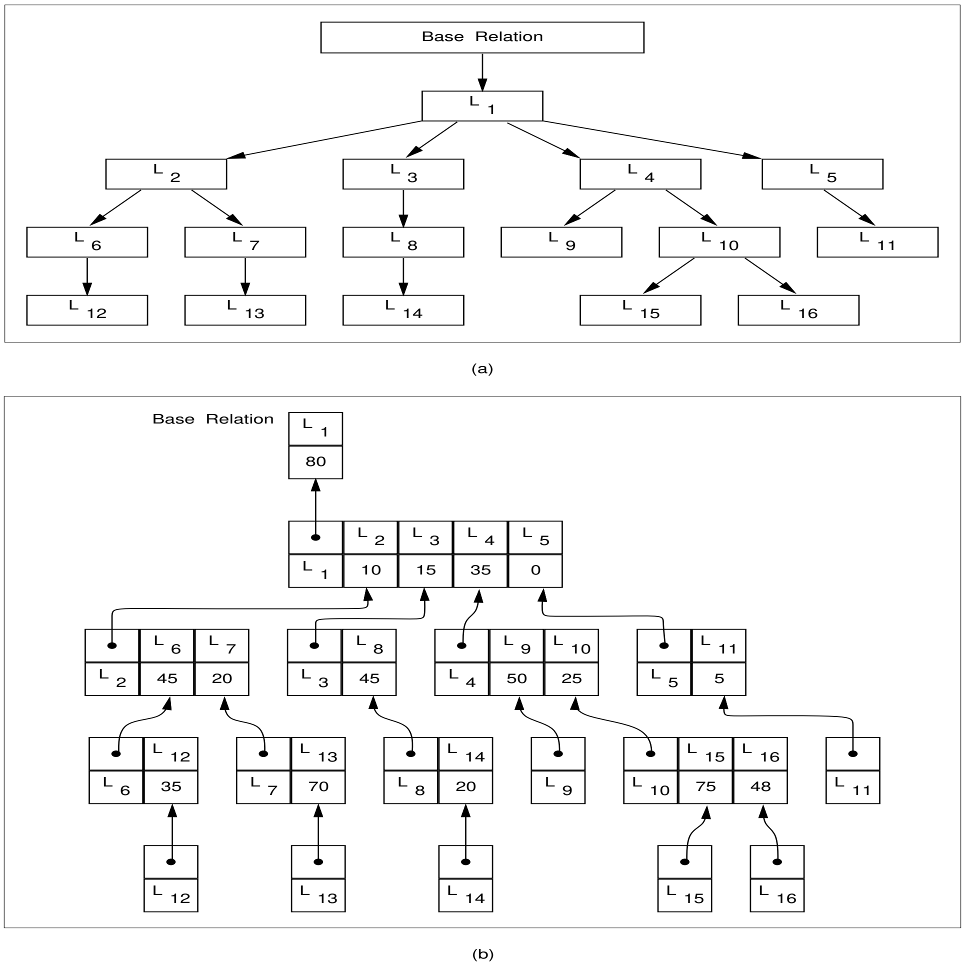

Figure 7: A View Network with its Log Map before and after an update.

Figure 7 (b) illustrates a log map for the View Network shown in Figure 7 (a). The log map contains a log record for each update log. Given an update log L_i, with j dependent logs, a log record for L_i contains an entry, $E_k, 1 \leq k \leq j$, for each dependent log. Each entry, E_k, records the number of updates in log L_i that have not been processed for log L_k. For example, the log record for L_1 contains an entry for logs L_2, L_3, L_4, and L_5. The log map indicates that L_1 contains 10 updates that have not been processed for L_2, 15 updates that have not been processed for L_3, and 25 updates that have not been processed for L_4. All of the updates in L_1 have been processed for log L_5.

Two processes operate on the log map. The first process estimates the number of log pages that will be read and written to refresh a given view node. The second process updates the log map whenever the logs are processed. Algorithms for these processes, and other implementation details, are given in [Pit95].

4.3 Managing View Nodes

The previous section presented several strategies for managing update logs. This section will discuss the problem of managing nodes in the View Network, primarily how to organize and the store the nodes on disk. Since disk access dominates the cost of metadata maintenance, disk I/O should be minimized when the view nodes are retrieved. However, the number of nodes to be refreshed should be maximized in order to reduce the update cost for future queries. Therefore, the objective of node management is to maximize the number of nodes that are refreshed while minimizing disk access.

We will make two assumptions for this discussion. First, a view node can always fit on a single memory page, and several nodes will typically fit on the same page. Second, the number of log pages will dominate the number of node pages. Therefore, it is more expensive to process the logs than to update the nodes. These assumptions are justified below.

The amount of storage required for the rules at a view node depends on the number of rules and the number of predicates in each rule. Predicates are composed of terms. A term is typically of the form <attribute_name, operator, string>. If we assume that the attribute names and the operators are both enumerated sets, then each attribute name and each operator requires one byte. If we assume that each string is ten characters, then each term of a predicate requires 12 bytes (10 bytes for the string plus one byte for the attribute name and one byte for the operator). Typically, a rule will require at least two predicates, one for the condition and one for the implication. Therefore, the smallest rule requires 24 bytes. If we assume a 2K page size, each node can have 40 rules (960 bytes) with 1K remaining for statistics. Since the page size can be as large as 8K, we assume that we can adjust the page size such that a view node can be stored on a single page. As the size of the view nodes decreases for the nodes deeper in the network, several smaller nodes can be stored together on one page.

Our second assumption assumes that the number of log pages will dominate the number of node pages. At the top levels of the network, the number of logs to be read will be small, but the number of updates, and the number of log pages, will be large because more updates apply to the views at higher levels of the network. At lower levels of the network, more logs must be read, but the number of pages in each log will be less. In addition, more nodes will fit on one page because the nodes are smaller.

The objective for managing view nodes is to maximize the number of nodes that are refreshed. Once we have paid the price to process the update logs and bring the updates into main memory, we want to maximize the number of nodes that are updated. Since we assume that the number of node pages is less than the number of log pages, we are willing to spend a few more cycles to retrieve additional nodes.

Realizing that view nodes depend on logs, our strategy for organizing and managing view nodes is based on two premises: (1) whenever a log is retrieved and processed, there is an opportunity to update the nodes that depend on the log, and (2) whenever one node is updated, all of the nodes that depend on that node's log should also be updated.

Node Groups

Consider a simple chain of nodes (V_1, V_6, and V_{10} as shown in Figure 8), that all depend on the same log (the view log at V_1). While it is true that a node depends on all of the logs in the log chain, when a node depends directly on a log, we say that the log *services* the node. In Figure 8, the view log at V_1 services V_1, V_6, and V_{10}. In order to update one of these nodes, the view log at V_1 must be updated. Since it is more expensive to process the logs than refresh the nodes (our second assumption), whenever one of the nodes in this group is refreshed, all of the nodes in the group should be refreshed. Therefore, nodes are grouped locally based on the nodes that are served by each log. All of the nodes that are served by a log will be called a *node group*, and all of the nodes in a node group will be stored together on disk. Six node groups (circled with dashed lines) are shown in Figure 8.

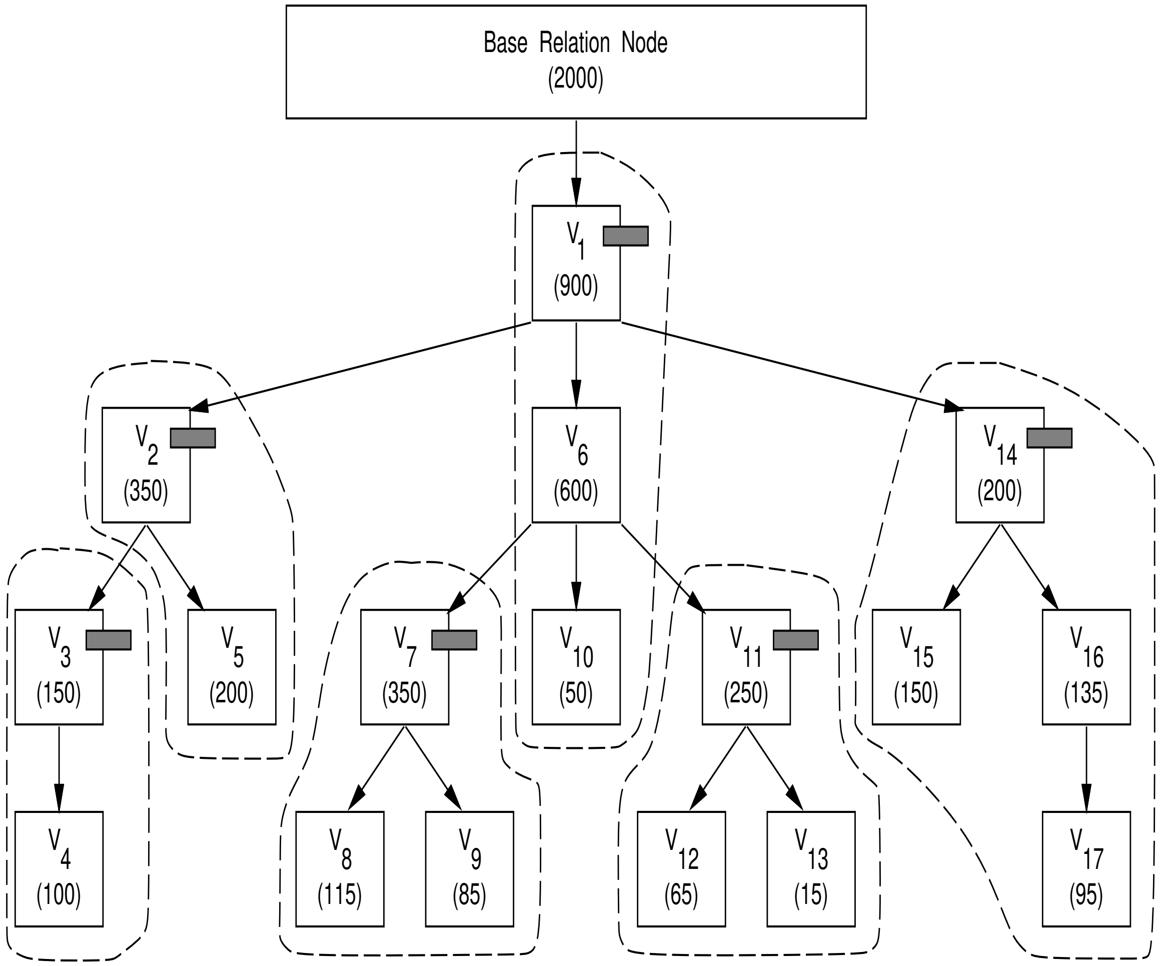

Figure 8: Node groups.

The next issue is how to order the nodes in a node group. Assuming that the nodes in a node group are stored on multiple pages, we want to store the nodes such that whenever a node is retrieved, the first page of the node group is most likely to be retrieved. The reason for this is because the operating system may return several pages from a single read, and may prefetch additional pages. Therefore, if we can retrieve the first page, we may get the remaining pages in the node group from the initial read or from future prefetched page reads. Therefore, the best way to order nodes in a node group is by frequency of access. If the most frequently accessed nodes reside on the first page of the node group, the first page of the node group is more likely to be retrieved.

4.4 Refreshing Dynamic Rules

When an update invalidates an instance-based rule, the update is referred to as an *exception* to the rule. Once the update logs have been processed and the view node to refreshed has been retrieved, exceptions can be detected using well-established integrity checking algorithms [Bla81, HS78]. Once a rule becomes invalid, the rule must be invalidated or rewritten.

As described in [SSS92], there are four methods for handling invalid rules. The simplest approach is to delete the invalid rule. However, deleting the rule may invalidate a semantic plan that is still valid for the current state of the database. Semantic plans require the contents of the database to meet a particular condition (e.g., the GPA of all graduate students is greater than 3.0). Rules represent the current contents of the database. Consequently, if the condition represented by the rule is identical to, or more restrictive than, the condition required by the semantic plan, then the semantic plan is valid. For example, a semantic plan may require all graduate students to have a GPA greater than 3.0. A rule representing the current contents of the database may assert that all graduate students have a GPA greater than 3.2. If a graduate student with a 3.1 grade point average is inserted into the database, the rule is violated, but the condition required by the semantic plan still holds. Therefore, if a rule is more restrictive than the condition required by a semantic plan, then an exception may invalidate the rule without invalidating the semantic plan. Furthermore, an invalid rule may become valid if future updates delete the exception. Consequently, invalid rules should never be deleted.

When an exception occurs, an invalid rule can be rewritten such that the modified rule is valid for the current state of the database. However, a rule that is repeatedly rewritten will eventually become useless. Another alternative involves the identification of autonomously computable updates [BCL89]. A rule can be modified if an update is found to be autonomously computable. The fourth approach is to leave the invalid rule unmodified, but require that the rule reference the exception. This approach requires the optimizer to modify queries using the rule and the exceptions to the rule. Methods for referencing exceptions are described in [Bor85, BCVG88].

4.5 Run-Time Maintenance Cost

As described in Section 3.1, when a query is processed at run-time, the query's execution packet is retrieved from the QEP Network and the query execution plans that apply to the given query are considered. The system will select the most efficient plan for execution. However, if a semantic plan is chosen, the cost of the plan must include the maintenance cost required to refresh the rules that the plan depends on. This section describes how the maintenance cost can be estimated for a given query execution plan.

The formula shown below calculates the cost of updating the metadata at a set of view nodes. This formula represents the number of log pages read (L_R) plus the number of log pages written (L_W) plus the number of node pages read (N_R) plus the number of node pages written (N_W).

$$L_R + L_W + N_R + N_W$$

All of the required information is maintained in the log map (Section 4.2.5. Each semantic plan maintains pointers to the view nodes that contain rules that the semantic plan depends on. These nodes must be refreshed before the semantic plan can be executed. The cost of processing the update logs can be estimated using the log map. In addition, the number of node pages in a node group is determined at compile-time and stored in the log record (in the log map) of the update log that serves the node group. In the worst case, every node page will be modified and written back to disk. Therefore, by doubling the number of node pages that are read (N_R), to account for N_W, the following formula can be used to estimate the maintenance cost.

$$L_R + L_W + (2 * N_R)$$

Although the maintenance cost can be estimated efficiently, the system must still determine whether or not maintenance is cost effective. The dynamic plans in the QEP Network do not depend on any dynamic rules (i.e., the dynamic plans are *non-semantic plans*). Therefore, the cost of the most efficient non-semantic plan can be used as a baseline. When a semantic plan is considered for execution, the system uses the estimated cost of the most efficient non-semantic plan to determine if metadata maintenance is cost effective. If the cost of refreshing the rules that the semantic plan depends on plus the cost of executing the semantic plan is less than the cost of the non-semantic plan, then the metadata should be refreshed. Therefore, metadata maintenance is cost effective when the following condition is met.

$$CP^- > CP^+ + MC$$

If the estimated cost of the non-semantic plan (CP^-) is greater than the estimated cost of the semantic plan (CP^+) plus the maintenance cost (MC) to refresh the rules that the semantic plan depends on, then metadata maintenance is cost effective because the savings produced by the semantic plan will cover the cost of refreshing the metadata.

The semantic plans under consideration can then be ranked in ascending order of cost. If the maintenance cost for the most efficient semantic plan is acceptable, then the metadata is refreshed. If the semantic plan is not valid, then the process is repeated for the next semantic plan until there are no more semantic plans to consider. Even though the semantic plans are ranked by cost, all of the semantic plans must be considered. Although a semantic plan may be expensive, if its maintenance cost is low enough, the total cost of the plan (maintenance plus execution) may be less than the cost of the most efficient non-semantic plan. Therefore, all of the semantic plans must be considered until a plan is found. When multiple semantic plans are considered and some of the metadata has been refreshed, the estimated maintenance cost must include the cost of any maintenance that has already been performed. For example, if the metadata is refreshed for the first semantic plan, but the plan is invalid, then the maintenance cost must be included in the maintenance cost of the remaining semantic plans under consideration. The algorithm for selecting a query execution plan from the QEP Network at run-time is given below.

```
INPUT:   QUERY;   // the query that was received at run-time
OUTPUT:  PLAN;    // the most efficient query execution plan

select_best_query_plan(QUERY)
qep ← query execution packet for QUERY
nsp ← most efficient non-semantic plan from qep.dynamic_plan
sp_list ← the semantic plans in qep.semantic_plans that apply to QUERY
sp_list ← sort sp_list in ascending order of estimated cost
maint_cost ← 0

While (sp_list <> nil)
    sp ← pop(sp_list)
    uc ← estimate_maintenance_cost(sp.view_nodes)
    If (nsp.cost > sp.cost + uc + maint_cost)
        refresh_metadata(sp.view_nodes)
        If (sp is valid)
            Return sp
        Else
            maint_cost ← maint_cost + uc

Return nsp
```

Algorithm 4.7: Select Best Query Plan

5 Conclusion

Previous experimental results have shown that dynamic semantic rules may significantly reduce the cost of query execution. Although several researchers have acknowledged the need to maintain dynamic semantic rules, no one has formally addressed the problem. The Metadata View Graph is a metadatabase that maintains semantic, statistical, and structural metadata for views of a database. Capturing dynamic rules for relevant views increases the amount of information contained in the rules and improves the performance of discovering, retrieving, maintaining, and using the rules. This paper has described how rules are retrieved, maintained, and used efficiently during query processing.

The paper presented several strategies for managing update logs and showed that the 50-Percent Logs strategy produces an optimal allocation of update logs in the View Network. The paper then presented a data structure called a *log map* which monitors the maintenance logs and allows the update cost to be estimated. The paper also presented strategies for grouping and organizing the view nodes on disk to maximize the amount of metadata that is refreshed. Finally, the paper presented a run-time cost model that determines when metadata maintenance is cost effective at query execution time.

References

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

[BCVG88] A. Buchmann, R. Carrerra, and M. Vasquez-Galindo. A generalized constraint and exception handler for an object-oriented cad-dbms. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, pages 38–49, Pacific Grove, California, 1988.

[Bla81] B. Blaustein. *Enforcing Database Assertions: Techniques and Applications*. PhD thesis, Harvard University, 1981.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In C. Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on the Management of Data*, pages 61–71, Washington, D.C., May 1986.

[Bor85] A. Borgida. Language features for flexible handling of exceptions in information systems. *ACM Transactions on Database Systems*, 10(4):565–603, December 1985.

[CG94] R.L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 150–160, Minneapolis, Minnesota, May 1994.

[CGM90] S. Chakravarthy, J. Grant, and J. Minker. Logic based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM-SIGMOD International Conference on the Management of Data*, pages 358–366, Portland, Oregon, 1989.

[HK93] C. Hsu and C.A. Knoblock. Reformulating query plans for multidatabase systems. In *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington, DC, 1993.

[HK94] Chun-Nan Hsu and Craig Knoblock. Rule induction for semantic query optimization. *Machine Learning*, pages 1–10, 1994.

[HS78] M. Hammer and S. Sarin. Efficient monitoring of database assertions. In *Proceedings of the 1978 ACM-SIGMOD International Conference on the Management of Data*, pages 38–49, Austin, Texas, 1978.

[HZ80] M. Hammer and S.B. Zdonik. Knowledge-based query processing. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 137–147, Montreal, October 1980.

[Kin81] J. King. Quist: A system for semantic query optimization in relational databases. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 510–517, 1981.

[Pit95] J. Pittges. *Metadata View Graphs: A Framework for Query Optimization and Metadata Management.* PhD thesis, Georgia Institute of Technology, 1995.

[PMN95] J. Pittges, L. Mark, and S. Navathe. Metadata view graphs: A framework for query optimization and metadata management. *ACM Transactions on Information Systems*, 1995. Submitted November 1994.

[Rou82] N. Roussopoulos. The logical access path scheme of a database. *IEEE Transactions on Software Engineering*, SE-8(6):563–573, November 1982.

[SO89] S.T. Shenoy and Z.M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, 1989.

[SSD92] S. Shekhar, J. Srivastava, and S. Dutta. A formal model of trade-off between optimization and execution costs in semantic query optimization. *Data and Knowledge Engineering*, 8:131–151, 1992.

[SSS92] M. Siegel, E. Sciore, and S. Salveter. A method for automatic rule derivation to support semantic query optimization. *ACM Transactions on Database Systems*, 17(4):563–600, December 1992.

[YS89] C. Yu and W. Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):362–375, September 1989.