Maintaining Instance-Based Constraints for Semantic Query Optimization

Jeff Pittges

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 853-9381
`pittges@cc.gatech.edu`
http://www.cc.gatech.edu/grads/p/Jeff.Pittges/pittges.html

May 3, 1995

Abstract

Semantic Query Optimization has traditionally relied upon scheme-based integrity constraints that are valid for all instances of a database. *Instance-based constraints*, which are only valid for certain states of a database, contain more information than scheme-based constraints because they are specific to the current contents of the database. This makes instance-based constraints more useful to semantic query optimization. However, instance-based constraints are highly sensitive to any changes made to the database and must therefore be updated and validated before they can be applied.

A Metadata View Graph (MVG) is a metadatabase that stores instance-based constraints, along with statistical and structural metadata, for logical views of the database. Constraints at this level are even more useful to semantic query optimization because they are specifically tailored to the intermediate results of a query. This paper reviews existing methods for constraint discovery, describes how constraints are stored in the Metadata View Graph at compile-time, and describes how the MVG Framework retrieves and maintains instance-based constraints at run-time. The paper then analyzes how to apply updates to instance-based constraints in order to refresh them.

Keywords: Metadata View Graph, Instance-Based Constraints, Semantic Query Optimization, Constraint Discovery, Metadata Maintenance.

1 Introduction

Semantic query optimization [HZ80, Kin81, SO89, CGM90, SSD92, HK93] uses transformation rules to reformulate a query into a semantically equivalent query that is more efficient to execute. Traditionally, transformation rules have been derived exclusively from scheme-based integrity constraints that are valid for all instances of a database. Transformation rules based on this type of constraint are desirable because the rules remain valid when changes are made to the database since changes cannot violate the integrity constraints. Unfortunately, scheme-based constraints are typically so general that they are of little use. For example, an integrity constraint may require an employee's salary to be greater than zero.

Recently, a number of researchers [YS89, SSS92, SHKC93, HK94] have proposed methods for discovering *instance-based constraints* (also referred to as *dynamic constraints* in [YS89] and *derived constraints* in [SSS92]) which are only valid for particular instances of the database. Instance-based constraints contain more information than scheme-based constraints because they represent the actual contents of the database. For example, an instance-based constraint may assert that the salary of all employees is greater than or equal to $22,000 and less than or equal to $85,000. Although this constraint is more useful than the integrity constraint given above, instance-based constraints are sensitive to changes made to the database, so they must be maintained whenever the database is updated.

A Metadata View Graph is a metadatabase that maintains instance-based constraints for logical views of the database. Constraints at this level are even more useful to semantic query optimization because they are specifically tailored to the intermediate results of a query. For example, one view might represent graduate students and another view might represent faculty. Both views would maintain an instance-based constraint on salary, but the salary range for graduate students would be much less than the salary range for faculty members. Therefore, when given a query involving graduate students, the semantic query optimizer could use the salary constraint for graduate students to reformulate the query.

In addition, capturing instance-based constraints for views of the database allows the query context to influence the optimization process during semantic query optimization. The *profitability* [SO87] of a rule can be adjusted for each view. Therefore, although the same rule may appear in several views, the rule can be applied differently based on the query context represented by each view. The rules for a particular view can be ordered by associating a context-sensitive salience (priority) with each rule. In this way, the views partition the semantic constraints which allows the constraints to be precisely tailored to particular data sets. Partitioning reduces the number of rules which must be searched during semantic query optimization and improves rule selection by prioritizing rules according to the query context.

The rest of this paper is organized as follows. Section 2 describes the Metadata View Graph Framework. This section provides a structural description of the Metadata View Graph, reviews existing methods for discovering instance-based constraints, and describes how instance-based constraints are stored in the Metadata View Graph. The section also describes how instance-based constraints are retrieved at run-time and used to select the best query execution plan. Section 3 presents the general problem of maintaining instance-based constraints, presents various representations that allow instance-based constraints to be maintained, and analyzes the problem of applying updates to instance-based constraints in order to refresh them. The last section summarizes the contributions of this paper and describes future tasks for this research.

2 The MVG Framework

The Metadata View Graph Framework [PMN95] supports the integration of various approaches to query optimization. As shown in Figure 1, the framework consists of an optimizer and a metadatabase. The MVG Framework has been developed with two objectives in mind: (1) improve query optimization, and (2) provide for a highly extensible query optimizer. Query optimization is improved by maintaining metadata, especially instance-based constraints which improve semantic query optimization, multiple query optimization, incremental query computation, and dynamic plans. Three types of knowledge are required by the query optimizer: (1) procedural knowledge specific to each type of query optimization, (2) control knowledge which integrates the various types of query optimization together, and (3) domain knowledge (metadata about the database). The MVG Framework maintains these three types of knowledge separately in order to facilitate a highly extensible architecture. The optimizer and metadatabase can be extended incrementally as new approaches to query optimization are developed.

This research focuses primarily on the metadatabase (the Metadata View Graph) which was inspired by considering six types of query optimization: syntactic, physical, semantic, dynamic, multiple, and caching and incremental query computation. Therefore, we treat the query optimizer as a black box that is capable of performing these six types of query optimization. Within that black box, the optimizers can be developed independently and loosely coupled, which requires less integration effort but reduces run-time efficiency, or the optimizers can be tightly coupled, which requires greater integration effort (each optimizer may have to be rewritten) but improves run-time efficiency. Although we envision a set of rule-based optimizers [Fre87, GD87, GM93], the actual implementation is irrelevant to our work on Metadata View Graphs.

2.1 Metadata View Graphs

A Metadata View Graph (MVG) is a collection of networks, as shown in Figure 2, for organizing and storing metadata (i.e., a metadatabase). The Metadata View Graph consists of four components: (1) a lexicon, (2) a semantic network, (3) a view network, and (4) a QEP Network of query execution plans.

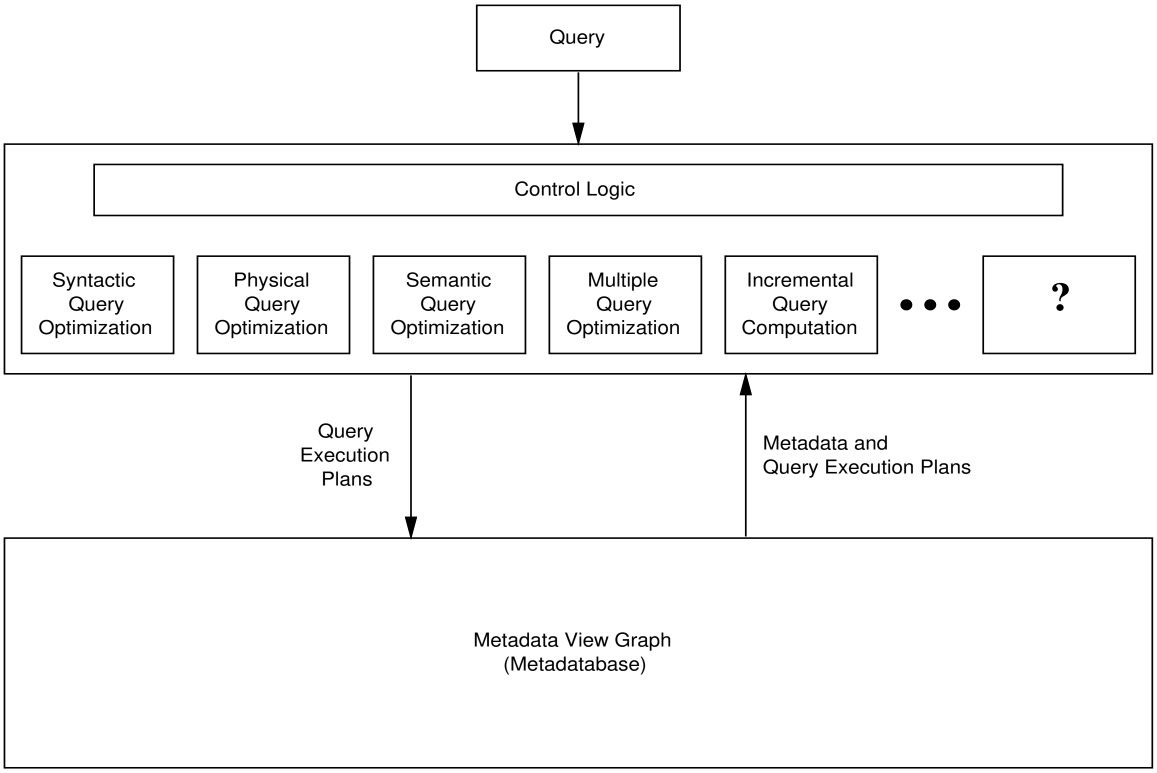

Figure 1: The MVG Framework.

The lexicon contains an entry for each term (word or phrase) recognized by the system (i.e., the system's vocabulary). A lexical entry provides information about the term, including a set of pointers to the semantic nodes that represent the term. In general, a lexicon will store any information about a term that is useful to the system.

Semantic networks represent domain knowledge about the concepts "understood" by the system. Each concept is represented as a node. Two nodes are linked together to represent their relationship to each other. In Figure 2, the semantic network is represented by nodes a_1 through a_8 and S_1 through S_4. Nodes a_1 through a_8 represent the attributes that participate in one or more of the base relations (nodes R_1 through R_3). The attribute nodes are linked directly to their corresponding base relations.

The attribute names and types are specified in the data definition of the database. These nodes form the foundation of the semantic network. The network can be extended by defining nodes and links for application specific concepts and relationships. For example, two nodes representing STUDENT and ADVISOR could be connected by the links ADVISED-BY and ADVISOR-OF. The lexicon and semantic network are not relevant to the research presented in this paper.

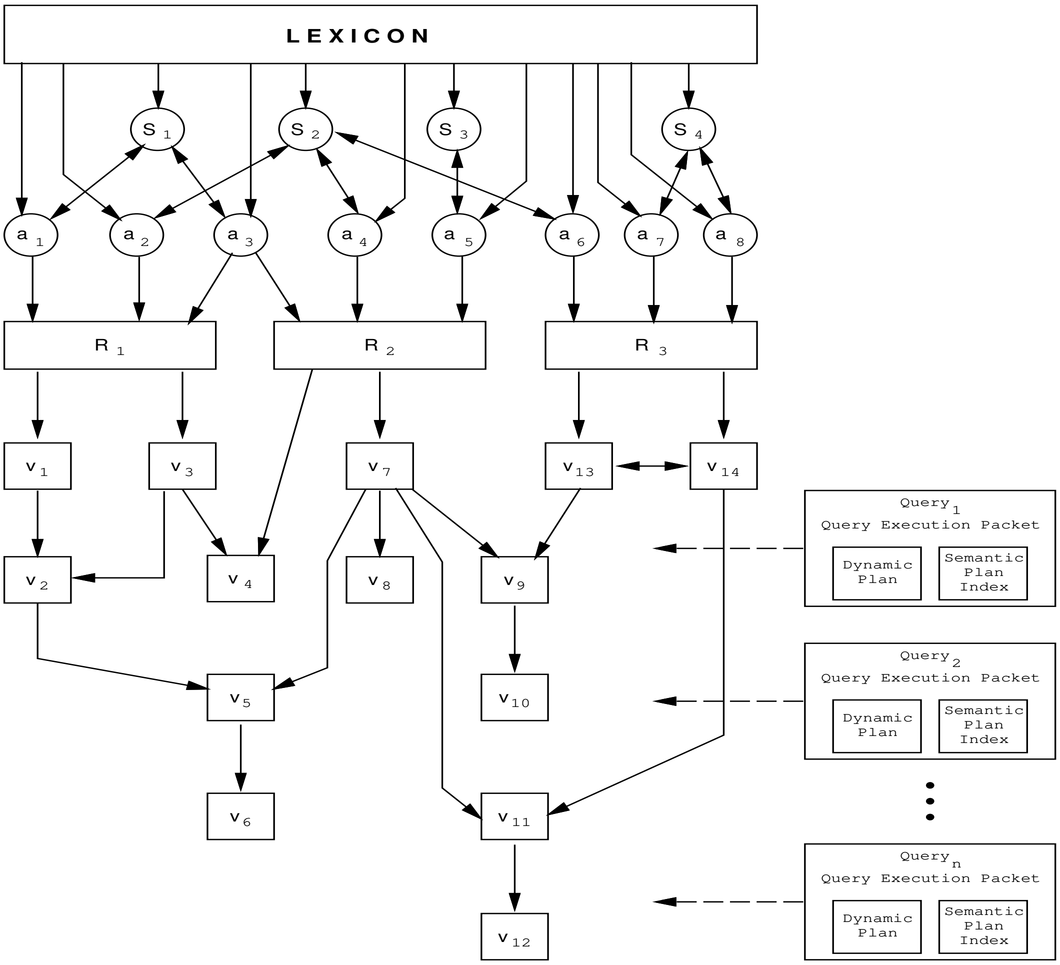

Figure 2: Conceptual Representation of a Metadata View Graph.

The View Network is an extension of Roussopoulos' Logical Access Path schema [Rou82]. The View Network stores semantic, statistical, and structural metadata that is useful to the query optimizer. The view nodes in the network, v_1 through v_{14}, represent *logical views* (intermediate results) and store metadata specific to the particular data set. A view is a projection of attributes which can be defined recursively as follows. All base relations are views. Additional views are the result of applying an operation (e.g., selection, projection, join) to a view or to a pair of views. The views represented by these nodes may or may not be materialized. The links represent logical operations and semantic relationships. The View Network, nodes R_1 through R_3 and v_1 through v_{14}, is essentially a collection of query graphs overlaid on top of each other where R_1 through R_3 represent base relations and v_1 through v_{14} represent the results of performing the operations specified by the links.

The View Network is a unified structure that applies to all of the application queries being served by the Metadata View Graph. The top level of the hierarchy consists of base relation nodes which anchor the Metadata View Graph and serve to connect the semantic network to the View Network. The remaining nodes represent logical views. A view node is defined by the links connecting the node to the base relations. Figure 3 illustrates an example View Network along with a semantic network and lexicon.

The QEP Network stores two types of query execution plans, *dynamic plans* [GW89, CG94] and *semantic plans*. A semantic plan is a query execution plan that is semantically equivalent to the original query. Semantic plans are generated during semantic query optimization and depend on integrity constraints and instance-based semantic constraints.

A dynamic plan links several query execution plans together with choose-plan operators. An example of a dynamic plan is shown in Figure 4. Choose-plan operators allow a decision to be postponed until run-time when the run-time conditions are known. In order to select the best plan from the dynamic plan, the dynamic plan is traversed and the best path is chosen at each choose-plan operator. The statistics stored at the view nodes can guide the decision process. Therefore, the dynamic plan contains pointers to the view nodes with relevant metadata. If the statistics at a view node are out of date, the statistics must be updated before a decision can accurately be made.

The QEP Network maintains a separate *query execution packet* (i.e., a dynamic plan and semantic plan index) for each application query. Figure 4 illustrates how the query execution plans are linked to the logical access paths of the View Network so that the relevant view nodes can be retrieved for each plan. When a query is received at run-time, the query's execution packet is retrieved from the QEP Network and the metadata at the view nodes is used to select the best plan. When a semantic plan is selected as a candidate, the constraints it depends on must be updated and verified against the current state of the database before the plan can be executed. If one of the dependencies has been violated, then the plan is no longer guaranteed to be correct.

2.2 Using Metadata View Graphs

Metadata View Graphs are used at compile-time and run-time. Metadata is collected at compile-time and stored in the View Network. Metadata is retrieved at run-time and used to select the best query execution plan.

2.2.1 Compile-Time

The following high-level algorithm describes how the View Network and the QEP Network are constructed incrementally when a query is compiled.

1. When a query is received at compile-time, the query is optimized with conventional optimization techniques in order to generate a set of logical access paths.

2. The logical access paths are used to construct a separate View Network for the query being compiled. The existing MVG View Network is searched for (partial) matching view nodes.

3. The logical access paths are translated into query execution plans. The cost of each plan is estimated and the plans are filtered to remove any non-competitive plans.

4. Constraints and statistics are collected for the view nodes in the query's View Network. If a node already exists in the MVG View Network, metadata may not have to be collected for that node if the node's metadata is up to date.

   In order to collect metadata, each logical access path will have to be executed. If the query being compiled contains variables, the query history will be used to substitute values for the variables. These are the variable bindings most likely to occur in future queries.

5. The constraints collected in step 4 are used by the optimizer to generate additional query execution plans. Semantic query optimization, multiple query optimization, and incremental query computation can apply the instance-based constraints that were collected.

6. If a new set of query execution plans are produced in step 5, the plans are evaluated using the statistics that were collected. The non-competitive plans are discarded and the query's View Network is modified to include any additional view nodes. Steps 4 and 5 are repeated until no new query execution plans are generated.

7. A query execution packet is created for the query. A dynamic plan is constructed for the non-semantic query execution plans, and an index is created for the semantic plans. The dynamic plan and the semantic plan index are stored in the query's execution packet.

8. The query's View Network is unified with the MVG's View Network (i.e., if a view node in the query's View Network does not already exist in the MVG's View Network, the node is added to the MVG's View Network at the correct location and the MVG View Network is reorganized).

Constraint Discovery

Our research does not address constraint discovery. This section describes existing methods for constraint discovery and discusses the advantages provided by the Metadata View Graph Framework.

Two primary problems hinder constraint discovery: (1) determining *where* to search, and (2) determining *what* to search for. Focusing on views of the database reduces the search space and produces more useful constraints. Searching smaller data sets, as opposed to searching the entire database, improves the performance of the discovery methods. Therefore, the view nodes of the Metadata View Graph determine where to search.

Two basic methods are used to determine what to search for: (1) query-driven methods, and (2) data-driven methods. Query-driven methods use a top-down process to search for constraints that would have been useful for previous queries. Data-driven methods use a bottom-up process to search (random) data sets for constraints. Metadata View Graphs provide a framework for integrating the top-down and bottom-up processes.

Query Driven Methods

[SSS92] presents a query-driven method for discovering constraints. Given a query, the semantic query optimizer identifies the *template* transformation rules that would have been useful to the optimizer, and the system discovers constraints that fit the rule templates. This strategy reduces the search space by only considering data sets that are relevant to queries that have been received. The disadvantage of this strategy is that a query will only benefit from the constraints that have been discovered if the query is similar to a previous query.

Reverse Engineering Method

After a query has been executed, [YS89, HK94] inspect the query result and attempt to discover relationships with other queries. For example, if the (intermediate) results of two queries are identical, then there must be some constraints relating the two queries. This is a type of query-driven approach that requires two similar queries before any constraints are discovered. In addition, this method requires that the results of previous queries be stored and matched against future queries.

Data Driven Methods

[SHKC93] has proposed a data-driven approach that uses grid files to inspect combinations of attribute values for a given data set. The zeros in the grid file indicate constraints. The advantage of this approach is that constraints can be found regardless of the query history. However, since it is impractical to search the entire database, there is no guarantee that the discovered constraints will apply to a query.

MVG Guidelines

The View Network is constructed for the application queries that have been compiled by the system, thus providing queries for the query-driven methods. In addition, the View Network identifies relevant data sets for the data-driven methods. Therefore, the View Network provides a foundation for integrating the top-down and bottom-up constraint discovery processes.

The semantic query optimization transformation types should be used to guide the discovery process. This guideline will focus the data-driven methods on constraints that are useful given the current structure of the database.

For example, one transition type attempts to introduce an index into the query condition. Therefore, the indexed attributes of each base relation should be explored since constraints involving these attributes could lead to rules that introduce indexes. Another transformation attempts to eliminate operations such as a join between two views. In some cases, range constraints for the join attributes of each view can determine that the result of a join is empty in which case the operation can be eliminated.

The structure of the View Network, which consists of chains of nodes organized in a subsumption hierarchy, can also be used to guide the data-driven techniques. The constraints that exist at higher level nodes can be propagated to the nodes below, provided they apply to the nodes below, and then tightened to reflect the contents of the more restricted view. At the top level, the scheme-based integrity constraints that already exist for a database can be restricted to reflect the actual contents of the base relations.

Run-Time

When a query is received at run-time, the query's execution packet (i.e., the query's dynamic plan and its semantic plan index) is retrieved from the QEP Network along with any relevant view nodes from the View Network. The semantic plans are indexed so that the run-time bindings of the query can be used to select the semantic plans that match the conditions of the query. A semantic plan contains a set of pointers to the instance-based constraints it depends on (i.e., the constraints used to generate the plan). These dependencies must be verified for the current state of the database before a semantic plan can be executed. If any of the constraints that a plan depends on are no longer valid, the semantic plan is not guaranteed to be correct.

Each intermediate result in a query execution plan indexes a (possibly empty) set of view nodes with relevant metadata (i.e., metadata that is useful for predicting statistics about the intermediate result). When plans are being compared, the statistics at the view nodes are used to estimate the cost of each plan. However, before the plans can be evaluated, the metadata must be refreshed to reflect any updates made to the database.

The query optimizer selects the best non-semantic plan from the query's dynamic plan. If it is cost effective to update the instance-based constraints, then the constraints are updated and the semantic plans are evaluated. The best query execution plan is selected for execution. The Metadata View Graph adds the query to its query log along with a time-stamp and any other data which may have been collected during execution of the query.

Although this scheme moves most of the optimization effort to the compile phase, it does not preclude run-time optimization. For example, if several queries are received within a reasonable time frame, multiple query optimization can be performed.

2.3 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 3 illustrates part of a view network that supports this query. In this example, there are four departments (Psychology, Math, Computer Science, and Business). The View Network is not required to contain a view node for every department. Only the most frequently accessed views, based on the query history, will be represented in the view network. For example, only three class views ($v_9 - v_{11}$) are represented for the Student base relation.

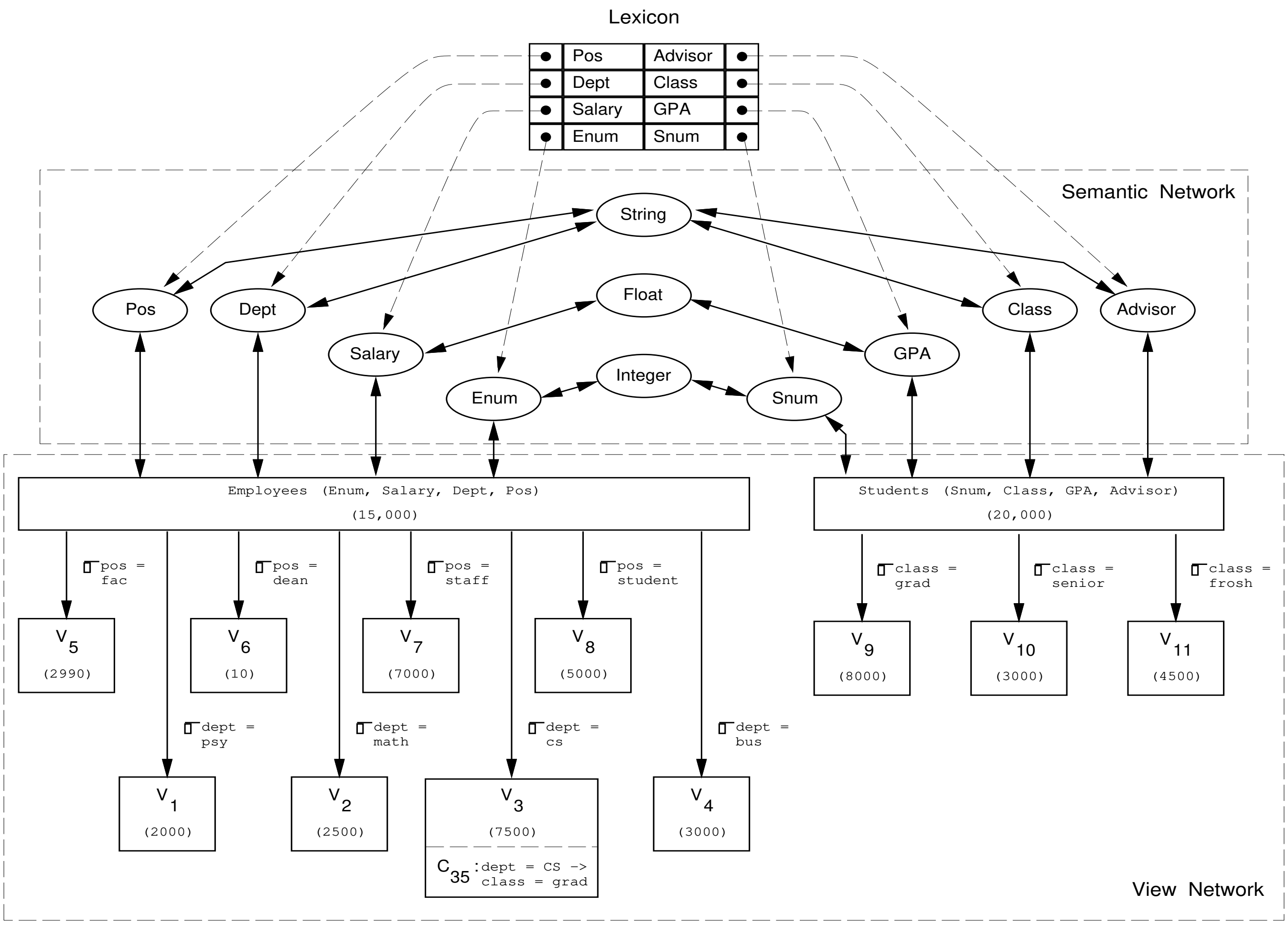

Figure 3: Example Lexicon, Semantic Network, and View Network.

Figure 4 shows the Query Execution Packet for query Q_1. The packet contains the dynamic plan and the semantic plan index. There is one choice to be made in the dynamic plan. The two selections can be performed in either order. The first two filter operations point to the view nodes that contain statistics to determine which path to take. 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.

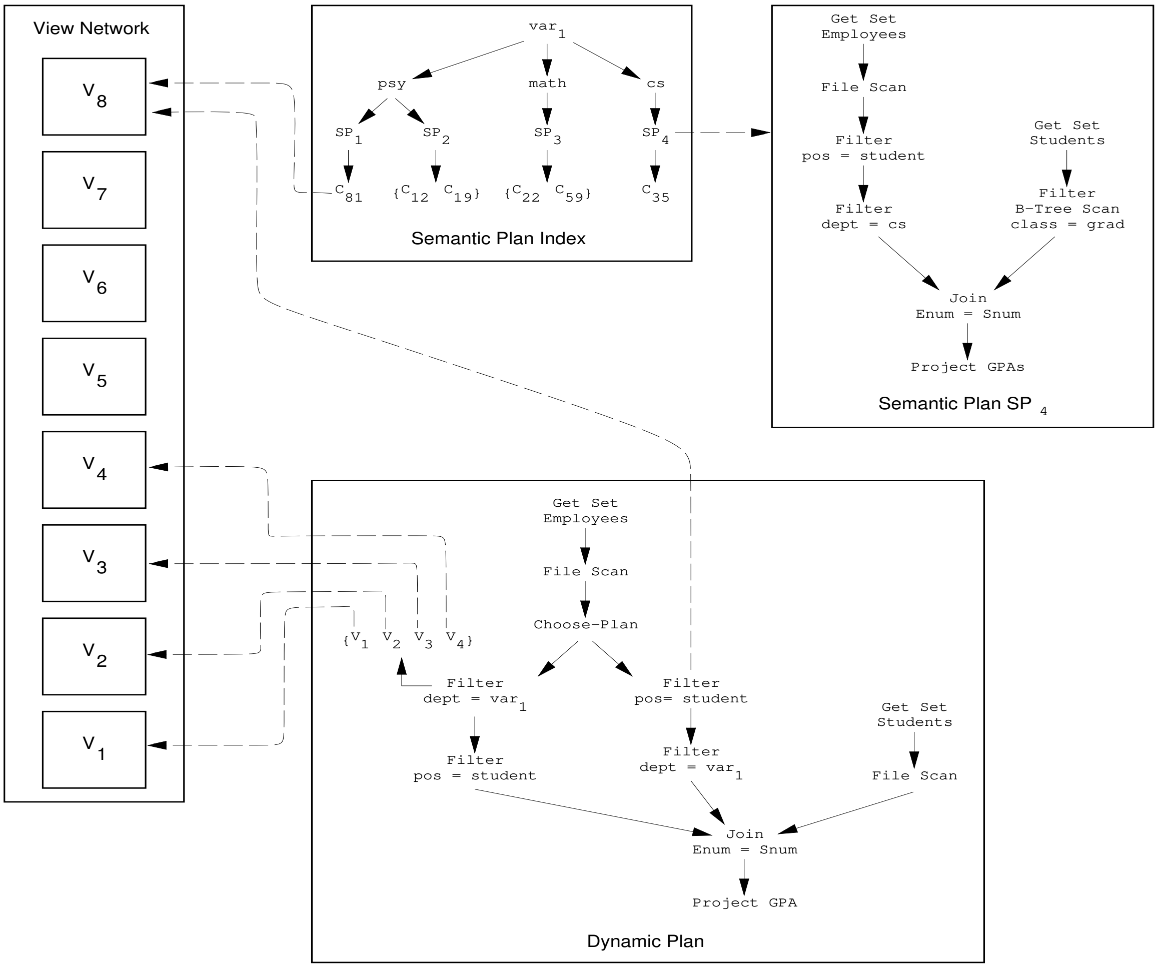

Figure 4: A global persepctive of selecting a query execution plan at run-time.

Semantic plans are indexed by the bindings of the variables. The packet contains four semantic plans, SP_1, SP_2, SP_3, and SP_4, corresponding to the variable bindings var_1 = PSY, var_1 = MATH, and var_1 = CS respectively. There are two semantic plans for the PSY binding. Each plan points to the set of constraints 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 constraint C_{35} which is stored at view node v_3. The semantic plan SP_4 is shown in Figure 4. Constraint C_{35}, shown in Figure 3, 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. Furthermore, because the semantic plan was designed for a specific binding of var_1, the choose-plan operator can be omitted since a drastic change in the database would be required before the number of students would be greater than the number of computer science employees.

Note that constraint C_{35} is not an integrity constraint. Undergraduate computer science students can be employed. Therefore, if the database were updated to include an employed undergraduate from the computer science department, the semantic plan SP_4 would no longer be valid. Consequently, the constraint C_{35} must be updated and verified against the current state of the database before the semantic plan can be selected for execution. If an update invalidates a constraint, then none of the semantic plans that depend on the invalid constraint can be executed because the plans are not guaranteed to be correct.

3 Maintaining Instance-Based Constraints

When a semantic plan is selected from the QEP Network, the constraints that the plan depends on must be refreshed and verified for the current state of the database. Constraints may be out of date if one or more updates have been received by the system since the last time the constraints were refreshed. However, only a subset of the updates will apply to a view node based on the definition of the node (e.g., GPA $\geq$ 3.5). Therefore, the updates must be *filtered* to remove the *irrelevant updates* (i.e., those updates that do not apply to the view node being updated) [BLT86]. An example is provided below.

Consider the Student base relation and the view node shown in Figure 5. The view node shown in this example contains seven constraints, 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 constraints and tuple count at the view node can be verified 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 or deleted. Updates to the database are maintained in a set of logs, one log per base relation. In order to refresh the metadata at this node, the updates in the base relation update log must be filtered to select the updates that satisfy the definition of the node. Each update with a GPA of 3.0 or greater can then be applied to each metadatum at the node.

Before the updates can be filtered and applied to the metadata, the update logs and the view nodes to be updated must be retrieved from disk. The cost of these disk accesses dominates the cost of the update 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 at a view node. The rest of this paper analyzes the problem of refreshing constraints once the updates and the constraints are in main memory.

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

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

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	Sno = 11 -> GPA = 4.0	T4
Tuple Count: 15		T12
distributions		T9
View Cache: nil		

Filtered Log

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

Figure 5: The Student base relation, base relation update log, view node, and filtered log.

3.1 Representing Constraints

Constraints are represented in First Order Predicate Logic. However, constraints must maintain additional information in order to be maintained efficiently. This section begins by considering the affects that insertions and deletions have on instance-based constraints, and then the section considers several representations that improve the maintainability of instance-based constraints.

3.1.1 Update Affects

The Metadata View Graph stores semantic query execution plans that depend on certain conditions. Consider a semantic plan, SP_1, that requires that all graduate students have a GPA greater than 3.0. If all of the graduate students in the database have a GPA of 3.2 or greater, then the instance-based constraint, C_1: CLASS $=$ GRAD $\Rightarrow$ GPA ≥ 3.2, indicates that the semantic plan SP_1 is valid.

Assume there is only one graduate student with a 3.2 GPA and assume that the student is deleted. The deletion does not have to be applied to C_1 because no deletion could violate the required condition, CLASS $=$ GRAD $\Rightarrow$ GPA > 3.0. In general, unless a constraint implies existence, deletions do not have to be applied to valid constraints because deletions cannot invalidate the required conditions with respect to the semantic plans that are stored in the Metadata View Graph.

However, assume a graduate student with a 2.8 GPA is inserted into the database. The insertion must be applied to C_1 because the update violates the required condition for SP_1 thus invalidating the plan. Now assume that the graduate student with GPA 2.8 is deleted and assume that the remaining graduate students all have a GPA greater than 3.0. The required condition for SP_1 is now satisfied. Therefore, this deletion should be applied to C_1 in order to validate the plan.

When insertions are made to the database, it is easy to modify constraints because all of the necessary information is contained in the update. However, when deletions occur, the constraint must represent additional information in order to *recompute* the correct constraint.

3.1.2 Efficient Representations

Consider a student base relation with four attributes: student number (Snum), class (e.g., frosh, grad), GPA, and advisor. Consider a view node defined for GPA ≥ 3.9 and assume that constraint C_1 is stored at the view node. Assume that student 4 is a graduate student advised by Smith with a 3.92 GPA. If student 4 is inserted at time T_2, constraint C_1 can be modified as shown below. However, if student 4 is deleted at time T_3, the constraint cannot be recomputed.

T_1: C_1: Class $=$ Grad $\Rightarrow$ Advisor $=$ Jones

T_2: Insert: Snum $= 4$, Class $=$ Grad, GPA $= 3.92$, Advisor $=$ Smith

T_2: C_1: Class $=$ Grad $\Rightarrow$ (Advisor $=$ Jones) OR (Advisor $=$ Smith)

T_3: Delete: Snum $= 4$, Class $=$ Grad, GPA $= 3.92$, Advisor $=$ Smith

T_3: C_1: Class $=$ Grad $\Rightarrow$ Advisor $=$ Jones

When student 4 is deleted, the constraint should be modified as shown at T_3. However, with this representation, the system must materialize the view (GPA ≥ 3.9) in order to discover that there are no graduate students in the view who are advised by Smith. We will consider two solutions to this problem.

1. Recompute the constraints at run-time during query execution by testing all of the tuples in the intermediate result. This can be done by saving the intermediate result, possibly to disk, and recomputing all of the constraints at a node, or by recomputing a few constraints on the fly while the query is being executed.

2. When the attribute of a term can be enumerated, as in the example above, keep a counter for each term.

The first solution requires additional processing during run-time to recompute the constraints. We can assume that the processing cost is negligible or that the processing is performed when the system is idle. However, the problem with this solution is that the constraints are recomputed during or after query execution. Therefore, the constraints cannot be used for the given query.

Consider semantic plan SP_1 which requires that all graduate students have a GPA greater than 3.0. Assume that SP_1 is valid at time T_1. At time T_2, a graduate student with a 2.8 GPA is inserted which invalidates semantic plan SP_1. At T_3, however, the graduate student with GPA 2.8 is deleted. At this point, semantic plan SP_1 is valid, but the plan cannot be used because constraint C_1 cannot be recomputed until the query is executed.

Therefore, the constraints will have to be recomputed every time a deletion affects the view node. Consequently, a constraint may thrash between being valid and invalid and the semantic plans that depend on the constraint will never be usable even though the required conditions are met.

The second solution is preferable, but this representation does not apply to attributes with non-enumerable values. Consider the following representation for constraint C_1. This representation maintains the number of tuples that apply to each condition. At time T_1, there are 10 graduate students in the view and all 10 are advised by Jones.

T_1: C_1: Class = Grad (10) $\Rightarrow$ Advisor = Jones (10)

T_2: Insert: Snum = 4, Class = Grad, GPA = 3.92, Advisor = Smith

T_2: C_1: Class = Grad (11) $\Rightarrow$ (Advisor = Jones (10)) OR (Advisor = Smith (1))

T_3: Delete: Snum = 5, Class = Grad, GPA = 3.94, Advisor = Jones

T_3: C_1: Class = Grad (10) $\Rightarrow$ (Advisor = Jones (9)) OR (Advisor = Smith (1))

T_4: Delete: Snum = 4, Class = Grad, GPA = 3.92, Advisor = Smith

T_4: C_1: Class = Grad (9) $\Rightarrow$ Advisor = Jones (9)

Student 4 is inserted at T_2 and the constraint is modified. There are now 11 graduate students in the view, 10 are advised by Jones and 1 is advised by Smith. Student 5 is deleted at T_3

and the constraint is modified accordingly. Once again there are 10 graduate students in the view, but now there are 9 students advised by Jones and 1 advised by Smith. Finally, student 4 is deleted at T_4. The constraint is modified to reflect that all 9 graduate students in the view are advised by Jones.

This representation works well for attributes with values that can be enumerated. Some constraints have attribute values that cannot be enumerated. For example, consider the range constraint on GPA shown below.

T_{10}: C_2: GPA >= 3.56

T_{11}: Insert: Snum = 4: GPA = 3.5

T_{11}: C_2: GPA >= 3.5

T_{12}: Delete: Snum = 4: GPA = 3.5

Initially, the lowest GPA for the given view is 3.56. When student 4 is inserted at time T_{11}, the lowest GPA becomes 3.5. However, when student 4 is deleted at T_{12}, there is no way to determine the exact value of the lowest GPA in the view without materializing the view. However, constraint C_2 still represents a lower bound on GPA. The constraint asserts that all of the GPAs are greater than or equal to 3.5, but the constraint does not represent the exact value of the lowest GPA.

The representation shown below maintains a *boundary value list* of the lowest GPAs. This representation can be used to determine the exact lower bound provided the boundary value list is not empty.

T_{10}: C_2: GPA >= (3.56, 3.57, 3.57, 3.58, 3.59)

T_{11}: Insert: Snum = 4: GPA = 3.5

T_{11}: C_2: GPA >= (3.5, 3.56, 3.57, 3.57, 3.58, 3.59)

T_{12}: Delete: Snum = 4: GPA = 3.5

T_{12}: C_2: GPA >= (3.56, 3.57, 3.57, 3.58, 3.59)

Initially the boundary value list contains the 5 lowest GPAs in the view. When student 4 is inserted at T_{11}, student 4's GPA is the lowest GPA in the view. Therefore, student 4's GPA is added to the front of the boundary value list. When student 4 is deleted at T_{12}, one instance of the value 3.5 (student 4's GPA) is removed from the boundary value list.

If the boundary value list becomes empty, because all of the values are deleted and no insertions replace them, then the view must be materialized in order to determine the exact lower bound. The last value in the boundary value list can be retained in order to provide a lower bound for the constraint. In this case, the retained value must be flagged as invalid so that it can be removed when the boundary value list is recomputed. If an insertion is received with a GPA less than or equal to the retained value, then the inserted GPA will replace the invalid GPA and the constraint will once again reflect the exact lowest GPA in the view.

For example, consider constraint C_2 shown above and assume that the students with GPA 3.56, 3.57, 3.57, 3.58, and 3.59 are deleted in that order. C_2 will become C_2: GPA >= (3.59), and the value 3.59 will be flagged as invalid. C_2 now represents an inexact lower bound. Now assume that a student with GPA 3.55 is inserted. The value 3.59 will be replaced by 3.55 and C_2 will represent the lowest GPA in the view, C_2: GPA >= (3.55).

Consider constraints C_3 and C_4 shown below. Each of these constraints involve attributes that require boundary value lists.

C_3: (GPA $\geq$ 3.54) AND (GPA $\leq$ 3.58) $\Rightarrow$ Advisor = Jones (3)

C_4: (SALARY $\geq$ 12K) AND (SALARY $\leq$ 15K) $\Rightarrow$ (GPA $\geq$ 3.2) AND (GPA $\leq$ 3.8)

The counter (3) associated with C_3's condition (Advisor = Jones) indicates that three students satisfy the constraint. Without boundary value lists for GPA, as shown below, the system cannot determine the middle GPA without materializing the view. Constraint C_4 would require four boundary value lists, one list for each term.

C_3: (GPA $\geq$ (3.54, 3.55)) AND (GPA $\leq$ (3.55, 3.58)) $\Rightarrow$ Advisor = Jones (3)

3.2 Refreshing Constraints

This section presents an algorithm for applying an update to an instance-based constraint in order to refresh the constraint. Consider constraint C_5, shown below, which asserts that if a student's GPA is greater than 3.9, then the student is advised by Jones.

C_5: GPA > 3.9 $\Rightarrow$ ADVISOR = JONES

The left hand side of a constraint is a list of terms. Each term represents a condition. The right hand side of a constraint is also a list a terms, but the terms on the right hand side represent assertions. Constraint C_5 has one condition on the left hand side (GPA > 3.9) and one assertion on the right hand side (ADVISOR = JONES).

An update applies to a constraint if it satisfies all of the conditions on the left hand side. As soon as one condition is not satisfied, the update can be disregarded with respect to the constraint being refreshed. If an update satisfies the conditions of the constraint, then the assertions on the right hand side must be considered. In this example, if the update's GPA is greater than 3.9 and the advisor is Jones then the update does not affect the constraint. However, if the update satisfied the condition (GPA ≥ 3.9) and the advisor is not Jones, then there are three options: (1) modify the assertion, (2) modify the condition, or (3) modify both the assertion and the condition by creating another constraint.

For example, assume a student is added to the database with a GPA of 3.92 and the student is advised by Smith. The student's GPA satisfies the condition of the constraint, but the assertion no longer holds. The following two constraints can be created.

$\text{GPA} > 3.92 \Rightarrow \text{ADVISOR} = \text{Jones}$

$\text{GPA} > 3.9 \Rightarrow (\text{ADVISOR} = \text{Jones}) \text{ OR } (\text{ADVISOR} = \text{Smith})$

The following algorithm applies one update to an instance-based constraint.

```
If the update satisfies the condition then
    If the assertion no longer holds then
        Modify the constraint
```

As described above, an update may alter the representation of a constraint without affecting the validity of the constraint. For example, a deletion may add or remove a value from the boundary value list. In this case, the constraint would have to be modified, but the assertion would still hold.

Range Constraints

Specific algorithms can be developed to efficiently process some of the constraint types. For example, if there are range constraints to be updated, instead of processing each update individually for each range constraint, a single pass through the updates can collect the high and low values for each attribute with a range constraint. The high and low values can then be applied to each constraint. Furthermore, the pass that collects the high and low values can be performed during the filtering process. As the new updates are filtered, all of the range constraints at a node can be updated with only a few additional operations.

For example, consider a view node that represents graduate students and assume that the lowest GPA in the view is 3.1. Constraint C_6 represents the lower bound on GPA at the view, C_6: GPA ≥ 3.8. Assume the following six insertions are received by the system: (grad, 3.3), (grad, 3.0), (frosh, 3.6), (soph, 3.4), (senior, 2.7), (grad, 3.8). In order to update the graduate student view, these updates must be filtered to remove the students that are not graduates. When the updates are filtered (i.e., tested for CLASS = GRAD), the system can maintain the lowest GPA for the graduate student updates. When these six updates are filtered, the three irrelevant updates will be removed and the lowest graduate GPA for the insertions will be recored as 3.0. The lowest GPA for the updates (3.0) will be compared with the GPA range constraint at the view node (3.1), and the range constraint will be modified if the insertions have a lower GPA than the current range constraint. In this example, the range constraint will be modified to reflect the 3.0 GPA that has been inserted.

4 Conclusion

Instance-based constraints are more useful to semantic query optimization because they contain more information than scheme-based constraints. This paper presented a framework for maintaining instance-based constraints. The Metadata View Graph Framework makes three contributions: (1) the framework maintains instance-based constraints for logical views of the database, (2) the framework provides a foundation that directs and integrates existing methods for constraint discovery, and (3) the framework allows instance-based constraints to be retrieved efficiently at run-time.

The problem of maintaining instance-based constraints in the Metadata View Graph can be decomposed into three sub-problems: (1) manage the update logs, (2) manage the view nodes, and (3) refresh the instance-based constraints. This paper analyzed the third sub-problem and considered various representations that improve maintenance efficiency.

Future research will develop efficient strategies for managing update logs and view nodes (i.e., the first two subproblems). Future research will also continue to analyze instance-based constraints in order to develop more efficient update strategies, such as the strategy presented for range constraints, and further classify the properties of instance-based constraints with respect to semantic query optimization and maintenance.

5 Acknowledgements

The author wishes to thank Leo Mark and Shamkant Navathe for their many comments and discussions regarding this research. The author has been generously supported by BNR Inc., the research and development subsidiary of Northern Telecom, and is especially grateful to Robert Bloedon and Deborah Stokes. The author also acknowledges the support of the Advanced Project Research Agency under contract number F33615-93-1-1338. The current work is part of the project entitled: "A Knowledge Based Approach to Integrating and Querying Distributed Heterogeneous Information Systems."

References

[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.

[CG94] Richard L. Cole and Goetz 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.

[Fre87] J.C. Freytag. A rule-based view of query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–180, San Francisco, May 1987.

[GD87] G. Graefe and D. DeWitt. The exodus optimizer generator. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 160–171, San Francisco, May 1987.

[GM93] G. Graefe and W.J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE Conference on Data Engineering*, pages 209–228, Vienna, April 1993.

[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.

[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.

[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. Unpublished – submitted, Nov. 1994.

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

[SHKC93] S. Shekhar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization: A data-driven approach. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):950–964, 1993.

[SO87] S.T. Shenoy and Z.M. Ozsoyoglu. A system for semantic query optimization. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 16(3):181–195, December 1987.

[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] Shashi Shekhar, Jaideep Srivastava, and Soumitra 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.