Semantic Query Optimization for Bottom-Up Evaluation *

P. Godfrey[1], J. Gryz[1], J. Minker[1,2]
{*godfrey, jarek, minker*}*@cs.umd.edu*
Department of Computer Science[1] and Institute for Advanced Computer Studies[2]
University of Maryland, College Park, Maryland 20742

Abstract

Semantic query optimization uses semantic knowledge in databases (represented in the form of integrity constraints) to rewrite queries and logic programs for the purpose of more efficient query evaluation. Much work has been done to develop various techniques for optimization. Most of it, however, is only applicable to top-down query evaluation strategies. Moreover, little attention has been paid to the cost of the optimization itself. In this paper, we address the issue of semantic query optimization for bottom-up query evaluation strategies with an emphasis on overall efficiency. We restrict our attention to a single optimization technique, join elimination. We discuss various factors that influence the cost of semantic optimization, and present two abstract algorithms for different optimization approaches. The first one pre-processes a query statically before it is evaluated; the second approach combines query evaluation with semantic optimization using heuristics to achieve the largest possible savings.

Keywords: Intelligent Information Systems, Databases, Semantic Query Optimization.

*This research was supported by by the following grant: NSF IRI-9300691

1 Introduction

Semantic query optimization uses semantic knowledge in the form of integrity constraints to rewrite queries and logic programs for the purpose of more efficient query evaluation. Several researchers have developed methods for semantic optimization in relational and deductive databases [1, 8, 16]. Recently, this work has been extended to programs with recursion [9, 10, 12] and negation [2]. In this paper we accomplish the following. First, our semantic query optimization method applies to bottom-up query evaluation strategies. Such optimization methods are crucial because bottom-up query evaluation is significantly more efficient, in most all cases, than top-down evaluation. To our knowledge only one paper [11] addresses specifically the issue of semantic query optimization for bottom-up evaluation. However, the optimizations they consider are restricted to certain types of integrity constraints, and no general technique for query rewriting is provided. Second, we present a cost analysis of our optimization approach and we show how our method exploits this analysis. Last, our optimization technique allows for the efficient use of integrity constraints which contain both EDB and IDB predicates. In most previous work, integrity constraints are restricted to contain only EDB predicates. (Papers [9, 12, 16] state this condition explicitly.)

The paper is organized as follows. Section 2 introduces notation, and discusses the advantage of bottom-up over top-down query evaluation. Section 3 describes the main focus of our optimization algorithms, namely the removal of joins of tables, which are known beforehand (via deduction over the ICs and rules) not to return any answers. The cost of this semantic optimization is discussed in Section 4. An overview of two such optimization algorithms is provided in Section 5. The paper concludes with a summary and future research directions in Section 6.

2 Preliminaries

2.1 Terminology

We assume familiarity with the terminology of relational and deductive databases [17]. A database, DB, consists of an extensional database (EDB), an intensional database (IDB), and a set of integrity constraints (IC). We assume that DB is function-free, the EDB consists of ground, positive facts and the IDB of rules. We also assume here that IDB rules are nonrecursive. An integrity constraint is a rule with an empty head and whose body contains nonground atoms. EDB predicates are those that appear only in bodies of rules. IDB predicates are the rest.

We also describe the concept of a query tree (an AND/OR tree). A query tree of an IDB predicate p is the "parse tree" of an expression in relational algebra

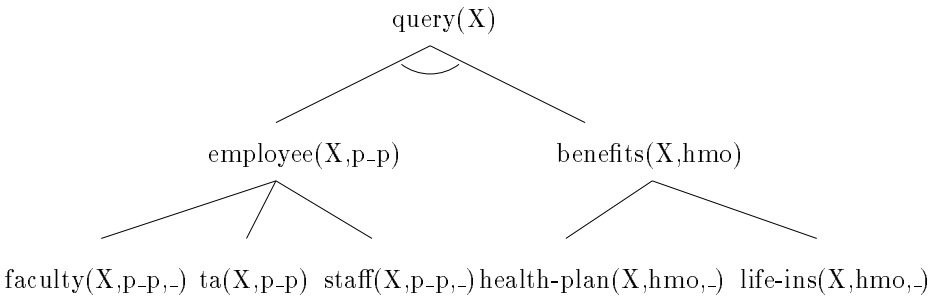

Figure 1: The query tree representation of the query of Example 1

that yields the relation p in terms of the EDB.[1] We are mainly interested in union and join operations, so we do not represent selections and projections explicitly in the tree. However, selections and projections are implicitly apparent. Also, whenever an intermediate node of the query tree represents an IDB predicate q, we likewise label that node as q.

The following example shows a query tree for the query $\leftarrow query(X)$.

Example 1 *Let the database contain five base relations:* faculty*(Name, Department, Rank),* staff*(Name, Department, Years_of_Employment),* ta*(Name, Department),* life_ins*(Name, Provider, Monthly_premium) and* health_plan *(Name, Provider, Monthly_premium).*

Let there be two rules in the database: the first one defines an employee *relation (via the union of the* ta *relation and projections from the* faculty *and* staff *relations); the second defines a* benefits *relation (via the union of projections from the* life_ins *and* health_plan *relations).*

$employee(X,Y) \leftarrow faculty(X,Y,Z).$ $\quad benefits(X,Z) \leftarrow life_ins(X,Z,W).$

$employee(X,Y) \leftarrow staff(X,Y,Z).$ $\quad benefits(X,Z) \leftarrow health_plan(X,Z,W).$

$employee(X,Y) \leftarrow ta(X,Y).$

Let a query ask for the names of all employees of the physical plant, p_p, *whose benefits are provided by* hmo*:*

$$query(X): \leftarrow employee(X,p_p),\ benefits(X,hmo)$$

The query tree representation of this query is given in Figure 1.

It is very easy to translate a tree representation of a query to an equivalent relational algebra representation. We often switch between the two representations. The relational algebra representation[2] of the query (call it Q) of Example 1 is:

$$Q = (\pi_X faculty(X,p_p,Y)) \cup \pi_X staff(X,p_p,Z) \cup ta(X,p_p)) \bowtie (\pi_X life_ins(X,hmo,W) \cup \pi_X health_plan(X,hmo,V))$$

Next we define several concepts which we will use in the paper.

[1] Algorithm 3.2 of [17] describes how such an expression is derived.

[2] We ignore for clarity explicit representation of select operations.

Definition 2.1 *Let Q be a query. U is an unfolding of Q in DB iff*

- $U = Q$;
- $U = q_1(v_1), ..., q_{i-1}(v_{i-1}), P\theta, q_{i+1}(v_{i+1}), ..., q_m(v_m)$, *where* $U' = q_1(v_1), ..., q_{i-1}(v_{i-1}), q_i(v_i), q_{i+1}(v_{i+1}), ..., q_m(v_m)$ *is an unfolding of Q and there is a rule* $< q_i(x_i) \leftarrow P >$ *in IDB for which* $q_i(v_i)\theta = q_i(x_i)\theta$ *and θ is an mgu.*

U is a complete unfolding of Q if it contains only EDB predicates .

Example 2 *A set of complete unfoldings of the query of Example 1 is:*

faculty(X,p_p,_),life_ins(X,hmo,_). *faculty(X,p_p,_),health_plan(X,hmo,_).*
ta(X,p_p),life_ins(X,hmo,_). *ta(X,p_p), health_plan(X,hmo,_).*
staff(X,p_p,_),life_ins(X,hmo,_). *staff(X,p_p,_), health_plan(X,hmo,_).*

Note that atoms in a (complete) unfolding of a query Q represent (leaf) nodes in the respective query tree for Q.

Definition 2.2 *Let U be an unfolding (not necessarily complete) of Q. U is a null unfolding of Q in DB iff $IDB \cup IC \models \neg\exists\ U$*

Example 3 *Let an integrity constraint be: <$\leftarrow$ ta(X,Y), life_ins(X,Z,W)>, which states that teaching assistants are not entitled to receive life insurance. Then, ta(X,p_p),life_ins(X,hmo,_) is a null unfolding of Q of Example 1.*

We assume that the null unfoldings of a query are identified beforehand and are provided as input to the optimization algorithm. The reader is refered to [4] for details on how this is done.

In this paper, we discuss the problem of semantic query optimization in the context of bottom-up query evaluation. We assume that the query evaluation proceeds bottom-up, as described in the semi-naive algorithms presented in [17] (chapter 3) and in [7]. The problem we address in this paper can be stated informally as follows: given a query tree and a set of null unfoldings, rewrite the query tree (that is, reformulate the query) to achieve the largest savings possible. We focus on the elimination of redundant joins. Thus, to achieve the largest savings means maximizing the difference between the savings conferred by the join elimination and the cost of computing the optimized form itself. This cost, as we show later in the paper, can be non-trivial. This is a very different scenario from semantic query optimization for a top-down evaluation strategies. In the top-down case, the cost of optimization is only the cost of identifying null unfoldings . We share that cost also, which can be non-trivial too, but we do not consider that issue here.

2.2 Top-Down versus Bottom-Up Query Processing

Most work in semantic query optimization has been done in the context of the top-down, PROLOG style query processing. This type of query processing, however, is impractical in most cases for database query evaluation. First, a top-down approach requires evaluating independently all the complete unfoldings of the query. The number of complete unfoldings for a given query can be exponential in the size of the rule base. In most contemporary databases, this

is not so problematic since the number of rules (views) is often small. The introduction of deductive databases (such as Aditi [14] and CORAL [13]), and heterogenous database systems, incurs increases to the size of the rule base, to the point where top-down query processing becomes unmanageable.

Another problem incurred with a top-down approach is *redundancy* in query processing. If a predicate P is defined by two rules, say R1 and R2, and each of these rules, given the selections and projections of the query, computes many of the same set of tuples, then the two unfoldings will compute many of the tuples twice. If there are k unfoldings, a given answer (tuple) may be computed k times. Bottom-up evaluation largely escapes this problem.

Example 4 *Consider again the database and the query of Example 1. Assume additionally that the provider* hmo *sells its life insurance and its health insurance as a single package, in other words,* $\pi_X(\sigma_{Y='hmo'}(health_plan(X,Y,Z)) = \pi_T(\sigma_{V='hmo'}\ life_ins(T,V,W))$*. Then, all answers to the query can be found via just three unfoldings:*

faculty(X,p_p,_),life_ins(X,hmo,_).

ta(X,p_p),life_ins(X,hmo,_).

staff(X,p_p,_),life_ins(X,hmo,_).

To evaluate any of the rest of the unfoldings would be redundant.

The difference between top-down and bottom-up query evaluation can be expressed as a difference in query representation. For top-down evaluation, the query is represented as a set of complete unfoldings; for bottom-up evaluation, as a query tree. These two representations differ greatly in their compactness when viewed as formulas in relational algebra written over the EDB predicates. If this compactness is measured (inversely) by the number of elementary join and union operations that need to be executed to evaluate the query, then the top-down query representation is the least compact of all equivalent (in relational algebra) query forms.[3] On the other hand, the query tree tends to minimize the number of these operations providing a compact form of query representation.

Example 5 *Consider the database and the query of Example 1. The query tree requires four operations to evaluate the query (two union operations over the three EDB relations in the left branch, one union over the two EDB relations in the right branch and the final join); whereas the top-down representation of the same query in Example 2 requires eleven operations (six joins and five unions). The query tree representation in this example is most compact.*

To *unfold* a query is to distribute one (or more) of the unions of its relational algebra representation; to *refold* it is to factor out one (or more) of its subexpressions from one (or more) of the unions of its relational algebra representation. Thus, unfolding a query increases the number of its operations, while refolding decreases the number.

[3] We assume non-redundancy in the formula.

3 General Optimization Strategy

The problem we need to address first is the following: given a query tree and a set of null unfoldings of the query,[4] rewrite the tree to guarantee that the joins that these unfoldings represent are not part of the transformed query. Note that by doing this (assuming that the tables to be joined are not empty) we *always* save in terms of query processing time by not having to do the unnecessary join (in other words, the join of tables from the null unfolding) which we know is not going to return any values anyway. In the extreme case, when a query itself is a null unfolding (such a query is called a *simple misconception* in the cooperative answering literature [3]), the query does not need to be evaluated at all since we know the answer set is empty. As stated in Section 2.2, redundant join elimination is straightforward for top-down query processing. Thus, the first approximation to an optimization algorithm can be stated as follows:

1. unfold the query in all possible ways;
2. remove null unfoldings; and
3. refold the query resulting from step 2 back as much as possible.

Clearly, we do not need to unfold the query in all possible ways. Rather, it is enough to unfold the query partially, just to the extent that the null unfoldings are explicitly represented so that they may be removed.

We show informally how this is done on the following example.

Example 6 *Consider again the database and the query of Example 1 and the null unfolding of Example 3. The query Q and the null unfolding N expressed in relational algebra are respectively:*

$$Q = (\pi_X faculty(X,p_p,Y)) \cup \pi_X staff(X,p_p,Z) \cup ta(X,p_p)) \bowtie (\pi_X life_ins(X,hmo,W) \cup \pi_X health_plan(X,hmo,V))$$

$$N = \pi_X(life_ins(X,hmo,W) \bowtie ta(X,p_p))$$

To represent the unfolding explicitly, we may unfold the query (partially) as:

$$Q' = (\pi_X faculty(X,p_p,Y) \cup \pi_X staff(X,p_p,Z)) \bowtie \pi_X life\text{-}ins(X,hmo,W) \cup ta(X,p_p) \bowtie \pi_X life\text{-}ins(X,hmo,W) \cup (\pi_X faculty(X,p_p,Y) \cup ta(X,p_p) \cup \pi_X staff(X,p_p,Z)) \bowtie \pi_X health\text{-}plan(X,hmo,V)$$

After the null unfolding is removed we obtain:

$Q_1 = Q_1^1 \cup Q_1^2$, *where:*

$Q_1^1 = (\pi_X faculty(X,p_p,Y) \cup \pi_X staff(X,p_p,Z)) \bowtie \pi_X life\text{-}ins(X,hmo,W)$

$Q_1^2 = (\pi_X faculty(X,p_p,Y) \cup ta(X,p_p) \cup \pi_X staff(X,p_p,Z)) \bowtie \pi_X health\text{-}plan(X,hmo,V)$

Q_1^1 *and* Q_1^2 *have straightforward representations as query trees as shown in Figure 2.*

Since unfolding a query *always* increases the number of operations in its respective relational algebra representation, it may be viewed as bringing the query form closer to its top-down representation. In the extreme case, when

[4] An algorithm for computing this set for a given database and a query is presented in [4].

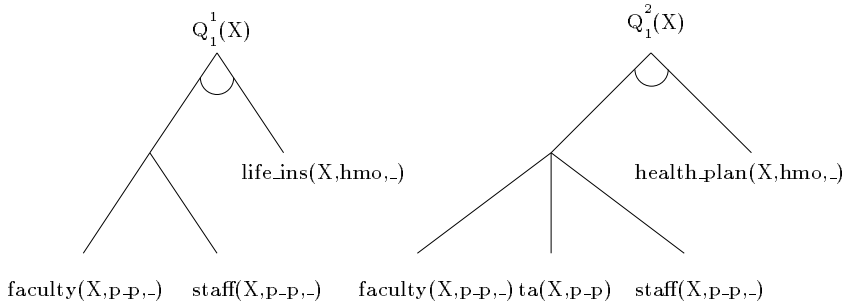

Figure 2: The query tree representation of Q_1^1 and Q_1^2

there are many null unfoldings the query may have to be unfolded almost completely and cannot be refolded to any extent, this worst case converges on the form of the query's top-down representation.

The last issue that deserves to be addressed is the compactness of the optimized query. As stated in Section 2.2, the difference between the top-down and bottom-up query representations can be expressed syntactically as a difference in their number of operations (unions and joins). The top-down approach maximizes that number, while the bottom-up approach tends to minimize it. We assumed above that the query is unfolded *only* to the degree necessary so that the null unfoldings are represented explicitly, and then after removing these null unfoldings *refolded* back from that form. This does not guarantee, however, that the final form of the query is in the most compact form (that is, has the least number of operations). The following example shows this.

Example 7 *Let the query be:*

$Q = G \bowtie (A \bowtie (B \cup C \bowtie D) \cup E \bowtie (F \cup C \bowtie D))$

and the null unfolding be:

$U = E \bowtie F$

By simply unfolding the query, removing the null unfolding and refolding the query back we get:

$Q' = G \bowtie (A \bowtie (B \cup C \bowtie D) \cup E \bowtie C \bowtie D)$

However, the most compact query form is:

$Q'' = G \bowtie (C \bowtie D \bowtie (A \cup E) \cup A \bowtie B)$.

There are two reasons for not trying to minimize (absolutely) the number of operations in the query. First, it can be shown easily that the minimization problem is NP-complete. Second, we adopt as a working hypothesis the assumption that the input to the optimization algorithm, the query tree, is close to any minimum representation of the query. We conjecture then that the optimized query is also close to any *ideal* optimized query. We argue, moreover, that even if a polynomial time algorithm were available for optimality, there are still good reasons for not choosing the number of operations as our sole criterion for designing an algorithm. In the next section, it is shown that other costs of

semantic optimization can easily overshadow the value of an optimal (in the above sense) algorithm.

4 Optimization trade-offs

As stated in the previous section, removing null unfoldings from a query produces a less compact query, bringing it closer to its top-down representation. As a consequence, some of the undesirable features of the top-down approach to query evaluation become prominent in the evaluation of the optimized query.

1. *Query fragmentation.*
   Query fragmentation is an inherent feature of semantic optimization for bottom-up query evaluation. There is only one type of null unfolding removal of which does not increase the number of operations in the optimized query. We call such null unfoldings which have this property *nice*.[5]
   Definition 4.1 *Let $Q = A_1 \bowtie ... \bowtie A_n$ be a query expressed in relational algebra and U be an unfolding of Q. U is nice iff $U = A_{i_1} \bowtie ... \bowtie A_{i_{n-1}} \bowtie B$, $i_j \in \{1, ..., n-1\}$, $i_j \neq i_k$ if $j \neq k$ where B is a subexpression of A_{i_n}.*
   On the other hand, there are cases where the optimized query will have almost twice the number of operation of the original one.
   The degree of query fragmentation depends also on the algorithm used. If the optimization algorithm is iterative, that is, it removes null unfoldings sequentially, independently of one another, query fragmentation can be worse than in the case of a global approach. Consider the following extension of Example 6.
   Assume that *health_plan* is not an extensional predicate, but is further defined as:
   health_plan(X, W, V)← subsidized_health_plan(X, W, V).
   health_plan(X, W, V)← unsubsidized_health_plan(X, W, V).
   Assume also that there is another null unfolding to consider:
   ta(X,p_p),subsidized_health_plan(X,hmo,_).
   Removing this unfolding from a query Q_1 of Example 6, (where *health_plan (X, hmo, _)* has been rewritten via the above rules fragments the query again.
   Notice, however, that if the original query were split in a different way, for instance, into:
   $Q_2(X) = Q_2^1(X) \cup Q_2^2(X)$ in which:
   $Q_2^1(X)$=*(faculty(X,p_p,_)*$\cup$ *staff(X,p_p,_))* $\bowtie$ *(subsidized_health_plan(X ,hmo,_)*
   $\cup$ *unsubsidized_health_plan(X,hmo,_)* $\cup$ *life_ins(X,hmo,_))*
   $Q_2^2(X)$=*ta(X,p_p)* $\bowtie$ *(subsidized_health_plan(X ,hmo,_)*
   $\cup$ *unsubsidized_health_plan(X,hmo,_))*
   then removing the second null unfolding does not increase the number of nodes in the tree (only the second subquery, $Q_2^2(X)$ needs to be rewritten).

[5] Such integrity constraints (with two atoms) were called *semi-complete join pairs* in [11].

One of the key differences between the two algorithms presented in the next section is the way they handle the removal of multiple null unfoldings.

2. *Recomputation of joins due to table overlaps.*
   Materialized tables in bottom-up query evaluation described here do not contain duplicates (we assume that duplicates are removed whenever tables are unioned). Consider again the database and the query of Example 6 and the condition of Example 4 holds, that is
   health_plan(X,hmo,Y)← life_ins(X,hmo,Z).
   Then, all answers to the query Q are computed by Q_1^2 and so Q_1^1 is redundant. This redundant computation would not have taken place if the query had not been optimized.
3. *Recomputation of joins due to independent computation of subqueries.*
   Consider yet another extension of Example 6: assume that *health_plan* is not an extensional predicate, but it is further defined as:
   health_plan(X,Y,Z)← personel(SSN,X),insurance(SSN,Y,Z).
   Also assume that the query has been split as in Q_3.
   $Q_3(X) = Q_3^1(X) \cup Q_3^2(X)$ in which:
   $Q_3^1(X)$=*(faculty(X,p_p,_)∪ staff(X,p_p,_)) ⋈ (personel(SSN,X)⋈ insurance(SSN,hmo,_) ∪ life_ins(X,hmo,_))*
   $Q_3^2(X)$=*ta(X,p_p) ⋈ personel(SSN,X) ⋈ insurance(SSN,hmo,_)*
   Note that both Q_3^1 and Q_3^2 involve computing the join of *personel(SSN,X) ⋈ insurance(SSN,hmo,_)*. However, since the two queries are evaluated independently, the optimal plan for query $Q_2^2(X)$=*ta(X,p_p) ⋈ personel (SSN,X) ⋈ insurance(SSN,hmo,_)* may require computing the join *ta(X,p_p) ⋈ personel(SSN,X)* first and then joining the result with *insurance (SSN, hmo, _)*. Clearly, considering the fact that the join *personel (SSN,X) ⋈ insurance (SSN, hmo, _)* has to be computed as part of Q_3^1, it may be better to compute Q_3^2 in a different order.
4. *Elusive savings.*
   The savings achieved by join elimination are proportional to the sizes of tables whose join can be eliminated from the query because of the presence of an appropriate null unfolding. These savings may be too small, however, to justify applying semantic optimization, given the overhead costs.
   Assume that *ta(X,p_p)* in Example 6 is empty; that is, there is no teaching assistant employed in the physical plant department. Thus, removing the null unfolding *ta(X,p_p)⋈ life_ins(X,hmo,_)* saves nothing. In the ideal situation, we should be able to discover such cases to avoid rewriting the query.

The problems discussed in points 2 and 3 can be solved easily. Recomputation of joins due to table overlap can be solved in the following fashion. Let $U_1, ..., U_n$ be a null unfolding, $U_i^1, ..., U_i^k$ be the siblings of U_i, $1 \leq i \leq n$, in the query tree. We state without formal proof that substituting U_i^j, $1 \leq j \leq k$ with $U_i^j - U_i$ does not change the answer set of the evaluated query while prevent-

ing at the same time recomputation of joins due to table overlaps.[6] Note also that this computation does not add an extra cost to query evaluation since it is equivalent to duplicate removal from materialized tables.

The problem of recomputation of joins due to independent evaluation of subqueries is addressed by *multiple query optimization* (see [6, 15] on this topic).

If the savings analysis is to be a part of an optimization algorithm, then the sizes of the tables in the query tree must be part of the input to the algorithm. But the only way[7] the sizes of intermediate nodes of the query tree can be learned is by materializing these nodes. Materializing *all* the nodes of the tree *before* doing any semantic optimization of course defeats the purpose of optimization (since this is equivalent to evaluating the query). Hence, materialization should be done in stages, interleaving the steps of evaluation and optimization. A consequence, however, is that this type of optimization algorithm must be iterative since we cannot know in advance which null unfoldings should be removed. Thus, there is a clear trade-off between a global optimization (which produces a potentially more compact query as point 1 of this section showed) and exploiting the savings analysis. The two algorithms in the next section emphasize these different choices.

5 The Algorithms

5.1 A Global Algorithm

First we consider a global approach in which all the null unfoldings are removed from the query in parallel. The algorithm will ensure that the rewritten query is minimal.

We wish to avoid "evaluating" any of the known null unfoldings of a query whenever we evaluate the query. Our approach is to rewrite the query as a set of unfoldings that are *independent* of the null unfoldings, but which, when evaluated, result in the same answer set as the query's.

This unfolding set, call it $\mathcal{S}$, should have the following properties. Let $\mathcal{N}$ be the set of null unfoldings.

- No unfolding in $\mathcal{S}$ should *overlap* with any unfolding in $\mathcal{N}$; that is, for $U \in \mathcal{S}$ and $V \in \mathcal{N}$, U and V have no unfolding in common. (Call U and V *independent* of one another in this case.)
- $\mathcal{N} \cup \mathcal{S}$ should be a *cover* of the query; that is, any complete unfolding of the query is an unfolding of some unfolding in $\mathcal{N}$ or $\mathcal{S}$.
- Set $\mathcal{S}$ should be *most general*:
  - no unfolding in $\mathcal{S}$ can be refolded at all, and still preserve the above properties; and

[6] Note that the difference operation makes sense because nodes in the tree represent tables. We also assume that all atoms of null unfoldings are OR nodes in the query tree.

[7] Estimating the sizes of intermediate nodes based on the sizes of the tree leaves may be very unreliable

– for any $U \in \mathcal{S}$, $(\mathcal{N} \cup \mathcal{S}) - \{U\}$ is not a cover of the query.

Techniques from the *Carmin* System can be used to determine the set of null unfoldings, $\mathcal{N}$, for a query [5, 4]. *Carmin* is a cooperative database system, which provides cooperative responses to queries, in addition to the answer sets. If *Carmin* can find a set of null unfoldings $\mathcal{N}$ for a query which is a *cover* of the query, and each such null unfolding is *provably* null, the query is said to be a *complex misconception*. *Carmin* detects misconceptions (both simple and complex), and *explains* them (based on the proofs) to the user.

Detecting that a query is a misconception is the ultimate in semantic query optimization; the query does not need to be evaluated because its answer set is known to be empty. A system like *Carmin* will determine $\mathcal{N}$ in attempting to deduce misconceptions. We are interested in intermediate cases where $\mathcal{N}$ is not a cover, and so we still need to evaluate (some optimized form of) the query.[8]

Given a query $\mathcal{Q}$ and its set of null unfoldings $\mathcal{N}$, the routine **find_cover** finds the unfolding set $\mathcal{S}$ as defined above.

find_cover ($\mathcal{Q}$, $\mathcal{N}$)
    $\mathcal{S} := \{\}$
    while **new_unfolding** ($\mathcal{Q}$, $\mathcal{N} \cup \mathcal{S}$, U)
        $V :=$ **refolding** (U, $\mathcal{N}$)
        $\mathcal{S} := \mathcal{S} \cup \{V\}$
    return $\mathcal{S}$

The routine **new_unfolding** is implemented in *Carmin*. It returns *false* if $\mathcal{N} \cup \mathcal{S}$ is a cover of $\mathcal{Q}$. Otherwise, it evaluates *true*, and returns a new complete unfolding as U which is not a sub-unfolding of any unfolding in $\mathcal{N} \cup \mathcal{S}$. (So U is a witness that $\mathcal{N} \cup \mathcal{S}$ is not a cover.) This routine is the computational bottleneck in this approach. It is NP-hard over the size of $\mathcal{N} \cup \mathcal{S}$. However, for reasonably small input sizes, it runs with good average case performance.

The refolding step is simple. It refolds the unfolding U as much as possible without *overlapping* with any of the null unfoldings in $\mathcal{N}$.

Evaluating each unfolded query in $\mathcal{S}$ and unioning the results is equivalent to evaluating the original query. It is also obvious this does not evaluate any of the join expressions of the null unfoldings. This approach's key advantage is that $\mathcal{S}$ can be guaranteed to be minimal. The disadvantage is that if $\mathcal{N}$ is large, the algorithm will tend towards intractable. This can also happen if the minimal $\mathcal{S}$ deduced is inherently large (although there are reasons to believe that this does not happen, in average case, unless $\mathcal{N}$ is large.)

A second disadvantage, perhaps, is that *every* null unfolding is "removed", regardless of whether that truly yields a worthwhile optimization. This method does not allow for potential savings to be estimated, to decide which null unfoldings to attend and which to ignore, in conjunction with the query rewrite and

[8] We do not discuss the costs of deducing $\mathcal{N}$ in this paper.

evaluation. Next, we consider an alternate, dynamic approach which integrates estimations, rewriting, and evaluation.

5.2 A Dynamic Iterative Algorithm

We have assumed so far that the null unfoldings of a query are given as input and are to be removed. This is an acceptable strategy if the number of null unfoldings is small (hence the degree of query fragmentation will be small). Contemporary databases, however, often contain too many integrity constraints, resulting in too many null unfoldings for a given query, to make the use of all of them for optimization feasible.[9] We could focus instead on a small subset of the null unfoldings for the given query. We would like this subset of unfoldings to be *optimal*, in the sense that it contains only unfoldings that yield large savings in the query's evaluation (and leaves out ones resulting in only insignificant savings). As we demonstrated earlier, the savings can be measured via the sizes of the tables involved in a given null unfolding. Then only those unfoldings that "save" the most in that sense are selected.

We propose a dynamic algorithm to rewrite the query *during* the evaluation, removing null unfoldings iteratively throughout the evaluation. The idea of this approach is to proceed with the evaluation in stages. Each stage proceeds until enough tables have been materialized so that there is enough information to estimate the savings that would be achieved by removing a given null unfolding. If the estimated savings are greater than some fixed threshold, the query is rewritten and the evaluation proceeds until the next checkpoint. Otherwise, that unfolding is not removed, and the query evaluation proceeds as normal.

The downside of this type of dynamic optimization is a non-optimal query fragmentation. Since we do not know in advance which null unfoldings are to be removed, we cannot predict what the best query rewriting at any given checkpoint will be. Point 1 *Query fragmentation* of Section 4 illustrates this problem.

We assume for this algorithm that the set of null unfoldings $\mathcal{N}$ for a given query Q with a query tree T is *ordered*.[10] We also assume that we are given some threshold value, say h (for example based on experiments) which indicates how big the tables to be joined ought to be to justify the optimization step. **remove_unfolding**(T,N) is a simple algorithm described informally in Example 6 that given a query tree and a null unfolding unfolds the query, removes that unfolding, and refolds back the query to its compact form.

[9] Besides the set of integrity constraints given by the database administrator one can also use for optimization the so-called propagated integrity constraints [10] as well as dynamic integrity constraints [18].

[10] Informally, N_1 is before N_2 in a sequence of ordered unfoldings if all atoms of N_1 are below atoms of N_2 in the query tree. A formal definition of the ordering and a discussion of a possibility of such ordering is beyond the scope of this paper.

We assume that all *nice* unfoldings have been removed prior to running this algorithm.

evaluate_optimize (T, $\mathcal{N}$)
    If $\mathcal{N} = \emptyset$
        Materialize all nodes of T, return $Root(T)$ [evaluated query]
    Else
        Let $\mathcal{N} = N_1, ..., N_n$
        Let $N_1 = N_1^1, ..., N_1^n$
        Materialize all subtrees rooted at $N_1^1, ..., N_1^n$
        If $Size(N_1^1) * ... * Size(N_1^n) > h$
            T:=**remove_unfolding**(T, N_1)
        $\mathcal{N} := \mathcal{N} - N_1$
        evaluate_optimize(T, $\mathcal{N}$)

6 Summary and Future Work

We discussed in this paper semantic query optimization for bottom-up query evaluation strategies. We presented the goal of such an optimization, analyzed its cost, and proposed two algorithms. Our optimization techniques are general enough so that they can be implemented with any particular bottom-up query evaluation method.

We believe that the bottom-up evaluation strategy will be a method of choice in future deductive and heterogenous database systems. Such databases tend to have a large number of integrity constraints which could be used to constrain the search space for answers by means of semantic optimization.

We plan to extend this work in two directions. First, algorithms for other types of semantic optimization (such as restriction elimination or restriction introduction) should be developed and tested. Second, the algorithms for join elimination presented here should be generalized to work for databases with resursion and negation (in queries, rules and ICs).

We intend to implement the algorithms on the testbed provided by the *Carmin* system [4, 5]. We plan to experiment with large databases to determine the conditions under which these algorithms perform well, and when to use one method over another.

References

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

[2] T. Gaasterland and J. Lobo. Processing negation and disjunction in logic programs through integrity constraints. *Journal of Intelligent Information Systems*, 2(3), 1993.

[3] Terry Gaasterland, Parke Godfrey, and Jack Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992. Invited paper.

[4] Parke Godfrey. *An Architecture and Implementation for a Cooperative Database System*. PhD thesis, University of Maryland, Dept. of Computer Science, University of Maryland, College Park, MD 20742, USA, December 1995.

[5] Parke Godfrey, Jack Minker, and Lev Novik. An architecture for a cooperative database system. In Witold Litwin and Tore Risch, editors, *Proceedings of the First International Conference on Applications of Databases*, Lecture Notes in Computer Science 819, pages 3–24. Springer Verlag, Vadstena, Sweden, June 1994.

[6] J. Grant and J. Minker. On optimizing the evaluation of a set of expressions. *International Journal of Computer and Information Sciences*, 11:179–191, 1982.

[7] L.J. Henschen and S.A. Naqvi. On compiling queries in recursive first-order databases. *J.ACM*, 31(1):47–85, January 1984.

[8] J.J. King. Quist: A system for semantic query optimization in relational databases. *Proc. 7th International Conference on Very Large Data Bases*, pages 510–517, September 1981.

[9] Laks V. S. Lakshmanan and R. Missaoui. On semantic query optimization in deductive databases. In *Proc. IEEE International Conference on Data Engineering*, pages 368–375, 1992.

[10] S. Lee and J. Han. Semantic query optimization in resursive databases. In *Proc. IEEE International Conference on Data Engineering*, pages 444–451, 1988.

[11] S. Lee, L.J.Henschen, and G.Z. Qadah. Semantic query reformulation in deductive databases. In *Proc. IEEE International Conference on Data Engineering*, pages 232–239, Los Amitos, CA, 1991. IEEE Computer Society Press.

[12] A.Y. Levy and Y. Sagiv. Semantic query optimization in datalog programs. In *Proc. PODS*, 1995.

[13] Raghu Ramakrishnan, Per Bothner, Devesh Srivastava, and S. Sudarshan. CORAL – A database programming language. Technical Report TR-CS-90-14, Dept. of Computing and Information Sciences, Kansas State University, 1990. Also in NACLP90 Workshop on Deductive Databases, ed Jan Chomicki.

[14] K. Ramamohanarao. An implementation overview of the aditi deductive database system. In *Proc. 3rd International Conference DOOD*, Arizona, 1993. (Invited Talk).

[15] T. Sellis. Global query optimization. *Proc. 1986 ACM-SIGMOD International Conference on Management of Data*, May 1986.

[16] 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, September 1989.

[17] J. D. Ullman. *Principles of Database and Knowledge-Base Systems II.* Principles of Computer Science Series. Computer Science Press, Rockville, Maryland 20850, 1988.

[18] Clement T. Yu and Wei Su. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):362–375, September 1989.