A Scalable Algorithm for Query Minimization

Isaac K. Kunen
zook@cs.washington.edu

Dan Suciu
suciu@cs.washington.edu

Abstract

Minimizing a query means finding an equivalent one with the least number of joins. Although this problem was conceived over 25 years ago, no scalable algorithm exists today for several reasons. First, the problem is NP-hard even for conjunctive queries, which seems to preclude a universally fast algorithm. More importantly, there was little need: historically most queries were handwritten and already minimal. Today view expansion has become a central theme of many systems. This tends to result in queries that are highly redundant and in need of minimization. In addition, recent theoretical work on tree-decomposition has provided a new approach to query containment, the major sub-problem in query minimization.

We describe a query minimization algorithm that scales to conjunctive queries with hundreds of joins. The algorithm incorporates four techniques: a fast heuristic for tree-decomposition, randomization, table-pruning, and incremental recomputation. Our experimental evaluation shows that each contributes significantly to the algorithm's performance.

1 Introduction

We consider the following problem: given a query, find an equivalent query with the minimum number of joins. The queries considered are conjunctive queries, which correspond to SELECT-DISTINCT-FROM-WHERE queries in SQL where the conditions in the WHERE clause are limited to equality predicates. This problem was first addressed by Chandra and Merlin in 1977 [8] and can be solved by eliminating redundant joins, one by one, until no more redundant joins remain. The problem is hard, because checking whether a join is redundant is NP-complete (it requires a query equivalence test).

Given these simple facts, it may be surprising that this problem has received little attention so far in practice. None of the major relational database systems performs general query minimization, and to our knowledge no scalable algorithm has been described so far. There are two reasons for this. First, SQL queries used to be written primarily by users, and humans tend to write queries that are already minimal. Second, the high complexity of the problem pushed this particular form of optimization down on the database vendors' priority list.

Today, however, neither reason is valid. First, most SQL queries are generated automatically by applications. An increasingly common situation is that of view expansion. Although this will sometimes happen inside of a relational engine, an extreme case can be seen by examining recent XML publishing systems [12, 23]. Here, the XML data is defined as a large virtual view over a relational database, with almost every element type associated width a relational view. XML queries expressed, for example, in XQuery are translated into SQL, by expanding these views. The more element and attribute tags are mentioned in the XML query, the more views end up expanded in the SQL translation. As shown in [12], this may result in highly redundant queries that require a minimization step. Worse, novel web service applications often require several levels of view expansion resulting in even higher redundancy.

Second, there have been recent theoretical advances in studying and analyzing the complexity of the conjunctive query containment and equivalence. These results generalize a well-known result by Yannakakis [29] that query containment of *acyclic* queries can be decided in PTIME, by relaxing the acyclicity condition to requiring a fixed value for a certain parameter. For example, Chekuri and Rajaraman [11] define *query width* and prove that for every k, containment of queries with query width less than k can be checked in PTIME. Similar results have been shown for other parameters, such as tree width [18], and hypertree width [14].

We describe here the first scalable query minimization algorithm, and validate it experimentally for queries with hundreds of joins. Given a query q, we construct the associated *canonical database* instance D, which is simply the database containing the atoms comprising the q. We then eliminate tuples one by one from D, checking for each tuple whether q still evaluates to `true` on the remaining database. The algorithm incorporates four techniques. The first is a fast heuristic for *tree-decomposition* which, as we show here, corresponds precisely to finding a query plan with joins and projections. The second technique is randomization: we introduce a random choice in our tree-decomposition algorithm and run it several times, picking the lowest cost. The third technique is *pruning*: after evaluating the query on the database the first time, we analyze all tuples in all intermediate relations and determine which really contribute to the output. All other tuples are removed from the database. Finally, the fourth technique is incremental evaluation: since the query needs to be evaluated repeatedly after each tuple deletion, we recompute it using a technique taken from from recursive query evaluation [27, 5] and incremental view maintenance [15]. Each technique brings significant improvements to the overall query minimization algorithm, as we show experimentally.

The paper is organized as follows. We motivate the problem in Section 2, provide some background in Section 3, and present some running examples in Section 4. In Section 5 we propose a class of query plans and discuss our plan-generation algorithm. Section 6 deals with several optimizations to the execution phase of our algorithm. We show scalability results in Section 7. We present related work in Section 8, and conclude in Section 9.

2 Motivation

Consider the relational schema `Emp(EmpID, Gender, MngrID)` and the following two SQL queries:

```
q:
SELECT DISTINCT e1.EmpID
FROM Emp AS e1,
     Emp AS e2,
     Emp AS e3,
     Emp AS e4
WHERE
e1.Gender = "female" AND
e1.EmpID = e2.MngrID AND e2.Gender = "female" AND
e2.EmpID = e3.MngrID AND e3.Gender = "female" AND
e4.EmpID = e3.MngrID AND e4.Gender = "female"

q':
SELECT DISTINCT e1.EmpID
FROM Emp AS e1,
     Emp AS e2,
     Emp AS e3
WHERE
```

```
e1.Gender = "female" AND
e1.EmpID = e2.MngrID AND e2.Gender = "female" AND
e2.EmpID = e3.MngrID AND e3.Gender = "female"
```

Instead of SQL we will use datalog notation [26] throughout the paper, and write the queries as:

$$\begin{array}{lll} q(x) & :- & Emp(x, \text{``}female\text{''}, \_) \\ & & Emp(y, \text{``}female\text{''}, x) \\ & & Emp(z, \text{``}female\text{''}, y) \\ & & Emp(y, \text{``}female\text{''}, \_) \\ q'(x) & :- & Emp(x, \text{``}female\text{''}, \_) \\ & & Emp(y, \text{``}female\text{''}, x) \\ & & Emp(z, \text{``}female\text{''}, y) \end{array}$$

q and q' compute the same thing: both return women managers who supervise other women managers, who in turn supervise women. However, q' is easier to compute than q because it requires only two joins instead of three. In fact q' has the smallest number of joins among all queries that are equivalent to q and is therefore called the *minimal query.* Although today's database systems perform query minimization in some limited cases, they do not do so in general: when presented with query q they will optimize and execute a query with three joins. The reason for the choice not to implement a full query minimization algorithm has been twofold: users typically don't write non-minimal queries like q, and query minimization would add an expensive step to the optimizer.

This decision warrants revisiting in light of the many applications that have been devised that rely on view expansion. We begin by considering a simple case of expanding a query written in terms of virtual views. Assume we have the following view:

$$v(x, y) \colon -Emp(x, \text{``}female\text{''}, \_), Emp(y, \text{``}female\text{''}, x)$$

The view computes all female employees y managed by a female manager x. Now we can answer the query above by using this view, as:

$$q''(x) \quad :- \quad v(x, y), v(y, \_)$$

When the system expands the view definition, the result is precisely query q above, with one redundant join. This is, of course, a toy example: in practice queries may be posed over dozens of views which, after expansion, result in many more redundant joins. In some systems there may be several layers of these rewritings, each compounding the redundancy.

A special case of view expansion is in XML publishing [12, 23]. The XML document is defined as a view over a relational database and XML queries are translated into SQL by a process with resembles view expansion, and result in highly redundant queries. In fact, our initial motivation for this work was precisely to minimize queries in SilkRoute [12].

We should note that a query that retains duplicates cannot be minimized; more precisely, no join can be eliminated without changing the query's semantics. A result by Chaudhuri and Vardi [10] shows that two queries under *bag semantics* are equivalent if and only if they have isomorphic bodies. However, if such a query occurs as a subquery, then we may be able to apply query minimization even if it retains duplicates. For example the query:

```
SELECT...
FROM...
WHERE EXISTS (SELECT... FROM... WHERE...)
```

has a subquery which can be first translated into a SELECT-DISTINCT query, and subsequently minimized.

Most databases contain some schema constraints which can be used to minimize queries. For example, if we assume that the CEO is her own manager, then using the fact that `EmpID` is a key in `Emp` and `MngrID` is a foreign key we can minimize the following query:

$$q(x) : -Emp(x, \text{“}female\text{''}, y), Emp(y, \_, \_)$$

to:

$$q'(x) : -Emp(x, \text{“}female\text{''}, y)$$

Here q and q' are equivalent even under bag semantics, and most relational systems today indeed rewrite q into q'. Query minimization in the presence of constraints can be done by adding a *chase* step to the basic minimization algorithm [2]. Schema-based minimizations is orthogonal to the minimization algorithm, and we do no address it in this paper.

3 Background and Problem Statement

We define here the problem formally. We are given a conjunctive query, usually written as a datalog rule with n goals:

$$q(\bar{x}) \ :- \ r_1(\bar{x}_1), \ldots, r_n(\bar{x}_n)$$

and are asked to find an equivalent query q' with a minimum number m of goals:

$$q'(\bar{x}') \ :- \ r'_1(\bar{x}'_1), \ldots, r'_m(\bar{x}'_m)$$

such that q and q' are equivalent. q' is called a minimal query: it is known that any other minimal query is isomorphic to q'. In other words, they differ only by renaming the variables. It suffices to consider only the minimization problem for *Boolean* queries, i.e., those without head variables. Minimizing $q(\bar{x})$ can be reduced to the problem of minimizing q_0 defined as:

$$q_0() \ :- \ h(\bar{x}), r_1(\bar{x}_1), \ldots, r_n(\bar{x}_n)$$

where h is a new predicate symbol. Hence, throughout the paper we will assume that the queries are Boolean.

There is a standard algorithm for minimizing q. Construct a decreasing sequence of queries, $q_0 = q, q_1, q_2, \ldots$, as follows. At each step $i = 1, 2, \ldots$ choose a subgoal in q_{i-1} such that, if we denote by q_i the query obtained by eliminating this subgoal, then q and q_i are equivalent. When no more subgoals can be eliminated stop and return the last query in the sequence.

It can be shown that we arrive at the same result (up to variable renaming) no matter in which order we attempt to remove subgoals, that we need only check the removal of each subgoal once, and that the result is indeed the minimal query.

The simple procedure takes at most n steps, but at each step it needs to check if two queries are equivalent, a problem which is know to be NP-complete. We focus now on this step.

The standard method for checking equivalence is to find a homomorphism from q to q_i (there is no need to search for one from q_i to q, in our case). This can be recast in terms of a query evaluation problem. First, construct the *canonical database* D_i for q_i, the database composed of the atoms in the body of q_i. To illustrate this idea, consider the case if q_i were defined as follows:

$$\begin{array}{lll} q_i(x) & :- & Emp(x, \text{“}female\text{''}, tmp1) \\ & & Emp(y, \text{“}female\text{''}, x) \\ & & Emp(z, \text{“}female\text{''}, y) \\ & & Emp(y, \text{“}female\text{''}, tmp2) \end{array}$$

This query is translated into a database with one relation, *Emp*, and a row in the table for each relation, with each variable converted to a constant:

D_i :

Emp		
"x"	"female"	"tmp1"
"y"	"female"	"x"
"z"	"female"	"y"
"y"	"female"	"tmp2"

Next, evaluate q on D_i. If the answer is `true`, then there exists a homomorphism. Our discussion leads to the following query minimization algorithm template:

```
Algorithm: Generic Query Minimization
Input: Query $q$
Output: Minimal query $q'$
Method
  Step 1 construct a query plan $p$ for $q$
         construct $q$'s canonical database $D$
  Step 2 for each tuple $t \in D$ do
           if $q(D - \{t\}) =$ true
             then $D := D - \{t\}$
  Step 3 Return the query $q'$ corresponding to $D$
```

Query minimization uses core techniques deployed in a database management system: optimization and query plan generation, followed by query execution on a database instance. Viewed this way, one may be tempted to simply use a relational engine to do so. There are however a few key differences, some making this approach not only inefficient but even impossible (as we shall see in more detail). First, the query is relatively large: in applications that require several levels of unfoldings the query may have hundreds of joins, even if the minimal query has many fewer. This rules out traditional optimization methods. By contrast, the database instance on which we evaluate the query is relatively small, "only" a few hundreds of tuples, but is not persistent, meaning that we don't have precomputed statistics or indexes. Finally, we need to execute the same query repeatedly on several databases that have only small differences, and so specific optimizations for this case need to be considered.

4 Running Examples

Here we introduce several classes of queries which will be used throughout the paper. Each query consists of a single binary predicate, so we can visualize them as directed graphs, and each class is parameterized by a single variable n.

Augmented path queries (Figure 1a) consist of a path of length n, with an additional dangling edge rooted at each node except the last. An augmented path query of length n has $2n$ subgoals.

Ladder queries (Figure 1b) are arranged in a ladder-like configuration with all edges pointing from the upper left to the lower right. A ladder query with n rungs contains $3n - 2$ subgoals.

Augmented ladder queries and *circular augmented ladder queries* (Figure 1c) are ladder queries that have additional, dangling edges attached to each node. Circular queries also loop back upon themselves. A linear augmented ladder query contains $5n - 2$ subgoals, and a circular augmented ladder query contains $5n$ subgoals, where n is the number of rungs in the query.

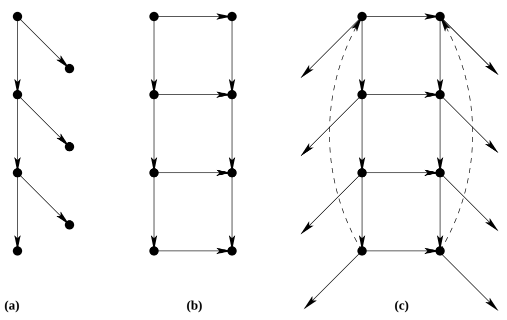

Figure 1: Augmented path, ladder, and augmented circular ladder queries, each with a highlighted minimal subquery.

5 The Query Plan

5.1 Join-Project Plans

Query plans for SELECT-DISTINCT-FROM-WHERE queries consist of joins, projections, and duplicate elimination. Figure 4 illustrates a query plan for the query in Fig. 3 (a), showing projection and join operators—we discuss the duplicate elimination in a moment. We describe in this section a query plan generation algorithm specifically designed for the problem of query minimization. As we shall see, it differs significantly from traditional query optimization algorithms.

Duplicate Elimination Database optimizers usually postpone duplicate elimination until the end, because of their high cost for large tables. In query minimization, however, plans with duplicate elimination at the end are simply impossible to evaluate. For a simple illustration, consider an augmented path query of length n. This query has 2^n possible evaluations on its own canonical database, and, absent duplicate elimination, each of these results in an output tuple. Our database system could not evaluate this query for n larger than 20 or so. Thus, in the query minimization algorithm we only consider plans that automatically perform duplicate elimination after each operation, and push projections down as far as possible. We call these *join-project plans*. Since these plans are executed in main memory, we implemented all intermediate relations as hash tables, and all joins as hash-joins.

Cost metric We chose a very simple function for our cost metric: the cost of a plan is defined to be the largest arity (width) of any intermediate relation in that plan. Clearly, a small width ensures a nice theoretical upper bound on the complexity of the plan, but is no guarantee that the plan will run better than others with a larger width. In Figure 2 we show an example of the correlation between our simple cost and the actual running time in one particular example. We considered a single query with 58 subgoals, generated 100 plans for it, and ran our complete minimization algorithm, with several optimizations turned on. The plan widths ranged from 4 to 13, and the running times ranged from 0.2 seconds to over 1800 seconds. We see that small width guaranteed fast running time, but the worst-case running time grew exponentially with the width. This positive correlation supports the choice of our cost model.

The optimization problem With this justification, can now formally state our optimization problem: given a conjunctive query, find a join-project plan with the minimum width (cost).

Before discussing our solution, it helps to remember the setting: we need to optimize queries with hundreds of joins, but then only evaluate them on a database with hundreds of tuples. This rules out expensive optimization algorithm, like dynamic programming, because the optimization time would dominate the total running time. Instead, we base our solution on recent graph-

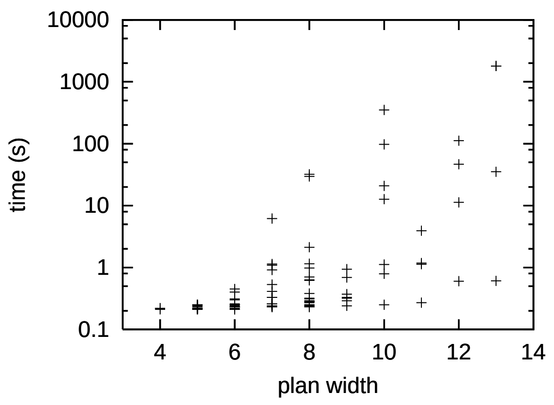

Figure 2: Plan widths and minimization times for a query with 58 subgoals. Run times have been limited to 1800 seconds.

theoretical work on the *tree-decomposition problem*. We review that work and its relationship to query optimization next.

5.2 Tree Decompositions and Query Plans

The *tree decomposition* problem (see, e.g., [6]) is the following. We are given a graph, G, and are asked to construct a tree T whose nodes are labeled with sets of graph vertices, and which satisfies the following conditions:

- Each vertex of the graph appears at some tree node.
- If an edge connects vertex x and vertex y, then x and y are co-located at some tree node.
- For each vertex x, the nodes of the tree which contain x form a connected sub-tree.

This definition extends to hypergraphs in a natural way: the tree decomposition of a hypergraph is defined to be the tree decomposition of its incidence graph. The incidence graph G of a hypergraph H consists of the vertices of H with an edge connecting vertices x and y in G iff there is a hyperedge in H containing both x and y.

The width of such a tree is defined to be $m - 1$, where m is the maximum number of elements in a node label. The *tree width* of a graph is the minimum width of all tree decompositions of the graph.

We denote by H_q the hypergraph of a query q, i.e., the hypergraph with vertices corresponding to the variables in q and edges corresponding to the subgoals of q. The tree decomposition and tree width of a query are defined to be the decomposition and width of its corresponding hypergraph. It has been shown that given a tree decomposition for a query q of width w, one can test whether r is contained in q in time $O(n^(w + 1))$ for an arbitrary query r.

There have been several refinements to this bound given by the widths of similar decompositions, namely query and hypertree decompositions [11, 14]. Rougly, these decompositions allow edges to appear in the decomposition and "cover" the variables in their bodies. The width of a particular node is the number of edges used at the node plus the number of uncovered vertices. For example, if a node in a tree decomposition contained $\{x, y, z\}$ then we would say the width was 2. If we were able to use the edge (x, y) to cover variables at at this node, then it would cover the x and y, leaving an edge and one uncovered variable for a width of 1.

In the case of query decomposition, the edges used must only cover variables present at the nodes, while in hypertree decompositions the edges may introduce variables not present.

If we denote the tree width, query width, and hypertree width of a query q by $TW(q)$, $QW(q)$, and $HW(q)$, then

$$kQW(q) \geq TW(q) \geq QW(q) \geq HW(q)$$

where k is the maximum arity of the predicates in q [11, 14]. Similar time bounds to that shown for tree decomposition have been found for both query and hypertree decompositions.

One should note that to achieve these bounds one needs to find a minimal decomposition of the form in question. There are several negative results on this front. Finding a decomposition of any of these forms is NP-hard. If we fix a maximum width we are willing to tolerate, then there exist algorithms for tree and hypertree decompositions that will either return a decomposition of that width or fail in polynomial time. The time bounds for these algorithms still grow exponentially in the width and in practice are not viable even for very small widths [7]. Query decomposition remains NP-hard even for a fixed width[14].

In our work we consider a restricted form of tree decomposition of a query. Instead of decomposing the incedence graph, we decompose a bipartite graph in which the nodes on one side correspond to vertices from the original hypergraph, the nodes on the other side correspont to edges in the original hypergraph, and there are edges connecting a variable node to an edge node iff the vertex appears in the edge. To obtain a restricted tree decomposition we find the tree decomposition of this graph, but remove the edge labels from the decomposition. (Although we find it useful to retain them as annotation, they are not technically part of the decomposition.)

Although formulated soemwhat differently, this restricted tree decomposition is not substantially different from a "normal" tree decomposition:

Theorem 5.1 *(i) Each restricted tree decomposition is a tree decomposition, and (ii) if there exists a minimal tree decompositon of a hypergraph H of width w, then there exists a restricted tree decomposition of graph H with width w as well.*

This restricted tree decomposition can be transofmed into query plan with relative ease. We illustrate with an example.

Example 5.2 Consider the query q in Figure 3a and its query graph G_q in Figure 3b. A restricted tree-decomposition is shown in Figure 3e. We obtain a query plan by applying the following rules bottom-up: (1) leaf nodes become base table in the plan, (2) internal nodes containing only variables become projections on those variables, and (3) internal nodes containing subgoals will be converted into a join of arity $c + 1$, where c is the number of children the join has—this will join the subgoal at the node with the tables from each of the child branches.

The conversion for the query in Figure 3 appears in Figure 4. Note that the table associated with each join node has width equal to the width of the corresponding node in the tree-decomposition, and that the width of the tree corresponds exactly to the width of the resulting query plan. These plans can be further simplified by eliminating unnecessary projections.

5.3 The Query Plan Generation Algorithm

Our plan generation algorithm for a conjunctive query q is the following. Let G_q be the query graph.

Step 1: Find a random spanning tree T for G_q

Step 2: Evolve T into a tree decomposition for G_q, by labeling its nodes with sets of vertices from G_q. Initially, each node of T is labeled with itself. For every edge (x, g) in G_q that is not part of the tree, we add x to the labels of all tree nodes on the unique path from x to g. The result is a restricted tree decomposition for G_q, and we compute its width, w.

Example 5.3 We illustrate the algorithm step-by-step in Figure 3. The steps are:

1. The query q (Figure 3a) is converted into a query graph G_q (Figure 3b). The query graph is a bipartite graph with variables on the left, subgoals on the right, and edges connecting variables to the subgoals in which they are mentioned.

2. A spanning tree is generated for the query graph. The tree is re-labeled so that the nodes carrying a variable are labeled with a set containing that variable, and the nodes carrying a subgoal are labeled with that subgoal and the empty set of variables (Figure 3c). The edges not used in the tree, the *residual* edges, still need to be accounted for.

3. The residual edges are taken into account. In our example the only residual edge is $w \longleftrightarrow p(z, w)$. The (unique) shallowest nodes containing the variable and the subgoal are located (indicated by arrows in Figure 3c), the path between the nodes is found, and the variable is inserted into each node along this path (Figure 3d).

4. Finally, any leaf nodes lacking a subgoal are pruned from the tree, and the variable sets at nodes containing subgoals are augmented to contain the variables present in those subgoals (Figure 3e).

Analyzing the Algorithm The algorithm is simple, and very fast, but it is not clear how close it comes to finding an optimal width plan. The first observation is that, if there exists a restricted tree-decomposition T of G_q of width w, then there exists a run of the algorithm that finds it: to see this simply remove labels from T until it becomes a spanning tree, and, hence, it could have been generated by the algorithm during Step 1. This means that the algorithm always has a chance to find the best query plan. However, it also has a chance to find arbitrarily bad plans, as the following example shows.

Example 5.4 Consider the ladder query with 2 rungs shown in in Figure 5a. Figures 5b and 5c illustrate two possible decompositions of this query. The decomposition in Figure 5b spans using all of the rungs of the ladder, and induces a plan of width 3, while the one in Figure 5c breaks all but one of the rungs and induces one of width 4. In fact, a width of 3 is optimal, so it is clear that our algorithm can miss the optimal decomposition.

This behavior is only exacerbated as the ladder's length is increased. Regardless of the ladder's length, it can be shown that the optimal width is 3. If the spanning tree splits all but one of the rungs, however, the derived width will grow linearly with the number of rungs, and our algorithm will achieve near-pessimal performance.

Thus, it is possible to produce arbitrarily poor plans, and the ladder queries are in some sense the worst cases for the algorithm. For this reason we added the following randomization step to the algorithm, to reduce the chance of making a bad choice.

Step 3: Randomization Repeat steps 1 and 2 several times and retain the tree decomposition with the smallest width.

This step results in significant speed-ups on the overall running times. Figure 6 shows the running times of the entire minimization algorithm without the randomization step. Here we ran

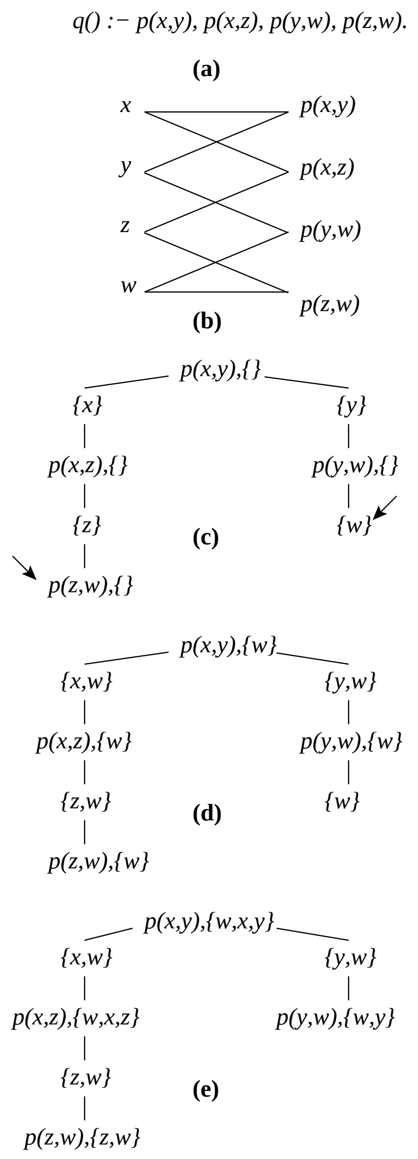

Figure 3: From Query to Decomposition

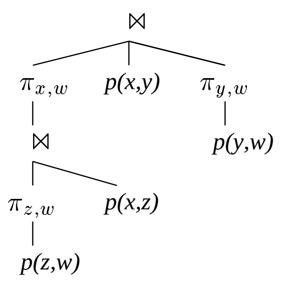

Figure 4: The Final Execution Plan—Duplicate eliminations are performed after each operation.

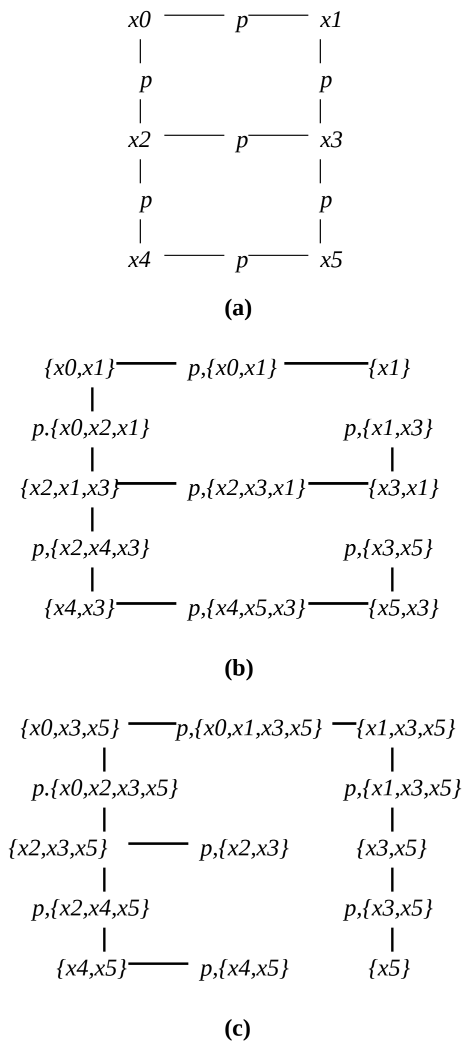

Figure 5: A 2-Rung Ladder and two Decompositions

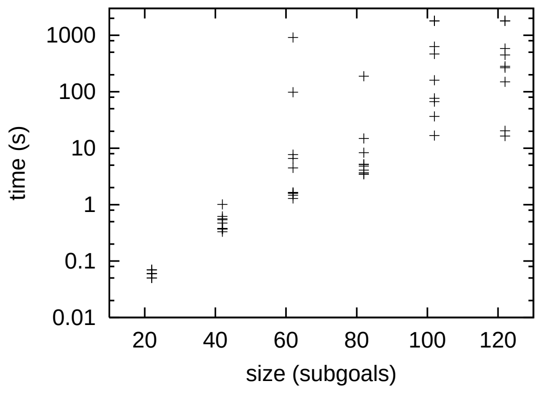

Figure 6: Augmented ladder queries minimized with the baseline implementation

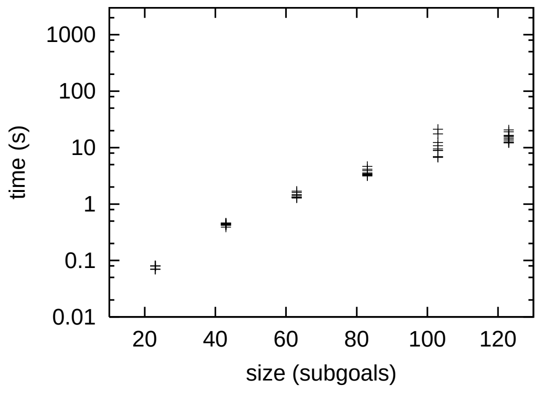

Figure 7: Augmented ladder queries minimized with randomization

the algorithm ten times on each query and reported the running time. Since we generated a different (random) query plan at each run, the running times were spread over a range of more than two orders of magnitude. Figure 7 shows exactly the same queries, run *with* the randomization step. As one can see, the spread decreases significantly, concentrating the points on the lower edge of the curve. In both graphs we did not use the optimizations discussed in the following section.

All of the results in the remainder of the paper are from experiments with this optimization turned on.

6 Query Execution

In this section we explore our base implementation and several optimizations we have layered on top of it to further enhance its performance. Our problem is simple: we have a query plan and we need to evaluate it repeatedly on a sequence of decreasing databases.

To speed this up we make a number of optimizations. Although our basic algorithm is successful because it is very effective at removing duplicates, many optimizations rely on duplicate information. We can retain the benefits of duplicate elimintation while retaining this information by collapsing duplicates into a single tuple and keeping a count of how many times it occurs. As multiplicities can grow exponentially we required a bignum implementation to store them which we obtained from the Gnu Multiple Precision Library.

In the following results, each method was tried 10 times and the results averaged. Randomization was used, choosing the best plan from 50 randomly created ones. To ensure a fair comparision, in each experiment the plans chosen were the same for each method. Differences in performance should therefore be based on the optimizations rather than the particular plans used.

We make the following optimizations:

Early Terminations: We have two early termination optimizations. First, if at any point during the execution of the plan an intermediate result is found to be empty, the end result must be empty too, and execution terminates. This holds in our case because of the monotonicity of conjunctive queries.

Second, since we are repeatedly executing our query on very similar databases, we can expect that we will often see internal tables that do not change from iteration to iteration. We cache the results of each join, and if the inputs to that join have not changed from the previous execution, we simply output the cached value.

We can determine if a table has changed in constant time by making use of a row-count on each table. Making use of the monotonicity property once more, we observe that removal of entries from the database can only remove rows from the internal tables, and so a table's size changes if and only if its entries have changed.

There is almost no cost to these tests, so we always perform these optimizations.

Table Pruning: Consider a join $D = E \bowtie F$. If there are rows in E which do not join with any row of F, or vice versa, then they can be removed preemptively. This will yield smaller joins during future evaluations of the query. We can carry this a step further by pruning in a top down fashion. Once we have pruned D, we can remove not only those rows in E and F which do not contribute to the original D, but all rows which do not contribute to the *pruned* version of D. This amounts to sequentially performing semi-join reductions down the join tree, with the result that no tuples are left which do not contribute to the final output [2].

This is implemented by executing the query once on the complete database, and then removing tuples that do not contribute to the pruned result of each join in a top-down fashion. Pruning can have a significant effect on minimization time, but it is not entirely free: pruning essentially requires an additional complete evaluation of the query.

Notice that pruning is done only once at the beginning of the minimization algorithm, and that all subsequent iterations benefit from the reduced intermediate tables.

We examine the effect of pruning using the augmented ladder queries. As we can see from the results in Figure 8, pruning improves performance by about an order of magnitude, reducing the time to minimize a 25-rung augmented ladder from 15.3 to 1.2 seconds. This query is very amenable to pruning. Since each edge can only be mapped to a small number of positions in the ladder, the corresponding tuples will be pruned out of most of the base tables.

Incremental Evaluation: Our next optimization is based on an idea taken from recursive query processing (see, e.g., [27, 5]) and incremental view maintenance [15]. Assume that we have an internal table D which is produced as a join of tables E and F, i.e., $D = E \bowtie F$. If E and F are updated to E' and F' we can either calculate D' as before, or we can compute the change in D, ΔD, as

$$\Delta D = (E \bowtie \Delta F) + (\Delta E \bowtie F) - (\Delta E \bowtie \Delta F)$$

Although this requires three joins instead of the one required for the naive calculation, if ΔE and ΔF are small, then these may indeed be faster calculations.

To examine the additional benefit of incremental evaluation, we use the example of circular augmented ladder queries. The results are shown in Figure 9. Note that this is a logarithmic scale,

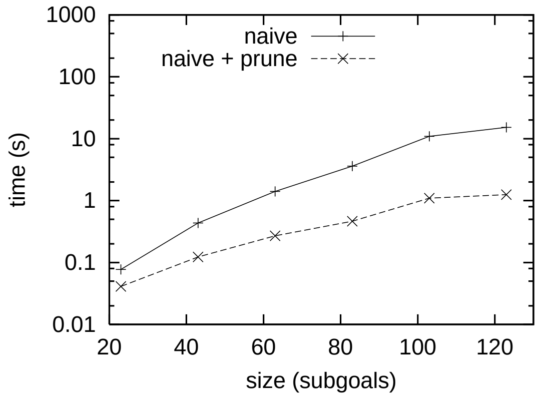

Figure 8: Average minimization time for augmented ladder queries

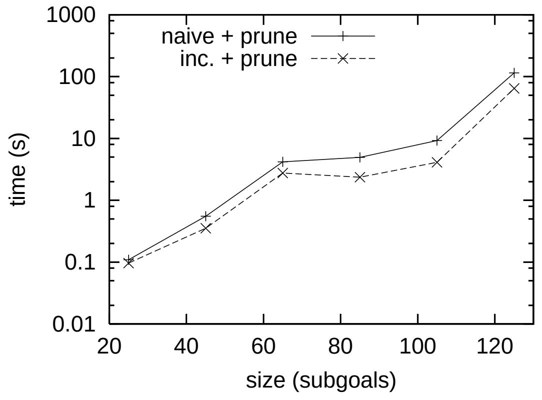

Figure 9: Average minimization time for augmented circular ladder queries

so the improvements are actually several-fold. We first see that the times are generally higher than in the linear ladder case. This is due in part to the fact that the best-possible plan width is 5 instead of 3 for these queries. Additionally, although pruning will have some effect, less is possible due to the symmetry of the query. We find, however, that incremental evaluation is effective in this case.

Combining Optimizations: These techniques can be used independently or combined. Note that each has its own niche, however. Pruning works very well in the case that many rows contained in base tables do not contribute to the result. Incremental evaluation works well in the case that many rows only contribute small amounts to the intermediate tables. These are not completely independent, however: rows that have little intermediate impact may often have no impact on the output and will therefore be pruned.

To illustrate, consider the case of augmented path queries. The results in Figure 10 show that for this class of queries incremental evaluation actually slows minimization when combined with pruning. In this case, pruning is very effective, leaving very small tables to begin with. The joins are already very small making them very inexpensive. The naive method therefore has an advantage in having to only calculate one, versus the three that the incremental method must perform.

It is clear that these optimizations are not always beneficial. Although pruning only adds an extra evaluation to the minimization and thus incurs limited overhead, incremental evaluation can

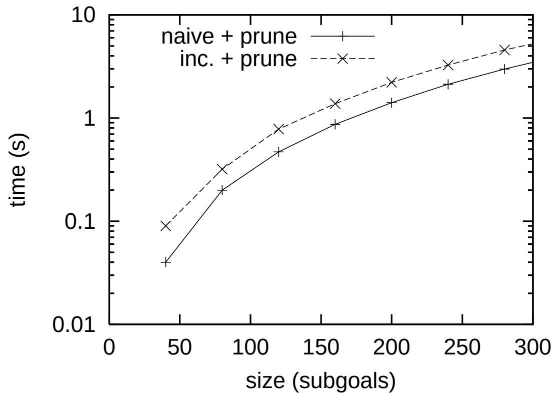

Figure 10: Augmented path queries minimized with the naive and incremental pruned approaches

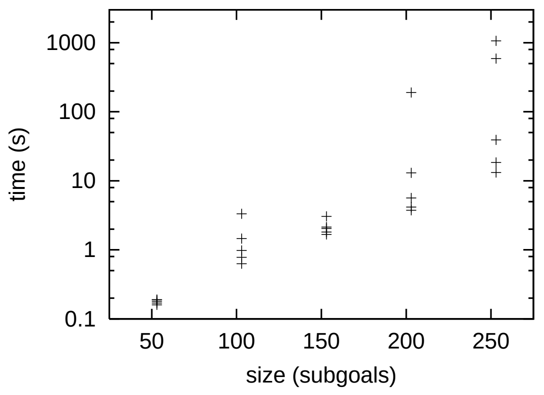

Figure 11: Augmented ladder queries minimized using the naive pruned method. Times are limited to 1800 seconds.

have a substantial cost depending on the structure of a particular query, and its use must be considered on a case-by-case basis.

7 Experiments and Results

In this section we examine how well our algorithm scales to large instances of several classes of queries.

Our implementation was written in C++ and compiled with gcc. All results were taken from a 1.7 GHz Intel Pentium 4 running Linux 2.4.7 with 1GB of main memory.

Tables are represented using hash sets from the Standard Template Library [25], and joins are implemented as hash-joins. Along with each table a count of the number of rows present is kept. This count is implemented using the arbitrary precision integers from the Gnu Multiple Precision Library [1].

As we saw in the previous section, our naive evaluation method paired with pruning performed the best on linear augmented ladder queries. In Figure 11 we show how this method performs as the size of the query grows into the hundreds of subgoals. In these trials we chose the lowest-width plan out of 50 generated.

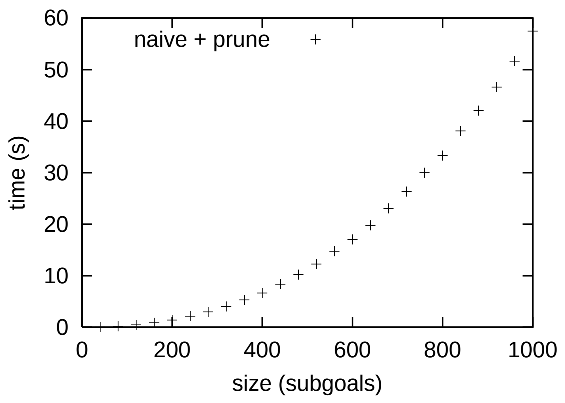

Figure 12: Augmented path queries minimized with the naive pruned approach

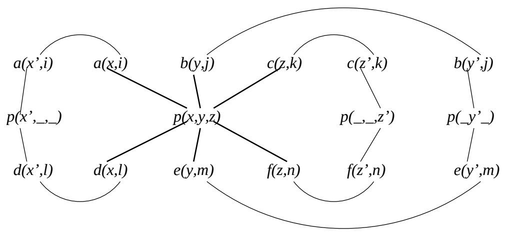

Figure 13: A redundant star query with a central predicate of arity 3 and one set of side loops. The central star is highlighted.

We see that for queries under 200 subgoals in length, the algorithm behaves quite well. For queries larger than 200 subgoals, however, the algorithm starts to exhibit poor results, as the probability of splitting rungs in the query grows, yielding larger plan-widths.

Although augmented ladder queries exhibit many qualities that make them useful in differentiating between our algorithms, they are not a natural class of queries. Path, star, and snowflake queries are more common construct found in actual queries.

We found that pruning was very effective in this case, and the best method was our naive pruned implementation. In Figure 12 we see that using pruning we can scale to minimize a 1000-subgoal query of this form in only 57.5 seconds.

We also consider the embellished star queries such as the one shown in Figure 13. These queries have a wide central subgoal, with each variable joined to a pair of side goals. This is made made more complex by adding a number of extra loops between these side goals, each through an extra, redundant copy of the central subgoal with its variables renamed. Such queries are often found in data warehousing applications.

The results in Figure 14 show the minimization times for a star query with a central predicate of arity 20, and between 1 and 20 complete sets of side loops. Although all of the methods performed roughly the same on this class of queries, the incremental pruned approach was somewhat better, and we have presented these results.

Despite the fact that the large central predicate induces large plan widths (roughly 40), this class is very fast to minimize, taking only 27.9 seconds to minimize an 861-subgoal query. This is due in part to the fact that there are several predicates, and these do not contribute rows to each other in the database. More fundamentally, although the central predicate is large, there are not

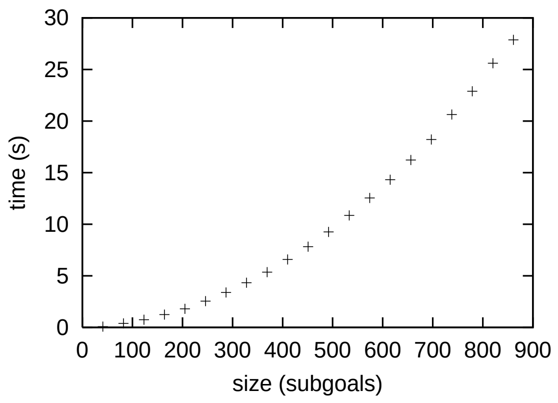

Figure 14: Redundant star queries minimized with the incremental pruned approach

many valid valuations for it; just because a table could contain a large number of distinct rows does not mean that these actually occur in the evaluation of a query.

8 Related Work

The problem of query containment on which our system is based dates to the work of Chandra and Merlin [8].

Subsequent work has considered containment for various extensions of conjunctive queries [21, 3], datalog [24, 22, 9], queries with order predicates [17, 28, 19, 30, 16], with complex objects [20]. with regular expressions [13], and under bag semantics [10].

In [7], Bodlaender shows that given a constant bound on the treewidth, there is a linear-time algorithm to recognize whether a graph has a treewidth less than that bound and produce a decomposition with that width. This algorithm is, however, exponential in the treewidth, and the constant factors are too large for it to be useful in practice.

In [11], Chekuri and Rajaraman introduce the notion of query-decomposition and its related notion of width as a a refinement of tree decomposition in the context of query execution.

Gottlob et al., show in [14] that unlike treewidth, it is NP-complete to recognize queries of even a constant width. They also introduce the notion of hypertree-decomposition and the associated concept of hypertree-width which further refine the bound on the work to execute a query.

Recently, the minimization problem has been considered for XPath expressions in [4].

9 Conclusions and Future Work

Although NP-hard in general, the query minimization problem is actually an instance of the quite practical query optimization/evaluation problem. However, unlike the usual setting, in this case the query is large, the database is quite small and has no indexes nor statistics, and we need to evaluate the query repeatedly on a sequence of decreasing database instances. This makes many of the traditional optimization methods useless.

We have proposed a fast *heuristic-based* query plan generation algorithm, designed specifically for query minimization. Our inspiration comes from recent theoretical work on tree decomposition: we generate a random spanning tree in the query graph, and construct the plan from that tree. Next, we have proposed an optimization to this query plan generation, based on *randomization*:

we run the (randomized) query plan generation several times, then pick the best generated plan. Our experiments show that randomization results in a dramatic reduction in the variation of the quality of the generated plans, and, hence in the total query minimization time.

Finally, we have proposed a number of run-time optimization that speed up the repeated evaluation of the same query plan on a sequence of database instances. The two major ones are *table pruning*, and *incremental evaluation*. The first eliminates tuples from all intermediate tables that do not participate to the end result. In theoretical terms, it eliminates partial homomorphisms that cannot be completed to total homomorphisms. Pruning needs to be done only once during the minimization, then all subsequent iterations will benefit from it. The second is *incremental evaluation*. This is based on previous work on incremental view maintenance, and computes on the changes to each intermediate table as the database decreases.

Put together, these techniques allowed us to scale the optimization algorithm to queries with hundreds of joins, for several realistic types of queries.

Future research involves improving two different aspects of our work. First, although randomization works in many cases, it would seem that as many queries grow the proportion of low-width plans goes down. To find these plans we will need a more directed search algorithm

Second, although our tests are encouraging, they consist entirely of artificial queries. We would like to use real redundant queries generated by various applications that use view expansion, providing a better idea of how our algorithms would be used in practice.

References

[1] Gnu multiple precision library. http://www.swox.com/gmp/.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley Publishing Co, 1995.

[3] Alfred Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalence of relational expressions. *SIAM Journal of Computing*, (8)2:218–246, 1979.

[4] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 497–508, 2001.

[5] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of ACM SIGMOD Conference on Management of Data*, May 1986.

[6] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[7] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[8] Ashok Chandra and Philip Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of 9th ACM Symposium on Thoery of Computing*, pages 77–90, Boulder, Colorado, May 1977.

[9] Surajit Chaudhuri and Moshe Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *The Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, CA.*, pages 55–66, 1992.

[10] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 59–70, Washington, D. C., May 1993.

[11] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 1997.

[12] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.

[13] Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 139–148, 1998.

[14] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *PODS*, pages 21–32, 1999.

[15] Timothy Griffin and Leonid Libkin. Incremental mainenance of views with duplicates. In *International Conference on Management of Data*, pages 328–339, San Jose, California, June 1995.

[16] Ashish Gupta, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint checking with partial information. In *Proceedings of the Thirteenth Symposium on Principles of Database Systems (PODS)*, pages 45–55, 1994.

[17] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, pages 35(1): 146–160, 1988.

[18] P. Kolaitis and M. Vardi. Conjunctive-query containment and constraint satisfaction. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1998.

[19] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In *Proceedings of the 19th VLDB Conference, Dublin, Ireland*, pages 171–181, 1993.

[20] Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects and aggregations. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona.*, 1997.

[21] Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1981.

[22] Yehoshua Sagiv. Optimizing datalog programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, Los Altos, CA, 1988.

[23] J. Shanmugasundaram, , J. Kiernana, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proceedings of VLDB*, pages 261–270, Rome, Italy, September 2001.

[24] Oded Shmueli. Equivalence of datalog queries is undecidable. *Journal of Logic Programming*, 15:231–241, 1993.

[25] A. Stepanov and M. Lee. The standard template library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[26] Jeffrey D. Ullman. *Principle of Database Systems.* Pitman, 2nd edition, 1982.

[27] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems II: The New Technologies.* Computer Science Press, Rockvill, MD 20850, 1989.

[28] Ron van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *The Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, CA.*, pages 331–345, 1992.

[29] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Zaniolo and Delobel(eds)*, 1981.

[30] X. Zhang and M. Z. Ozsoyoglu. On efficient reasoning with implication constraints. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 236–252, 1993.