Intensional Query Optimization

P. Godfrey[1,2]
godfrey@arl.mil

J. Gryz[1]
jarek@cs.umd.edu

[1] Department of Computer Science at the
University of Maryland at College Park
College Park, Maryland, USA

and

[2] U.S. Army Research Laboratory
Adelphi, Maryland, USA

Abstract

We have introduced a new query optimization framework called *intensional query optimization* (IQO), which enables existing optimization techniques to be applied to queries that use views. In particular, we consider that view definitions may employ unions. Advanced database technologies and applications—such as federation and mediation over heterogeneous database sources—lead to such complex view definitions, and to the need to handle complex, expensive queries.

Query rewriting techniques have been proposed which exploit semantic query caches, materialized views, and semantic knowledge about the database domain to optimize query evaluation. These can augment syntactic optimization to reduce evaluation costs further. Such techniques include semantic query caching, query folding, and semantic query optimization. However, most proposed rewrite techniques ignore views in queries; that is, the views are considered as other tables. The IQO framework enables rewrites to be applied to various expansions of the query, even when no such rewrite is applicable directly to the query itself. With IQO, we optimize the query tree, not just the query.

The IQO framework introduces the notion of a discounted query, which is a query with some of its expansions "separated out", so the query can be recast into pieces that can be optimized. For this approach to be effective, the sum of the costs of evaluating each piece must be less than the cost of evaluating the query itself. This includes the discounted query. We develop an evaluation plan for discounted queries that is generally more efficient than the evaluation of the queries themselves.

1 Introduction

We have introduced a new query optimization framework called *intensional query optimization* (IQO) [8], which enables existing optimization techniques to be applied to queries that use views. In particular, the framework allows for view definitions that employ unions. Advanced database technologies and applications—federation and mediation over heterogeneous database sources, object oriented databases and query languages, and data warehousing for decision support—lead to such complex view definitions, and tend to incur complex, expensive queries.

Query rewriting techniques have been proposed which exploit semantic query caches, materialized

This research was supported by grant NSF IRI-9300691.

views, and semantic knowledge about the database domain to optimize query evaluation. These can augment syntactic optimization to reduce evaluation costs further. Such techniques include semantic query caching, query folding, and semantic query optimization. However, most proposed query rewrite techniques ignore views in queries; that is, the views are considered as other tables. The IQO framework enables rewrites to be applied to various expansions of the query, even when no such rewrite is applicable directly to the query itself. With IQO, we optimize the query tree, not just the query.

The IQO framework introduces the notion of a *discounted query*, which is a query with some of its expansions "separated out". In this way, the query can be recast into pieces that can be optimized. For this approach to be effective, the sum of the costs of evaluating each piece must be less than the cost of evaluating the query itself. This includes the discounted query. In this paper, we develop an evaluation plan for discounted queries that is generally more efficient than the evaluation of the queries themselves.

Section 2 provides background on semantic optimization and query evaluation, and illustrates a problem of compatibility between preferred query evaluation strategies and existing semantic optimization approaches. Section 3 provides an example. This motivates our general approach. Section 4 presents a formal framework for *discounted queries*, Section 5 shows how the framework can be used for optimization, and Section 6 presents a technique through three progressive refinements for evaluating discounted queries. Section 7 concludes with issues and future work.

2 Background

2.1 Optimization via Semantics

Rewrite techniques which exploit semantic information about the database and query can improve the efficiency of query evaluation. These methods include *semantic query caching* (SQC) [2, 3, 12], *query folding* (QF) [14, 18, 15], and *semantic query optimization* (SQO) [1, 16, 20]. These are primarily suitable for distributed and federated databases, although they (in particular, SQO) can also be applied in centralized databases.

The benefit of SQC lies in saving a part (or all) of query processing by using the cached results of previous queries. In a client-server environment, it is assumed that the client maintains a semantic description of the data in cache, instead of maintaining a list of physical pages or tuple identifiers. Query processing can make use of the semantic description to determine what data are locally available in cache and what data are needed from the server.

QF refers to the activity of determining whether (and how) a query can be answered using a given set of resources, for instance, materialized views. Rewriting a query using this technique does not lead, in general, to an equivalent query, but to a query which is *contained* in the original query [21].

SQO uses semantic knowledge in the form of *integrity constraints* to reformulate a query into a semantically equivalent query that can be evaluated more efficiently. The ultimate win in SQO is the discovery that a query is a *misconception* [7, 5, 10]; that is, it is subsumed by an integrity constraint, hence cannot return any answers. In such cases, the query does not need be evaluated at all.

2.2 Evaluation Strategies

Database queries containing views (also called intensional predicates) can be evaluated in two different ways, often referred to as *top-down* and *bottom-up* [21]. In the top-down approach, all *extensional unfoldings*[1] of the query are generated, each is evaluated and the answer sets are unioned together to produce the final answer set.

Bottom-up query evaluation does not require evaluating separately all extensional unfoldings of a query. Rather, the intensional atoms (views) in the query are substituted with their definitions and evaluated as specified by these definitions. The relational algebra representation of a query—in which all intensional predicates have been substituted with their definitions—implies through the order of the operations an evaluation plan for the query. Union and join operations are intermixed in the query formula.

2.3 Problem

Previous techniques, which we shall call *extensional*,[2] can only be applied to the *conjunctive* query. They do not operate over the *query tree*. If a top-down evaluation is used, all unfoldings of the query are manifested. In this case, these extensional optimization techniques can be applied to each unfolding (a conjunctive query as well), and thus be interleaved with the evaluation process.

The problem is that the top-down query evaluation is considered impractical. It may introduce redundancies both in the join evaluations done and between retrieved answer sets for different unfoldings.[3] More importantly, it may require evaluating an exponential number (in the number of views plus query) of unfoldings. For relational databases, the use of nested views can render top-down evaluation strategies intractable.

Bottom-up evaluation strategies, however, tend to be more efficient, precisely because the unfoldings of the query are *not* manifested. Unfortunately, this means that extensional optimization techniques are not applicable when a bottom-up evaluation is used, except when a technique fortuitously applies directly to the query itself.

The intensional query optimization framework allows the exploitation of information such as misconceptions,[4] cached queries, and materialized views, and can incorporate previous optimization techniques, enabling them to be used in conjunction with bottom-up optimization strategies. The key idea of the approach is to rewrite the query so that certain unfoldings of it are "removed". Such an unfolding may be a null unfolding, or be easily derived from a materialized view or a previously cached query. These unfoldings and the modified query may then be evaluated in a more efficient way than the original query itself.

[1]Informally, an *unfolding* is a result of *expanding* all the views in the query in *one* possible way; that is, choosing one of the unioned definitions for each view. Thus an *extensional unfolding* does not contain any views or unions.

[2]We call these *extensional* since they treat the query as if it were extensional. Any views in the query are considered to be materialized beforehand.

[3]The intersection of two answer sets may be substantial, representing redundant evaluation work.

[4]Let us call *any* unfolding which evaluates to the empty answer set a *null* unfolding.

3 Example

Let DB contain five base relations:

Faculty	(Name, Department, Rank)
Staff	(Name, Department, Years_of_Employment)
Ta	(Name, Department)
Life_ins	(Name, Insurer, Monthly_premium)
Health_plan	(Name, Insurer, Monthly_premium)

Let there be two views (defined here in Datalog) in the **DB**: 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).

$$\begin{array}{ll} employee(X,Y) \leftarrow faculty(X,Y,Z). & benefits(X,Z) \leftarrow life\_ins(X,Z,W). \\ employee(X,Y) \leftarrow staff(X,Y,Z). & benefits(X,Z) \leftarrow health\_plan(X,Z,W). \\ employee(X,Y) \leftarrow ta(X,Y). & \end{array}$$

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

$$Q\!: \leftarrow employee(X,p\_p),\ benefits(X,hmo).$$

A relational algebra (RA) representation[5] of this query is:

$$\begin{array}{ll} Q\!: & (\pi_X faculty(X,\ p\_p,\ Y) \cup \pi_X staff(X,\ p\_p,\ Z) \cup \pi_X ta(X,\ p\_p)) \\ & \qquad\qquad\qquad\qquad \bowtie \\ & (\pi_X life\_ins(X,\ hmo,\ W) \cup \pi_X health\_plan(X,\ hmo,\ V)) \end{array}$$

Query Q can also be represented as a parse tree of its RA representation, which is an AND/OR tree, as shown in Figure 1.

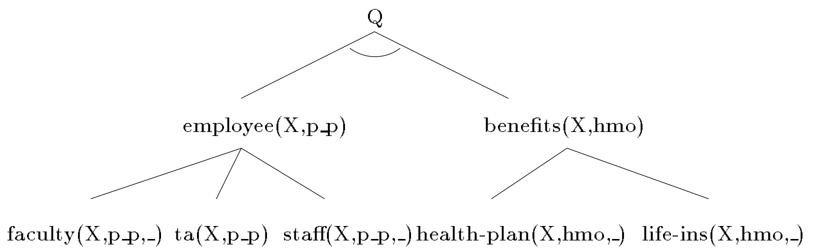


Figure 1: The query tree representation of the original query.

Evaluating the query as specified—in the order of operations as specified in its RA representation—is equivalent to materializing all nodes of the query tree (the AND/OR tree expansion of the query). This requires evaluating the leaves first, then executing the unions to materialize the intermediate nodes *employee*$(X,\ p\_p)$ and *benefits*(X,hmo), and then joining them to materialize the root node, which represents the answer set of the query Q. This type of evaluation is what we call *bottom-up*.

[5]We ignore for simplicity explicit representation of select operations.

Now assume that the following two queries have been asked before and their results are stored in cache. (Equivalently, we can assume that these formulas represent materialized views or are subsumed by integrity constraints, so are null unfoldings.)

$$\mathcal{F}\colon \leftarrow \mathit{ta}(X,\ Y),\ \mathit{life\_ins}(X,\ \mathit{hmo},\ \_).$$

$$\mathcal{G}\colon \leftarrow \mathit{staff}(X,\ \mathit{p\_p}),\ \mathit{health\_plan}(X,\ Z,\ \_).$$

If $\mathcal{F}$ and $\mathcal{G}$ represent integrity constraints, then the joins:

$$\mathcal{U}_1\colon \quad \mathit{ta}(X,\ \mathit{p\_p}) \bowtie \mathit{life\_ins}(X,\ \mathit{hmo},\_)$$

and

$$\mathcal{U}_2\colon \quad \mathit{staff}(X,\ \mathit{p\_p}) \bowtie \mathit{health\_plan}(X,\ \mathit{hmo},\_)$$

which are computed as part of the query cannot possibly return any answers. Thus they can be eliminated from the query without changing the result. Otherwise, if $\mathcal{F}$ and $\mathcal{G}$ are cached queries or materialized views, it may still be advantageous to evaluate the expressions $\mathit{ta}(X,\ \mathit{p\_p}) \bowtie \mathit{life\_ins}(X,\ \mathit{hmo},\ \_)$ and $\mathit{staff}(X,\ \mathit{p\_p}) \bowtie \mathit{health\_plan}(X,\ \mathit{hmo},\ \_)$ separately, as it would require executing single select operations over $\mathcal{F}$ and $\mathcal{G}$. In each of these cases, it is beneficial to "remove" $\mathcal{U}_1$ and $\mathcal{U}_2$ from the query. This leads to the following RA representation of the query:

$$\begin{array}{ll} \mathcal{Q}\colon & \pi_X((\mathit{faculty}(X,\ \mathit{p\_p},\ \_) \cup \mathit{staff}(X,\ \mathit{p\_p},\ \_) \bowtie \mathit{life\_ins}(X,\ \mathit{hmo},\ \_)) \\ & \bigcup \\ & \pi_X((\mathit{faculty}(X,\ \mathit{p\_p},\ \_) \cup \mathit{ta}(X,\ \mathit{p\_p})) \bowtie \mathit{health\_plan}(X,\ \mathit{hmo},\ \_)) \\ & \bigcup \\ & \pi_X(\mathit{staff}(X,\ \mathit{p\_p},\ \_) \bowtie \mathit{health\_plan}(X,\ \mathit{hmo},\ \_)) \\ & \bigcup \\ & \pi_X(\mathit{ta}(X,\ \mathit{p\_p}) \bowtie \mathit{life\_ins}(X,\ \mathit{hmo},\ \_)) \end{array}$$

The AND/OR tree for the modified query is in fact a query *forest*; it contains four trees as presented in Figure 2.

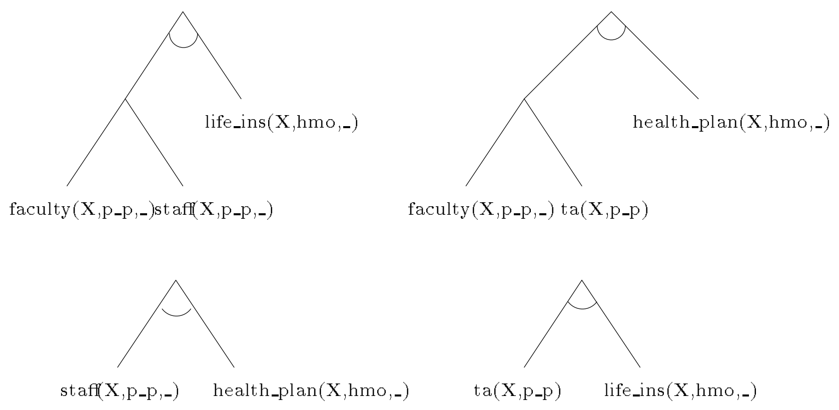


Figure 2: The query tree representation of the optimized query.

The new query does not require evaluating the joins $\mathit{ta}(X,\ \mathit{p\_p}) \bowtie \mathit{life\_ins}(X,\ \mathit{hmo},\ \_)$ and $\mathit{staff}(X,\ \mathit{p\_p},\ \_) \bowtie \mathit{health\_plan}(X,\ \mathit{hmo},\ \_)$ although it is equivalent to the original query (that is, returns the same set of answers). The cost of evaluating the joins is saved, so it can be considered the gross savings due to the optimization.

This naïve approach, however, has several problems; these stem from the fact that this type of query rewriting brings the query closer to its top-down representation. In the extreme case, when there are many unfoldings to remove, the query takes the form of a union of pure conjunctive queries. As a consequence, the disadvantages (discussed in Section 2.3) of the top-down approach to query evaluation become manifest in the evaluation of the "optimized" query. The worst of these—an exponential number of subqueries generated in the top-down approach—renders the naïve optimization impractical for complex queries. Section 6 is devoted to the presentation of an approach to intensional query optimization that scales up for arbitrarily large queries and number of removed unfoldings.

4 Discounted Queries

We make a second pass over the concepts presented above, and formalize the notion of *discounting* unfoldings from a query. First, we are obliged to give a definition for *query.* We shall define a query to be a set of atoms—to be interpreted as a conjunction—and the set of variables to be considered as *free* in the query; that is, the *distinguished* variables of the query.

Definition 1. Define a *query set* to be any set of atoms, composed over the predicates, variables, and constants of the database language. A *query* is a pair $\langle \mathcal{Q}, \mathcal{V} \rangle$ in which $\mathcal{Q}$ is a query set, and $\mathcal{V}$ is a set of variables—each of which must appear in $\mathcal{Q}$—to be called the *distinguished variables* of the query.

Next, we define what an *answer set* is of a query.

Definition 2. The *answer set* for a query with respect to a database is the set of all its answers[6] with respect to that database. Denote the answer set for query $\langle \mathcal{Q}, \mathcal{V} \rangle$ as $[\![\mathcal{Q}]\!]^{\mathcal{V}}$. (We assume it is always known by context with respect to which database a query's answer set is intended.)

We shall call a query with an empty answer set an *null query.*

We give a formal definition for an *unfolding* of a query, introduced earlier.

Definition 3. Given query sets $\mathcal{Q}$ and $\mathcal{U}$, call $\mathcal{U}$ a 1-*step unfolding* of query set $\mathcal{Q}$ with respect to database **DB** *iff*, given some $Q_i \in \mathcal{Q}$ and a rule $\langle A \leftarrow B_1, \ldots, B_n. \rangle$ in **DB** such that $Q_i\theta \equiv A\theta$ where θ is a *most general unifier* [17], then

$$\mathcal{U} = \mathcal{Q} - \{Q_i\} \cup \{B_1\theta, \ldots, B_n\theta\}$$

Denote this by $\mathcal{U} \leq^1 \mathcal{Q}$. Call $\mathcal{U}_1$ simply an *unfolding* of $\mathcal{Q}$, written as $\mathcal{U}_1 \leq \mathcal{Q}$, *iff* there is some finite collection of query sets $\mathcal{U}_1$,...,$\mathcal{U}_k$ such that $\mathcal{U}_1 \leq^1 \ldots \leq^1 \mathcal{U}_k \leq^1 \mathcal{Q}$.

A query set (unfolding) $\mathcal{Q}$ is called *extensional iff*, for every $Q_i \in \mathcal{Q}$, atom Q_i is written with an extensional predicate. Call the query set *intensional* otherwise.

[6]An *answer* is any ground substitution θ over $\mathcal{V}$ such that $\mathcal{Q}\theta$ follows from the database [17].

One of the 1-step unfoldings of the query in the Example of Section 3 is: $\{$*employee*$(X,\ p\_p)$, *life_ins*$(X,\ hmo,\ \_)\}$. One of the extensional unfoldings of the query is: $\{$*faculty*$(X, p\_p,\ \_)$, *life_ins*$(X,\ hmo,\ \_)\}$.

A query can be considered to be equivalent to the union of all its extensional unfoldings.

Definition 4. Given any query set $\mathcal{Q}$, let **unfolds**/*1* denote the set of all extensional unfoldings of $\mathcal{Q}$.

$$\mathbf{unfolds}\,(\mathcal{Q}) \quad = \quad \{\mathcal{U} \mid \mathcal{U} \text{ is an extensional query set and } \mathcal{U} \leq \mathcal{Q}\}$$

We make a natural assumption of our semantics that

$$[\![\mathcal{Q}]\!]^{\mathcal{V}} \quad = \quad \bigcup_{\mathcal{U} \in \mathbf{unfolds}\,(\mathcal{Q})} [\![\mathcal{U}]\!]^{\mathcal{V}}$$

and that this is another way to define a query's answer set.

We introduce a new query structure to be called a *discounted query*, which can be considered to be the query with some of its unfoldings "removed" (discounted).

Definition 5. Define a new type of query structure to be called a *discounted query set.* Given a query set $\mathcal{Q}$ and unfoldings $\mathcal{U}_1,\ldots,\mathcal{U}_k$ of $\mathcal{Q}$, then $\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\}$ is a discounted query set, to be read as "query set $\mathcal{Q}$ discounting unfoldings $\mathcal{U}_1,\ldots,\mathcal{U}_k$." Define a *discounted query* to be a query with a discounted query set. We call $\mathcal{U}_1,\ldots,\mathcal{U}_k$ *unfoldings-to-discount* and the tuples in the answer sets of these unfoldings *tuples-to-discount.*[7]

Definition 6. Let us extend the domain of **unfolds**/*1* to include discounted query sets, and its definition as follows. Given discounted query set $\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\}$,

$$\mathbf{unfolds}\,(\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\}) \quad = \quad \{\mathcal{R} \mid \mathcal{R} \text{ is extensional, } \mathcal{R} \leq \mathcal{Q}, \text{ but } \mathcal{R} \not\leq \mathcal{U}_i, \text{ for } i \in \{1,\ldots,k\}\}$$

Since the unfoldings-to-discount in our example are $\{$*staff*$(X,\ p\_p,\ \_)$, *health_plan*$(X,\ hmo,\ \_)\}$ and $\{$*ta*$(X,\ p\_p)$, *life_ins*$(X,\ hmo, \_)\}$ then the unfolds of the query are:

- $\{$*ta*$(X,\ p\_p)$, *health_plan*$(X,\ hmo, \_)\}$
- $\{$*faculty*$(X,\ p\_p)$, *life_ins*$(X,\ hmo, \_)\}$
- $\{$*faculty*$(X,\ p\_p)$, *health_plan*$(X,\ hmo, \_)\}$
- $\{$*staff*$(X,\ p\_p)$, *health_plan*$(X,\ hmo, \_)\}$

Let us now define the answer set semantics of discounted queries as follows:

$$[\![\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\}]\!]^{\mathcal{V}} \quad = \quad \bigcup_{\mathcal{R} \in \mathbf{unfolds}\,(\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\})} [\![\mathcal{R}]\!]^{\mathcal{V}}$$

(Note that this is not the same as relational set minus.)

Clearly, given a query $\langle \mathcal{Q}, \mathcal{V}\rangle$, and a collection of unfoldings $\mathcal{U}_1,\ldots,\mathcal{U}_k$ of $\mathcal{Q}$, then

$$[\![\mathcal{Q}]\!]^{\mathcal{V}} \quad = \quad [\![\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\}]\!]^{\mathcal{V}} \cup [\![\mathcal{U}_1]\!]^{\mathcal{V}} \cup \ldots \cup [\![\mathcal{U}_k]\!]^{\mathcal{V}}$$

[7]This definition is semantic. We could likewise give a syntactic definition, which is more what we intend. However, such would be more cumbersome, and this distinction is not important here.

This provides us a means to treat various unfoldings of the query independent of the query itself.

5 Optimization by Discounting

5.1 Costs

For discounted queries to be useful for optimization, clearly the following cost equation must hold:

$$\mathbf{cost}\,(\mathbf{dis\_eval}\,(\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\},\mathcal{V})) + \sum_{i=1}^{k}\mathbf{cost}\,(\mathbf{eval}\,(\mathcal{U}_i,\mathcal{V})) + \mathbf{C} < \mathbf{cost}\,(\mathbf{eval}\,(\mathcal{Q},\mathcal{V})) \qquad \mathbf{1)}$$

where **eval**/*2* is a procedure which determines an evaluation strategy for its query (and executes the chosen strategy), **cost**/*1* measures the evaluation time of the query (to manifest a strategy and execute it), and **C** is the cost involved in somehow choosing $\mathcal{U}_1$,. . .,$\mathcal{U}_k$. There must be a specialized evaluation strategy for discounted queries, denoted as **dis_eval**/*2*, which is a focus of our research. It must be that discounted queries can be evaluated less expensively than the corresponding query, else this approach will not work. We briefly discuss strategies for **dis_eval**/*2* in the next section.

If we have *multiple query optimization* strategies [11, 19] available—that is, strategies which exploit similarities between queries to be evaluated in batch in order to optimize evaluation—we can weaken this cost equation. Let **mult_eval**/*2* represent such a strategy, which takes a set of queries (and discounted queries) to be evaluated in batch.

$$\mathbf{cost}\,(\mathbf{mult\_eval}\,(\{\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\},\mathcal{U}_1,\ldots,\mathcal{U}_k\},\mathcal{V})) + \mathbf{C} < \mathbf{cost}\,(\mathbf{eval}\,(\mathcal{Q},\mathcal{V})) \qquad \mathbf{2)}$$

We shall not consider multiple query optimization here.

5.2 Applications of Discounting

If it can be determined that a query is null, this is the ideal optimization: the query does not need to be evaluated because the answer set is already known (to be empty). Likewise, if a collection of null unfoldings can be found for a query, these may be quite useful to optimize evaluation. Let $\mathcal{N}_1$,. . .$\mathcal{N}_k$ be null unfoldings of query $\langle\mathcal{Q},\mathcal{V}\rangle$. Then one can optimize simply if

$$\mathbf{cost}\,(\mathbf{dis\_eval}\,(\mathcal{Q}\backslash\{\mathcal{N}_1,\ldots,\mathcal{N}_k\},\mathcal{V})) + \mathbf{C} < \mathbf{cost}\,(\mathbf{eval}\,(\mathcal{Q},\mathcal{V}))$$

The null unfoldings of course cost nothing to evaluate.[8]

Another case when it might be advantageous to separate out a certain unfolding from the query is when the unfolding can be easily answered from cached queries or materialized views. This can be particularly pertinent in a client/server environment, in which network communication costs can dominate the query processing. If the cached queries or materialized views are local, the unfolding can be evaluated locally, and so not involve any network communication overhead. When the unfolding can be completely evaluated from a single cached query or materialized view—that is, it can be computed simply via projections and selects on the cache—the cost is negligible, and could

[8]Of course they do cost something to discover, and this cost, so far, is hidden in **C**.

be counted as zero. Thus, this is similar to the null unfolding case. Otherwise, it may be that the unfolding can be computed by the joins of several caches; this is the method that QF explores. In this case, the cost of evaluating the unfolding via caches (and QF) must be weighed against re-evaluating the unfolding, or, perhaps, not discounting it from the query at all.

The ability to separate an unfolding from a query can extend dramatically the domain of applications of traditional SQO techniques. Almost all SQO algorithms apply to conjunctive queries only[9] hence cannot be used when the query is evaluated bottom-up. After an unfolding is separated, however, SQO can be used to rewrite it by eliminating some of the atoms (because they are redundant) or by adding new atoms (when the addition does not change the semantics, but may simplify the evaluation by, for example, adding a selection or an indexed attribute).

The intensional optimization framework offers a way to incorporate these techniques more generally. We can simply replace the **eval/***2* operations in cost equation **1)** with, say, **sem_eval/***2*, an evaluation strategy which incorporates semantic optimization techniques. An important question is how to decide *which* unfoldings ought to be discounted from the query so that they may be semantically optimized.

5.3 Identifying Useful Unfoldings

So far we have ignored the first stage of intensional optimization: determining which unfoldings of a query are advantageous to discount; that is, finding the collection $\mathcal{U}_1,\ldots,\mathcal{U}_k$. This is the cost **C** in all the above cost equations.

These costs can be non-trivial. However, efficient techniques for such—in particular, identifying the null unfoldings of a query—have been explored [9], and developed within the context of the Carmin project, a cooperative database system [5, 6, 10]. For edification of the reader, we briefly sketch an approach to finding null unfoldings. The general approach that we have developed is as follows.

- Rewrite each integrity constraint ($\mathcal{IC}$) by replacing its empty head with the special predicate *bottom*, '$\perp$', so it becomes a rule (view).
- Assume an "answer" to the query, using Skolem constants.
- Employ standard deduction over these facts (a hypothesized answer), the set of database views, and the converted $\mathcal{IC}$s, to determine whether '$\perp$' is derivable.

These techniques could be extended to identify when cached queries and materialized views can be used to evaluate certain unfoldings of queries.

[9]The papers [13, 16] are the only exceptions known to us.

6 A Discounted Evaluation Strategy

6.1 Possible Strategies

Whether intensional optimization is feasible hinges on whether there exist good evaluation strategies for discounted queries, **dis_eval**/*2*, which are less expensive than evaluating the query itself. Since, the unfoldings of the query are not manifested in a bottom-up evaluation, there is a question of how to "discount" unfoldings during the evaluation and whether this can be exploited for optimization. We shall demonstrate that discounted queries can be, in general, evaluated bottom-up less expensively than their corresponding queries [9]. We briefly outline two basic approaches (described in more detail in [9]) for a bottom-up **dis_eval**.

One approach, sketched in the example of Section 3, is to *rewrite* a discounted query into a collection of conjunctive queries, for which evaluation strategies already exist [9]. Thus, given unfoldings $\mathcal{U}_1,\ldots,\mathcal{U}_k$ to discount from $\mathcal{Q}$, one should find unfoldings $\mathcal{W}_1,\ldots,\mathcal{W}_l$ of $\mathcal{Q}$ such that

$$\textbf{unfolds}\,(\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\}) \quad = \quad \bigcup_{i=1}^{l} \textbf{unfolds}\,(\mathcal{W}_i)$$

Ideally, the collection of $\mathcal{W}_1,\ldots,\mathcal{W}_l$ is *minimal* in the sense that there is no $\mathcal{R} > \mathcal{W}_i$ such that $\mathcal{W}_1,\ldots,\mathcal{W}_{i-1}$, $\mathcal{R}$, $\mathcal{W}_{i+1},\ldots,\mathcal{W}_l$ has the above property, and there are no distinct $\mathcal{W}_i$ and $\mathcal{W}_j$ such that $\mathcal{W}_i > \mathcal{W}_j$. Then we have a natural evaluation strategy for $\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_k\}$:

$$\bigcup_{i=1}^{l} \textbf{eval}\,(\mathcal{W}_i, \mathcal{V}) \qquad \text{or} \qquad \textbf{mult\_eval}\,(\{\mathcal{W}_1,\ldots,\mathcal{W}_l\}, \mathcal{V})$$

The advantage of this approach is that it accounts for all the unfoldings to be discounted "in parallel". The disadvantage is that the collection of $\mathcal{W}_i$'s may be large. In essence, this rewrite approach mixes top-down and bottom-up representation. The query is partially evaluated top-down, just enough so that the unfoldings to discount are explicitly represented, as shown in the example in Section 3. Therefore, this approach is equivalent in the limit to top-down evaluation.

A second type of approach is to modify a bottom-up evaluation strategy to account for the unfoldings to be discounted during the evaluation process. The advantage here is that the approach is dynamic: it interleaves evaluation and optimization planning. Thus, cost measures can be judged during the evaluation to decide whether to discount a given unfolding, depending on whether there would be any actual cost savings. The disadvantage is that the query is still explicitly split into subqueries, hence cannot scale up for complex queries.

This dynamic approach can be modified, however, to avoid explicit splitting of a query. We present below a dynamic approach which would store label information about the unfoldings-to-discount in the materialized tables during bottom-up evaluation. These labels are then used during the evaluation to ensure that the the tuples which would result from (the evaluation of) the unfoldings-to-discount do not contribute to the answer set of the discounted query.

6.2 Overview

Our strategy shall be a bottom-up materialization strategy for the query tree with the union and join operations modified to account for the discounted unfoldings. The strategy will ensure two things:

- that tuples-to-discount do not contribute to the answer set of the discounted query, and
- the joins representing unfoldings-to-discount are never evaluated.

Of course, when the unfoldings-to-discount are null unfoldings, the first property is ensured trivially. In the case of discounting an unfolding which is a cached query or a materialized view, the tuples resulting from this unfolding have to be explicitly removed. This can be done either during or after the actual query evaluation. To ensure the second property, we want somehow to avoid evaluating the unfoldings-to-discount.

The method shall be to keep extra information in the temporary tables created during the materialization of the query tree. In essence, each table will have an extra column for each unfolding-to-discount. The domain of these *tag* columns is boolean. The value of a tag column for a given tuple shall indicate whether that tuple belongs to the answer set of the corresponding unfolding-to-discount. (It is marked as *true* if yes, and *false* if no.) On each union or join operation (which creates a new temporary table), these tag columns' values must be maintained properly.

By keeping this derivation information for each tuple during evaluation, we can easily ensure the first property from above: after evaluation of the query, select those tuples which have all *false* values in the tag columns (and project away these columns). However, this alone does not ensure the second property. We shall be able further to use the tag columns to determine during a join operation which tuples should be joined, and which should not be (because the resulting tuple would be from an unfolding-to-discount). This can be done whenever the join operation is the last one relevant to a given unfolding-to-discount. The computation saved is the joining of sections from the two tables. If these sections (sub-tables) are large, this can be significant. We shall see that this savings is, in a sense, optimal.

We present the evaluation strategy in three progressive versions. The simplest version will ensure only the first property; that is, the final answer set will not contain tuples from any unfolding-to-discount. This strategy *does not* optimize query evaluation in centralized databases; in fact, it adds overhead to traditional bottom-up evaluation. However, by reducing the size of the answer set, it does generate savings in bandwidth for database systems in which query results are sent over a network. The second version removes the tuples-to-discount *during* query evaluation, as soon as is possible. This strategy can reduce the cost of query evaluation, even when it does not involve network traffic. The third, most complex, strategy ensures the second property from above; that is, the join operations which represent the unfoldings-to-discount are never evaluated.

6.3 Bottom-up Query Materialization

Evaluation will be done over the query tree. We insist that each node in the query tree be either an AND or an OR node. (So we do not allow mixed nodes in the tree.) An AND node is a node whose parent is generated through a join operation; an OR node is a node whose parent is generated through an union operation. An AND/OR tree can always be rewritten to this effect. Extra nodes may have to be added to separate AND and OR nodes. This can be done in an obvious manner.

The first step of our algorithm requires that we rewrite the query tree into *binary form*. This means that every node in the tree has either two children or none. Again, this can be done in an obvious manner by the addition of new interior nodes to the tree. The binary tree representation of the query of Section 3 is presented in Figure 3.

Given a binary query tree, a bottom-up evaluation strategy is easy to specify.

Definition 7. Let **QT** be a binary query tree. For any node $\mathcal{N}$ in the query tree, call it a *leaf* if it has no children, and a *branch* if it does.

Any branch node $\mathcal{N}$ in a binary query tree has two children, by definition. We arbitrarily fix an order on the tree, without loss of generality. We refer to one child of $\mathcal{N}$ as $\mathcal{L}_{\mathcal{N}}$ (for *left* child), and the other as $\mathcal{R}_{\mathcal{N}}$ (for *right* child).

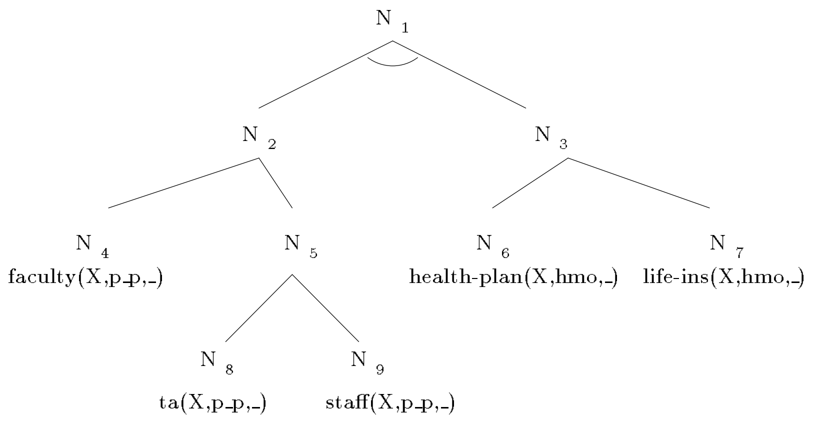


Figure 3: The binary tree representation of the original query.

We assume that any leaf $\mathcal{N}$ in the (binary) query tree has a corresponding table in the database; that is, the answer set for $\mathcal{N}$ is derivable from some table in the database via selects and projections. Call $\mathcal{N}$'s table (with any selects and projections implicit) $\mathbf{T}_{\mathcal{N}}$.

Algorithm 1 shows an evaluation strategy. In any implementation of this, we assume that all the natural optimizations are made: all relevant selects in the query tree are pushed as far down the query tree as possible to the relevant nodes, and all columns that can be projected when making a (temporary) table are projected out.

Note that there are choices when transforming the query tree into binary form. The final binary form dictates the order of the join operations to be performed. Of course join order is quite important in the efficiency of evaluation. The transformation into binary form should be done with the syntactic query optimizer in order to choose a good join order.

6.4 Optimization I

We shall modify Algorithm 1 to handle discounted queries. This shall involve replacing the union and join operations with specialized versions, which handle and exploit the tag columns for the unfoldings-to-discount, as discussed above.

```
While there exists a branch in $\mathbf{QT}$
    % Choose the next operation.
    Choose a branch $\mathcal{N}$ from the query tree $\mathbf{QT}$
    such that $\mathcal{L}_{\mathcal{N}}$ and $\mathcal{R}_{\mathcal{R}}$ are leaves.

    Cases:
        $\mathcal{N}$ is an AND node.
            Union $\mathbf{T}_{\mathcal{L}_{\mathcal{N}}}$ and $\mathbf{T}_{\mathcal{R}_{\mathcal{N}}}$ to create $\mathbf{T}_{\mathcal{N}}$.
        $\mathcal{N}$ is an OR node.
            Join $\mathbf{T}_{\mathcal{L}_{\mathcal{N}}}$ and $\mathbf{T}_{\mathcal{R}_{\mathcal{N}}}$ to create $\mathbf{T}_{\mathcal{N}}$.

    % Clean up.
    Erase nodes $\mathcal{L}_{\mathcal{N}}$ and $\mathcal{R}_{\mathcal{N}}$ from $\mathbf{QT}$.
    For $\mathcal{A} \in \{\mathcal{L}_{\mathcal{N}}, \mathcal{R}_{\mathcal{N}}\}$
        If $\mathbf{T}_{\mathcal{A}}$ is a temporary table then
            Remove $\mathbf{T}_{\mathcal{A}}$ from the database.
```

Algorithm 1: Bottom-up Materialization of the Query Tree

Given discounted query $\mathcal{Q}\backslash\{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$, let $\mathbf{QT}_{\mathcal{Q}}$ be a binary query tree for query $\mathcal{Q}$ as in Figure 3. We shall introduce new columns, $\mathbf{C}_{\mathcal{U}_i}$, for $i \in \{1, \ldots, k\}$, as the tag columns corresponding to the unfoldings-to-discount. Thus, $\mathbf{C}_{\mathcal{U}_1}$ is added to tables $\mathbf{T}_{N_6}$ and $\mathbf{T}_{N_8}$ and $\mathbf{C}_{\mathcal{U}_2}$ is added to tables $\mathbf{T}_{N_7}$ and $\mathbf{T}_{N_9}$. During the materialization process, tables will include certain of these tag columns, and the new union and join operators will set their values accordingly.

6.4.1 The Modified Union Operation

We assume for a well-formed query tree that the tables to be unioned at any union step are union-compatible. With our addition of tag columns, this may now be violated. The two tables to be unioned may not be union-compatible over the tag columns. Thus, we need to modify the union step first to make the tables union-compatible by adding any tag columns that are needed. Algorithm 2 shows this. These are the only ways in which we need to modify the union step.

Note that if the operation involves two nodes both subsumed by a given unfolding-to-discount, neither corresponding table has a tag column for that unfolding. We only add the column when the operation involves one node subsumed by the unfolding and one not. It is only after that point that the tag column is needed. Thus we add the tag columns only on a need-by basis.

6.4.2 The Modified Join Operation

We must assign the correct values to tag columns of joined tables. If a tuple results from the join of one tuple which was derived under a given unfolding-to-discount $\mathcal{U}$ (hence the value of its $\mathbf{C}_{\mathcal{U}}$ is *true*), and a second tuple which was not (hence the value of its $\mathbf{C}_{\mathcal{U}}$ is *false*), then the resulting tuple is not in the answer set of $\mathcal{U}$. (So $\mathbf{C}_{\mathcal{U}}$ for the resulting tuple should have the value *false*.) Only when both tuples being joined were derived under $\mathcal{U}$ should the resulting tuple's column $\mathbf{C}_{\mathcal{U}}$ be set to *true*.

```
% Add new tag columns on a need-by basis.
For each $\mathcal{U}_i$
    For $\{\mathcal{A}, \mathcal{B}\} = \{\mathcal{L}_\mathcal{N}, \mathcal{R}_\mathcal{N}\}$
        If $\mathcal{A} \in \mathcal{U}_i$ then
            % Clearly $\mathcal{B}$ does not belong to $\mathcal{U}_i$, by definition.
            Add column $\mathbf{C}_{\mathcal{U}_i}$ to $\mathbf{T}_\mathcal{A}$.
            Instantiate all values of $\mathbf{C}_{\mathcal{U}_i}$ in $\mathbf{T}_\mathcal{A}$ to true.
% Make union-compatible.
For each $\mathcal{U}_i$
    For $\{\mathcal{A}, \mathcal{B}\} = \{\mathcal{L}_\mathcal{N}, \mathcal{R}_\mathcal{N}\}$
        If $\mathbf{C}_{\mathcal{U}_i}$ belongs to $\mathbf{T}_\mathcal{A}$ but not to $\mathbf{T}_\mathcal{B}$ then
            Add column $\mathbf{C}_{\mathcal{U}_i}$ to $\mathbf{T}_\mathcal{B}$.
            Instantiate all values of $\mathbf{C}_{\mathcal{U}_i}$ in $\mathbf{T}_\mathcal{B}$ to false.
% Union the two tables.
Union $\mathbf{T}_{\mathcal{L}_\mathcal{N}}$ and $\mathbf{T}_{\mathcal{R}_\mathcal{N}}$ to create $\mathbf{T}_\mathcal{N}$.
% Clean up.
Remove $\mathcal{L}_\mathcal{N}$ and $\mathcal{R}_\mathcal{N}$ from QT.
For $\mathcal{A} \in \{\mathcal{L}_\mathcal{N}, \mathcal{R}_\mathcal{N}\}$
    If $\mathbf{T}_\mathcal{A}$ is a temporary table then
        Remove $\mathbf{T}_\mathcal{A}$ from the database.
```

Algorithm 2: The Modified Union Operation

We define an auxiliary notion of $\mathbf{merges}_\mathcal{N}$ and then define a modified notion of join.

Definition 8. Let $\mathbf{merges}_\mathcal{N}$ be the collection of column assignment statements

$$(\mathbf{T}_{\mathcal{L}_\mathcal{N}}.\mathbf{C}_\mathcal{U} \wedge \mathbf{T}_{\mathcal{R}_\mathcal{N}}.\mathbf{C}_\mathcal{U}) \text{ as } \mathbf{C}_\mathcal{U}$$

for each unfolding-to-discount $\mathcal{U}$ which has a tag column in both tables $\mathbf{T}_{\mathcal{L}_\mathcal{N}}$ and $\mathbf{T}_{\mathcal{R}_\mathcal{N}}$

Definition 9. Define the following criteria for the join.

- $\mathbf{cols}_\mathcal{N}$: the relevant query columns to be kept for the query at node $\mathcal{N}$ plus any tag columns from tables $\mathbf{T}_{\mathcal{L}_\mathcal{N}}$ and $\mathbf{T}_{\mathcal{R}_\mathcal{N}}$ not accounted for in $\mathbf{merges}_\mathcal{N}$.
- $\mathbf{join}_\mathcal{N}$: the relevant query join criteria for $\mathcal{L}_\mathcal{N}$ and $\mathcal{R}_\mathcal{N}$ to produce $\mathcal{N}$.
- $\mathbf{sel}_{\mathcal{L}_\mathcal{N}}$ and $\mathbf{sel}_{\mathcal{R}_\mathcal{N}}$: the relevant query selects for $\mathcal{L}_\mathcal{N}$and $\mathcal{R}_\mathcal{N}$, respectively.

Now we can state a relational algebra expression for $\mathbf{T}_\mathcal{N}$ in terms of $\mathbf{T}_{\mathcal{L}_\mathcal{N}}$ and $\mathbf{T}_{\mathcal{R}_\mathcal{N}}$.

$$\mathbf{T}_\mathcal{N} \quad := \quad \Pi_{\mathbf{cols}_\mathcal{N}, \mathbf{merges}_\mathcal{N}}(\sigma_{\mathbf{sel}_{\mathcal{L}_\mathcal{N}}} \mathbf{T}_{\mathcal{L}_\mathcal{N}} \bowtie_{\mathbf{join}_\mathcal{N}} \sigma_{\mathbf{sel}_{\mathcal{R}_\mathcal{N}}} \mathbf{T}_{\mathcal{R}_\mathcal{N}})$$

This expression represents the modified join operation. This can be readily implemented in SQL.

The modified union and join operations have no influence (except for adding extra columns) on the final answer set of a query. Their only purpose is to keep the trace information about the unfoldings-to-discount via the tag columns. The last step of the optimization algorithm then consists in using this information to select only the tuples that are known not to be in an answer set of any unfolding-to-discount of the query. To ensure this, it is sufficient to select the tuples that have the value *false* for all their tag columns.

6.5 Optimization II

As stated in Section 6.2, removing tuples-to-discount from the final query answer set according to the optimization described above does not improve efficiency of query evaluation (unless the query's answer set is shipped over a network). For complex queries, however, such removal can be executed *during* query evaluation; that is, before the final answer set is produced. In other words, we can push some of the selects for *false* over the tag columns further down in the query tree. Consider the following example.

Let the query be $(A \cup B) \bowtie (C \cup D) \bowtie E$, and the unfolding-to-discount be $A \bowtie C \bowtie E$. Then, all tuples in the answer set of the join $A \bowtie C$ can be eliminated from the answer set of the join $(A \cup B) \bowtie (C \cup D)$ as soon as it is evaluated. This can be done because the tuples generated by the join $A \bowtie C$ contribute *only* to the answer set of the unfolding-to-discount and no other unfoldings of the query. Note that the gross savings achieved through this optimization is equal to the cost of the join of the result of $A \bowtie C$ with E.

We introduce the notion of a *closing* of an unfolding-to-discount by a node in a query tree—not necessarily the root node—and modify our original algorithm by allowing the elimination of certain tuples-to-discount from the tables representing such nodes.

Definition 10. Unfolding $\mathcal{U}$ is *closed* by node $\mathcal{N}$ with respect to binary query tree **QT** if every node in $\mathcal{U}$ that cannot be reached from the root of **QT** through AND nodes only, is in the sub-AND/OR tree rooted by $\mathcal{N}$, and there is no other node below $\mathcal{N}$ in the sub-AND/OR tree rooted by $\mathcal{N}$ which has this property.

Given discounted query $\mathcal{Q}\backslash\{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$ and binary query tree $\mathbf{QT}_{\mathcal{Q}}$ for $\mathcal{Q}$, define $\mathbf{close}_{\mathcal{N}}$ for each node in $\mathbf{QT}_{\mathcal{Q}}$ as the subset of the unfoldings-to-discount, $\{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$, which are closed by $\mathcal{N}$.

The intuition of the definition of closing is as follows. Consider the nodes of $\mathcal{U}$ that do not lie under $\mathcal{N}$. None of these nodes contain a column for $\mathcal{U}$ and since none of these tables will be used in a union (because there are AND nodes only between these nodes and the query tree root) such columns will never be created for the tuples from these tables. Hence, no tuple in the table represented by the node $\mathcal{N}$ could contribute to a change in the value in the column $\mathbf{C}_{\mathcal{U}}$ when they are joined during evaluation with other tables represented by the nodes of $\mathcal{U}$. Thus, the tuples with the value *true* for any unfoldings in $\mathbf{close}_{\mathcal{N}}$ can be removed immediately after the table $\mathbf{T}_{\mathcal{N}}$ is materialized.

6.6 Optimization III

In the next step, one can push the tag selects even further down, in a sense *beneath* the joins. That is, for a join which is closing an unfolding-to-discount, say $\mathcal{U}$, the join itself can be optimized: for any pair of tuples—one from the first table and the other from the second—to be considered for join comparison, if both have the value *true* in tag column $\mathcal{U}$, the pair should be immediately dropped from consideration. Going a step further, it would be better to ensure somehow that such pairs never even arise for consideration during the join operation.

To ensure that such pairs never arise can be accomplished by joining only *portions* of the two tables, such that tuple pairs each having *true* for a tag column being closed inherently cannot occur. Thus, the original join can be replaced with the *union* of a collection of such joins over appropriate selects on the tables. We shall show a partitioning of the tables which accomplishes this, and call the modified join procedure the *partitioned join* of the tables.

The intuition of how the partitioned join can be an optimization is as follows. The partitioned join reduces the number of operations to be done during the join execution, compared with a straight join of the two tables. If a naïve cross-join strategy is chosen—that is, to materialize the cross-join of the two tables, and then select for equality of the join attributes—clearly the partitioned join saves steps: the tuple pairs we want to avoid simply do not appear in the cross-join. If a sort-scan-merge style join strategy is chosen, the number of scan operations that are done in sum for the partitioned join will be inherently fewer than would be done by the straight join. (Both tables must still be sorted in full and indexed, in either case.) If the percentage of tuples with the value *true* in tag columns for unfoldings being closed is large, the number of saved scan steps is significant.

The partitioned join adds overhead. The tables must be sorted and indexed over the tag columns for the unfoldings being closed so that the appropriate partitions of the tables can be efficiently accessed during the individual joins. If both tables are small to begin with, or if the percentage of tuples marked *true* is small, this overhead may outweigh any benefit.

The partitioned join operation will be defined similarly to the modified join defined in Section 6.4.2. Instead of considering *all* tuples from the tables during the join, we will only look at the tuples selected according to the *select patterns* defined below.

Definition 11. Let $\mathcal{J} \subseteq \mathbf{close}_{\mathcal{N}}$. Define select patterns $\mathbf{pat}_{\mathcal{L}_{\mathcal{N}},\mathcal{J}}$ and $\mathbf{pat}_{\mathcal{R}_{\mathcal{N}},\mathcal{J}}$ as follows.

$$\begin{aligned}
\mathbf{pat}_{\mathcal{L}_{\mathcal{N}},\mathcal{J}} &:= \bigwedge_{\mathcal{U}\in\mathcal{J}} (\mathbf{T}_{\mathcal{L}_{\mathcal{N}}}.\mathbf{C}_{\mathcal{U}} = \mathit{false}) \\
\mathbf{pat}_{\mathcal{R}_{\mathcal{N}},\mathcal{J}} &:= \bigwedge_{\mathcal{U}\in\mathcal{J}} (\mathbf{T}_{\mathcal{R}_{\mathcal{N}}}.\mathbf{C}_{\mathcal{U}} = \mathit{true}) \quad \wedge \bigwedge_{\mathcal{U}\in(\mathbf{close}_{\mathcal{N}}-\mathcal{J})} (\mathbf{T}_{\mathcal{R}_{\mathcal{N}}}.\mathbf{C}_{\mathcal{U}} = \mathit{false})
\end{aligned}$$

We describe the intuition behind this definition using our example. The result of the join of the tables $\mathbf{T}_{N_2}$ and $\mathbf{T}_{N_3}$ should not contain tuples that result from the joins *ta*$(X,\ p\_p) \bowtie$ *life_ins*$(X,\ hmo,\ \_)$ and *staff*$(X,\ p\_p,\ \_) \bowtie$ *health-plan*$(X,\ hmo,\ \_)$. To ensure this exclusion, it is enough to select for the join $\mathbf{T}_{N_2} \bowtie \mathbf{T}_{N_3}$ only such pairs of tuples from $\mathbf{T}_{N_2}$ and $\mathbf{T}_{N_3}$ that do not both have the value *true* for either $\mathbf{C}_{\mathcal{U}_1}$or $\mathbf{C}_{\mathcal{U}_2}$. Another way of stating this is to select from $\mathbf{T}_{N_2}$ and $\mathbf{T}_{N_3}$ tuples whose values for $\mathbf{C}_{\mathcal{U}_1}$ and $\mathbf{C}_{\mathcal{U}_2}$ are respectively:

$\mathbf{T}_{N_2}$		$\mathbf{T}_{N_3}$	
$\mathbf{C}_{\mathcal{U}_1}$	$\mathbf{C}_{\mathcal{U}_2}$	$\mathbf{C}_{\mathcal{U}_1}$	$\mathbf{C}_{\mathcal{U}_2}$
false	false	true	true
false	true	true	false
false	false	true	false
true	false	false	true
false	false	false	true
true	true	false	false
true	false	false	false
false	true	false	false
false	false	false	false

Note that tuples with values of $\mathbf{C}_{\mathcal{U}_1}$ and $\mathbf{C}_{\mathcal{U}_2}$ as described in line 1 represent the select pattern with $\mathcal{J} = \{\mathcal{U}_1 \cup \mathcal{U}_2\}$, lines 2 and 3 with $\mathcal{J}=\{\mathcal{U}_1\}$, lines 4 and 5 with $\mathcal{J}=\{\mathcal{U}_2\}$, and lines 6-9 with $\mathcal{J}=\{\emptyset\}$.

The definition of a partitioned join operation is now straightforward.

Definition 12. (Partitioned Join) We modify slightly Definition 8 for $\mathbf{merges}_\mathcal{N}$ to not include $\mathbf{close}_\mathcal{N}$. We also modify Definition 9 of $\mathbf{cols}_\mathcal{N}$ likewise. Then the *partioned join* for $\mathcal{N}$ is as follows.

$$\mathbf{T}_\mathcal{N} \quad := \bigcup_{\mathcal{J} \subseteq \mathbf{close}_\mathcal{N}} \Pi_{\mathbf{cols}_\mathcal{N}, \mathbf{merges}_\mathcal{N}}(\sigma_{\mathbf{sel}_{\mathcal{L}_\mathcal{N}}, \mathbf{pat}_{\mathcal{L}_\mathcal{N}, \mathcal{J}}} \mathbf{T}_{\mathcal{L}_\mathcal{N}} \bowtie_{\mathbf{join}_\mathcal{N}} \sigma_{\mathbf{sel}_{\mathcal{R}_\mathcal{N}}, \mathbf{pat}_{\mathcal{R}_\mathcal{N}, \mathcal{J}}} \mathbf{T}_{\mathcal{R}_\mathcal{N}})$$

Notice that we would not need to partition for all the unfoldings-to-discount in $\mathbf{close}_\mathcal{N}$; we could pick a subset of them. This can be done dynamically, and a subset can be chosen which would provide the best optimization. (This is discussed in the next section.) The tuples-to-discount from any unfolding-to-discount in $\mathbf{close}_\mathcal{N}$ which was not chosen to partition over can be removed after the join is completed, just as described in Section 6.5.

6.7 Savings Analysis

We present an analysis of the potential savings associated with IQO. Some of the savings can be expressed as the proportion of tuples eliminated from the query's answer set by the optimization algorithm. The purpose of this analysis is twofold. First, this is a first-level approximation of the net savings via IQO in any practical system. A complete analysis would have to account not only for the number of tuples eliminated, but also the point in the query evaluation at which the removal takes place (the sooner it happens, the larger the savings), as well as any overhead costs incurred by the algorithm. Second, the number of eliminated tuples serves as a simple heuristic that could be used within an optimization algorithm. Such a heuristic would be useful to decide when an "optimization" step should be performed, and when it should be skipped.

The proportion of tuples-to-discount in the final answer set depends on three variables: the *probability* that a tuple is tagged with *true* for an unfolding-to-discount, the *number* of unfoldings-to-discount and the *size* of the join (how many tables participate).

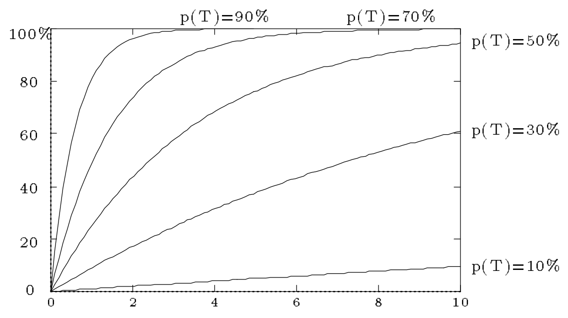
Let us consider the case of a two-way join and assume a uniform distribution of tuples-to-discount among all tuples in each table. Let $p^i_\mathcal{L}(F)$ and $p^i_\mathcal{R}(F)$ be the probabilities that a tuple in the left or right tables, respectively, have a value *false* for i-th unfoldings-to-discount to be closed by the join. The tuple is *not* eliminated by the join if it results from a join of two tuples, one each from the left and the right tables, such that the two tuples do not both have the value *true* for any tag column corresponding to an unfoldings-to-discount being closed. Hence, the probability that a tuple is not eliminated can be expressed as:

$$\Pi^n_{i=1}[p^i_\mathcal{R}(F) + (1 - p^i_\mathcal{R}(F))p^i_\mathcal{L}(F)]$$

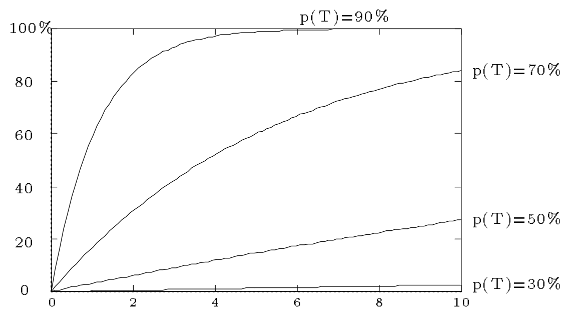
where n is the number of unfoldings-to-discount closed by the node. The probability that a tuple *is* eliminated (that is, the *proportion* of tuples that *are* eliminated) is then simply:

$$1 - \Pi^n_{i=1}[p^i_\mathcal{R}(F) + (1 - p^i_\mathcal{R}(F))p^i_\mathcal{L}(F)]$$

Consider again the example in Section 3. Assume that in $\mathbf{T}_{N_2}$ in Figure 3, 40% of the tuples



(a) 2-way join



(b) 5-way join

Figure 4: Potential answer set reduction with respect to the number of unfoldings-to-discount.

came from the *ta* table and 30% of the tuples came from the *staff* table. Similarly, assume that in $\mathbf{T}_{N_3}$ 20% of the tuples came from the *life_ins* table and 80% from the *health_plan* table. Assume independence of the tag columns' values, and that the set of tuples (with the tag columns projected out) marked *true* and the set marked *false* for any tag column are mostly disjoint. Given, $\mathcal{U}_1$=*ta*(*X*, *p_p*) $\bowtie$ *life_ins*(*X*, *hmo*,_) and $\mathcal{U}_2$= *staff*(*X*, *p_p*) $\bowtie$ *health_plan*(*X*, *hmo*,_), we know: $p^1_{\mathcal{L}}(F) = 0.6$, $p^2_{\mathcal{L}}(F) = 0.7$, $p^1_{\mathcal{R}}(F) = 0.8$ and $p^2_{\mathcal{R}}(F) = 0.2$. This means, the proportion of tuples eliminated from the final answer set is equal to $\approx$0.3, 30%.

The formula above can be generalized to an m-way join. Let $p^i_j(F)$ be the probability that a tuple in $\mathbf{T}_j$ is *false* in tag column $\mathbf{C}_i$. The formula then has the form:

$$1 - \Pi^n_{i=1} R^i_m$$

where R^i_m is defined recursively as:

- $R^i_1 = p^i_1(F)$
- $R^i_m = p^i_m(F) + (1 - p^i_m(F))R^i_{m-1}$

In the special case when $p^i_j(F) = p(F)$ $(= 1 - p(T))$, for $1 \leq j \leq m$, $1 \leq i \leq n$, this formula reduces to:

$$1 - [1 - p(T)^m]^n$$

This function is plotted in Figure 4 for 2-way and 5-way joins.

7 Conclusions

Intensional query optimization provides framework to apply effective semantic optimization such as semantic query caching [2, 3, 12], query folding [14, 18, 15], and semantic query optimization [1, 16, 20], to queries in realistic domains in which views may be complex and may use unions.

The concept of a *discounted* query—which is the query with some of its expansions (unfoldings) "removed"—enables this. This then leaves the question of how to evaluate efficiently the discounted query.

We have developed a bottom-up evaluation strategy (in three progressive versions) for discounted queries which, in general, is more efficient than the evaluation of the query itself. This optimization arises from two sources. First, the number of tuples in the discounted query's answer set is a subset of the query's. We show an analysis of how much smaller the discounted query's answer set may be with respect to the unfolding-to-discount. This translates to a cost savings whenever this answer set must be shipped across a network, as in a client-server environment, and an I/O savings for the RDBMS because fewer tuples are written during evaluation. Second, the cost of the join during the evaluation can be reduced by avoiding the consideration of tuples which would have resulted from an unfolding-to-discount. With this evaluation strategy for discounted queries, the IQO framework now provides a viable approach for optimization.

There are many further issues we must explore with respect to IQO framework. This includes exploring which types of interaction with syntactic optimizer would be most beneficial. Currently IQO is done in a prior step, and syntactic optimization occurs over the resulting queries. We also need to understand better the various cost trade-offs that can arise in IQO, and how best to balance them.

We are also exploring other pertinent applications of IQO. The framework allows for discounting queries for purposes other than just optimization. For instance, unfoldings may be discounted which would violate data security if answered. Thus, the (discounted) query still could be answered and database security maintained.

References

[1] U. 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.

[2] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proc. of the* 4th *International Conference on Extending Database Technology*, Cambridge, UK, 1994.

[3] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, Bombay, India, Sept. 1996. To appear.
URL: http://www.cs.umd.edu/projects/dimsum/papers/semantic_caching.ps.gz

[4] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992. Invited paper.

[5] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. Studies in Logic and Computation 3, chapter 1, pages 1–40. Clarendon Press, Oxford, 1994. Appears orginally as [4].
URL: http://karna.cs.umd.edu:3264/papers/GGM92:survey/GGM92:survey.html

[6] T. Gaasterland, P. Godfrey, J. Minker, and L. Novik. A cooperative answering system. In A. Voronkov, editor, *Proceedings of the Logic Programming and Automated Reasoning Conference*, Lecture Notes in Artificial Intelligence 624, pages 478–480. Springer-Verlag, St. Petersburg, Russia, July 1992.
URL: http://karna.cs.umd.edu:3264/papers/GGMN92:lpar/GGMN92:lpar.html

[7] A. Gal and J. Minker. Informative and cooperative answers in databases using integrity constraints. pages 277–300. North Holland, 1988.

[8] P. Godfrey and J. Gryz. A framework for intensional query optimization. In F. G. Dimitri Boulanger, Ultrich Geske and D. Seipel, editors, *Proceedings of the Workshop on Deductive Databases and Logic Programming, held in conjunction with the Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, GMD-Studien Nr. 295, pages 57–68, Bonn, Germany, Sept. 1996. GMD-Forschungszentrum.
URL: http://karna.cs.umd.edu:3264/papers/gg96:framework/paper.html

[9] P. Godfrey, J. Gryz, and J. Minker. Semantic query optimization for bottom-up evaluation. In Z. W. Raś and M. Michalewicz, editors, *Foundations of Intelligent Systems: Proceedings of the 9th International Symposium on Methodologies for Intelligent Systems*, Lecture Notes in Artificial Intelligence 1079, pages 561–571, Berlin, June 1996. Springer.
URL: http://karna.cs.umd.edu:3264/papers/GGM95:optimization/paper.html

[10] P. Godfrey, J. Minker, and L. Novik. An architecture for a cooperative database system. In W. Litwin and T. 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.
URL: http://karna.cs.umd.edu:3264/papers/GMN94:architecture/gmn94.html

[11] 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.

[12] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(2):35–47, Apr. 1996.
URL: ftp://ftp.cs.concordia.ca/pub/laks/papers/pods91.ps.gz

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

[14] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Princliples of Data Systems*. ACM, May 1995. To appear.
URL: ftp://twist.db.toronto.edu/pub/papers/pods95LMSS.ps.Z

[15] A. Y. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. 22nd VLDB*, 1996.

[16] A. Y. Levy and Y. Sagiv. Semantic query optimization in datalog programs. In *Proceedings of Principles of Database Systems*, 1995.

[17] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation—Artificial Intelligence. Springer-Verlag, Berlin, second edition, 1987.

[18] X. Qian. Query folding. In *Proceedings of the 12th International Conference on Data Engineering*, pages 48–55, 1996.
URL: http://www.csl.sri.com/∼qian/icde96.ps.Z

[19] T. Sellis and S. Shosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2), June 1990.

[20] 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, Sept. 1989.

[21] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I & II.* Principles of Computer Science Series. Computer Science Press, Incorporated, Rockville, Maryland, 1988.