Available online at www.sciencedirect.com

ScienceDirect

Journal of Computer and System Sciences 73 (2007) 941–961

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

www.elsevier.com/locate/jcss

View disassembly: A rewrite that extracts portions of views

Parke Godfrey, Jarek Gryz *

York University, Toronto ON M3J 1P3, Canada

Received 26 February 2001; received in revised form 27 April 2006

Available online 12 March 2007

Abstract

We explore a new form of view rewrite called *view disassembly*. The objective is to rewrite views in order to "remove" certain sub-views (or *unfoldings*) of the view. This becomes pertinent for complex views which may be defined over other views and which may involve union. Such complex views arise necessarily in environments such as data warehousing and mediation over heterogeneous databases. View disassembly can be used for view and query optimization, preserving data security, making use of cached queries and materialized views, and view maintenance.

We provide computational complexity results of view disassembly. One question is whether the unfoldings to be removed effectively *cover* the view, meaning that the disassembled view is effectively the empty view, evaluating to the empty table. We illustrate the complexity to determine when a collection of unfoldings cover the view definition. The problem is **NP**-*hard* with respect to the number of unfoldings to remove, but not with respect to the size (complexity) of the view definition. We next consider rewrites optimal in the size of the rewritten (disassembled) view. We prove that this task is **NP**-*hard* for a special class of views, but this time **NP**-*hard* in a worse way: it is **NP**-*hard* this time with respect to the size of the view definition in addition to the number of unfoldings to be removed. In general, we suspect the problem is computationally even harder, and we show that the general problem is in Π_2^p.

However, we provide good news too. We identify a pertinent class of unfoldings for which the removal of such unfoldings always results in a simpler disassembled view than the original view itself. We also develop an algorithm that finds rewrites equivalent to the disassembled view which are, in a sense, *locally* optimal. The algorithm establishes a *cover completion* by finding a (minimal) collection of unfoldings of the view that, along with the unfoldings to be removed, covers the original view. This approach is effectively tractable, unlike the search for globally, or absolutely, optimal rewrites. Furthermore, we show that these cover-completion rewrites are preferable to absolutely optimal rewrites in many ways.

Keywords: Views; Query rewriting; Rewrite complexity; Query optimization

1. Introduction

Many database applications and environments, such as mediation over heterogeneous database sources and data warehousing for decision support, lead to complex view definitions. Views are often nested, defined over previously defined views, and may involve unions. The union operator is a necessity in mediation, as views in the meta-schema are

* Corresponding author.
E-mail addresses: godfrey@cs.yorku.ca (P. Godfrey), jarek@cs.yorku.ca (J. Gryz).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2007.03.002

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" components, or *sub-views*, from a given view, or from a query which involves views.[1] Let us call a sub-view an *unfolding* of the view, as the view can be *unfolded* via its definition into more specific sub-views. These reasons include the following.

1. Some unfoldings of the view may be effectively cached from previous queries [2], or may be materialized views [16].
2. Some unfoldings may be known to evaluate empty, by reasoning over the integrity constraints [1] or rules discovered via data mining [3].
3. Some unfoldings may match protected queries, which, for security, cannot be evaluated for all users [18].
4. Some unfoldings may be subsumed by previously asked queries, so are not of interest to the user.

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

In case 1, one might want to separate out certain unfoldings, because they can be evaluated much less expensively (and perhaps, in a networked, distributed environment, be evaluated locally). Then, the "remainder query" could be evaluated separately [2]. There is a large body of research devoted to the question of when one query (unfolding) can be answered by means of another query [12]. In case 2, the unfoldings are free to evaluate, since it is known in advance that they must evaluate empty. If the remainder query is less expensive to evaluate than the original, this is an optimization. In case 3, when some unfoldings are protected, this does not mean that the "rest" of the query or view 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 returning. So any previously seen answers are no longer of interest. In environments in which there are monetary charges for information, there is an additional advantage of not having to pay repeatedly for the same information.

In this paper, we address this problem of how to remove efficiently and correctly sub-views from views. We call this problem *view disassembly*. We present the computational complexities of, and potential algorithmic approaches to, view disassembly tasks. On first consideration, it may seem that the view disassembly problem is trivial, that a view could always be "trimmed" to exclude any given sub-view. On further consideration, however, one quickly sees that this is not true. To remove a sub-view, or especially a collection of sub-views, can be a quite complex task. Certain unfolding removals are, in fact, simple: the view's definition can be trimmed to exclude them. We call these unfoldings *simple*, and we characterize these in this paper. In the general case, however, unfoldings require that the view's definition be *rewritten* to effectively remove them. We are interested in *compact* rewrites that accomplish this.

The paper proceeds as follows.

– In Section 2, we present and motivate the view disassembly problem.
– In Section 3, we discuss related work.
– In Section 4, we profile the general complexity of the problem.
– In Section 5, we identify an important class of unfoldings, to be called *simple unfoldings*, which can be removed from a view without increasing the view's complexity.
– In Section 6, we define globally optimal rewrites for disassembly. By this, we mean a view that is the smallest algebraically of all possible rewrites that effectively "remove" the specified unfoldings from the original query. We prove, however, that finding globally optimal rewrites of this kind is intractable. It is **NP**-*hard* for a special case, and we show that the set-theoretic problem is in Π_2^p generally.
– In Section 7, we define locally optimal rewrites for disassembly. Such rewrites preserve in a fundamental sense the structure of the original view. Locally optimal disassembly rewrites are locally optimal insofar as the rewritten view cannot be transformed further to a yet smaller view and still represent the disassembled view. We show that locally optimal rewrites are often preferable to globally optimal rewrites for a number of reasons, even though

[1] In this paper, we use *view* and *query* synonymously.

they may not be as compact. We present an algorithm for disassembly that produces locally optimal rewrites. Combined with the complexity results established in Section 4, we demonstrate that (locally optimal) disassembly is effectively tractable.
– In Section 8, we outline and discuss further issues with view disassembly, and show directions for future work.
– In Section 9, we make concluding remarks.

2. Motivating example

We represent queries and views in Datalog. Thus, we consider databases under the *logic model* [19]. For this paper, we do not consider recursion or negation. A database **DB** is considered to consist of two parts: the *extensional database* ($\mathbf{E_{DB}}$), a set of atomic facts; and the *intensional database* ($\mathbf{I_{DB}}$), a set of clausal rules. Predicates are designated as either *extensional* or *intensional*. Extensional predicates are defined solely via the facts in the $\mathbf{E_{DB}}$. Intensional predicates are defined solely via the rules in the $\mathbf{I_{DB}}$. Thus, extensional predicates are equivalent to base relations defined by relational *tables* in the relational database realm, and intensional predicates to relational *views*. We shall employ the term *view* to refer to any intensional predicate or any query that uses intensional predicates.

In [8], we called the notion of a view or query with some of its unfoldings (sub-views) "removed" a *discounted query* or *view*, and called the "removed" unfoldings the *unfoldings-to-discount*. A view (or query) can be represented as an AND/OR tree that represents the expansion of its definition via the rules in the $\mathbf{I_{DB}}$. In view disassembly, we consider algebraic rewrites of the view's corresponding AND/OR tree to find AND/OR trees that properly represent the discounted view. Consider the following example, which we shall discuss throughout this section.

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)
– **Life_ins** (eid, premium, provider)

Let there also be three views defined in terms of these relations:

$$\mathit{academic\_units}(X, Y) \leftarrow \mathit{departments}(X, Y).$$
$$\mathit{academic\_units}(X, Y) \leftarrow \mathit{institutes}(X, Y).$$
$$\mathit{employees}(X, Y) \leftarrow \mathit{faculty}(X, Y, Z).$$
$$\mathit{employees}(X, Y) \leftarrow \mathit{staff}(X, Y, Z).$$
$$\mathit{benefits}(X, Y, Z) \leftarrow \mathit{health\_ins}(X, Y, Z).$$
$$\mathit{benefits}(X, Y, Z) \leftarrow \mathit{life\_ins}(X, Y, Z).$$

Define the following query $\mathcal{Q}$ that asks for addresses of all academic units with any employees receiving benefits from *Ætna*:[2]

$$\mathcal{Q}: \quad q(Y) \leftarrow a(X, Y), e(Z, X), b(Z, W, \mathit{aetna}).$$

Query $\mathcal{Q}$ from Example 1 can be represented as a parse tree of its relational algebra representation, which is an AND/OR tree, as shown in Fig. 1. Evaluating the query—in the order of operations as indicated by its relational

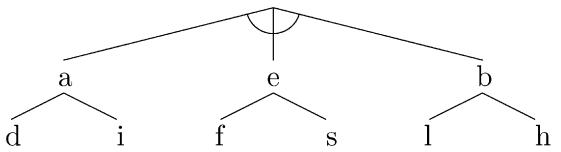

Fig. 1. The AND/OR tree representation of the original query.

[2] We henceforth abbreviate the predicate names.

algebra representation—is equivalent to materializing the internal nodes of the query's AND/OR tree. We refer to this type of evaluation (and representation) as *bottom-up*. Consider the following three scenarios:

Case 1. Assume that the answers of the following three queries $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ have been cached. Equivalently, we could assume that these represent materialized views, or that they are known to evaluate empty (say, by reasoning over integrity constraints). Let f_1, f_2, and f_3 be the corresponding cached predicates.

$\mathcal{F}_1$: $f_1(Y) \leftarrow a(X, Y), f(Z, X, V), b(Z, W, \text{ætna})$.
$\mathcal{F}_2$: $f_2(Y) \leftarrow d(X, Y), e(Z, X), b(Z, W, \text{ætna})$.
$\mathcal{F}_3$: $f_3(Y) \leftarrow i(X, Y), s(Z, X, V), b(Z, W, \text{ætna})$.

$\mathcal{Q}$ does not have to be evaluated, since its answer set is equal to the union of answer sets of cached queries.[3] We say that queries $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ *cover* the query $\mathcal{Q}$.

Case 2. Consider just query $\mathcal{F}_2$ from *Case* 1 again, and assume that it has been cached. As $\mathcal{F}_2$ provides a subset of the answer set to $\mathcal{Q}$, we can rewrite $\mathcal{Q}$ as $\mathcal{Q}'$ to retrieve only the remaining answers:

$\mathcal{Q}'$: $q'(Y) \leftarrow i(X, Y), e(Z, X), b(Z, W, \text{ætna})$.

Note that unless special tools are available to evaluate efficiently the join of **Academic_Units** $\bowtie$ **Employees**, the rewrite of $\mathcal{Q}$ to $\mathcal{Q}'$ provides an optimization. This is because the rewrite amounts to a removal of one of the leaves of the query tree resulting in a simpler query. We call unfoldings that lead to such rewrites *simple* (to be formally defined in Section 5).

Case 3. Assume that the following query $\mathcal{F}_4$ is cached.

$\mathcal{F}_4$: $f_4(Y) \leftarrow d(X, Y), f(Z, X, V), l(Z, W, \text{ætna})$.

Again, we may want to "remove" $\mathcal{F}_4$ from the rest of the query, as its answers are locally available. One way to do this is to rewrite $\mathcal{Q}$ as a union of join expressions over base tables, to remove the join expression represented by $\mathcal{F}_4$, and then to evaluate the remaining join expressions. We call this type of evaluation *top–down*. A top–down evaluation of $\mathcal{Q}$ from Fig. 1 is the union of the eight join expressions from

$$\{d, i\} \times \{f, s\} \times \{h, l\}.$$

After the unfolding $\mathcal{F}_4$ is removed, the following join expressions need to be evaluated:

$$d \bowtie f \bowtie h, \quad d \bowtie s \bowtie l, \quad i \bowtie f \bowtie l, \quad i \bowtie f \bowtie h,$$
$$d \bowtie s \bowtie h, \quad i \bowtie s \bowtie l, \quad i \bowtie s \bowtie h.$$

Top–down query evaluation is impractical in most cases. There are several problems:

1. The number of extensional unfoldings (that is, unfoldings with extensional predicates only) for a given query can be exponential in the size of the rule base. This has not been problematic so far since the number and complexity of rules (views) in most databases has been small. However, security issues and the necessity of providing flexible access to databases for different groups of users will require creating a large number of more complex views.[4] Moreover, the introduction of heterogeneous database systems and mediators (middleware) may require creating yet another layer of views increasing the size of the rule base to the point where top–down query evaluation becomes unmanageable.
2. Many of the same joins may appear in different extensional unfoldings. If each of the intensional predicates in a database is defined by more than one rule, then for any extensional unfoldings of a query, there is another unfolding which differs from the former by only one atom. Since the two extensional unfoldings are evaluated independently, most of the joins in them need to be computed at least twice.

[3] We assume set semantics for answer sets in this paper.
[4] Most contemporary databases allow defining views in terms of other views.

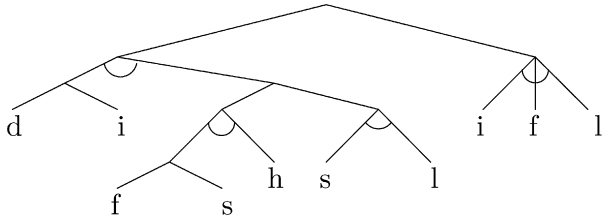

Fig. 2. The AND/OR tree representation of the modified query.

3. Sets of answers retrieved by different extensional unfoldings can overlap or can be even entirely redundant. If a predicate P is defined by two rules, say $\mathcal{R}_1$ and $\mathcal{R}_2$, and each of these rules, given the selections and projections of the query, computes many of the same set of tuples, then the two extensional unfoldings will compute many of the tuples twice. If there are k extensional unfoldings, a given answer (tuple) may be computed k times.
Consider again our running example. Assume additionally that the provider Ætna sells its life insurance and its health insurance as a single package. Then, instead of evaluating the seven join expressions listed above, it is sufficient to evaluate only the following four:

$$d \bowtie f \bowtie h, \qquad i \bowtie f \bowtie h,$$
$$d \bowtie s \bowtie h, \qquad i \bowtie s \bowtie h.$$

To evaluate any of the rest of the unfoldings would be redundant since they will not produce any new answers.

The question then arises on how to efficiently evaluate discounted queries. Consider the following two extensional unfoldings: $d \bowtie f \bowtie h$ and $d \bowtie s \bowtie h$. Since they share a pair of atoms, an obviously more efficient evaluation strategy would be to distribute unions over joins so that the following query is evaluated instead: $d \bowtie (f \cup s) \bowtie h$. The strategy of distributing unions over joins always leads to fewer joins (without increasing the number of union operations). As a side effect of this operation, the redundancy in join evaluation, as well as the redundancy in answer tuple computation, is reduced [11]. Figure 2 shows the result of this operation applied to the union of the seven extensional unfoldings of *Case* 3.[5]

Clearly, the size of the query expression is just one factor that can be used in predicting the efficiency of query evaluation. It is quite likely that some well chosen, larger query expression will evaluate faster than the minimum sized expression of the query. However, succinctness of the query expression is an important component which cannot be ignored. Especially when alternative expressions can be exponentially larger than the minimal expression, controlling the succinctness of the query expression is vital. We focus on the succinctness issue in this paper.

As in *Case* 1 above, it may be that the unfoldings-to-discount (removed sub-views) *cover* the view. This means that the discounted view is equivalent to the *null view* (defined to evaluate empty). We need to determine the complexity of deciding coverage (Section 4). It may be that there are certain types of unfoldings-to-discount that are easy to remove, and namely that the view can be *always* rewritten into a simpler form to accomplish the removal. A rewrite like this is always an algebraic optimization. *Case* 2 above illustrates such a case. We show that there are, in fact, natural cases for this, and we shall call such unfoldings *simple unfoldings* (Section 5).

Our general goal is to find good view disassemblies; that is, rewrites that result in small AND/OR trees. So our aim is to optimize over the number of nodes of the resulting AND/OR tree. An ultimate goal then might be to find a rewrite that results in the *smallest* AND/OR tree possible. Therefore, we should consider the general case of rewriting a view in an algebraically absolute optimal way, as we did in *Case* 3 and shown in Fig. 2. We shall prove that the complexity of (a set-theoretic version of) a special case of this task (a sub-class of fairly simple views) is **NP**-*complete* over the size of the view's AND/OR tree. We demonstrate the general problem is even harder (Section 6).

In lieu of absolute optimality then, we should consider "approximation" solutions for rewrites for view disassembly. We shall develop a rewrite algorithm that produces a disassembled view (thus equivalent semantically to the

[5] We have a naïve algorithm that can find the optimal rewrite in the case of a single unfolding-to-discount. Such a result, even for this small example, would be rather difficult to find by hand. We used this algorithm to produce the tree in Fig. 2.

discounted view expression) for which the algorithm's complexity is over the number of unfoldings-to-discount, and *not*, per se, over the size of the view's AND/OR tree. Hence, this approach is effectively tractable, while resulting in rewrites that are reasonably compact (Section 7).

We do not address in this paper the issue of *determining* when one query semantically *overlaps* with another query; that is, we assume that the unfolding that represents the overlap has already been identified. This, of course, may not be a trivial task. For example, queries $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ from our motivating example might not have been the original cached queries themselves, but instead could have been constructed from them.

3. Related work

The work most closely related to view disassembly is [15]. The authors consider queries that involve nested union operations, and 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 in [15] applies, however, only to a class of simple queries, and no complexity issues are addressed.

Another research area related to view disassembly is *multiple query optimization* (MQO) [17]. The goal in multiple query optimization is to optimize batch evaluation of a collection of queries, rather than just a single query. The techniques developed for MQO attempt to find and reuse common sub-expressions from the collection of queries, and are heuristics-based. We do not expect that the MQO techniques could result in the rewrites we propose in this paper.

The problem of query tree rewrites for the purpose of optimization has been also considered in the context of deductive databases with recursion. In [13], the problem of detecting and eliminating redundant subgoal occurrences in proof trees generated by programs in the presence of functional dependencies is discussed. In [14], the residue method of [1] is extended to recursive queries.

In [8], we introduced a framework we call *intensional query optimization* which enables rewrites to be applied to non-conjunctive queries and views (that is, ones which involve union). An initial discussion of complexity issues and possible algorithmic solutions appear in [11]. In [10], we present an algorithm which incorporates unfolding removal into the query evaluation procedure. Hence the method in [10] is *not* an explicit query rewrite. In [9], we defined when two queries *semantically overlap*. We did this by extending formally the concept of discounting in [7] for discounting a second *query* from a first query, not just discounting the query's unfoldings from a query.

Our work in view disassembly is naturally related with all work on view and query rewrites. However, most all work in view rewrites strives to find views that are semantically *equivalent* with, or *contained* in, the original. This, of course, is relevant for our work as the original view has to be equivalent to the removed components (unfoldings) together with the modified view. But our goal—to remove unfoldings from a view—is different than that of previous view rewrite work, and so this requires a different treatment. Aside from the work listed above, we are not aware of any work on view rewrites that bears directly on view disassembly.

4. Discounting and covers

We represent queries and views in Datalog. For simplicity, we shall refer to a query or view as sets of atoms. For instance, $\{a, e, b\}$ represents query $\mathcal{Q}$ in Example 1.[6] Call this a *query set*. Some of the atoms in the query set may be intensional; that is, they are written with view predicates defined over base table predicates and, perhaps, other views. Since some of atoms may be intensional, the query's corresponding AND/OR tree, with respect to the $\mathbf{I_{DB}}$'s rules, may involve both joins (ANDs) and unions (ORs).

We now provide a formal definition for an *unfolding* of a query, in terms of query sets.

Definition 2. Given query sets $\mathcal{Q}$ and $\mathcal{U}$, call $\mathcal{U}$ a 1-*step unfolding* of query set $\mathcal{Q}$, to be denoted by $\mathcal{U} \preccurlyeq^1 \mathcal{Q}$, with respect to the intensional database $\mathbf{I_{DB}}$ *iff* there is a $q_i \in \mathcal{Q}$ and a rule $\mathcal{R} \in \mathbf{I_{DB}}$

$$\mathcal{R}: \quad a \leftarrow b_1, \ldots, b_n.$$

[6] We ignore the ordering of the atoms in the query, without loss of generality. Furthermore, we also only present "propositional" examples for simplicity's sake, but we make it clear how the techniques and definitions apply in the general case with variables.

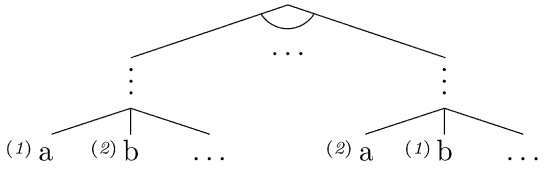

Fig. 3. An AND/OR tree with repeated elements.

such that $q_i\theta \equiv a\gamma\theta$, for which γ is a variable substitution that *standardizes apart* the variables of $\mathcal{Q}$ and $\mathcal{R}$,[7] and θ is a *most general unifier* of q_i and $a\gamma$, such that

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

Define inductively that $\mathcal{U} \preccurlyeq^k \mathcal{Q}$ (with respect to the $\mathbf{I_{DB}}$), for a finite $k > 1$, if there is a $\mathcal{V}$ such that $\mathcal{V} \preccurlyeq^{k-1} \mathcal{Q}$ and $\mathcal{U} \preccurlyeq^1 \mathcal{V}$. Call $\mathcal{U}$ simply an *unfolding* of $\mathcal{Q}$, denoted by $\mathcal{U} \preccurlyeq \mathcal{Q}$, *iff* there is a finite k such that $\mathcal{U} \preccurlyeq^k \mathcal{Q}$.

An unfolding $\mathcal{U}$ is called *extensional iff*, for every $q_i \in \mathcal{U}$, atom q_i is written with an extensional predicate. Call the unfolding *intensional* otherwise.

When $\mathcal{U} \preccurlyeq \mathcal{V}$ (with respect to the $\mathbf{I_{DB}}$), we say that $\mathcal{V}$ *subsumes* $\mathcal{U}$ (with respect to the $\mathbf{I_{DB}}$). We say that $\mathcal{V}$ *properly subsumes* $\mathcal{U}$ when $\mathcal{U} \preccurlyeq \mathcal{V}$ but $\mathcal{V} \not\preccurlyeq \mathcal{U}$, and denote this by $\mathcal{U} \prec \mathcal{V}$.

Call $\mathcal{U}$ and $\mathcal{V}$ *incomparable* if $\mathcal{U} \not\preccurlyeq \mathcal{V}$ and $\mathcal{V} \not\preccurlyeq \mathcal{U}$. We say that $\mathcal{U}$ and $\mathcal{V}$ *overlap* if there exists a $\mathcal{W}$ such that $\mathcal{W} \preccurlyeq \mathcal{U}$ and $\mathcal{W} \preccurlyeq \mathcal{V}$. If $\mathcal{U}$ and $\mathcal{V}$ are incomparable *and* do not overlap, we say that they are *independent*.

One of the 1-step unfoldings of the query $\mathcal{Q}$, $\{a, e, b\}$, in Example 1 is $\{a, e, l\}$. One of the extensional unfoldings of $\mathcal{Q}$ is $\{d, f, h\}$.

A query (set) $\mathcal{U}$ is an *unfolding* of query (set) $\mathcal{Q}$ (with respect to the $\mathbf{I_{DB}}$) if $\mathcal{U} \preccurlyeq \mathcal{Q}$ according to Definition 2. If $\mathcal{U} \preccurlyeq \mathcal{Q}$, query $\mathcal{U}$'s AND/OR tree representation (with respect to the $\mathbf{I_{DB}}$) is *embeddable* in $\mathcal{Q}$'s AND/OR tree representation. In essence, $\mathcal{U}$'s tree is simpler than $\mathcal{Q}$'s; it can be obtained by removing some of the nodes of $\mathcal{Q}$'s tree.

Thus, unfolding $\mathcal{U}$ of query $\mathcal{Q}$ can be represented by a *marking* in $\mathcal{Q}$'s AND/OR tree. In Example 1, the three unfoldings $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$, of query $\mathcal{Q}$ are shown marked by labels 1, 2, and 3, respectively, in $\mathcal{Q}$'s AND/OR tree in Fig. 4. We are interested in this notion of *embedded* AND/OR trees because it is these embedded (marked) AND/OR trees in the query's AND/OR tree—that correspond to the unfoldings that we want to "remove"—that we want to excise syntactically by rewriting the query's AND/OR tree.

When each of the unfoldings in question corresponds to a single embedded AND/OR tree (see Example 4 below), the distinction between *unfoldings*—a semantic notion as are query expressions or query sets—and the (embedded) AND/OR trees that represent them—a syntactic notion as are query plans or query trees—is not pertinent for us. The correspondence, however, between unfoldings of a view and embedded AND/OR trees in its AND/OR tree is not always unique. This can happen if the same atom appears multiple times in the view's definition, and hence, AND/OR tree. Consider the AND/OR tree in Fig. 3. Does unfolding $\{a, b, \ldots\}$ correspond to the embedded tree marked by (1) or that marked by (2)? Or does it correspond to both?

To remove this complication for the sake of this paper, we avoid these ambiguities. We limit our focus to a sub-class of views for which this ambiguity does not arise: views for which no atom is employed twice in its definition. For this sub-class, the correspondence of unfoldings to embedded AND/OR trees is one-to-one. Since the complexity results we derive in this paper are with respect to this sub-class of views, they provide a legitimate lower bound on the complexity on the view disassembly problems for the class of all views. Certainly, a view query with repetition in its definition presents *further* opportunities for optimization via rewrites to remove some of the duplication; but we do not consider these rewrite issues here.

[7] Thus γ renames variables in $\mathcal{R}$ in a most general way to ensure the variables in $\mathcal{R}\gamma$ are distinct from the variables in $\mathcal{Q}$. The substitution is most general in that $\mathcal{R}$ and $\mathcal{R}\gamma$ remain logically equivalent. (So γ does not map X to A *and* Y also to A, for instance. It renames variables one-to-one.)

This is commonly done implicitly in logic programming, to be understood by the reader. We make the standardize-apart step explicit here for clarity.

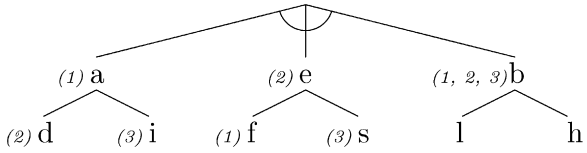

Fig. 4. Cover of the AND/OR tree in Example 4.

Definition 3. Let $\mathcal{Q}$ be a view and $\mathcal{U}_1, \ldots, \mathcal{U}_k$ be unfoldings of $\mathcal{Q}$. We define the *discounted view* (*query*) of $\mathcal{Q}$ with *unfoldings-to-discount* $\mathcal{U}_1, \ldots, \mathcal{U}_k$, denoted as $\mathcal{Q} \setminus \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$. Define **unfolds**$(\mathcal{Q})$ to be the set of all extensional unfoldings of $\mathcal{Q}$. The answer set of $\mathcal{Q} \setminus \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$ is defined to be

$$\textbf{unfolds}(\mathcal{Q}) - \left(\bigcup_{i=1}^{k} \textbf{unfolds}(\mathcal{U}_i) \right).$$

We shall assume for this paper that for any discounted query we consider, $\mathcal{Q} \setminus \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$, it is guaranteed that each $\mathcal{U}_i$ is, in fact, an unfolding of $\mathcal{Q}$. Thus, $\mathcal{Q}$ logically *contains* each $\mathcal{U}_i$. Furthermore, each $\mathcal{U}_i$ logically contains some unfolding of $\mathcal{Q}$, and is logically contained by that unfolding, and so is *equivalent* to that unfolding. Furthermore, given our assumption of an unambiguous mapping of unfoldings to embedded AND/OR trees, any syntactically equivalent unfoldings are, indeed, the same unfolding. Two unfoldings $\mathcal{U}$ and $\mathcal{V}$ of view $\mathcal{Q}$ might differ in their variable names—among their variables not in common with $\mathcal{Q}$—due to the standardizing apart step in Definition 2. If there is a containment mapping from $\mathcal{U}$ to $\mathcal{V}$, and a containment mapping from $\mathcal{V}$ to $\mathcal{U}$, they are considered the same unfolding of $\mathcal{Q}$. Again, because no predicate is repeated twice in $\mathcal{Q}$'s tree, this mapping is straightforward to check. Beyond this, we need not be concerned with containment when discussing discounted queries.

In [9], we extended the definition of discounted queries to allow *any* other *query* to be "removed" from the query to be discounted, not just known unfoldings of the query as here. Of course, this extended definition of discounted queries does require consideration of containment and is significantly more complex. For applications, the extended definition of discounted queries from [9] is more realistic, since we want to *match* the query in question to existing views, queries, and semantic caches, not necessarily directly to unfoldings of the query. However, for purposes of this paper, we assume this matching has been done already, and that the pertinent *unfoldings* of the query have been identified. We believe it is useful to separate these two tasks—matching existing queries and views to the query's unfoldings (which we do not do in this paper), and then discounting those unfoldings from the query (which we do study in this paper)—and allows us to study these steps independently.

In our consideration of discounted queries, the first case we ought to consider is when the set of extensional unfoldings of the discounted view is empty. In such a case, we say that the unfoldings-to-discount—that is, the unfoldings that we effectively want to remove—*cover* the view (or query). The degenerate case is $\mathcal{Q} \setminus \{\mathcal{Q}\}$, when the view itself is to be removed. At the opposite end of the spectrum is $\mathcal{Q} \setminus$ **unfolds**$(\mathcal{Q})$, when all extensional unfoldings are to be removed from the view. When a discounted view is covered, the most succinct disassembled view is the *null view*, which we define to evaluate to the empty answer set. Thus, we are interested in how to test when a discounted view is covered. As it happens, there are interesting, and unobvious, cases of discounted views which turn out to be covered. Furthermore, cover detection is computationally hard.

Example 4. $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ represent three unfoldings of query $\mathcal{Q}$ in Example 1. Figure 4 shows $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ marked in the tree (with labels 1, 2, and 3, respectively) from Fig. 1. Since $\textbf{unfolds}(\mathcal{Q}) \subseteq \bigcup_{i=1}^{3} \textbf{unfolds}(\mathcal{F}_i)$ the set $\{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3\}$ is a cover of $\mathcal{Q}$.

We establish that determining that a discounted view is covered is *co***NP**-*complete* over the number of unfoldings-to-discount. For a set-theoretic version of this problem that is appropriate to establish the computational complexity, the input can be considered to be the view's AND/OR tree and the trees of the unfoldings-to-discount, and the question is whether the view's AND/OR tree is *covered* by the trees of the unfoldings-to-discount.

Definition 5. A *discounted view instance* $\boldsymbol{\mathcal{V}}$ is a pair of an AND/OR tree, which represents the view (and which contains no duplicate atoms by our restriction), and a set of AND/OR trees, which correspond to the unfoldings-

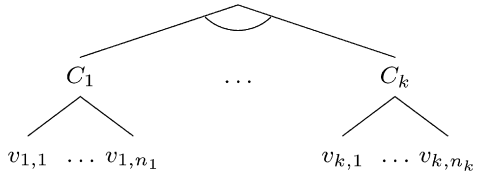

Fig. 5. AND/OR tree representing a CNF propositional theory.

to-discount and which can be (trivially) marked uniquely as embedded AND/OR trees in the view's AND/OR tree. Define **COV** as the set of all discounted view instances that are covered.

Call an embedded AND/OR tree *extensional* if it corresponds to an extensional unfolding. (In this case, there are no OR branches left in the tree, and so it is essentially an AND-tree.) When represented by a marking in the view's AND/OR tree, all its marks will be on leaves of the tree.

Theorem 6. COV *is co***NP**-*complete.*

Proof. Let $\boldsymbol{\mathcal{V}}$ be a discounted view instance. Let $\boldsymbol{\mathcal{V}} = \langle \mathcal{T}, \boldsymbol{\mathcal{D}} \rangle$, in which $\mathcal{T}$ is the AND/OR tree of the view, and $\boldsymbol{\mathcal{D}}$ is the collection of AND/OR trees corresponding to the unfoldings-to-discount.

$\overline{\textbf{COV}}$ is in NP. A witness that $\boldsymbol{\mathcal{V}} \in \overline{\textbf{COV}}$ is an extensional AND/OR tree embeddable in $\mathcal{T}$, call it $\mathcal{E}$, but which cannot be embedded in any $\mathcal{D} \in \boldsymbol{\mathcal{D}}$ (those trees representing the unfoldings-to-discount). This means that there is an extensional unfolding $\mathcal{U}$ (uniquely) corresponding to tree $\mathcal{E}$ that is not an unfolding of any of the unfoldings-to-discount. To check that $\mathcal{E}$ cannot be embedded in any $\mathcal{D} \in \boldsymbol{\mathcal{D}}$ (and that it is embeddable in $\mathcal{T}$) is polynomial in the size of the input ($\boldsymbol{\mathcal{V}}$).

A reduction of **SAT** to $\overline{\textbf{COV}}$. Consider any **CNF** propositional theory $\mathcal{P}$, a **SAT** instance candidate, restricted without loss of generality so that no propositional variable occurs more than three times (negated and not).[8] Furthermore, assume without loss of generality that, for any propositional variable p appearing in $\mathcal{P}$, p and $\neg p$ do not occur together in any clause in $\mathcal{P}$. Let $C_1, \ldots, C_k$ represent the clauses of $\mathcal{P}$. For each clause C_i, let $v_{i,1}, \ldots, v_{i,n_i}$ represent the occurrences of the propositional variables (positive or negative) in C_i.

Transform the **SAT** instance $\mathcal{P}$ into an AND/OR tree $\mathcal{T}$ as in Fig. 5. The tree consists of two layers: an AND of nodes $C_1, \ldots, C_k$ representing the clauses of $\mathcal{P}$; and, for each C_i node, an OR of $v_{i,1}, \ldots, v_{i,n_i}$ nodes, representing the propositional variables of clause C_i. (The $v_{i,x}$'s are distinct, so the tree $\mathcal{T}$ we construct obeys our restriction that no predicate appears more than once in it.)

Construct a set of AND/OR trees to discount as follows. For each pair $v_{i,x}$ and $v_{j,y}$ such that $i \neq j$, $v_{i,x}$ represents propositional variable p, and $v_{j,y}$ represents $\neg p$, construct an AND/OR tree, call it $\mathcal{D}$, as in Fig. 6. Tree $\mathcal{D}$ is specialized in just two ways from the tree $\mathcal{T}$: it contains just node $v_{i,x}$ under C_i, not all the nodes $v_{i,1}, \ldots, v_{i,n_i}$; and likewise, just $v_{j,y}$ under C_j.

Tree $\mathcal{D}$ represents a "non-model" of **CNF** theory $\mathcal{P}$. We cannot designate both $v_{i,x}$ and $v_{j,y}$ as *true*, because we would be setting both p and $\neg p$ as *true*, which would be contradictory. Therefore, no extensional tree embeddable in $\mathcal{T}$ that represents a model of $\mathcal{P}$ can be embedded in tree $\mathcal{D}$.

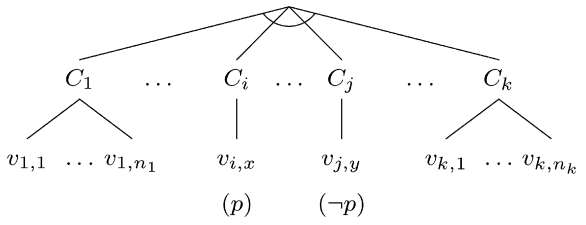

Fig. 6. Unfolding tree representing a "contradiction" in the CNF propositional theory.

[8] **SAT** restricted in this way remains **NP**-*complete*. See theorem [L01] on p. 259 in [5].

Call the collection of trees to discount that we have constructed $\boldsymbol{\mathcal{D}}$. Consider the size of $\mathcal{P}$. Let n denote the number of occurrences of propositional variables in $\mathcal{P}$. So $n = \sum_{i=1}^{k} n_i$. Thus the size of $\mathcal{P}$ is $\mathcal{O}(n+k)$ (in which k, recall, is the number of clauses in $\mathcal{P}$). Since any propositional variable p appears at most three times (negated or not) in $\mathcal{P}$, there can be at most two pairs $v_{i,x}$ and $v_{j,y}$ corresponding to p and $\neg p$. $|\boldsymbol{\mathcal{D}}|$ is thus bounded by $2n$, and so is polynomial in the size of $\mathcal{P}$.

Now ask if there exists an extensional AND/OR tree $\mathcal{E}$ embeddable in tree $\mathcal{T}$ (as seen in Fig. 6) that is not embeddable in any of the trees in $\boldsymbol{\mathcal{D}}$. There is, if and only if $\mathcal{P}$ is in **SAT** (that is, satisfiable). Such an AND/OR tree would not include any $v_{i,x}$ and $v_{j,y}$ corresponding to p and $\neg p$, for any propositional variable p. It would, however, include a v_{i,x_i} from each clause C_i. The set of the propositional variables corresponding to the v_{i,x_i}'s, call this $\mathcal{M}$, represents a model of the propositional theory $\mathcal{P}$. Thus, $\mathcal{M}$ is a witness that the propositional theory $\mathcal{P}$ is in **SAT**. Otherwise, if there is no such tree $\mathcal{E}$, then there is no model $\mathcal{M}$ of $\mathcal{P}$, and so $\mathcal{P}$ in not in **SAT**. □

Consider a variation of the set-theoretic problem **COV**, call it k-**COV**, for each finite positive integer k, such that k-**COV** is the set of all discounted view instances for which the number of trees to discount (unfoldings-to-discount) is k or less, and the discounted view instance is in **COV**. We could not prove that k-**COV**, for any fixed k, is *co***NP**-*complete*, by the same means we did for **COV** in Theorem 6. Our proof for Theorem 6 relies on a translation of a **SAT** instance into a number of trees to discount proportional to the input size. If the number of trees to discount is bounded by a fixed k, this does not work.

Furthermore, any k-**COV** is not, in fact, *co***NP**-*complete*. It is polynomial. Let n be the size of the view tree to be discounted (as in the proof of Theorem 6). We can write an algorithm that is $\mathcal{O}(n^k)$ that finds a tree not covered by any of the trees to discount (or likewise, determines that no such tree exists). (See Algorithm 2 in Section 7 and the related discussion.) Given a fixed k, this is polynomial. (Of course, this is not all that encouraging when k is large.)

This offers a perhaps surprising insight: the complexity of deciding the cover question for discounted views depends on the number of unfoldings-to-discount; it does not depend fundamentally, per se, on the size of the AND/OR tree to be discounted. Of course, the syntactic complexity of the view (n) is relevant, but the number of unfoldings-to-discount (k) is exceedingly more critical. This is quite good, in practice, as often the number of unfoldings being considered is manageably small.

Thus the first step in view disassembly is to check whether the discounted view is covered. We investigate next what can be done when it is discovered that it is not.

5. Simple unfoldings

A disassembled view may cost more to evaluate than the original view. A degenerate case is, of course, the case of the disassembled view that is the union of all the extensional unfoldings. In general, it cannot be guaranteed that the AND/OR tree for a *best* disassembled view would be more compact (hence would require fewer operations to evaluate) than the original view. (*Case* 3 in Section 2 on p. 944 demonstrated this.) In this section, we define a type of unfolding for which discounting is *guaranteed* to produce a tree that is more compact than the tree of the original view. We call such unfoldings *simple*.

The intuition behind the concept of a simple unfolding is as follows. We strive to define an unfolding whose discounting from a query amounts to a removal of one or more nodes from the query's tree *without* rewriting the rest of the tree. In general, removal of a node from a query tree is equivalent to discounting more than just one unfolding, because such a node represents an atom belonging to more than one unfolding. Consider again the query tree in Fig. 1: removal of node f effectively causes a removal of unfoldings $\mathcal{F}_1$, $\mathcal{F}_4$, and every other unfolding that contains that atom. Indeed, removal of a node from a query tree amounts to discounting *all* unfoldings that contain the atom represented by the removed node. Hence, if there is a single unfolding $\mathcal{U}$ that subsumes all unfoldings affected by a removal of node N but *no other* unfoldings, we are guaranteed that removing N from the tree of $\mathcal{Q}$ results in a query tree for $\mathcal{Q} \setminus \{\mathcal{U}\}$.

Let us define first the concept of a *choice point atom*. A choice point atom is an atom (in a query or view) that refers to a view defined with a union operator (at the top level).

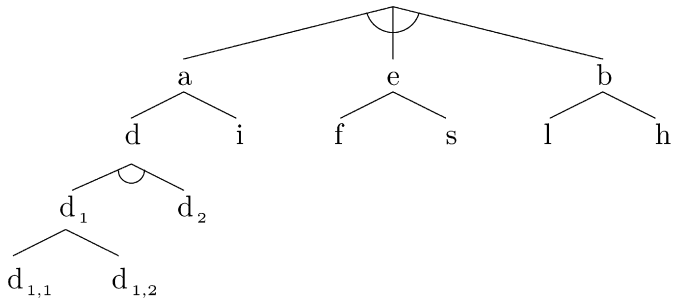

Fig. 7. The AND/OR tree representation of the query of Example 9.

Definition 7. Let $\mathcal{Q} = \{q_1, \ldots, q_n\}$ be a view. Then q_i is called a *choice point atom iff* there exists more than one rule $\langle a \leftarrow b_1, \ldots, b_n. \rangle$ in the $\mathbf{I_{DB}}$ such that $q_i\theta \equiv a\theta$, with respect to a most general unifier θ. (These correspond to OR-nodes in the trees.)

All atoms in query $\mathcal{Q}$ of Example 1 (that is, a, e, and b) are choice point atoms.

Definition 8. Let $\mathcal{Q}$ be a query and $\mathcal{U}_0$ be an unfolding of $\mathcal{Q}$ such that $\mathcal{U}_0 \preccurlyeq^1 \mathcal{U}_1 \preccurlyeq^1 \cdots \preccurlyeq^1 \mathcal{U}_k = \mathcal{Q}$. Then, $\mathcal{U}_0$ is a *simple* unfolding of $\mathcal{Q}$ *iff*, for all i, $0 < i \leqslant k$, the set $\mathcal{U}_i - \mathcal{U}_0$ contains at most one choice point atom.

Example 9. Let the query $\mathcal{Q}$ be as in Example 1, but now assume that **Departments** is also a view defined as follows.

$$d(Y, V) \leftarrow d_1(X, Y, Z), d_2(X, V, W).$$
$$d_1(X, Y, Z) \leftarrow d_{1,1}(X, Y, Z).$$
$$d_1(X, Y, Z) \leftarrow d_{1,2}(X, Y, Z).$$

The query tree for $\mathcal{Q}$ in this new database is shown in Fig. 7.

Consider the following unfoldings of $\mathcal{Q}$:

$\mathcal{U}_1$: $\{d, e, b\}$,
$\mathcal{U}_2$: $\{d_{1,1}, d_2, e, b\}$,
$\mathcal{U}_3$: $\{d, f, b\}$.

Unfolding $\mathcal{U}_1$ is simple, since $\mathcal{U}_1 \preccurlyeq^1 \mathcal{Q}$ and $\mathcal{Q} - \mathcal{U}_1 = \{a\}$. Removing node d (with the entire subtree rooted at d) produces a tree which contains all unfoldings of the original query $\mathcal{Q}$ except for the unfolding $\mathcal{U}_1$ (and all unfoldings subsumed by $\mathcal{U}_1$). In other words, the new tree represents the query $\mathcal{Q} \setminus \{\mathcal{U}_1\}$. Note that this is clearly a case of optimization. The new tree has fewer operations than the original tree.

Unfolding $\mathcal{U}_2$ is simple, since $\mathcal{U}_2 \preccurlyeq^1 \mathcal{U}_4 \preccurlyeq^1 \mathcal{U}_1 \preccurlyeq^1 \mathcal{Q}$, for which $\mathcal{U}_4 = \{d_1, d_2, e, b\}$, since we can show that $\mathcal{U}_4 - \mathcal{U}_2 = \{d_1\}$, $\mathcal{U}_1 - \mathcal{U}_2 = \{d\}$, and $\mathcal{Q} - \mathcal{U}_2 = \{a\}$. Similarly to the case discussed above, it is sufficient to remove node $d_{1,1}$ to produce a tree for $\mathcal{Q} \setminus \{\mathcal{U}_2\}$; no other rewrite is necessary.

Consider unfolding $\mathcal{U}_3$. Now, $\mathcal{Q} - \mathcal{U}_3 = \{a, e\}$. Since both a and e are choice point atoms, the unfolding is not simple. This case illustrates the intuition behind Definition 8. Since $\mathcal{Q} - \mathcal{U}_3 = \{a, e\}$, then there must be at least two atoms in $\mathcal{U}_3$ (d and f, in this case) that lie under a and e, respectively. Since both a and e are choice point atoms (OR-nodes in the tree in Fig. 7), d and f each must have siblings (they are i and s, respectively, in this case). Consider removing d or f (or both) to generate a tree for $\mathcal{Q} \setminus \{\mathcal{U}_3\}$. Removing d from the tree means that an unfolding $\{d, s, b\}$ is also removed; removing f means that $\{i, f, b\}$ is removed as well. Thus, it is not possible to produce a tree for $\mathcal{Q} \setminus \{\mathcal{U}_3\}$ by simple node removal.

We can now state an optimization theorem.

Theorem 10. *Let $\mathcal{Q}$ be a query and $\mathcal{U}$ be a simple unfolding of $\mathcal{Q}$. Then, an AND/OR tree for $\mathcal{Q} \setminus \{\mathcal{U}\}$ can be produced by removing one or more nodes from the AND/OR tree for $\mathcal{Q}$.*

Proof. Let $\mathcal{U} = \mathcal{U}_0 \preccurlyeq^1 \mathcal{U}_1 \preccurlyeq^1 \cdots \preccurlyeq^1 \mathcal{U}_n = \mathcal{Q}$ be a sequence of unfolding steps. Let $V_i \in \mathcal{U}_i$ be the atom whose unfolding produces $\mathcal{U}_{i-1}$, via application of the rule $\mathcal{R}_i^j$: $\langle A_i^j \leftarrow B_{i,1}^j, \ldots, B_{i,k_{i,j}}^j . \rangle$, with a *most general unifier* θ_j, such that $V_i\theta \equiv A_i^j \gamma_j \theta_j$.[9] Then

$$\mathcal{U}_{i-1} = \big(\mathcal{U}_i - \{V_i\} \cup \big\{B_1^j, \ldots, B_{k_{i,j}}^j\big\}\gamma_j\big)\theta_j.$$

Clearly there can be more than one rule $\mathcal{R}_i^j$ via which V_i can be unfolded. There are two cases.

1. For each i, $1 \leqslant i \leqslant n$, $\mathcal{R}_i^0$ is the only rule with head A_i such that there exists a most general unifier θ_0 such that $V_i\theta_0 \equiv A_i\gamma_0\theta_0$. Then, $\textbf{unfolds}(\mathcal{Q}) = \textbf{unfolds}(\mathcal{U}_{n-1}) = \cdots = \textbf{unfolds}(\mathcal{U}_0)$.[10] Thus $\mathcal{Q} \setminus \{\mathcal{U}_0\}$ is null and the query requires no operations to evaluate.
2. Let $\mathcal{U}_i$ be the first unfolding (that is, the unfolding with the lowest index) in the sequence $\mathcal{U}_0 \preccurlyeq^1 \cdots \preccurlyeq^1 \mathcal{U}_i \preccurlyeq^1 \cdots \preccurlyeq^1 \mathcal{U}_n = \mathcal{Q}$ for which there is more than one rule through which V_i can be evaluated. That is, there exist rules $\mathcal{R}_i^0, \ldots, \mathcal{R}_i^m$ such that $\mathcal{R}_i^j$: $\langle A_i^j \leftarrow B_{i,1}^j, \ldots, B_{i,k_{i,j}}^j \rangle$ and there exists a most general unifier θ_j such that $V_i\theta_j \equiv A_i^j\gamma_j\theta_j$. Then $\mathcal{U}_i$ is equivalent to the union of unfoldings $\mathcal{W}_0, \ldots, \mathcal{W}_m$ where $\mathcal{W}_j = (\mathcal{U}_i - \{V_i\} \cup \{\mathcal{B}_{i,1}^j, \ldots, \mathcal{B}_{i,k_{i,j}}^j\}\gamma_j)\theta_j$, for $1 \leqslant j \leqslant m$. Note that replacing a particular one of the $\mathcal{W}_j$'s, say $\mathcal{W}_l$, by its definition results in $\mathcal{U}_{i-1}$. Since each of the subsequent unfolding steps between $\mathcal{U}_{i-1}$ and $\mathcal{U}_0$ each involves a single rule, we have by case 1 from above:

$$\textbf{unfolds}\big(\mathcal{Q} \setminus \{\mathcal{U}_0\}\big) = \textbf{unfolds}\big(\mathcal{Q} \setminus \{\mathcal{U}_{i-1}\}\big) = \textbf{unfolds}\big(\mathcal{Q} \setminus \{\mathcal{W}_l\}\big).$$

Hence, ignoring rule $\mathcal{R}_i^l$ in the unfolding of V_i is equivalent to discounting unfolding $\mathcal{U}_0$ from the query. This is equivalent to removing the entire subtree rooted at $\mathcal{W}_l$ (and nothing else) from the original query tree. □

Simple unfoldings are ideal when the goal of disassembly is optimization (that is, the discounted query or view must cost less to evaluate than the original). They are easy to detect (by definition) and remove (as shown for $\mathcal{U}_1$ in Example 9). The disassembled view is then *guaranteed* to be smaller than the original view. This means the disassembled view will almost always cost less to evaluate than the original view.

We note also that even when a collection of unfoldings of $\mathcal{Q}$, for which none is simple itself, may imply (that is, *cover*) an unfolding of $\mathcal{Q}$ which *is* simple. Consider a non-simple unfolding $\mathcal{U}_3$ of Example 9, and another non-simple unfolding $\mathcal{U}_5 = \{d, s, b\}$. Unfoldings $\mathcal{U}_3$ and $\mathcal{U}_5$ *together* cover unfolding $\mathcal{U}_6 = \{d, e, b\}$, which is simple. We address the problem of *merging* non-simple unfoldings into simple ones in Section 7.

6. Globally optimal solutions

When the unfoldings-to-discount are not simple, finding an AND/OR tree of the disassembled view requires, in general, more than just a pruning of the tree of the original view. Moreover, as we stated in Section 1, one might want to find not just *any* tree, but the most *compact* one; that is, the one with the smallest number of union and join operations with respect to all trees that represent the disassembled view. We call such a rewrite a globally optimal rewrite, since it is smallest with respect to all possible rewrites. One way to achieve this is to take the union of all extensional unfoldings of the view, remove all that are extensional unfoldings of any unfolding-to-discount, and minimize (by distributing unions over joins) the number of operations in the remaining unfoldings. Since distributing unions over joins always decreases the number of joins, minimizing the overall number of operation will minimize the number of joins. This procedure is clearly intractable since there can be an exponential number of extensional unfoldings with respect to the size of the view's tree. As it is, we cannot do any better. Finding the most compact tree is intractable, even for very simple views. In this section, we consider a view which is a two-way join over unions of

[9] As there can be more than one such rule for V_i, we superscript $\mathcal{R}_i$ with j. As in Definition 2, γ_j is a most general substitution that standardizes apart V_i and $\mathcal{R}_i^j$.

[10] With appropriate, straightforward containment mapping of variables understood.

base tables. In a sense, this is the simplest class of views for which the minimization problem is non-trivial (insofar as this is the least complex type of view that can contain non-simple unfoldings).

Let the view $\mathcal{Q}$ be:

$$q \leftarrow a, b.$$

in which a and b are defined as follows:

$$\begin{array}{ll} a \leftarrow a_1. & b \leftarrow b_1. \\ \vdots & \vdots \\ a \leftarrow a_n. & b \leftarrow b_n. \end{array}$$

Define the set of unfoldings-to-discount as follows. Let $\mathcal{U}_1 = \{a_{i_1}, b_{j_1}\}, \ldots, \mathcal{U}_k = \{a_{i_k}, b_{j_k}\}$, such that $i_l, j_l \in \{1, \ldots, n\}$, $1 \leqslant l \leqslant k$. Assume that, for every $\mathcal{S} \subseteq \{a_1, \ldots, a_n\}$, there is an atom (with the intensional predicate) $\mathcal{A}_\mathcal{S}$, and a rule $\langle \mathcal{A}_\mathcal{S} \leftarrow a. \rangle$, for each $a \in \mathcal{S}$. We call the collection of all such atoms $\boldsymbol{\mathcal{A}}$; that is,

$$\boldsymbol{\mathcal{A}} = \big\{ \mathcal{A}_\mathcal{S} \bigm| \mathcal{S} \subseteq \{a_1, \ldots, a_n\} \big\}.$$

Similarly, define $\boldsymbol{\mathcal{B}}$ with respect to $\{b_1, \ldots, b_n\}$.

The discounted query can be evaluated as a union of joins over atoms $\mathcal{A}_\mathcal{S} \in \boldsymbol{\mathcal{A}}$ and $\mathcal{B}_\mathcal{R} \in \boldsymbol{\mathcal{B}}$. We are interested in the maximal pairs of $\mathcal{A}_\mathcal{S}$'s and $\mathcal{B}_\mathcal{R}$'s such that none of the unfoldings-to-discount can be found in the cross.[11] So given such a pair $\mathcal{A}_\mathcal{S}$ and $\mathcal{B}_\mathcal{R}$, $a_{i_l} \notin \mathcal{S}$ *or* $b_{i_l} \notin \mathcal{R}$, for $1 \leqslant l \leqslant k$. Let $\boldsymbol{\mathcal{C}}$ be the collection of all such maximal, consistent pairs $\langle \mathcal{A}_\mathcal{S}, \mathcal{B}_\mathcal{R} \rangle$.

$$\textbf{unfolds}\big(\mathcal{Q} \setminus \{\mathcal{U}_1, \ldots, \mathcal{U}_l\}\big) = \bigcup_{\langle \mathcal{A}_\mathcal{S}, \mathcal{B}_\mathcal{R} \rangle \in \boldsymbol{\mathcal{C}}} \textbf{unfolds}\big(\{\mathcal{A}_\mathcal{S}, \mathcal{B}_\mathcal{R}\}\big).$$

Consider the cardinality of $\boldsymbol{\mathcal{C}}$. Let $t = |\boldsymbol{\mathcal{C}}|$. This represents the number of trees that are needed to evaluate the discounted query. Let $\mathcal{W}_i$, $1 \leqslant i \leqslant t$, enumerate the queries (unfoldings) of $\boldsymbol{\mathcal{C}}$, and $op(\mathcal{W}_i)$ be the total number of operations required to evaluate $\mathcal{W}_i$ (that is, a single join and all its unions). Then, $op(\bigcup_{i=1}^{t} \mathcal{W}_i) = (\bigcup_{i=1}^{t} op(\mathcal{W}_i)) + t - 1$. We also require, without loss of generality, that the $\mathcal{W}_i$'s do not overlap; that is, there does not exist an unfolding U such that $U \preccurlyeq \mathcal{W}_i$ and $U \preccurlyeq \mathcal{W}_j$, for $i \neq j$.

We can now state the problem of *Minimization of Discounted Query* as follows.

Definition 11. Define the class *Minimization of Discounted Query* (**MDQ**) as follows. An instance is the triplet of a query set $\mathcal{Q}$, a collection of unfoldings-to-discount $\boldsymbol{\mathcal{U}}$, and a positive integer K. An instance belongs to **MDQ** *iff* there is a collection of unfoldings $\mathcal{W}_1, \ldots, \mathcal{W}_t$ defined as above with respect to $\mathcal{Q}$ and $\boldsymbol{\mathcal{U}}$, such that $op(\bigcup_{i=1}^{t} \mathcal{W}_i) \leqslant K$.

Theorem 12. *Minimization of Discounted Query* (**MDQ**) *is* **NP**-*complete.*

Proof. It can be shown that **MDQ** is **NP**-*hard* by reducing a known **NP**-*hard* problem, minimum order partition into bipartite cliques **MOP** [4], to it. **MOP** can be defined as follows: Let $\mathcal{G}(U, V, E)$ be a bipartite graph with the vertex sets $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$ and the edge set E. A *bipartite clique* in $\mathcal{G}$ is a complete bipartite graph, and its order is the number of vertices in it. A *clique partition* for $\mathcal{G}$ is a collection of bipartite cliques $\mathbf{C} = \{C_1, \ldots, C_t\}$ such that edge sets $E(C_1), \ldots, E(C_t)$ form a partition of the edge E. The order of a collection of bipartite cliques, $or(\mathbf{C})$, is the sum of orders of the individual cliques. The **MOP** problem can be stated as follows.

Minimum Order Partition (MOP)

Instance: A bipartite graph $\mathcal{G}(U, V, E)$ with the vertex sets $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$ and the edge set E, and a positive integer K.

Question: Is there a collection of bipartite cliques $\mathbf{C} = \{C_1, \ldots, C_t\}$ that partition $\mathcal{G}$ such that $or(\mathbf{C}) \leqslant K$.

The reduction is a straightforward mapping of atoms of a query to vertices of the graph as in the following example. Consider a bipartite graph shown in Fig. 8. The graph can be partitioned into two bipartite cliques $(u_1, u_2, u_3, v_1, v_2)$

[11] By maximal, it is meant that no super-set of $\mathcal{S}$ (from the $\mathcal{A}_\mathcal{S}$ chosen) or of $\mathcal{R}$ (from the $\mathcal{B}_\mathcal{R}$ chosen) would also have this property.

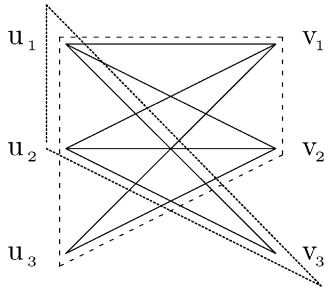

Fig. 8. The graph representing the query in the example of the proof of Theorem 12.

and (u_1, u_2, v_3) marked with broken and dotted lines, respectively. It is easy to see that this partition is minimal. Now consider a query $\mathcal{Q} = (u_1 \cup u_2 \cup u_3) \bowtie (v_1 \cup v_2 \cup v_3)$, where u_1, u_2, u_3, v_1, v_2, and v_3 represent atoms. Assume that there is one unfolding-to-discount, $\mathcal{U} = \{u_3, v_3\}$. Note that all extensional unfoldings of the query can be represented as edges in the graph, where the missing one, $\langle u_3, v_3 \rangle$, represents the unfolding $\mathcal{U}$. Partitioning the graph into cliques is equivalent to clustering the set of all extensional unfoldings into subsets such that they do not overlap, and when unioned, are equivalent to the discounted query. We show in detail in the proof below that minimizing the number of vertices in the cliques is equivalent to minimizing the number of operations in the discounted query.

MDQ $\in$ NP, since a non-deterministic algorithm needs only guess a collection of $\mathcal{W}_i$'s, which can then be checked in polynomial time whether the collection can be evaluated with fewer than K operations. Note that, in the worst case, the collection of $\mathcal{W}_i$'s is of size n^2, which is all of extensional unfoldings of $\mathcal{Q}$.

We transform a **MOP** (minimum order partition of a graph into bipartite cliques) to **MDQ**, which is known to be **NP**-*complete* [4].

Let $\mathcal{G}(U, V, E)$ and the positive integer K be an instance of **MOP**.

Let $U = u_1, \ldots, u_n$ and $V = v_1, \ldots, v_n$ be the sets of vertices in two subgraphs of the bipartite graph. Let each of the vertices represent an atom of a query defined as: $\mathcal{Q} = (u_1 \cup \cdots \cup u_n) \bowtie (v_1 \cup \cdots \cup v_n)$. We also define a collection of unfoldings-to-discount, $\mathcal{U}_1, \ldots, \mathcal{U}_l$ as follows: $\mathcal{U}_i = \{u_{i_1}, v_{i_k}\}$ *iff* $(u_{i_1}, v_{i_k}) \notin E$, that is, unfoldings-to-discount represent the missing edges between the two subgraphs of the bipartite graph. Then, the remaining edges of the graph represent all extensional unfoldings of the query $\mathcal{Q} \setminus \{\mathcal{U}_1, \ldots, \mathcal{U}_l\}$. It is easy to see that each of the cliques $\mathcal{C}_i$, $1 \leqslant i \leqslant t$, in $\mathbf{C}$ represents an unfolding $\mathcal{W}_i$. Since no two cliques share edges, no two unfolding representing them could share extensional unfoldings either.

What remains to be shown is the fact that the number of operations required to evaluate $\mathcal{W}_1, \ldots, \mathcal{W}_l$, that is $op(\bigcup_{i=1}^{t} \mathcal{W}_i)$, is equal (or differs by a constant) to the order of the collections of cliques, $or(\mathbf{C})$. Consider a clique $\mathcal{C}_i$ with vertices $u_{i_1}, \ldots, u_{i_k}, v_{j_1}, \ldots, v_{j_l}$. The order of this clique, $or(\mathcal{C}_i)$, is $k + l$. Consider an unfolding $\mathcal{W}_i$ that represents this clique in our transformation. It has the form of a query: $(u_{i_1} \cup \cdots \cup u_{i_k}) \bowtie (v_{j_1} \cup \cdots \cup v_{j_l})$, hence it requires $(k - 1) + (l - 1) + 1 = k + l - 1$ operations. Then, $op(\mathcal{W}_i) = or(\mathcal{C}_i) - 1$. Let $\mathbf{C}$ contain t cliques. Then, by definition, the order of the graph, $or(\mathbf{C})$ is equal to the sum of orders of all cliques; that is, $or(\mathbf{C}) = \bigcup_{i=1}^{t} or(\mathcal{C}_i)$, where $\mathcal{C}_i$ is the ith clique. On the other hand, evaluating the discounted query requires evaluating the union of all $\mathcal{W}_i$'s; that is, $\bigcup_{i=1}^{t} \mathcal{W}_i$. Hence, we have that

$$op\left(\bigcup_{i=1}^{t} \mathcal{W}_i\right) = \left(\bigcup_{i=1}^{t} op(\mathcal{W}_i)\right) + t - 1 = \bigcup_{i=1}^{t} \big(or(\mathcal{C}_i) - 1\big) + t - 1 = \left(\bigcup_{i=1}^{t} or(\mathcal{C}_i)\right) - 1 = or(\mathbf{C}) - 1. \qquad \square$$

The **NP**-*completeness* result can be trivially generalized to the case where a and b in the query $\mathcal{Q}$ are defined through different numbers of rules.

Theorem 13. *We generalize the* **NP**-*completeness result of Theorem* 12 *to a query* $\mathcal{Q} = (a_1 \cup \cdots \cup a_m) \bowtie (b_1 \cup \cdots \cup b_n)$, *where m is not necessarily equal to n. We call the minimization problem for such queries* **MDQ**$'$. *Minimization of Discounted Query* (**MDQ**$'$) *is* **NP**-*complete.*

Proof. It is easy to see that **MDQ**$' \in$ NP, since a non-deterministic algorithm need only guess a collection of $\mathcal{W}_i$'s and check in polynomial time whether that collection can be evaluated with fewer than K operations.

Since **MDQ** is a special case of **MDQ′**, it is clearly reducible to it. □

The above result can be further generalized (by reduction to **MDQ′**) to a query for which its AND/OR tree representation is arbitrarily deep (an alternation of ANDs and ORs).

Corollary 14. *Let* $\mathbf{MDQ}_{\text{ext}}$ *be the same problem as* **MDQ** *for a query* $\mathcal{Q}$ *with the query tree of unrestricted depth. Then* $\mathbf{MDQ}_{\text{ext}}$ *is* **NP***-hard.*

The minimization of these more complex queries does not remain **NP***-complete*. Consider a query $\mathcal{Q} = \{p_1, \ldots, p_n\}$, where each of p_i's is defined through multiple rules as $\langle p_i \leftarrow p_i^j . \rangle$, for $1 \leqslant j \leqslant k_i$. Since the number of extensional unfoldings for this query is exponential in the size of the original query, verifying the solution cannot be, in general, done in polynomial time. It can be shown, however, that minimization of such a query is in the class Π_2^p.

Theorem 15. *Let* $\mathcal{Q}$ *be a query. Then, minimization of* $\mathcal{Q}$ *is in* Π_2^p.

Proof. Minimization of query's operations is equivalent to minimization of operations in a propositional logic formula, where joins are mapped to conjunctions and unions are mapped to disjunctions. This problem, called *Minimum Equivalent Expression*, is known to be in Π_2^p [5]. □

We conjecture that minimization of $\mathcal{Q}$ may be complete in this class.

7. Locally optimal solutions

In the previous section, we showed that to find an algebraic rewrite for view disassembly which optimizes absolutely the number of algebraic operations (that is, the size of the AND/OR tree) is intractable. In this section, we investigate a locally optimal approach. In this approach, we want to find a collection of unfoldings of the view which, when taken together with the unfoldings-to-discount, cover the view. We call this *cover completion*. We are interested in cover completions that are equivalent to the discounted view. The query tree that is the union (OR) of the unfoldings in the cover completion is then a query tree for the discounted view. This approach is tied to the syntax of the $\mathbf{I_{DB}}$ (and hence, the view's tree). Thus, a semantically equivalent, but syntactically different, original view could result in a different disassembled rewrite.

There can be many such cover completions. We limit our scope to *minimal* cover completions. (We define the minimality criteria below.) The collection of unfoldings found is optimal in that none can be eliminated and still for the collection to cover the view. Still, some minimal cover completions may be algebraically smaller than others. So we do not know whether some other cover completion exists which is smaller. Finding *all* minimal cover completions would be very expensive, as there can be a exponential number of them. Thus, we are guaranteed only local optimality.

A collection of unfoldings that completes the cover, call it $\boldsymbol{\mathcal{C}}$, should have the following properties. Let $\boldsymbol{\mathcal{N}}$ be the collection of unfoldings-to-discount and $\mathcal{Q}$ be the view to be discounted. Given $\mathcal{U}$ and $\mathcal{V}$ such that $\mathcal{V} \preccurlyeq \mathcal{U}$ with respect to the $\mathbf{I_{DB}}$, let us call $\mathcal{U}$ a *refolding* of $\mathcal{V}$.

1. $\boldsymbol{\mathcal{N}} \cup \boldsymbol{\mathcal{C}}$ should be a *cover* of the view $\mathcal{Q}$. That is, for any extensional unfolding $\mathcal{U} \preccurlyeq \mathcal{Q}$, for some $\mathcal{V} \in \boldsymbol{\mathcal{N}} \cup \boldsymbol{\mathcal{C}}$, $\mathcal{U} \preccurlyeq \mathcal{V}$.
2. Any two unfoldings $\mathcal{U} \in \boldsymbol{\mathcal{C}}$ and $\mathcal{V} \in \boldsymbol{\mathcal{N}} \cup \boldsymbol{\mathcal{C}}$ should be independent (Definition 2). That is, for $\mathcal{U} \in \boldsymbol{\mathcal{C}}$ and $\mathcal{V} \in \boldsymbol{\mathcal{N}} \cup \boldsymbol{\mathcal{C}}$, $\mathcal{U} \not\preccurlyeq \mathcal{V}$, $\mathcal{V} \not\preccurlyeq \mathcal{U}$, and $\mathcal{U}$ and $\mathcal{V}$ subsume no unfolding in common.
3. Set $\boldsymbol{\mathcal{C}}$ should be *most general*:
   a. no unfolding $\mathcal{U}$ in $\boldsymbol{\mathcal{C}}$ can be replaced by $\mathcal{V}$ such that $\mathcal{U} \prec \mathcal{V}$ and still preserve the above properties; and
   b. for any $\mathcal{U} \in \boldsymbol{\mathcal{C}}$, $(\boldsymbol{\mathcal{N}} \cup \boldsymbol{\mathcal{C}}) - \{\mathcal{U}\}$ is not a cover of the view.

We present an algorithm to accomplish such a rewrite called the *unfold/refold* algorithm (Algorithm 1). It works as follows. First, find an extensional unfolding which is not subsumed by (and does not subsume) any of the unfoldings-

```
C := {}
while ((U := new_unfolding(N ∪ C, Q))! = null)
  V := refolding(U, N, C)
  C := C ∪ {V}
return parsimonious(C, N, Q)
```

Algorithm 1. Unfold/refold algorithm for view disassembly.

to-discount ($\mathcal{U} \in \boldsymbol{\mathcal{N}}$), or any of the unfoldings generated so far towards the cover completion ($\mathcal{U} \in \boldsymbol{\mathcal{C}}$). This is the co-problem of determining *cover*, discussed in Section 4.

The procedure *new_unfolding* performs this step. A high-level implementation of *new_unfolding* is shown in Algorithm 2. Let q be the atom that represents the query $\mathcal{Q}$, and make q is the root node of $\mathcal{Q}$'s AND/OR tree. The procedure starts with $\mathcal{N} = \{q\}$. Of course, this "unfolding" $\mathcal{N}$ necessarily subsumes each unfolding in the collection $\boldsymbol{\mathcal{U}}$. The objective is to specialize $\mathcal{N}$ so it no longer subsumes any unfolding in the collection, but still is not subsumed by any the collection. The recursive routine *find_unfolding* does this, specializing $\mathcal{N}$ with respect to each unfolding in the collection.

If $\mathcal{N} \preccurlyeq \mathcal{U}$, where $\mathcal{U}$ is the currently considered unfolding from collection $\boldsymbol{\mathcal{U}}$, this attempt at constructing $\mathcal{N}$ fails, and the routine backtracks. Otherwise, if $\mathcal{U} \not\preccurlyeq \mathcal{N}$, no modification to $\mathcal{N}$ is needed for $\mathcal{U}$'s sake; else $\mathcal{U} \prec \mathcal{N}$, and we need to modify $\mathcal{N}$. This can be done by finding an atom C in $\mathcal{U}$ that is the child of an OR-node in $\mathcal{Q}$'s AND/OR tree (these are the nodes in $\mathcal{U}$ that are children of choice point atoms, as defined in Definition 7), and is a descendent of some node in $\mathcal{N}$'s tree. By specializing $\mathcal{N}$ to include a sibling of C (instead of an ancestor of C, which it currently does), then $\mathcal{U} \not\preccurlyeq \mathcal{N}$.

If *find_unfolding* succeeds in finding such a sequence of specialization to $\mathcal{N}$ that distinguishes it from each $\mathcal{U}$ in the collection (and the resulting $\mathcal{N}$ is not subsumed by any of the $\mathcal{U}$'s), a new unfolding has been discovered. Otherwise, there is no such unfolding, and so the collection $\boldsymbol{\mathcal{U}}$ covers $\mathcal{Q}$. Note that if a new unfolding $\mathcal{N}$ is found, it need not be extensional. The routine *new_unfolding* calls routine *extension* (not shown) which finds an extensional unfolding $\mathcal{N}'$ subsumed by $\mathcal{N}$. This is trivial to do. For each $N \in \mathcal{N}$, an extensional descendent D is chosen. Note that this $\mathcal{N}'$ must also be incomparable with each $U \in \mathcal{U}$. There usually is not a unique new unfolding. Thus, *new_unfolding* "non-deterministically" finds a new unfolding (if there is one), according to the *choose* step and the order in which *choices* and *siblings* are tried.

What is *new_unfolding*'s complexity? Let n be the number of nodes in $\mathcal{Q}$'s tree. Let k be $|\boldsymbol{\mathcal{U}}|$, the number of unfoldings in the collection. The recursive stack for *find_unfolding* can only go k deep. In worst-case, the number of recursive calls an instance of *new_unfolding* can make is the number of *choice* atoms that appear in its chosen $\mathcal{U}$. This is *quite* loosely bounded by n. Thus *new_unfolding* is $\mathcal{O}(n^k)$. The average-case performance of *new_unfolding* should be much better. Note that its branching is based on the number of siblings nodes have. If the branching width of the query tree and the number of *choices* in the designated unfoldings are reasonably low, performance will be significantly better.

```
new_unfolding (U, Q){
  N := find_unfolding(U, Q, {q})
  if (N ! = null) return extension (N)
  else return null
}
find_unfolding (U, Q, N){
  if (U = ∅) return N
  choose U ∈ U
  if (N ≼ U) return null
  if (U ⋠ N) return find_unfolding (U − {U}, N)
  foreach (C ∈ choices(U))
    if (descendent(C, N))
    foreach (S ∈ siblings(C))
      N' := find_ unfolding(U − {U}, N ∪ {S}-ancestors(S))
      if (N'! = null) return N'
  return null
}
```

Algorithm 2. The *new_unfolding* procedure for view disassembly.

Many optimizations can be made to the approach presented in Algorithm 2. In [6], we developed a dynamic-programming algorithm for this. We have empirically observed, using this algorithm, that the average case for the cover check is efficient for reasonable numbers of unfoldings-to-discount (on the order of dozens) and views of reasonable complexity (the base in 'n^k'), even though the algorithm's performance does decay—as it must—as more unfoldings-to-discount are added.

Second, *refold* the unfolding $\mathcal{U}$ found by *new_unfolding*—that is, find a most general refolding $\mathcal{V}$ of $\mathcal{U}$—such that the refolding $\mathcal{V}$ remains independent with respect to each of the unfoldings-to-discount, $\boldsymbol{\mathcal{N}}$, and each of the unfoldings in the cover set, $\boldsymbol{\mathcal{C}}$, so far.

This is performed by the routine *refolding* (not shown). The *refold* operation, unlike *new_unfolding*, is inexpensive and straightforward. The unfolding $\mathcal{U}$ can be mapped to a marking in $\mathcal{Q}$'s tree trivially. The *refolding* routine chooses a marked node in the tree, unmarks it, and marks its parent instead. Any other marked nodes under this parent node (siblings of the original marked node) are also unmarked. The new marked tree corresponds uniquely to an unfolding of $\mathcal{Q}$; call it $\mathcal{V}$. By definition, $\mathcal{U} \preccurlyeq^1 \mathcal{V}$. The *refolding* routine iteratively does this until every possible further refolding found is not independent with respect to one of the unfoldings already in $\boldsymbol{\mathcal{N}} \cup \boldsymbol{\mathcal{C}}$; it then returns the last refolding found, so which *is* independent with respect to each of the others. The *refolding* procedure too is "non-deterministic." Depending on the order in which atoms are chosen in $\mathcal{U}$ to replace with parents, a different refolding $\mathcal{V}$ may be found.

The entire algorithm is then repeated until the *new_unfolding* step fails; that is, there is no such extensional unfolding remaining, meaning that a cover of $\mathcal{Q}$ has been established. In the end, *parsimonious* is called to ensure that the unfolding collection $\boldsymbol{\mathcal{C}}$ returned is minimal; that is, no member unfolding of $\boldsymbol{\mathcal{C}}$ can be removed and leave $\boldsymbol{\mathcal{N}} \cup \boldsymbol{\mathcal{C}}$ still as a cover. It is possible to evaluate *parsimonious* efficiently with appropriate bookkeeping added to the unfold/refold algorithm [6].

At completion, the union of the unfoldings produced is a disassembled view. It is equivalent semantically to the discounted view. The set of the unfoldings unioned with the set of the unfoldings-to-discount is equivalent semantically to the original view.

Example 16. Consider again Case 3 of the motivating example from Section 2. The algorithm is initialized with $\mathcal{C} := \{\}$. Assume that the first extensional unfolding to consider is $\mathcal{V} = \{i, f, l\}$. Refolding $\mathcal{V}$, we arrive at an unfolding $\{i, e, b\}$. The next extensional unfolding that does not overlap either with $\{i, e, b\}$ or with the unfolding-to-discount (which is $\{d, f, l\}$ in this case), is $\mathcal{V} = \{d, f, h\}$. Refolding it produces an unfolding $\{d, e, h\}$ (which is pair-wise incomparable with $\{i, e, b\}$ in $\mathcal{C}$ from before). The last remaining extensional unfolding to consider is $\mathcal{V} = \{d, s, l\}$. This one cannot be refolded any further. The AND/OR query tree representing the most-general unfoldings is shown in Fig. 9.

Note that the resulting rewrite in the example has ten internal nodes (algebraic operations) compared with nine nodes of the tree in Example 1 which represented the most compact rewrite.[12]

The run-time complexity of the unfold/refold algorithm is dictated by the *new_unfolding* step of each cycle. This depends on the size of the collection of unfoldings generated so far plus the number of unfoldings-to-discount. While this collection remains small, the algorithm is tractable. Only when the collection has grown large does the algorithm

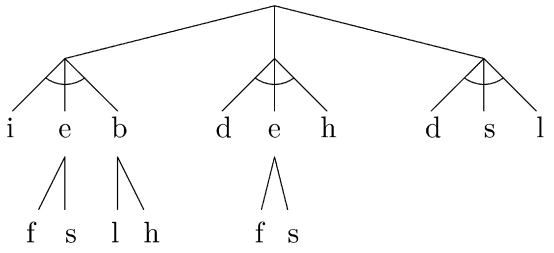

Fig. 9. The result of *unfold/refold* algorithm applied to the query and the unfolding-to-discount of Example 1.

[12] We count each multi-way branch as the requisite number of binary operations it would require.

tend towards intractable. An advantage of the approach is that a threshold can be set, beyond which the rewrite computation is abandoned. On average, we expect the final cover not to be large.

A simple variation of the *refolding* procedure can be applied to find a *most general* collection of unfoldings that is equivalent to the collection of unfoldings-to-discount, $\boldsymbol{\mathcal{N}}$. An unfoldings-to-discount $\mathcal{N}$ is chosen, and is refolded as long as the refolding $\mathcal{N}'$ does not subsume any unfolding $\mathcal{E}$ that is not also subsumed by some unfolding in the current collection. This is done until no unfolding in the current collection can be further refolded. The *parsimonious* routine is then applied to throw away any redundant unfoldings in the final collection. In the extreme case, we might end with a collection consisting just of $\mathcal{Q}$, the view itself. In that case, it is clear that the initial collection $\boldsymbol{\mathcal{N}}$ of unfoldings-to-discount *cover* $\mathcal{Q}$.[13] Again, the most general collection derived is not unique.

Refolding to a most general collection is beneficial. A most-general collection of unfoldings-to-discount is guaranteed always to be smaller—or the same size, at worst—as the initial collection. Thus, a most general collection is better input to the unfold/refold algorithm for view disassembly.

These techniques avail us many tools for rewriting views and queries for a number of purposes. For instance, by finding most-general unfoldings-to-discount, one may also identify (most-general) simple unfoldings that are entailed by the unfoldings-to-discount. For view or query optimization, the simple unfoldings can be pruned from the tree, resulting in a smaller, simpler tree to evaluate (as shown in Section 5). If we "remove" only the simple unfoldings, but *not* the others, we are not evaluating the discounted view, but something in-between the view and the discounted view. When our goal is optimization, this is acceptable.

Example 17. Consider removing the following six (extensional) unfoldings from the AND/OR tree in Fig. 1: $\{d, f, l\}$, $\{d, f, h\}$, $\{d, s, l\}$, $\{d, s, h\}$, $\{i, f, l\}$, and $\{i, f, h\}$. The $\mathbf{I_{DB}}$ again is

$$a \leftarrow d. \qquad e \leftarrow f. \qquad b \leftarrow l.$$
$$a \leftarrow i. \qquad e \leftarrow s. \qquad b \leftarrow h.$$

Choose $\{d, f, l\}$ for refolding. It can be refolded to $\{a, f, l\}$. This also subsumes $\{i, f, l\}$, but this is also subsumed by some unfolding in the collection. (Actually, it is one of them.) $\{a, f, l\}$ cannot be refolded to $\{a, e, l\}$ though, since that also subsumes $\{i, s, l\}$, which is not subsumed by any in the collection. $\{a, f, l\}$ cannot be refolded to $\{a, f, b\}$ either, since $\{a, f, h\}$ is not subsumed by any in the collection. No further refolding is possible for $\{a, f, l\}$ (so far).

Choose $\{d, f, h\}$. It can be refolded to $\{a, f, h\}$. No further refolding is possible.

Choose $\{a, f, l\}$ again. Now it can be refolded to $\{a, f, b\}$, since $\{a, f, h\}$ is *now* subsumed by some unfolding in the collection. No more refolding possibilities remain for $\{a, f, h\}$.

Choose $\{d, s, l\}$. It cannot be refolded to $\{a, s, l\}$; that subsumes $\{i, s, l\}$, which is not subsumed in the collection. It can be refolded to $\{d, e, l\}$; this subsumes $\{d, f, l\}$ too, which is in the collection. No further refolding is possible for $\{d, e, l\}$ (so far).

Choose $\{d, s, h\}$. It cannot be refolded to $\{a, e, h\}$. It can be refolded to $\{d, e, h\}$.

Choose $\{d, e, l\}$ again. Now it can be refolded to $\{d, e, b\}$. This also subsumes $\{d, e, h\}$, but we now know that is covered.

Nothing else can be refolded. When we apply *parsimonious*, just $\{a, f, b\}$ and $\{d, e, b\}$ remain. This represents a most general collection of unfoldings-to-discount with respect to the original collection.

Note that both of these are simple unfoldings, so the original tree can be pruned by simply removing nodes d and f.

Minimal cover completion rewrites for disassembled views have a number of advantages over absolute optimal rewrites.

- As we have demonstrated, one advantage is that they are tractable to find (at least when the number of unfoldings-to-discount plus the number of unfoldings in the final cover is small), while finding absolutely optimal rewrites is generally intractable, dependent on the size of the initial view.
- Since the cover completion approach preserves the structure of the original view—in essence choosing unfoldings of the view—the resulting rewrite may be handled better by query optimizers. The joins and unions are interleaved

[13] It is not guaranteed though that the refolding procedure applied to $\boldsymbol{\mathcal{N}}$ will result in $\{\mathcal{Q}\}$, even if $\boldsymbol{\mathcal{N}}$ covers $\mathcal{Q}$.

in the view trees, as they are in Datalog programs and $\mathbf{I_{DB}}$'s. Absolute optimized trees, as in Fig. 2, however, can be troublesome for cost-based and rewrite optimization because the operations are mixed in quite non-standard patterns. In [10], we have some initial evidence that this is the case.
– Also because cover completion preserves the structure of the original views employed, the evaluation of the disassembled view can effectively use materialized views. For a disassembled view obtained from a globally optimal rewrite, it would be exceedingly rare for any view in the database to match and therefore be applicable.
– Cover completion disassemblies are easier for database programmers and designers to understand, and perhaps, for which to prove correctness.

8. Issues and future work

There is much additional work that could be done on this topic of view disassembly. Some relevant issues are as follows.

1. *Query optimization using view disassembly*.
   a. We should study how view disassembly can be incorporated with existing query optimization techniques. Clearly cost-based estimates should be used to help determine which view disassemblies appear more promising for the given view, the given database, and the given database system.
   b. There are novel ways that view disassembly can be applied to effect query optimization beyond removing empty sub-views, as discussed in Section 1. Such applications to optimization warrant a study.
   c. Good containment algorithms for matching views to a query's unfoldings are needed. There has been significant work on matching views to queries, and this should be extended to query unfoldings. In [9], we address when queries semantically "overlap"—which is essentially the unfolding matching problem—and some initial approaches on how overlaps can be found.
2. *Non-rewrite approaches to view disassembly*.
   a. It may not be necessary to rewrite explicitly views to discount unfoldings to be removed. This could be done during query evaluation. (We have explored such a technique that we call *tuple-tagging* [10].) Other techniques to achieve discounting without any type of rewrite disassembly could potentially offer the advantages of discounting without the cost of rewriting.
   b. Given complementary approaches to discounting (for instance, tuple-tagging in [10] and view disassembly in this paper), which techniques are better in which circumstances? Rewrite techniques are advantageous when the evaluation engine is outside of our control and for view evolution. Evaluation techniques may be better for various types of query optimization.
3. *Extending view disassembly ideas*.
   a. It is possible that a combined approach of the unfold/refold algorithm with other view rewrite procedures might provide generally better performance.
   b. The basic approach of the unfold/refold algorithm ought to be extended for the additional computations discussed in Section 7, as exploiting simple unfoldings.
4. *Devising better complexity profiles for view disassembly*.
   a. We conjecture that the complexity of globally optimal rewrites for disassembly is Π_2^p*-complete*. It would be nice to prove this.
   b. It would be useful to know the average case complexity for computing cover completions.

In [7], we described a framework for intensional query optimization and first introduced the notion that a type of view disassembly could be used for query optimization. In [9], we extended formally the concept of discounting to define discounting a second query from a first query (rather than just discounting some of a query's unfoldings). This builds upon the definition of discounting as presented in [7]. We showed that relational operations such as UNION and EXCEPT (set difference) could be potentially optimized via the use of discounting. We hope to develop such techniques in combination with the rewrite techniques for disassembly developed here.

In [8,10], we have been exploring methods to evaluate discounted queries directly, with no need to algebraically rewrite the query. We have seen with initial experimental results that a method we call *tuple tagging* tends to evaluate

discounted queries less expensively than the evaluation of the queries themselves. Thus, it is necessary to determine which applications need rewrites, and which can be handled by other means.

Certain cover completions may result in more simple unfoldings than others. However, *all* simple unfoldings can be pruned from the view (as discussed in Section 5) and each pruning reduces the view complexity. We should develop techniques to identify (implicit) simple unfoldings. Related is that once one simple unfolding is removed, other (perhaps implicit) unfoldings-to-discount may become effectively simple. This should be exploited.

Sometimes if the original view tree were syntactically rewritten in some given, semantically preserving way, a candidate unfolding could be removed easily, while it cannot be "easily" removed with respect to the original tree as is. We should study how view disassembly could be combined effectively with other semantically preserving view rewrite procedures to enable such rewrites.

9. Conclusions

In this paper, we have defined the notion of a discounted view, which is conceptually the view with some of its sub-views (unfoldings) "removed." We have explored how to rewrite effectively the view into a form equivalent to a discounted view expression, thus "removing" the unfoldings-to-discount. We called such a rewrite a disassembled view. Disassembled views can be used for optimization, data security, and streamlining the query/answer cycle by helping to eliminate answers already seen.

View disassembly, as are most forms of view and query rewrites, can be computationally hard. We showed that optimal view disassembly rewrites is at least **NP**-*hard*. However, effective disassembled views can be found which are not necessarily algebraically optimal, but are compact. We explored an approach called the unfold/refold algorithm which can result in compact disassembled views. The complexity of the algorithm is driven by the number of unfoldings-to-discount, rather than by the complexity of the view definition to be disassembled. Thus, we have shown there are effective tools for view disassembly.

We also have identified a class of unfoldings we called simple unfoldings which can be easily removed from the view definition to result in a *simpler* view definition. This offers a powerful tool for semantic optimization of views. We also established how we can infer when a collection of unfoldings-to-discount *cover* the original view, meaning that the discounted view is void. This result has general application, and is a fundamental part of determining when a view is subsumed by a collection of views.

View disassembly offers new types of view rewrites not explored previously. Little attention has been paid to rewrites that semantically remove portions of the query. However, such techniques look promising for a wide array of applications, from query optimization techniques and query cache usage, to database security. We hope to make useful application of disassembly.

References

[1] U.S. Chakravarthy, J. Grant, J. Minker, Logic-based approach to semantic query optimization, ACM Trans. Database Systems (TODS) 15 (2) (1990) 162–207.

[2] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, M. Tan, Semantic data caching and replacement, in: Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB), Bombay, India, September 1996.

[3] J. Edmonds, J. Gryz, D. Liang, R.J. Miller, Mining for empty rectangles in large data sets, in: Proceedings of the 8th ICDT, London, UK, 2001, pp. 174–188.

[4] T. Feder, R. Motwani, Clique partitions, graph compression, and speeding-up algorithms, in: Proceedings of the ACM Symposium on the Theory of Computing (STOC), 1991, pp. 123–133.

[5] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, A Series of Books in the Mathematical Sciences, W.H. Freeman and Company, New York, 1979.

[6] P. Godfrey, An architecture and implementation for a cooperative database system, PhD thesis, University of Maryland at College Park, College Park, MD, 1999.

[7] P. Godfrey, J. Gryz, A framework for intensional query optimization, in: D. Boulanger, U. Geske, F. Giannotti, D. Seipel (Eds.), Proceedings of the Workshop on Deductive Databases and Logic Programming, in: GMD-Studien, vol. 295, GMD-Forschungszentrum, Bonn, Germany, September 1996, pp. 57–68. Held in conjunction with IJCSLP '96.

[8] P. Godfrey, J. Gryz, Overview of dynamic query evaluation in intensional query optimization, in: F. Bry, R. Ramakrishnan, K. Ramamohanarao (Eds.), Fifth International Conference on Deductive and Object-Oriented Databases (DOOD), in: Lecture Notes in Comput. Sci., vol. 1341, ACM Press, 1997, pp. 425–426.

[9] P. Godfrey, J. Gryz, Answering queries by semantic caches, in: T. Bench-Capon, G. Soda, A.M. Tjoa (Eds.), Proceedings of the Tenth International Conference on Database and Expert Systems Applications (DEXA '99), Florence, Italy, Springer, August 1999, pp. 485–498.

[10] P. Godfrey, J. Gryz, Partial evaluation of views, J. Intell. Inform. Systems 16 (1) (January/February 2001) 21–40.
[11] P. Godfrey, J. Gryz, J. Minker, Semantic query optimization for bottom-up evaluation, in: Z.W. Raś, M. Michalewicz (Eds.), Foundations of Intelligent Systems: Proceedings of the 9th International Symposium on Methodologies for Intelligent Systems, in: Lecture Notes in Artificial Intelligence, vol. 1079, Springer, Berlin, June 1996, pp. 561–571.
[12] A. Halevy, Answering queries using views: A survey, VLDB J. 4 (4) (2001) 270–294.
[13] L.V.S. Lakshmanan, H. Hernandez, Structural query optimization: A uniform framework for semantic query optimization in deductive databases, in: Proceedings of the ACM Symposium on the Principles of Database Systems, Denver, CO, May 1991, pp. 102–114.
[14] L.V.S. Lakshmanan, R. Missaoui, Pushing constraints inside recursion: A general framework for semantic optimization of recursive queries, in: Proceedings of the International Conference on Data Engineering, Taipei, Taiwan, February 1995.
[15] S.-G. Lee, L.J. Henschen, G.Z. Qadah, Semantic query reformulation in deductive databases, in: Proceedings of the Seventh International Conference on Data Engineering (ICDE), IEEE Computer Society, 1991, pp. 232–239.
[16] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, D. Srivastava, Answering queries using views, in: Proceedings of the Fourteenth ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), ACM Press, 1995, pp. 95–104.
[17] T.K. Sellis, S. Ghosh, On the multiple-query optimization problem, IEEE Trans. Knowledge Data Eng. (TKDE) 2 (2) (1990) 262–266.
[18] B. Thuraisingham, W. Ford, Security constraint processing in a multilevel secure distributed database management system, IEEE Trans. Knowledge Data Eng. 7 (2) (April 1995) 274–293.
[19] J.D. Ullman, Principles of Database and Knowledge-Base Systems, vols. I & II. Principles of Computer Science Series, Computer Science Press, Inc., Rockville, MD, 1988/1989.