Partial Evaluation of Views

Parke Godfrey Jarek Gryz

Department of Computer Science,
York University, Toronto, Canada
{godfrey,jarek}@cs.yorku.ca

Abstract

Many database applications and environments, such as mediation over heterogeneous database sources and data warehousing for decision support, lead to complex queries. Queries are often nested, defined over previously defined views, and may involve unions. There are good reasons why one might want to "remove" pieces (*sub-queries* or *sub-views*) from such queries: some sub-views of a query may be effectively cached from previous queries, or may be materialized views; some may be known to evaluate empty, by reasoning over the integrity constraints; and some may match protected queries, which for security cannot be evaluated for all users.

In this paper, we present a new evaluation strategy with respect to queries defined over views, which we call *tuple-tagging*, that allows for an efficient "removal" of sub-views from the query. Other approaches to this are to *rewrite* the query so the sub-views to be removed are effectively gone, then to evaluate the rewritten query. With the tuple tagging evaluation, no rewrite of the original query is necessary.

We describe formally a *discounted query* (a query with sub-views marked that are to be considered as removed), present the tuple tagging algorithm for evaluating discounted queries, provide an analysis of the algorithm's performance, and present some experimental results. These results strongly support the tuple-tagging algorithm both as an efficient means to effectively remove sub-views from a view query during evaluation, and as a viable optimization strategy for certain applications. The experiments also suggest that rewrite techniques for this may perform worse than the evaluation of the original query, and much worse than the tuple tagging approach.

1 Introduction

1.1 Motivation and Objectives

Many current database applications and environments, as mediation over heterogeneous database sources and data warehousing for decision support, incur complex queries. Queries are often nested, defined over previously defined views, and may involve unions. (A special type of such queries called *fusion queries*, which are self-joins of views defined over unions, was discussed in [25]). This is a necessity in mediation, as views in the meta-schema are defined to combine data from disparate sources. In these environments, view definition maintenance is of paramount importance.

There are many reasons why one might want to "remove" pieces (*sub-queries*) from a given query. Let us call a "sub-query" an *unfolding*, as the query can be *unfolded* via view definition into more specific sub-queries. These reasons include the following:

1. Some unfoldings of the query may be effectively cached from previous queries [5, 9], or may be materialized views [16] themselves.
2. Some unfoldings may be known to evaluate empty, by reasoning over the integrity constraints [1, 3].
3. Some unfoldings may match protected queries, which for security cannot be evaluated for all users [22].
4. Some unfoldings may be subsumed by previously asked queries, so are not of interest.
5. An unfolding shared by two queries in an **except** (difference) operation can be removed from both queries *before* the operation is carried out.

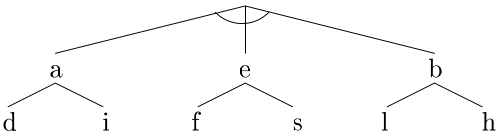


Figure 1: The AND/OR tree representation of the original query.

What does it mean to remove unfoldings from a query? The modified query should not subsume—and thus when evaluated should never evaluate—the removed unfoldings, but should subsume "everything" else of the original query.

In case 1, one might want to separate out certain unfoldings, because they can be evaluated much less expensively, and in a networked, distributed environment be evaluated locally. Then, a "remainder query" can be evaluated independently to find the remaining answers [5]. If the remainder query is less expensive to evaluate than the original, this is an optimization. In case 2, the unfoldings are free to evaluate, since it is known in advance that they must evaluate empty. In case 3, when some unfoldings are protected, it does not mean that the "rest" of the query cannot be safely evaluated. In case 4, when a user is asking a series of queries, he or she may just be interested in the stream of answers returned. So any previously seen answers are no longer of interest. In case 5, **except** queries might be optimized by this technique.

Consider the following example.

Example 1. Let there be six relations defined in the database **DB**: **Departments**(did, address), **Institutes**(did, address) **Faculty**(eid, did, rank), **Staff**(eid, did, position), **Health_Ins**(eid, premium, provider), and **Life_ins**(eid, premium, provider).

There are also three views defined in terms of these relations:

create view
Academic_Units as
(*select* did, address
from **Departments**)
union
(*select* did, address
from **Institutes**)

create view
Employees as
(*select* eid, did
from **Faculty**)
union
(*select* eid, did
from **Staff**)

create view
Benefits as
(*select* eid, premium, provider
from **Life_Ins**)
union
(*select* eid, premium, provider
from **Health_ins**)

Define the following query $\mathcal{Q}$:

$\mathcal{Q}$: *select* E.eid
from **Academic Units** A, **Employees** E, **Benefits** B
where A.did=E.did and E.eid=B.eid and B.provider="Blue Cross"

Query $\mathcal{Q}$ can be represented as a parse tree of its relational algebra representation, which is an AND/OR tree, as shown in Figure 1 (we ignore for brevity explicit representation of project and select operations). Evaluating the query—in the order of operations as specified in its relational algebra representation—is equivalent to materializing all nodes of the query tree. This type of evaluation (and representation) is referred to as *bottom-up*.

Now assume that the the following query $\mathcal{F}$ has been asked before and its result is stored in cache.

$\mathcal{F}$: *select* E.eid
from **Departments** D, **Faculty** F, **Life_Ins** L
where D.did=F.did and F.eid=L.eid and L.provider="Blue Cross"

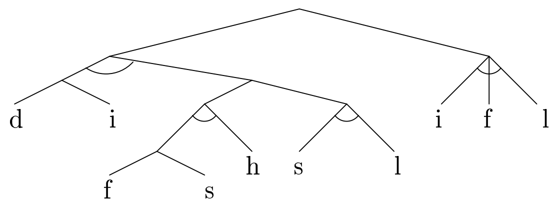


Figure 2: The AND/OR tree representation of the modified query.

Equivalently, we can assume that this formula represents a materialized view or that it matches a protected query whose answers should not be displayed. If $\mathcal{F}$ is a protected query, then the join expression it computes should not be evaluated. Thus, it has to be eliminated from the query. Otherwise, if $\mathcal{F}$ is a cached query or a materialized view, it may still be beneficial to "remove" it from the query. We call the result of this a *discounted query*.

One way to achieve this is to rewrite $\mathcal{Q}$ as a union of joins over base tables, then to remove the join represented by $\mathcal{F}$ (which then is explicitly present), and finally to evaluate the remaining join expressions. This may be very inefficient, however. The number of join expressions that remain to be evaluated may be exponential in the size of the collection of view definitions. Furthermore, we have shown in [11] that such an evaluation plan (which is often called *top-down*) may require evaluating the same joins many times, and incur the expense that a given tuple may be computed many times (whenever the projected parts of base tables overlap significantly). A top-down evaluation of $\mathcal{Q}$ from Figure 1 is the union of the eight following join expressions:

Departments ⋈ Faculty ⋈ Life_Ins	Departments ⋈ Faculty ⋈ Health_Ins
Departments ⋈ Staff ⋈ Life_Ins	Departments ⋈ Staff ⋈ Health_Ins
Institutes ⋈ Faculty ⋈ Life_Ins	Institutes ⋈ Faculty ⋈ Health_Ins
Institutes ⋈ Staff ⋈ Life_Ins	Institutes ⋈ Staff ⋈ Health_Ins

A seemingly more efficient evaluation plan for the discounted query can be devised by rewriting the query so that the number of operations (unions and joins) is minimized. (See Figure 2.) As a side effect of this approach, the redundancy in join evaluation, as well as redundancy in answer computation, is reduced. However, such redundancy is not entirely eliminated: for example, the join **Institutes** ⋈ **Faculty** is computed twice (implicitly in the left sub-tree and explicitly in the right sub-tree). One can verify that there is no rewrite of the query tree that removes the join **Departments** ⋈ **Faculty** ⋈ **Life_Ins** and yet guarantees at the same time that there is no redundancy in the computation of other joins. Thus, the "optimized" query can sometimes cost more to evaluate than the original query; our experimental results show (see Section 4) that this is, indeed, the case for this example.

In this paper, we present a new strategy for partial evaluation of queries defined over views, which we call *tuple-tagging*, which offers many advantages over explicit query rewrites.

- Tuple-tagging is easily implemented at the *logical level*, and can be accomplished by rewriting the SQL expression of the query.
- The technique is *modular*: it can be implemented independently of other optimization stages (in particular, of traditional relational database query optimization) to work in conjunction with the other stages.
- Tuple-tagging is *not* an algebraic rewrite of the query: it preserves the query tree *as is*, and thus scales up for complex queries.
- The tuple-tagging is interleaved with query evaluation. Thus, reliable heuristics can be devised and employed to decide step-wise whether a given "optimization" step should be applied.

The paper is organized as follows. Section 2 presents a formal framework for *discounted queries* (that section can be skipped on the first reading). Section 3 presents a tuple-tagging technique for evaluating discounted queries. Performance analysis of the technique, and experimental results over a TPC/D benchmark database in DB2, are presented in Section 4. Section 5 concludes with issues and future work.

1.2 Related Work

There is a substantial body of research in rewrite based query optimization [4, 6, 7, 19]. However, all of the techniques discussed in literature considered rewriting a query into a logically *equivalent* form. This is a different goal than the one we consider here: we are interested in generating and efficiently evaluating a query (view) from which an unfolding has been removed. Thus, the resulting query is *not* equivalent to the original query.

The problem we address is also different from the problem of answering queries using materialized views [2, 14, 16, 20]. In the latter problem, the goal is to *replace* subqueries of a query by views (or other queries) to generate a query equivalent or contained in the original query. Our goal, again, is to *remove*, or more precisely to *avoid evaluating*, parts of a query for optimization or security reasons.

The work that is most closely related to our work here is [15], in which the authors consider queries that involve nested union operations. They propose a technique for rewriting such queries, when it is known that some of the joins evaluated as part of the query are empty. The technique applies, however, only to a simple class of queries, and no complexity issues are addressed.

Another research area related to our problem is *multiple query optimization* [21]. The goal here is to optimize evaluation of a set of queries, rather than a single query. Since the queries in the set are arbitrary, they may not be related in any structured way to allow use of the techniques that we propose here. The developed techniques for multiple query optimization are focused towards finding and reusing common subexpressions across collections of queries and are heuristics-based.

The problem of query tree rewrites for the purpose of optimization has also been considered in the context of deductive databases with recursion. In [12], the problem of detecting and eliminating redundant subgoal occurrences in proof trees generated by programs in the presence of functional dependencies is discussed. In [13], the residue method of [1] is extended to recursive queries.

We investigated the computational complexity of query rewrites in [10]. We showed that the optimal rewrite of a query is NP-hard. We also identified a special class of queries and unfoldings for which rewrites result is a simpler query, thus always providing an optimization.

2 Discounted Queries

In this section, we formally define the notion of an *unfolding* and a *discounted query*. For preciseness, we use the notation of Datalog [24]. We will write a query as a set of atoms, to be interpreted as a conjunction of the atoms. For instance, $\{a, e, b\}$ represents the query of Example 1.[1] Some of the atoms may be intensional; that is, they are written with view predicates defined over base table predicates and, perhaps, other views.

Definition 2. 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 $\langle a \leftarrow b_1, \ldots, b_n . \rangle$ defining a view for a, such that $q_i\theta \equiv a\theta$ (for *most general unifier* θ [17]), then

$$\mathcal{U} = (\mathcal{Q} - \{q_i\} \cup \{b_1, \ldots, b_n\})\theta$$

[1] We present only "propositional" examples for simplicity's sake. They can be extended in a obvious way to queries with the variables made explicit.

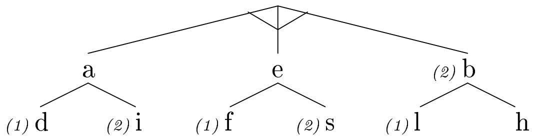


Figure 3: AND/OR tree of Example 1 with two unfoldings marked.

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 $\mathcal{U}_1 \leq^1 \ldots \leq^1 \mathcal{U}_k \leq^1 \mathcal{Q}$.

An unfolding $\mathcal{U}$ is called *extensional iff*, for every $q_i \in \mathcal{U}$, atom q_i refers to a base table. Call the unfolding *intensional* otherwise (in other words, some of the atoms refer to views).

Example 3. The views of Example 1 can be represented in simplified Datalog (where letters represent atoms) as:

$$\begin{array}{lll} a \leftarrow d. & e \leftarrow f. & b \leftarrow l. \\ a \leftarrow i. & e \leftarrow s. & b \leftarrow h. \end{array}$$

Since b in the query $\mathcal{Q}$ can be unfolded into l using a single rule $\langle b \leftarrow l. \rangle$, then $\{a, e, l\}$ is one of the 1-step unfoldings of $\mathcal{Q}$.

Since all of the atoms in unfolding $\mathcal{F}$ of $\mathcal{Q}$ (d, f, and l) are extensional, it is an extensional unfolding of $\mathcal{Q}$.

It is easy to see how an unfolding's AND/OR tree can be "inscribed" in the query's AND/OR tree. The atoms of an unfolding can be *marked* in the query's tree as shown in Figure 3 for unfoldings $\mathcal{F} = \{d, f, l\}$ and, say, $\mathcal{G} = \{i, s, b\}$.

A query is considered to be equivalent to the union of all its extensional unfoldings; define **unfolds** ($\mathcal{Q}$) to be the set of all such extensional unfoldings of $\mathcal{Q}$. We can now define the concept of a *discounted query*, which is to represent the query with some of its unfoldings "removed" (or *discounted*).

Definition 4. Given a query set $\mathcal{Q}$ and unfoldings $\mathcal{U}_1$,…,$\mathcal{U}_k$ of $\mathcal{Q}$, then the expression $\mathcal{Q}\backslash\{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$ is a *discounted query*. We define its meaning to be:

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

We call $\mathcal{U}_1$, …, $\mathcal{U}_k$ the *unfoldings-to-discount*, and the tuples in the answer sets of these unfoldings the *tuples-to-discount*.

Example 5. Consider again the query $\mathcal{Q}$ of Example 1. Since

$$\textbf{unfolds}(\mathcal{Q}) = \big\{\{d, f, l\}, \{d, f, h\}, \{d, s, l\}, \{d, s, h\}, \{i, f, l\}, \{i, f, h\}, \{i, s, l\}, \{i, s, h\}\big\}$$

and

$$\textbf{unfolds}(\mathcal{F}) = \big\{\{d, f, l\}\big\},$$

then

$$\textbf{unfolds}(\mathcal{Q}\backslash\{\mathcal{F}\}) = \big\{\{d, f, h\}, \{d, s, l\}, \{d, s, h\}, \{i, f, l\}, \{i, f, h\}, \{i, s, l\}, \{i, s, h\}\big\}.$$

Similarly, for the unfolding $\mathcal{G}$, $\{i, s, b\}$, in Figure 3, we have

$$\textbf{unfolds}(\mathcal{G}) = \big\{\{i,s,l\},\{i,s,h\}\big\}$$

and

$$\textbf{unfolds}(\mathcal{Q}\backslash\{\mathcal{G}\}) = \big\{\{d,f,h\},\{d,f,l\},\{d,s,l\},\{d,s,h\},\{i,f,l\},\{i,f,h\}\big\}.$$

Lastly,

$$\textbf{unfolds}(\mathcal{Q}\backslash\{\mathcal{F},\mathcal{G}\}) = \big\{\{d,f,h\},\{d,s,l\},\{d,s,h\},\{i,f,l\},\{i,f,h\}\big\}.$$

We discuss more formally the semantics of discounted queries in [8].

3 The Tuple-Tagging Evaluation Strategy

3.1 Overview

Our strategy is a bottom-up materialization strategy for the query tree with the union and join operations *modified* to account for the discounted unfoldings. The strategy ensures two things:

- that tuples resulting from an unfolding-to-discount do not contribute to the answer set of the discounted query; and
- any join represented by an unfolding-to-discount is never fully evaluated.

The tuples resulting from an unfolding-to-discount can be removed either during or after the actual query evaluation. To ensure the second property and thus to gain optimization, we need somehow to avoid evaluating the unfoldings-to-discount; that is, to prevent those tuples from being materialized *during* query evaluation.

Our proposed method, *tuple-tagging*, is 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 is *true* when that tuple is derived from the corresponding unfolding-to-discount; it is *false* otherwise. During each union or join operation (which creates a new temporary table), these tag columns' values must be maintained properly.

Example 6. Consider query $\mathcal{Q}$ of Example 1. Query $\mathcal{F}$, which represents the unfolding-to-discount, is a join of **Departments**, **Faculty** and **Life_Ins**. For each of these tables, a new column $\mathbf{C}_{\mathcal{F}}$ is added and its values initialized to *true*. Similarly, for tables **Institutes**, **Staff**, and **Health_Ins** which are unioned with the above tables, the same column is added and its values initialized to *false*.

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 also project away the tag columns). We shall be able further to use the tag columns—and ultimately satisfy our second property—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). The computation saved will primarily depend on the size of the *true* section in the table.

We present the evaluation strategy in two versions. The simpler version ensures only the first property; that is, the final answer set contains no tuples that arise solely from unfoldings-to-discount. The strategy is useful in the case when unfoldings are removed for security reasons. It does little, however, to optimize query evaluation: gross savings are equal to the difference in the cost of writing back the results of the original query versus the cost of writing back the results of the discounted query (which can sometimes be

% 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
            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 $\mathbf{T}_\mathcal{N}$.

Algorithm 1: The Modified Union Operation

significantly smaller). These savings can only be substantial when 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. Both versions of the algorithm require modifying the union and join operations. This is what we define next.

3.2 The Modified Union Operation

We assume that the query tree contains only union and join nodes (that is, all other operations are implicit). Furthermore, without loss of generality, we assume the the tree is binary. We refer to one child of any branch (non-leaf) node $\mathcal{N}$ in a binary query tree as $\mathcal{L}_\mathcal{N}$ (for *left* child), and the other as $\mathcal{R}_\mathcal{N}$ (for *right* child). We assume that any leaf $\mathcal{N}$ in the query tree has a corresponding table in the database; that is, the answer set for $\mathcal{N}$ is derivable from some table (perhaps temporary) in the database via selects and projections. Call $\mathcal{N}$'s table (with any selects and projections implicit) $\mathbf{T}_\mathcal{N}$.

To handle discounted queries, we modify the traditional algorithm for bottom-up query evaluation. This involves replacing the union and join operations with specialized versions, which handle and exploit the tag columns for the unfoldings-to-discount, as discussed above. Given discounted query $\mathcal{Q}\backslash\{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$, we introduce new columns, $\mathbf{C}_{\mathcal{U}_i}$, for $i \in \{1, \ldots, k\}$, as the tag columns corresponding to the unfoldings-to-discount (as described in Example 6).

We assume for any well-formed query tree that the tables to be unioned at any union step are union-compatible. With our addition of tag columns, this could 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 1 shows this. This is the only way in which we need to modify the union step.

Note that Algorithm 1 can be efficiently implemented in SQL by adding tag columns and initializing their values not before, but *during* the execution of the union operator. As we show in Section 4, this adds very little overhead to the cost of the query execution.

3.3 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*. Thus, tags are *conjunctively* combined.

Let $\mathcal{N}$ be a join node in a query tree for which the children are $\mathcal{L}_{\mathcal{N}}$ and $\mathcal{R}_{\mathcal{N}}$ and let $\mathcal{U}$ be an unfoldings-to-discount with the tag column $\mathbf{C}_{\mathcal{U}}$ in both tables $\mathbf{T}_{\mathcal{L}_{\mathcal{N}}}$ and $\mathbf{T}_{\mathcal{R}_{\mathcal{N}}}$. The modified join operation executed at node $\mathcal{N}$ is modified by adding the following assignment statement[2] for each unfolding-to-discount $\mathcal{U}$:

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

Example 7. Consider the final join of the query of Example 1. The three joined tables, **Academic Units**, **Employees**, **Benefits** will each contain an extra column, $\mathbf{C}_{\mathcal{F}}$, storing the values for the unfolding-to-discount $\mathcal{F}$. This column has been introduced during the execution of the union operations (as described in Algorithm 1). The query with the modified join operation is as follows.

select E.eid, (A.$\mathbf{C}_{\mathcal{F}}$ AND E.$\mathbf{C}_{\mathcal{F}}$ AND B.$\mathbf{C}_{\mathcal{F}}$) as $\mathbf{C}_{\mathcal{F}}$
from **Academic Units** A, **Employees** E, **Benefits** B
where A.did=E.did and E.eid=B.eid and B.provider="Blue Cross"

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 tuple-tagging algorithm in its first version then consists in using this information to select only the tuples that are known to be derivable from some unfolding other than an unfolding-to-discount. To ensure this, it is sufficient to select the tuples that have the value *false* for all their tag columns.

We show that the tuple-tagging algorithm (that is, the modified union and join operations) is correct. To prove it we need to show that all tuples, and only tuples, having the value *true* in a tag column for a given unfolding-to-discount in the final answer set of a query are the tuples from the answer set of that unfolding.

Theorem 8. Let $\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_n\}$ be a discounted query and let $[\![\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_n\}]\!]$ denote the answer set (table) representing the result of evaluating $\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_n\}$ using the tuple-tagging algorithm. Then, for any tuple $\tau \in [\![\mathcal{Q}\backslash\{\mathcal{U}_1,\ldots,\mathcal{U}_n\}]\!]$, $\tau.\mathbf{C}_{\mathcal{U}_i} = \mathit{true}$ iff $\tau \in [\![\mathcal{U}_i]\!]$, for $1 \leq i \leq n$.

Proof. Assume without loss of generality that all unfoldings-to-discount are extensional (an intensional unfolding can be always represented as a union of extensional unfoldings). Let unfolding $\mathcal{U}_i$ be a join $R_1 \bowtie \ldots \bowtie R_k$.

($\Rightarrow$)

Consider a tuple τ which is marked *true* in the column for unfolding $\mathcal{U}_i$, that is, $\tau.\mathbf{C}_{\mathcal{U}_i} = \mathit{true}$. Assume that $\tau \notin [\![\mathcal{U}_i]\!]$, that is, $\tau \notin [\![R_1 \bowtie \ldots \bowtie R_k]\!]$. But then, it cannot be that the initial value (set by the modified union operator) for $\mathbf{C}_{\mathcal{U}_i}$ is *true* for each of $R_1, \ldots, R_k$. Thus, if at least one of them is *false*, then the value of $\mathbf{C}_{\mathcal{U}_i}$ will be changed to false (and will remain such) sometime during the evaluation of joins. This contradicts our assumption, hence $\tau \in [\![\mathcal{U}_i]\!]$.

($\Leftarrow$)

[2]The **as** statement (as per the SQL'92 standards [18]) perform the requisite logical *ands* between tag columns and introduces the tag columns back into the new table.

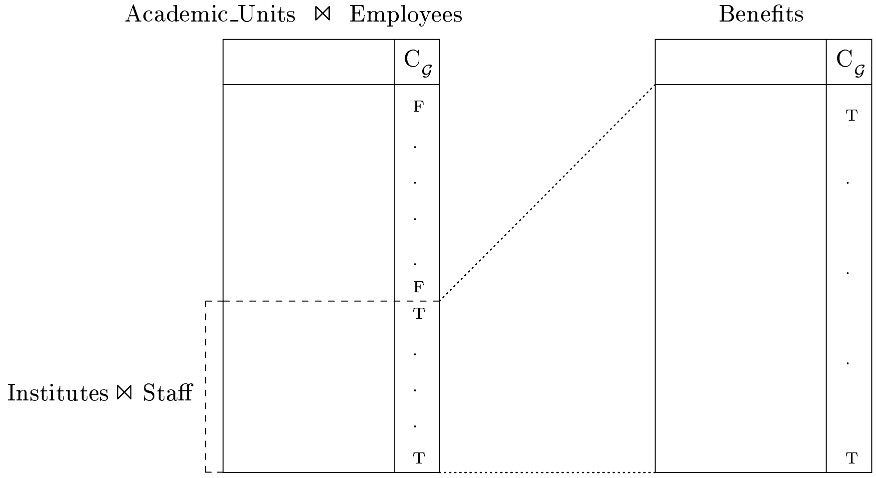


Figure 4: Final join for the query of Example 9.

Assume $\tau \in [\![\mathcal{U}_i]\!]$. Then, $\mathbf{C}_{\mathcal{U}_i}$ will be initialized to *true* for each of $R_1, \ldots, R_k$. Since the modified join operator assigns a conjunction of the values of $\mathbf{C}_{\mathcal{U}_i}$ from the join tables, $\mathbf{C}_{\mathcal{U}_i}$ will remain *true* for the duration of the evaluation process (note that the modified union operator never *changes* values for tag columns). Hence, $\tau.\mathbf{C}_{\mathcal{U}_i} = \mathit{true}$. □

3.4 Optimization

As stated in Section 3.1, removing tuples-to-discount from the final query answer set according to the optimization described above does not, in general, improve efficiency of query evaluation. 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. This constitutes the tuple-tagging algorithm in its second version. Consider the following example.

Example 9. Let the query be as in Example 1 and the unfolding-to-discount be $\mathcal{G} = \{i, s, b\}$ (marked as *(2)* in Figure 3). Thus, all tuples in the join **Intitutes** ⋈ **Staff** ⋈ **Benefits** should be removed from the query's answer set. Let us assume that the final join of the query **Academic_Units** ⋈ **Employees** ⋈ **Benefits** is executed as specified in the query tree (that is, left to right). Consider the result of evaluating **Academic_Units** ⋈ **Employees**. Some tuples in the result of that join will have the value *true* for the column $\mathbf{C}_\mathcal{G}$ (see Figure 4). Now, *all* tuples in the table for **Benefits** have the value *true* for that column (because the table is a part of the unfolding-to-discount $\mathcal{G}$). Thus, all of the tuples marked *true* in the result of the join **Academic_Units** ⋈ **Employees** will remain *true* after the join with **Benefits**, hence will be removed from the final query answer set.[3] If so, they can be eliminated as soon as **Academic_Units** ⋈ **Employees** is evaluated. Note that this provides optimization because the size of one of the tables in the input to the final join decreases. The gross savings achieved through this optimization can be estimated to the cost of the join of the result of **Institutes** ⋈ **Staff** with **Benefits** (marked with dotted lines in Figure 4).

We introduce the notion of a *closing* of an unfolding-to-discount by a node in a query tree.

Definition 10. Unfolding-to-discount $\mathcal{U}$ is *closed* by node $\mathcal{N}$ with respect to binary query tree **QT** if all tuple marked *true* in $\mathbf{T}_\mathcal{N}$ contribute to all and only tuples of $\mathcal{U}$.

[3]In the final tuple-tagging query plan, we would not even need to add a tag column $\mathbf{C}_\mathcal{G}$ to **Benefits** for this very reason.

Thus, the node representing the join of **Academic_Units** $\bowtie$ **Employees** is a closing node of unfolding-to-discount $\mathcal{G}$ in Example 9. Of course, the root of a query tree is a closing node for all unfoldings-to-discount.

Next, we prove a theorem that specifies a method of identifying closing nodes in a query tree.

Theorem 11. 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 join nodes only, is in the subtree rooted by $\mathcal{N}$.

Proof. 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 join 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 closed in $\mathcal{N}$ can be removed immediately after (or while) the table $\mathbf{T}_{\mathcal{N}}$ is materialized. □

Since there may be several closing nodes for a given unfolding, it is useful to identify the first one (in the sequence of operations specified by the query tree) in order to eliminate tuples-to-discount as soon as possible. Again, the condition for this property is simple. If $\mathcal{N}$ is a closing node for $\mathcal{U}$ and $\mathcal{U}$ does not have a closing node in the subtree rooted at $\mathcal{N}$, then $\mathcal{N}$ is the first of the closing nodes for $\mathcal{U}$.

The tuple-tagging algorithm can utilize (as described in Example 9) the existence of closing nodes not only to eliminate unfoldings-to-discount, but also to provide optimization. Our experimental results confirm that this is indeed the case.

4 Performance Analysis

4.1 Experiments

The purpose of our experiments is threefold:

1. to evaluate the overhead (in query evaluation time) introduced by the tuple-tagging algorithm through the modification of union and join operations;
2. to compare the performance of tuple-tagging versus query rewrite techniques in eliminating unfoldings-to-discount; and
3. to evaluate the optimization of query execution through tuple-tagging.

We used TPC/D benchmark database of size 100MB for our experiments (for details on this benchmark see [23]) installed on DB2 in Windows-NT. In order to be able to define views with unions, we modified slightly the TPC/D schema. Thus, the three tables **Supplier**, **Partsupp**, and **Lineitem**, which are base tables in the TPC/D schema, have been partitioned horizontally in half, and new base tables representing each of the sub-tables (**Supplier1** and **Supplier2**, and so forth) have been created. Then, views **Supplier-v**, **Partsupp-v**, and **Lineitem-v** have been defined as unions over **Supplier1** and **Supplier2**, **Partsupp1** and **Partsupp2**, and **Lineitem1** and **Lineitem2**, respectively. Thus each of the views has exactly the same content as the original tables in the TPC/D benchmark database.

We tested several versions of the following three queries.

$\mathcal{Q}_1$: *select* count(*)
from **Supplier-v S, Partsupp-v P**
where **S**.suppkey=**P**.suppkey

$\mathcal{Q}_1'$: *select* *
from **Supplier-v S, Partsupp-v P**
where **S**.suppkey=**P**.suppkey

$\mathcal{Q}_2$: *select* *

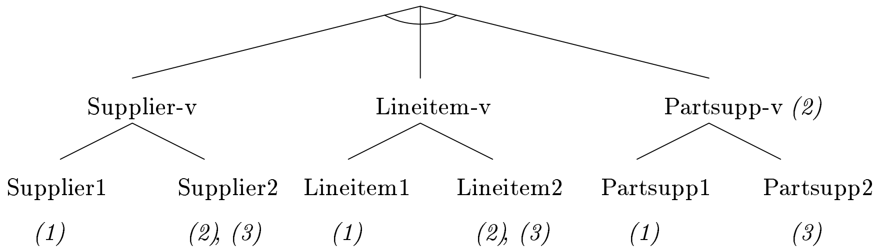


Figure 5: Query tree for $\mathcal{Q}_2$ with marked unfoldings-to-discount.

from **Supplier-v S**, **Partsupp-v P**,
Lineitem-v L
where **S**.suppkey=**P**.suppkey and
P.partkey=**L**.partkey and

For $\mathcal{Q}_1$ and $\mathcal{Q}_1^{'}$, we defined one unfolding-to-discount:

- $\mathcal{U}_0$, a join of **Supplier1** and **Partsupp1**.

For $\mathcal{Q}_2$, we defined the following three unfoldings-to-discount (marked in the query tree in Figure 5):

1. $\mathcal{U}_1$, a join of **Supplier1**, **Lineitem1**, and **Partsupp1**
2. $\mathcal{U}_2$, a join of **Supplier2**, **Lineitem2**, and **Partsupp-v**
3. $\mathcal{U}_3$, a join of **Supplier2**, **Lineitem2**, and **Partsupp2**

Then the following discounted queries were tested: $\mathcal{Q}_1\backslash\{\mathcal{U}_0\}$, $\mathcal{Q}_1^{'}\backslash\{\mathcal{U}_0\}$, $\mathcal{Q}_2\backslash\{\mathcal{U}_1\}$, $\mathcal{Q}_2\backslash\{\mathcal{U}_1,\mathcal{U}_3\}$, $\mathcal{Q}_2\backslash\{\mathcal{U}_2\}$.

Note that $\mathcal{Q}_2$ and $\mathcal{Q}_2\backslash\{\mathcal{U}_1\}$ have exactly the same structure as, respectively, $\mathcal{Q}$ and $\mathcal{Q}\backslash\{\mathcal{F}\}$ of Example 1. Similarly, $\mathcal{Q}_2\backslash\{\mathcal{U}_2\}$ has the same structure as $\mathcal{Q}\backslash\{\mathcal{G}\}$ of Example 9.

Since DB2 does not support boolean data types, we used integers (0 and 1) for tags, and we used multiplication instead of logical AND for their manipulation. For example, the query $\mathcal{Q}_2\backslash\{\mathcal{U}_1,\mathcal{U}_3\}$ evaluated under the tuple-tagging strategy has the following form:

select *
from **Supplier-v S**, **Partsupp-v P**, **Lineitem-v L**
where **S**.suppkey=**P**.suppkey and **P**.partkey=**L**.partkey and **P**.suppkey=**L**.suppkey
and **S**.tag1 * **P**.tag1 * **L**.tag1 = 0 and **S**.tag2 * **P**.tag2 * **L**.tag2 = 0;

Here *tag1* and *tag2* stand for tags for $\mathcal{U}_1$ and $\mathcal{U}_3$ respectively, and their values have been set up in the definitions of the views **Supplier-v**, **Partsupp-v**, and **Lineitem-v**.

Table 1 presents the results of tests measuring the evaluation cost of several queries. We used both tuple-tagging and explicit rewrites in evaluating the discounted queries. Thus, the suffixes *tag*, *rew*, and *top* mean that a discounted query was evaluated using respectively tuple-tagging, explicit rewrite of its query tree to minimize the number of operations (as described in Example 1), and the top-down approach (that is, by evaluating all joins of base tables). In particular, the query tree for $\mathcal{Q}_2\backslash\{\mathcal{U}_1\}^{rew}$ has the same structure as the query tree of Figure 2. Similarly, $\mathcal{Q}_2\backslash\{\mathcal{U}_1,\mathcal{U}_3\}^{rew}$ has the following form:[4]

$$\prod_{All\,Attributes} [(\textbf{Supplier1} \cup \textbf{Supplier2}) \bowtie (\textbf{Lineitem1} \bowtie \textbf{Partsupp2} \cup \textbf{Lineitem2} \bowtie \textbf{Partsupp1})] \cup \textbf{Supplier1} \bowtie \textbf{Lineitem2} \bowtie \textbf{Partsupp2}$$

[4]We express this in relational algebra only for brevity.

Query	Total Execution Time (s)	Number of Retrieved Rows
$\mathcal{Q}_1$	3.485	1
$\mathcal{Q}_1 \backslash \{\mathcal{U}_0\}^{tag}$	3.495	1
$\mathcal{Q}_1^{'}$	25.6	9600
$\mathcal{Q}_1^{'} \backslash \{\mathcal{U}_0\}^{tag}$	17.5	7242
$\mathcal{Q}_2$	402.25	73943
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1\}^{tag}$	364.38	65219
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1\}^{rew}$	582.82	
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1\}^{top}$	788.68	
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}^{tag}$	356.18	55537
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}^{rew}$	482.65	
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}^{top}$	653.23	
$\mathcal{Q}_2 \backslash \{\mathcal{U}_2\}^{tag}$	298.88	56002

Table 1: Experimental Results

$\cup$ **Supplier2** $\bowtie$ **Lineitem1** $\bowtie$ **Partsupp1**

One can easily verify that this is the most compact representation of $\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}$. Recall that $\mathcal{Q}_2 \backslash \{\mathcal{U}_1\}^{top}$ was evaluated as a union of all its extensional unfoldings—that is, a union over the joins of base tables—and similarly for $\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}^{top}$.

4.2 Discussion

The purpose of testing $\mathcal{Q}_1$ and $\mathcal{Q}_1 \backslash \{\mathcal{U}_0\}$ is to measure the overhead of adding and manipulating tags. The query is designed in such a way that no optimization in query execution time can possibly be achieved by using tags. This is done by making the root the closing node for $\mathcal{U}_0$ in $\mathcal{Q}_1 \backslash \{\mathcal{U}_0\}$ (so that the joined tables are of identical sizes as in $\mathcal{Q}$) and minimizing the size of the answer set (thus making sure that there is no benefit in writing less data back to a disk). Indeed, query execution time for $\mathcal{Q}_1 \backslash \{\mathcal{U}_0\}$ is 3.495 seconds which is larger than that for $\mathcal{Q}_1$, which is 3.485 seconds. The good news is that the difference is negligible: 0.29% in the case of $\mathcal{Q}_1$ versus $\mathcal{Q}_1 \backslash \{\mathcal{U}_0\}$. Once the size of the answer becomes substantial, tuple-tagging begins to optimize. Queries $\mathcal{Q}_1^{'}$ and $\mathcal{Q}_1^{'} \backslash \{\mathcal{U}_0\}$ project all attributes of the two joined tables. By reducing the size of the answer set, $\mathcal{Q}_1^{'} \backslash \{\mathcal{U}_0\}$ provides optimization in query execution time over $\mathcal{Q}_1^{'}$: 17.5 seconds for $\mathcal{Q}_1^{'} \backslash \{\mathcal{U}_0\}$ vs. 25.6 seconds for $\mathcal{Q}_1^{'}$.

The purpose of queries $\mathcal{Q}_2$, $\mathcal{Q}_2 \backslash \{\mathcal{U}_1\}$, and $\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}$ is to compare tuple-tagging with two other techniques for removing unfolding from a query: top-down query evaluation; and an explicit rewrite of a query tree to minimize algebraic form. Tuple-tagging outperforms the other two techniques by a respectable margin. In fact, both the top-down approach and the explicit query rewrite approach add substantial overhead to the cost of evaluating discounted queries (see Table 2). As we conjectured in Section 1, this is due to the introduction of redundancy in join evaluation for both techniques. On the other hand, both $\mathcal{Q}_2 \backslash \{\mathcal{U}_1\}$ and $\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}$ evaluated under the tuple-tagging strategy provide modest optimization over $\mathcal{Q}_2$ (see Figure 6). This is still achieved only through the reduction of the size of the answer set, and not through reduction of the sizes of the joined tables.

The last query, $\mathcal{Q}_2 \backslash \{\mathcal{U}_2\}$, provides a crucial test for our technique. The temporary table created through the join **Supplier-v** $\bowtie$ **Lineitem-v** is a closing node for unfolding $\mathcal{U}_2$. This means, that all tuples marked as *true* in that table can be eliminated before the next join (with **Partsupp-v**) is executed, thus reducing

Query	Evaluation Strategy		
	Tuple-Tagging	Top-Down	Explicit Rewrite
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1\}$	364.38	582.82	788.68
$\mathcal{Q}_2 \backslash \{\mathcal{U}_1, \mathcal{U}_3\}$	356.18	482.65	653.23
$\mathcal{Q}_2$	402.25 (standard evaluation)		

Table 2: Evaluation time (in seconds) for discounted queries under different evaluation strategies.

the cost of that last join. Indeed, the reduction of the execution cost in $\mathcal{Q}_2 \backslash \{\mathcal{U}_2\}$ over the original query $\mathcal{Q}_2$ is 25.7%.

4.3 Heuristics

As stated in Section 1, one of the advantages of tuple-tagging over explicit query tree rewrites is that reliable heuristics can be devised to decide step-wise whether a given "optimization" step should be applied. Such a decision will depend on the reduction of the size of a table representing a closing node for some unfolding (or a set of unfoldings) in the query tree. For example, in query $\mathcal{Q}_2 \backslash \{\mathcal{U}_2\}$ half of the tuples in **Supplier-v** and half of the tuples in **Lineitem-v** are marked *true*. Hence, the proportion of the tuples marked *true* in the join **Supplier-v** $\bowtie$ **Lineitem-v** (assuming a uniform distribution of values) is estimated to be 25%. Once these tuples are eliminated, the cost of the join with **Partsupp-v** is roughly estimated to be reduced by 25% as well. We show formally how such a reduction can be estimated in general, thus providing a main component for heuristics that can be used with the tuple-tagging algorithm.

The proportion of tuples-to-discount versus all of the tuples, p, in a table representing a closing node $\mathcal{N}$ depends on three factors: the **proportion** of tuples tagged with *true* for some unfolding-to-discount closed by $\mathcal{N}$; the **number** of unfoldings-to-discount closed by $\mathcal{N}$; and the **size** of the join (how many tables participate). We state this dependence formally as follows. (We assume a uniform distribution of tuples-to-discount among all tuples in each table.)

Theorem 12. Let table $\mathbf{T}_0$ be created through a join of tables $\mathbf{T}_1$,...,$\mathbf{T}_m$ each with n different tag columns $\mathbf{C}_1$,..., $\mathbf{C}_n$. Let $p_j^i(T)$ be the proportion of tuples in the table $\mathbf{T}_j$ with the value *true* in the column $\mathbf{C}_i$. Then, p, the proportion of tuples in $\mathbf{T}_0$ for which all tag columns have the value *true*, is:

$$p = 1 - \Pi_{i=1}^{n}[1 - \Pi_{j=1}^{m}\ p_j^i(T)]$$

Proof. Induction on n.

1. $n = 1$.
   If there is only one unfolding-to-discount, then a tuple τ is discounted only if it was created by joining m tuples $\tau_1, ...\tau_m$ from the tables participating in a join, such that each tuple τ_i $1 \leq i \leq m$ had the value **true** in column $\mathbf{C}_{\mathcal{U}}$. The probability of this is $\Pi_{j=1}^{m}\ p_j(T)$, which is the value of the above formula for $n = 1$.
2. Assume that the formula holds for n unfoldings-to-discount. We show it holds for $n + 1$.
   A tuple is discounted when $n + 1$ unfoldings-to-discount are considered in one of the following cases:
   a. It would have been discounted when only n unfoldings-to-discount were considered.
   b. It is discounted because it is in the answer set of the $n + 1$-st unfolding-to-discount.

   The probability of 2a is given by the induction hypothesis as: $p(disc) = 1 - \Pi_{i=1}^{n}[1 - \Pi_{j=1}^{m}\ p_j^i(T)]$.

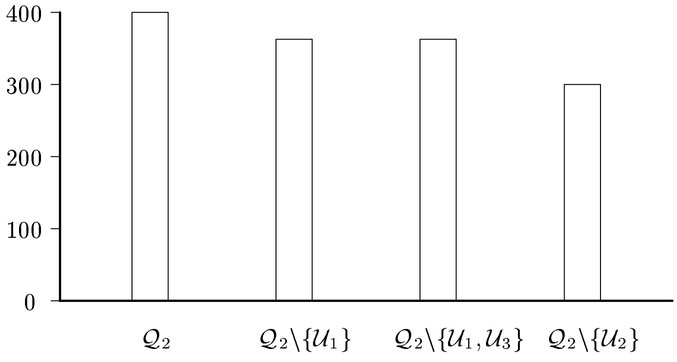


Figure 6: Comparison of execution times (s) of the original query and queries evaluated with tuple-tagging.

The probability of 2b is a conditional probability that the tuple has not been discounted when only n unfoldings-to-discount were considered, but is discounted when the $n+1$-st unfolding-to-discount is considered. This probability can be expressed as: $\Pi_{i=1}^{n}[1-\Pi_{j=1}^{m}\ p_j^i(T)]*\Pi_{j=1}^{m}\ p_j^{n+1}(T)$. Thus, the probability that a tuple is discounted when $n+1$ unfoldings-to-discount are considered is equal to the sum of the probabilitues of cases 2a and 2b:

$$\begin{aligned} p(\text{disc}) &= 1-\Pi_{i=1}^{n}[1-\Pi_{j=1}^{m}\ p_j^i(T)]+\Pi_{i=1}^{n}[1-\Pi_{j=1}^{m}\ p_j^i(T)]*\Pi_{j=1}^{m}\ p_j^{n+1}(T) \\ &= 1-\Pi_{i=1}^{n}[1-\Pi_{j=1}^{m}\ p_j^i(T)](1-\Pi_{j=1}^{m}p_j^{n+1}(T)) \\ &= 1-\Pi_{i=1}^{n+1}[1-\Pi_{j=1}^{m}\ p_j^i(T)] \end{aligned} \qquad \square$$

In the special case when the proportion of tuples marked *true* in all tables and all tag columns is identical and equal to some value $P(T)$ (that is, $p_j^i(T)=p(T)$, for $1\leq j\leq m,\ 1\leq i\leq n$), this formula reduces to:

$$p = 1-[1-p(T)^m]^n \qquad \mathbf{1)}$$

We show the application of this result to predict the size of tables for closing nodes in our experiments.

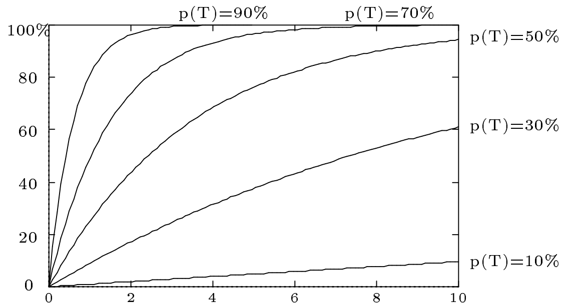
Example 13. For queries $\mathcal{Q}_2\backslash\{\mathcal{U}_1\}$ and $\mathcal{Q}_2\backslash\{\mathcal{U}_1,\mathcal{U}_3\}$, the root of the query tree is the closing node. The proportion of tuples marked *true* for both $\mathcal{U}_1$ and $\mathcal{U}_3$ is 0.5 for all three tables: **Supplier-v**, **Partsupp-v**, and **Lineitem-v**. Thus, for query $\mathcal{Q}_2\backslash\{\mathcal{U}_3\}$, Formula **1)** has the form $p=1-[1-0.5^3]=0.125$. Since the size of the answer of the original query $\mathcal{Q}$ was 73943 rows, reducing it by 12.5% would produce a table of size 64700. This is very close to the actual size of 65219 rows observed for $\mathcal{Q}_2\backslash\{\mathcal{U}_3\}$.

Similarly, for query $\mathcal{Q}_2\backslash\{\mathcal{U}_1,\mathcal{U}_3\}$, Formula **1)** has the form $p=1-[1-0.5^3]^2=0.234$. The predicted size of the final query answer set is thus 56566, which again is close to the actual size of 55537 for $\mathcal{Q}_2\backslash\{\mathcal{U}_1,\mathcal{U}_3\}$.

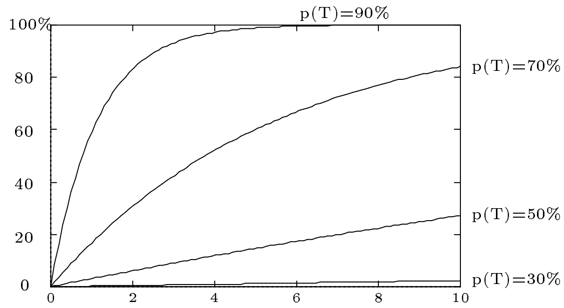
The values for Formula **1)** (p as a function of n) is plotted in Figure 7 for 2-way and 5-way joins with several values of $p(T)$. Not surprisingly, the number of discounted tuples grows with the number of unfoldings-to-discount as well as the proportion of the tuples-to-discount in the tables participating in the join. There is also, however, a strong *inverse* relationship between the number of joins involved in a multiway join and the number of discounted tuples. Even if each of the tables participating in the join has 30% tuples-to-discount and 10 different unfolding are removed, the number of discounted tuples is very small for a 5-way join.

5 Conclusions

In this paper, we introduced a new framework in which a query is represented as a collection of selected unfoldings of the query and a *discounted query*, which represents the query with those unfoldings "removed".



(a) 2-way join



(b) 5-way join

Figure 7: Proportion of discounted tuples in the result of a join with respect to the number of unfoldings-to-discount.

The selected unfoldings may be removed for security reasons, or because their answers are readily available (through caching or materialized views). We presented an efficient evaluation strategy for discounted queries called tuple-tagging. We showed through experiments that a discounted query can be, in general, evaluated more efficiently than the query itself. The experiments also suggested that rewrite techniques, which seem to be an intuitive approach to removing unfoldings from a query, may perform worse than the evaluation of the original query, and much worse than the tuple tagging approach. Thus the discounting framework and the tuple-tagging algorithm offer a viable approach to optimization of queries which employ views.

There are numerous issues to explore with respect to optimization of queries over views. This type of optimization is orthogonal to other optimization techniques, and so can be directly used in conjunction with existing optimizers. It would be beneficial to identify the types of interaction with the traditional query optimizer that could increase overall optimization. Currently tuple-tagging is done in a prior stage, and optimization is applied over the resulting queries. We also need to understand better the various cost trade-offs in tuple-tagging, and how best to balance them.

References

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

[2] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th ICDE*, pages 190–200, 1995.

[3] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *Proceedings of the 25th VLDB*, Edinburgh, Scotland, 1999.

[4] M. Cherniack and S. Zdonik. Rule languages and internal algebras for rule_based optimizers. In *Proc. SIGMOD*, pages 401–412, 1996.

[5] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of 22nd VLDB*, pages 330–341, 1996.

[6] D. Das and D. Batory. Prairie: A rule specification framework for query optimizers. In *Proceedings of ICDE*, pages 201–210, 1995.

[7] J. Freytag. A rule-based view of query optimization. In *SIGMOD Proceedings*, pages 173–180. ACM, 1987.

[8] P. Godfrey and J. Gryz. A framework for intensional query optimization. In *Proceedings of DDLP'96*, pages 57–68, Bonn, Germany, Sept. 1996.

[9] P. Godfrey and J. Gryz. Answering queries by semantic caches. In *Proceedings of 10th DEXA*, Florence, Italy, Aug. 1999.

[10] P. Godfrey and J. Gryz. View disassembly. In *Proceedings of 7th ICDT*, pages 417–434, Jerusalem, Israel, Jan. 1999.

[11] P. Godfrey, J. Gryz, and J. Minker. Semantic query optimization for bottom-up evaluation. In Z. Ras and M. Michalewicz, editors, *Proc. of the 9th. ISMIS*, pages 561–571, Zakopane, Poland, June 1996.

[12] L. V. S. Lakshmanan and H. J. Hernandez. Structural query optimization - a uniform framework for semantic query optimization in deductive databases. In *Proc. PODS*, pages 102–114, 1991.

[13] L. V. S. Lakshmanan and R. Missaoui. Pushing semantics inside recursion: A general framework for semantic optimization of recursive queries. In *Proc. ICDE*, pages 211–220, 1995.

[14] P.-A. Larson and H. Yang. Computing queries from derived relations. In *Proc. of 11th VLDB*, pages 259–269, 1985.

[15] S. Lee, L.J.Henschen, and G. Qadah. Semantic query reformulation in deductive databases. In *Proc. ICDE*, pages 232–239, 1991.

[16] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS*, pages 95–104, 1995.

[17] J. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, second edition, 1987.

[18] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Mateo, California, 1993.

[19] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. SIGMOD*, pages 39–48, 1992.

[20] X. Qian. Query folding. In *Proceedings of the* 12^{th} *ICDE*, pages 48–55, 1996.

[21] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *TKDE*, 2(2):262–266, June 1990.

[22] B. Thuraisingham and W. Ford. Security constraint processing in a multilevel secure distributed database management system. *TKDE*, 7(2):274–293, Apr. 1995.

[23] Transaction Processing Performance Council, 777 No. First Street, Suite 600, San Jose, CA 95112-6311, www.tpc.org. *TPC Benchmark*^{TM} *D*, 1.3.1 edition, Feb. 1998.

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

[25] R. Yernani, Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Fusion queries over internet databases. In *Proceedings of the 6th EDBT*, pages 57–71, 1998.