Optimizing Recursive Information Gathering Plans

Eric Lambrecht & Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85287
Corresponding author email: rao@asu.edu

Abstract

In this paper we describe two optimization techniques that are specially tailored for information gathering. The first is a greedy minimization algorithm that minimizes an information gathering plan by removing redundant and overlapping information sources without loss of completeness. We then discuss a set of heuristics that guide the greedy minimization algorithm so as to remove costlier information sources first. In contrast to previous work, our approach can handle recursive query plans that arise commonly in practice. Second, we present a method for ordering the access to sources to reduce the execution cost. Sources on the Internet have a variety of access limitations and the execution cost in information gathering is affected both by network traffic and by the connection setup costs. We describe a way of representing the access capabilities of sources, and provide a greedy algorithm for ordering source calls that respects source limitations. It also takes both access costs and traffic costs into account, without requring full source statistics. Finally, we will discuss implementation and empirical evaluation of these methods in *Emerac*, our prototype information gathering system.

1 Introduction

The explosive growth and popularity of the world-wide web have resulted in thousands of structured queryable information sources on the Internet, and the promise of unprecedented information-gathering capabilities to lay users. Unfortunately, the promise has not yet been transformed into reality. While there are sources relevant to virtually any user-queries, the morass of sources presents a formidable hurdle to effectively accessing the information. One way of alleviating this problem is to develop *information gatherers* (also called mediators) which take the user's query, and develop and execute an effective *information gathering plan*, that accesses the relevant sources to answer the user's query efficiently.[1] Figure 1 illustrates the typical architecture of such a system for integrating diverse information sources on the internet. Several first steps have recently been taken towards the development of a theory of such gatherers in both database and artificial intelligence communities.

The information gathering problem is typically modeled by building a virtual global schema for the information that the user is interested in, and describing the accessible information sources as materialized views on the global schema.[2] The user query is posed in terms of the relations of the global schema. Since the global schema is virtual (in that its extensions are not stored explicitly anywhere), computing the query requires rewriting (or "folding" [21]) the query such that all the EDB predicates in the rewrite correspond to the materialized view predicates that represent information sources. Several researchers [19, 21, 15] have addressed this rewriting problem. Recent research by Duschka and his co-workers [6, 7] subsumes most of this work, and provides a clean methodology for constructing information gathering plans for user queries posed in terms of a global schema The plans produced by this methodology are "maximally contained" in that any other plan for answering the given query is contained in them.[3]

Generating source complete plans however is only a first step towards efficient information gathering. A crucial next step, which we focus on in this paper, is that of query plan optimization. Maximally contained plans produced by Duschka's methodology are conservative in that they in essence wind up calling any information source that may be remotely relevant to the query. Given the autonomous and decentralized nature of the Internet, sources tend to have significantly overlapping contents (e.g. mirror sources), as well as varying access costs (premium vs. non-premium sources, high-traffic vs. low-traffic sources). Naive execu-

[1] Notice that this is different from the capability provided by the existing search engines, which supply a list of pointers to the relevant sources, rather than return the requested data.

[2] See [17] for a tutorial.

[3] In other words, executing a maximally contained plan guarantees the return of every tuple satisfying the query that can be returned by executing any other query plan.

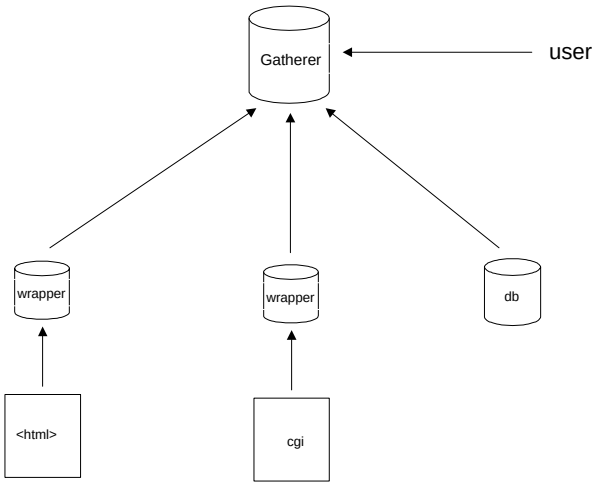


Figure 1: The information gatherer acts as an intermediary between the user and information sources on the Internet.

tion of maximally contained plans will generate all relevant sources and be prohibitively costly, in terms of the network traffic, response time, and access costs (in the case of "premium" sources that charge for access).

At first blush, it would seem that we should be able to directly apply the rich body of work on query optimization in databases [3]. Unfortunately, this does not work because many of the assumptions made in the traditional database query optimization do not hold in information gathering scenarios. To begin with, in traditional databases, redundancy and overlap among different sources is not a major issue, while it is a very crucial issue in information gathering. Similarly, traditional query optimization methods depend on elaborate statistical models (histograms, selectivity indices etc.) of the underlying databases. Such statistical models are not easily available for sources on the Internet. [4] Finally, even the work on optimizing queries in the presence of materialized views (c.f. [4]) is not directly relevant as in such work materialized views are assumed to be available *in addition* to the main database. In contrast, the global database in information gathering is "virtual" and the only accessible information resides in materialized views whose statistical models are not easily available. For all these reasons, it is now generally recognized (c.f. [10]) that query optimization for information gathering is a very important open problem.

In this paper we describe the query optimization techniques that we have developed in the context of *Emerac*, a prototype information gathering system under development. Figure 2 provides a schematic illustration of the query planning and optimization process in *Emerac*. We start by generating a query plan using the source inversion techniques described Duschka [6, 7]. This polynomial time process gives us a "maximally contained" query plan which

[4] It would of course be interesting to try and "learn" the source statistics through judicious probing. See [27] for a technique that does it in the context of multi-databases.

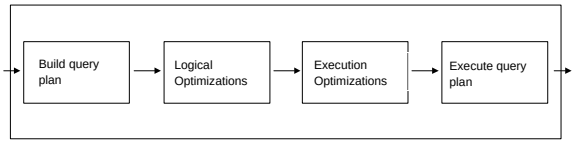


Figure 2: The full process of query planning

serves as the input for the optimization methods. As in traditional databases, our optimization phase involves two steps: logical optimization and execution optimization. In traditional databases, logical optimization involves rewriting a query plan, using relational algebra equivalences, to make it more efficient; while execution optimization involves steps such as ordering the accesss to the base relations to make computations of joins cheaper. For *Emerac*, the logical optimization step involves minimizing the maximally contained query plan such that access to redundant sources is removed. Execution optimization involves ordering the access to the information sources in the minimized plan so as to reduce execution cost.

Our contributions: For logical optimization, we present a technique that operates on the recursive plans generated by Duschka's algorithm and greedily minimizes it so as to remove access to costly and redundant information sources, without affecting the completeness of the plan. For this purpose, we use the so-called localized closed world (LCW) statements that characterize the completeness of the contents of a source relative to either the global (virtual) database schema or the other sources. Our techniques are based on an adaptation of Sagiv's [22] method for minimizing datalog programs under uniform equivalence. Although there exists some previous research on minimizing information gathering plans using LCW statements [5, 11], none of it is applicable to minimization of information gathering plans containing recursion. Our ability to handle recursion is significant because recursion appears in virtually all information gathering plans either due to functional dependencies, binding constraints on information sources, or recursive user queries [6]. Additionally, in contrast to existing methods, which do pairwise redundancy checks on source accesses, our approach is capable of exploiting cases where access to one information source is rendered redundant by access to a combination of sources together. Large performance improvements in our prototype information gatherer, *Emerac*, attest to the cost-effectiveness of our minimization approach.

Ultimately plan execution in our context boils down to doing joins between the sources efficiently. When gathering information on the Internet, we typically cannot instruct two sources to join with each other. It is thus necessary to order the access to the sources. The existing methods for subgoal ordering assume that the plan is operating on a single "fully relational" (i.e., no binding restrictions) database, and that the plan execution cost is dominated by the number of tuples transferred. In contrast, sources on the Internet have a variety of access limitations and the execution

cost in information gathering is affected significantly by the connection setup costs. We describe a way of representing the access capabilities of sources, and provide a greedy algorithm for ordering source calls that respects source limitations, and takes both access costs and traffic costs into account, without requring full source statistics.

Although there exist other research efforts that address source redundancy elimination and optimization in the presence of sources with limited capabilities, *Emerac* is the first to consider end-to-end the issues of redundancy elimination and optimization in recursive information gathering plans. It is also the first system system to consider the source access costs as well as traffic costs together in doing optimization.

Organization: The rest of the paper is organized as follows. In Section 2, we review the work on integrating diverse information sources by modeling them as materialized views on a virtual database. We pay special attention to the work of Duschka [6, 7], which forms the basis for our own work. Section 3 briefly reviews the use of LCW statements and Sagiv's algorithm for datalog program minimization under uniform equivalence. Section 4 presents our greedy minimization algorithm that adapts Sagiv's algorithm to check of source redundancy in the context of the given LCW statements. We also explain how the inter-source subsumption relations can be exploited in addition to LCW statements (Section 4.1). We then discuss the complexity of the minimization (Section 4.3), and present heuristics for biasing the greedy minimization strategy (Section 4.2). Section 5 describes our algorithm for ordering source accesses during execution. Section 6 describes the architecture of *Emerac*, our prototype information gatherer, and presents an empirical evaluation of the effectiveness of our optimization techniques. Section 7 discusses related work. Section 8 presents our conclusions, and outlines our current research directions.

2 Building Query Plans: Background

Suppose our global schema contains the world relation $advisor(S, A)$, where A is the advisor of S. Further more, suppose we have an information source ADDB, such that for every tuple (S, A) returned by it, A is the advisor of S. This can be represented as a materialized view on the global schema as follows:

ADDB*(S,A)* → *advisor(S, A)*

Suppose we want to retrieve all the students advised by Weld. We can represent our goal by the query $\mathcal{Q}$:

query(S, A) :- *advisor(S, A)* ∧ *A = "Weld"*

Dushcka et. al. [6, 7] show how we can generate an information gathering plan that is "maximally contained" in that it returns every query-satisfying tuple that is stored in any of the accessible information sources. This method works by *inverting* all source (materialized view) definitions, and adding them to the query. The inverse, v^{-1}, of the materialized view definition with head $v(X_1, \ldots, X_m)$ is a set of logic rules in which the body of each new rule is the head of the original view, and the head of each new rule is a relation from the body of the original view. When we invert our definition above, we get:

advisor(S,A) :- ADDB*(S,A)*

When this rule is added to the original query $\mathcal{Q}$, we effectively create a datalog[5] program whose execution produces all the tuples satisfying the query.

2.1 Constrained sources & Recursion

The materialized view inversion algorithm can be modified in order to model databases that have binding pattern requirements. Suppose we have a second information source, CONDB that requires the student argument to be bound, and returns the advisor of that given student. We denote this in its view as follows:

CONDB*($S, A)* → *advisor(S, A)*

The '$' notation denotes that *S* must be bound to a value for any query sent to CONDB. A straightforward inversion of this source will get us a rule of the form:

advisor(S,A) :- CONDB*($S, A)*

which is unexecutable as S is not bound. This is handled by making up a new relation called dom whose extension is made to correspond to all possible constants that can be substituted for S. In our example, assuming that we have both the ADDB source and the CONDB source, the complete plan for the query, which we shall refer to as $\mathcal{P}$, is:

r1: query(S, A) :- *advisor(S , A)* ∧ *A="Weld"*
r2: advisor(S, A) :- ADDB*(S, A)*
r3: advisor(S, A) :- *dom(S)* ∧ CONDB*(S, A)*
r4: dom(S) :- ADDB*(S, A)*
r5: dom(A) :- ADDB*(S, A)*
r6: dom(A) :- *dom(S)* ∧ CONDB*(S, A)*

Notice that all extensional (EDB) predicates in the progam correspond to source predicates (materialized views). Notice also the presence of $dom(S)$ relation in the rule $r3$. Rules $r4, r5$ and $r7$ define the extension of dom by collecting all possible constants that can be derived from source calls. Finally, note that rule $r6$ is recursive, which makes the overall plan recursive, *even though* the original query as well as the source description are non-recursive. Given the ubiquitousness of constrained sources on the Internet, it is thus important that we know how to handle recursive information gathering plans.

[5] Things get a bit more complicated when there are variables in the body of the view that do not appear in the head. During inversion, every such variable is replaced with a new function term $f_N(X_1, \ldots, X_m)$. The function symbols can then be eliminated by a flattening procedure, as there will be no recursion through them in the eventual plan, resulting in a datalog program in the end [6].

3 Plan minimization preliminaries

The plan $\mathcal{P}$ above accesses two different advisor databases to answer the query. It would be useful to try and cut down redundant accesses, as this would improve the execution cost of the plan. To do this however, we need more information about the sources. While the materialized view characterizations of sources explicate the world relations that are respected by each tuple returned by the source, there is no guarantee that all tuples satisfying those properties are going to be returned by that source.

One way to support minimization is to augment the source descriptions with statements about their relative coverage, using the so-called localized closed world (LCW) statements [8]. An LCW statement attempts to characterize what information (tuples) the source is *guaranteed* to contain in terms of the global schema. Suppose, we happen to know that the source ADDB is guaranteed to contain all the students advised by Weld and Hanks. We can represent this information by the statement (note the direction of the arrow):

ADDB*(S, A)* ← *advisor(S, A)* ∧ *A="Weld"*
ADDB*(S, A)* ← *advisor(S, A)* ∧ *A="Hanks"*

Information sources might also have pairwise or n-ary LCW statements among themselves. That is, the completeness of one or more information sources could be defined in terms of one or more other information sources. The following statement means that the source $sourceD$ contains all the information in $sourceE$:

SOURCED*(X, Y)* ← SOURCEE*(X,Y)*

This is particularly useful in modeling "mirror sites" on Internet, which guarantee keeping a complete and up-to-date copy of another source. Inter-source LCW statements were first proposed in [11].

3.1 Pair-wise rule subsumption

Given the LCW statement above, intuitively it is obvious that we can get all the tuples satisfying the query $\mathcal{Q}$ by accessing just ADDB. We now need to provide an automated way of making these determinations. Suppose we have two datalog rules, each of which has one or more materialized view predicates in its body that also have LCW statements, and we wish to determine if one rule subsumes the other. The obvious way of checking the subsumption is to replace the source predicates from the first rule with the bodies of their view description statements, and the source predicates from the second rule with the bodies of the LCW statements corresponding to those predicates. We now have the transformed first rule providing a "liberal" bound on the tuples returned by that rule, while the transformed second rule gives a "conservative" bound. If the conservative bound subsumes the liberal bound, i.e., if the transformed second rule "contains" (entails) the transformed first rule, we know that second rule subsumes the first rule. Duschka [5] shows that this check, while sufficient, is not a necessary condition for subsumption. He proposes a modified version that involves replacing each source predicate s with $s \wedge v$ in the first rule, and with $s \vee l$ in the second rule, where v is the view description of s, and l is the conjunction of LCW statements of s. If after this transformation, the second rule contains the first, then the first rule is subsumed by it.[6]

3.2 Minimization under uniform equivalence

Pair-wise rule subsumption checks alone are enough to detect redundancy in non-recursive plans [18, 11], but are inadequate for minimizing recursive plans. Specifically, recursive plans correspond to infinite union of conjunctive queries and checking if a particular rule of the recursive plan is redundant will involve trying to see if that part is subsumed by any of these infinite conjuncts [23, pp. 908]. We instead base our minimization process on the notion of uniform containment for datalog programs, presented in [22]. To minimize a datalog program, we might try removing one rule at a time, and checking if the new program is equivalent to the original program. Two datalog programs are equivalent if they produce the same result for all possible assignments of EDB predicates [22]. Checking equivalence is known to be undecidable. Two datalog programs are uniformly equivalent if they produce the same result for all possible assignments of EDB and IDB predicates. Uniform equivalence is decidable, and implies equivalence. Sagiv [22] offers a method for minimizing a datalog program under uniform equivalence that we illustrate by an example (and later adapt for our information gathering plan minimization). Suppose that we have the following datalog program:

r1: p(X) :- *p(Y)* ∧ *j(X, Y)*
r2: p(X) :- *s(Y)* ∧ *j(X, Y)*
r3: s(X) :- *p(X)*

We can check to see if *r1* is redundant by removing it from the program, then instantiating its body to see if the remaining rules can derive the instantiation of the head of this rule through simple bottom-up evaluation. Our initial assignment of relations is *p("Y"), j("X", "Y")* . If the remaining rules in the datalog program can derive *p("X")* from the assignment above, then we can safely leave rule *r1* out of the datalog program. This is indeed the case. Given *p("Y")* we can assert *s("Y")* via rule *r3*. Then, given *s("Y")* and *j("X", "Y")*, we can assert *p("X")* from rule *r2*. Thus the above program will produce the same results without rule *r1* in it.

4 Greedy Minimization of Recursive plans

We now adapt the algorithm for minimizing datalog programs under uniform equivalence to remove redundant sources and unnecessary recursion from the information gathering plans. Our first step is to transform the query plan such that the query predicate is directly related to the

[6] The next section contains an example illustrating this strategy.

Replace all global schema predicates in $\mathcal{P}$
with bodies of their inversion rules.
repeat
let r be a rule in $\mathcal{P}$ that has not yet been considered
let $\hat{\mathcal{P}}$ be the program obtained by deleting rule r from $\mathcal{P}$
and simplifying it by deleting any unreachable rules.
let $\hat{\mathcal{P}}'$ be $\hat{\mathcal{P}}[s \mapsto s \vee l]$
let r' be $r[s \mapsto s \wedge v]$
if there is a rule, r_i in r',
such that r_i is uniformly contained by $\hat{\mathcal{P}}'$
then replace $\mathcal{P}$ with $\hat{\mathcal{P}}$
until each rule in $\mathcal{P}$ has been considered once

Figure 3: The greedy plan minimization algorithm

source calls. This is done by removing global schema predicates, and replacing them with bodies of inversion rules that define those predicates (see [23, Sec. 13.4]).Note that this step is safe because there is no recursion through global schema predicates. This step also removes any new predicates introduced through flattening of function symbols. Our example plan $\mathcal{P}$, after this transformation looks as follows:

r2: query(S, A) :- *adDB(S , A)* $\wedge$ *A="Weld"*
r3: query(S, A) :- *dom(S)* $\wedge$ CONDB*(S, A)* $\wedge$ *A="Weld"*
r4: dom(S) :- ADDB*(S, A)*
r5: dom(A) :- ADDB*(S, A)*
r6: dom(A) :- *dom(S)* $\wedge$ CONDB*(S, A)*

We are now ready to consider minimization. Our basic idea is to iteratively try to remove each rule from the information gathering plan. At each iteration, we use the method of replacing information source relations with their views or LCW's as in the rule subsumption check (see previous section) to transform the removed rule into a representation of what could possibly be gathered by the information sources in it, and transform the remaining rules into a representation of what is guaranteed to be gathered by the information sources in them. Then, we instantiate the body of the transformed removed rule and see if the transformed remaining rules can derive its head. If so, we can leave the extracted rule out of the information gathering plan, because the information sources in the remaining rules guarantee to gather at least as much information as the rule that was removed. The full algorithm is shown in Figure 3.

For our example plan above, we will try to prove that rule *r3*, containing an access to the source CONDB, is unnecessary. First we remove *r3* from our plan, and then transform it and the remaining rules so they represent the information gatherered by the information sources in them. For the removed rule, we want to replace each information source in it with a representation of all the possible information that the information source could return. Specifically, we want to transform it to $r[s \mapsto s \wedge v]$. This produces:

query(S,A) :- *dom(S)* $\wedge$ CONDB*(S, A)*
$\wedge$ *advisor(S, A)* $\wedge$ *A="Weld"*

For the remaining rules, P, we transform them into $P[s \mapsto s \vee l]$, which represents the information guaranteed to be produced by the information sources in the rules. For our example, we produce:

r21: query(S, A) :- ADDB*(S, A)* $\wedge$ *A="Weld"*
r22: query(S, A) :- *advisor(S, A)* $\wedge$ *A="Weld"*
r23: query(S, A) :- *advisor(S, A)* $\wedge$ *A="Hanks"*
dom(S) :- ADDB*(S, A)*
dom(S) :- *advisor(S, A)*
dom(A) :- ADDB*(S, A)*
dom(A) :- *advisor(S, A)*
dom(A) :- *dom(S)* $\wedge$ CONDB*(S, A)*
dom(A) :- *dom(S)* $\wedge$ *advisor(S, A)*

When we instantiate the body of the transformed removed rule $r3$, we get the ground terms: *dom("S"), conDB("S", "A"), A="Weld", advisor("S", "A").* After evaluating the remaining rules given with these constants, we find that we can derive *query("S", "A")*, using the rule $r22$, which means we can safely leave out the rule $r3$ that we've removed from our information gathering program.

If we continue with the algorithm on our example problem, we will not be able to remove any more rules. The remaining *dom* rules can be removed if we do a simple reachability test from the user's query, as they are not referenced by any rules reachable from the query.

4.1 Handling inter-source subsumption relations

The algorithm above only makes use of LCW statements that describe sources in terms of the global schema. It is possible to incorporate inter-source subsumption statements into the minimization algorithm. Specifically, suppose we are considering the removal of a rule r containing a source relation s from the plan P. Let U be the set of inter-source subsumption statements that have s in the tail, and $U^{\leftarrow\mapsto:-}$ be the statements of U with the $\leftarrow$ notation replaced by $:-$ notation (so U is a set of datalog rules). We have to check if $r[s \mapsto s \wedge v]$ is uniformly contained in $(P - r + U^{\leftarrow\mapsto:-})[s \mapsto s \vee l]$ If so, then we can remove r.

As an example, suppose we know that $s1$ and $s2$ are defined by the views:

s1(x) $\rightarrow$ *r(x)*
s2(x) $\rightarrow$ *r(x)*

Suppose we know that s1 contains all tuples that $s2$ contains. This corresponds to the statement

s1(x) $\leftarrow$ *s2(x)*

Suppose we have the query:

Q(x) :- *r(x)*

The corresponding maximally contained plan P will be:

Q(x) :- *r(x)*
r(x) :- *s1(x)*
r(x) :- *s2(x)*

To recognize that we can remove third rule from this plan because of the source subsumption statements, we check if that rule is uniformly contained in the program

$(P - r +$ "$s1(x)$: $-$s2(x))$''$, which is:

Q(x) :- *r(x)*
r(x) :- *s1(x)*
s1(x) :- *s2(x)*

The uniform containment holds here since if we add the tuple $s2(A)$ to the program, bottom up evaluation allows us to derive $s1(A)$ and subsequently $r(A)$, thus deriving the head of the removed rule.

4.2 Heuristics for guiding the minimization

The final information gathering plan that we end up with after executing the minimization algorithm will depend on the order in which we remove the rules from the original plan. In the example of the previous section, we could have removed *r2* from the original information gathering plan before *r3*. Since both rules will lead to the generation of the same information, the removal would succeed. Once *r2* is removed however, we can no longer remove *r3*. Note that the plan with rule *r3* in it is much costlier to execute than the one with rule *r2* in it. This is because the presence of *r3* triggers the *dom* recursion through rules *r4* $\cdots$ *r6*, which would have been eliminated otherwise. Recursion greatly increases the execution cost of the plan, as it can generate potentially boundless number of accesses to remote sources (see Section 6.2).

Given that the quality of the plan returned by the minimization algorithm is sensitive to the order in which the rules are considered, we have two alternatives. The first is to do an exhaustive search with all possible removal orderings, and the second is to bias the greedy minimization algorithms with appropriate heuristics so it is more likely to find good quality plans. The exhaustive search option in the worst case involves considering all permutations of rule removal, and is prohibitively expensive given that each removal check is worst case exponential to begin with.

So we concentrate on heuristics for selecting the rules to be removed. Good heuristics order the rules by information source execution cost, from highest to lowest. The cost of the access to a source depends on several factors, including:

Contact time: The time it takes to contact a source. Depends on such factors as whether the source is high-traffic vs. low-traffic.

Query time: The time it takes to send the query. This essentially includes the time taken for filling the "forms" for sources with forms-based interface.

Tuple response time: The time it takes to return first tuples of the answer to the query. This is useful when the user is interested in getting at least the first few answers fast.

Probability of access: The probability that you can even contact the information source

Cost of access: Any monetary cost for accessing this source (in case of premium sources).

By keeping track of rudimentary access statistics from past accesses to the sources (c.f. [1]), we can get a reasonable qualitative estimates of these cost factors. The exact way in which these costs should be combined to get a global cost depends upon the general objectives of the user (e.g., minimize the time taken to get all the tuples, minimize the time taken to get the first n tuples, or minimize the monetary cost for getting the tuples etc.)

In addition to the above source-specific costs, there are certain general heuristics for rule removal that are valid most of the time. These include:

- Rules which have a *dom* term should have a large extra cost added to them, since recursion often arises due to *dom* rules, and recursion implies high execution time (in terms of connection setup costs; see below).[7]

- In considering rules for removal, we should first consider source inversion rules (those which have a predicate of the global schema in their head and some source relations in their tail). The *dom* rules should be considered after source inversion rules are considered since many *dom* rules may be removed by reachability analysis if the rules that have *dom* predicates get removed.

4.3 Complexity of the minimization algorithm

The complexity of the minimization algorithm in Figure 4 is dominated by the cost of uniform containment checks. The containment check is called at most once per rule in the information gathering plan. The other phases of the algorithm, such as the translation using LCW statements, and the reachability analysis to remove unreachable rules, can all be done in polynomial time.

As Sagiv [22] points out, the running time of the uniform containment check is in the worst case exponential in the size of the query plan being minimized. However, things are brighter in practice for two reasons. First, if the arity of the predicates is bounded (as is often the case in practice) then the running time is polynomial. Second, the exponential part of the complexity comes from the "evaluation" of the datalog program. The evaluation here is done with respect to a "small" database –consisting of the grounded literals of the tail of the rule being considered for removal.

5 Ordering source calls during Execution

A crucial practical choice we have to make during the evaluation of datalog programs is the order in which predicates

[7] There is a tradeoff between this idea of trying to get rid of sources with binding restrictions, and the goal of reducing network traffic. Sources with binding restrictions tend to send less data over the network and thus reduce the network traffic, but introducing them into the plan may also lead to costly recursion which can worsen the execution time arbitrarily by increasing the connection costs; see Section 5.

are evaluated. Our objective is to reduce the "cost" of execution, where cost is a function of the access cost (including connection time), traffic costs (the number of tuples transferred), and processing cost (the time involved in processing the data). Typcially, traffic and processing costs are closely correlated.

In our cost model, we assume that the access cost, dominates the other terms. This is a reasonable assumption given the large connection setup delays involved in accessing sources on the Internet. While the traffic costs can also be significant, this is offset to some extent by the fact that many data sources on the Internet do tend to have smaller extractable tables.[8]

Although the source call ordering problem is similar to the "join ordering" phase in the traditional database optimization algorithms [3], there are several reasons why the traditional as well as distributed-database techniques are not suitable:

- Join ordering algorithms assume that all sources are relational databases. The sources on the Internet are rarely fully relational and tend to support limited types of queries. These limitations need to be represented and respected by the join ordering algorithm.

- Join ordering algorithms in distributed databases typically assume that the cost of query execution is dominated by the number of tuples transferred during execution. Thus, the so-called "bound-is-easier" assumption makes good sense. In the Internet information gathering scenario, the cost of accessing sources tends to dominate the execution cost. We thus we cannot rely solely on the bound-is-easier assumption and would need to consider the number of source calls.

- Typically, join ordering algorithms use statistics about the sizes of the various predicates to compute an optimal order of joining. These techniques are not applicable for us as our predicates correspond to source relations, about which we typically do not have complete statistics.

- The fact that source latencies make up a significant portion of the cost of execution argues for parallel (or "bushy") join trees instead of the "left-linear" join trees considered by the conventional algorithms [3].

5.1 Representing source access capabilities

As we mentioned, in the information gathering domain, the assumption that information sources are fully relational databases is not valid. An information source may now be a wrapped web page, a form interfaced database, or a fully relational database. A wrapped web page is a WWW document interfaced through a wrapper program to make it appear as a relational database. The wrapper retrieves the web page, extracts the relational information from it, then answers relational queries. Normal selection queries are not supported. A form-interfaced database refers to a database with an HTML form interface on the web which only answers selection queries over a subset of the attributes in the database. A WWW airline database that accepts two cities and two dates and returns flight listings is an example of a form interfaced database.

We devised a simple way to inform the gatherer as to what types of queries on an information source would accept. We use the "$" annotation to identify bound attribute requirements on the source and "%" annotation to identify unselectable attributes. Thus a fully relational source would be adorned $source(X, Y)$, a form interfaced webpage that only accepts bindings for its first argument would be adorned $source(X, \%Y)$, while a wrapped web-page source would have all its attributes marked unselectable, represented as $source(\%X, \%Y)$. Finally, a form interfaced web-page that requires bindings for its first argument, and is able to do selections only on the second argument would be adorned $source(\$X, Y, \%Z)$.

The $ and % annotations are used to identify feasible binding patterns for queries on a source, to establish generality relations between two binding patterns, and to ensure that soundness is preserved in pushing variable selection constraints (such as "$X = 7$") into source calls. Given a source with annotations $S_1(\$X, \%Y, Z)$, only the binding patterns of the form S_1^{b--} are feasible (where "$-$" stands for either $b$ound or $f$ree argument). Similarly, we are not allowed to push selection constraints on Y to the source (they must be filtered locally). Thus the call S_1^{bbf} is modeled as S_1^{bff} filtered locally with the binding on Y.

Finally, a binding pattern S^p is more general than S^q (written $S^p \succ_g s^q$, if every selectable (non "%"-annotated) variable that is free in q is also free in p, but not *vice versa*. Thus, for the source S_1 above, the binding pattern S_1^{bbf} is more general than the binding pattern S_1^{bfb} (while such a relation would not have held without "%" annotations [23]). Intuitively, the more general a binding pattern, the higher the number of tuples returned by the source when called with that binding pattern. We ignore binding status of "%"-annotated variables since by definition they will not have any effect on the amount of data transferred.

5.2 A greedy algorithm for ordering source calls

In this section, we describe a greedy algorithm that produces bushy join trees. We note that the optimal plans will need to minimize:

$$\sum_s (C_a^s * n_s + C_t^s * D_s)$$

Where n_s is the number of times a source s has been accessed during the plan and C_a^s is the cost per access, and

[8] Even those that have large tables regulate their data output, by paginating the tuples and sending them in small quanta (e.g., first 10 tuples satisfying the query), which will avoid the network congestion if the users needed only a certain percentage of the tuples satisfying the query.

```
Inputs: FBP: table of forbidden binding patterns
        HTBP: table of high traffic binding patterns
        V := all variables bound by the head;
        C[1 ··· m]: Array where C[i] lists sources chosen at i^th stage;
        P[1 ··· m]: Arrays where P[i] lists sources postponed at i^th stage
for i := 1 to m (where m is the number of subgoals) do begin
        C[i] := ∅; P[i] := ∅;
        for each unchosen subgoal S do begin
            Generate all feasible binding patterns B for S w.r.t. V and FBP;
            Do a topological sort on B in terms of ≻_g relation
            for each b ∈ B examined from most to least general do begin
                if there exists b ∈ B s.t. b ∉ FBP and b ∉ HTBP,
                  then begin
                        Push S with binding pattern b into C[i];
                        Mark S as "chosen";
                        add to V all variables appearing in S;
                  end
            end
            if B ≠ ∅ and S is not chosen
                then Push S with the least general b ∈ B into P[i];
        end
        if C[i] = ∅ and P[i] ≠ ∅
            then begin
                Take any one element from P[i] and push it into C[i];
                add to V all variables appearing in S;
            else fail
        end
        Return the array C[1..i].
```

Figure 4: A greedy source call ordering algorithm that considers both access costs and traffic costs.

C_t^s is the per tuple transfer cost for source s, and D_s is the number of tuples transferred by s. We note that this cost metric imposes a tension between the desire to reduce network traffic, and the desire to reduce access costs. To elaborate, reducing the network traffic involves accessing sources with less general binding patterns. This in turn typically increases the number of separate calls made to a source, and thus increase the access cost. To illustrate this further, consider the subgoals:

$$\text{SOURCEA}(W, X) \wedge \text{SOURCEB}(X, Y)$$

Suppose that the query provides bindings for W. How should we access the sources? The conventional wisdom says that we should access SOURCEA first since it has more bound arguments. As a result of this access, we will have bindings for X which can then be fed into calls to SOURCEB. The motivation here is to reduce the costs due to network traffic. However, calling SOURCEA and using its outputs to bind the arguments of SOURCEB may also lead to a potentially large number of separate calls to SOURCEB (one per each of the distinct X values returned by **sourceA**), and this can lead to a significant connection setup costs, thus worsening the overall cost.

Exact optimization of the execution cost requires access to source selectivity statistics. As such statistics are not easily available, *Emerac*'s current optimization strategy avoids dependence on statistics. To simplify the optimization problem, we also assume that by default source access cost is the dominating cost of the plan, and thus concentrate primarily on reducing the access costs. If this assumption were always true, then we can issue calls to any source as soon as its binding constraints are met. Furthermore, we need only access the source with the most general feasible binding pattern. Such plans may sometimes lead to high costs in scenarios where the sources can transfer arbitarily large amounts of data for calls with sufficiently general binding patterns.

High-traffic Binding Patterns: To ensure that we don't get penalized excessively for concentrating primarily on access costs, we maintain a table called "HTBP" that lists, for each source, all binding patterns that are *known* to violate this assumption (i.e., cause high traffic by returning a large number of tuples). The general idea is to postpone calling a source as long as all the feasible binding patterns for that source supported by the currently bound variables are listed in the HTBP table. An underlying assumption of this approach is that while full source statistics are rarely available, one can easily gain partial information on the types of binding patterns that cause excessive traffic. For example, given a source that exports the relation $Book(Author, Title, ISBN, Subject, Price, Pages)$, we might easily find out that calls that do not bind at least one of the first four attributes tend to generate high traffic. There are two useful internal consistency conditions on HBTP. First, if S^b is listed in HBTP, then every $S^{b'}$ where $b' \succ_g b$ is also implicitly in HBTP. Similarly, if S^b is in HBTP, then it cannot be the case that the only free variables in b are all "%"-annotated variables.

A greedy algorithm to order source calls based on these ideas appears in Figure 4. It takes as input a rule with m subgoals and a given binding pattern for its head. The input also includes FBP, a table of forbidden binding patterns for each source (constructed from the $ annotations), and the table HTBP, which contains all source binding patterns that are known to be high-traffic producing. At each level i, the algorithm considers the feasible binding patterns of each unchosen source from most general to least general, until one is found that is not known to be high-traffic producing (i.e., not in HTBP). If such a binding pattern is found, that source, along with that binding pattern is added to the set of selected sources at that level (maintained in the array of sets $C[i]$). If not, the source, along with the least general binding pattern is temporarily postponed (by placing it in $P[i]$). If at the end of considering all unchosen sources at level i, we have not chosen at least one source (i.e., all of them have only high-traffic inducing binding patterns), then one of the sources from the list of postponed sources ($P[i]$) is chosen and placed in $C[i]$. Anytime a source is placed in $C[i]$, the set V of variables that currently have available bindings is updated. This ensures that at the next stage more sources will have feasible, as well as less general, binding patterns available. This allows the algorithm to progress since less general binding patterns are also less likely to be present in the HTBP table. Specifically, when $C[i]$ is empty and $P[i]$ is non-empty, we push only one source from $P[i]$ into $C[i]$ since it is hoped that the up-

dated V will then support non-high-traffic binding patterns at the later stages. (By consulting HTBP, and the set of unchosen sources, the selection of the source from $P[i]$ can be done more intelligently to ensure that the likelihood of this occurrence is increased).

When the algorithm terminates successfully, the array C specifies which sources are to be called in each stage, and what binding patterns are to be used in those calls. Execution involves issuing calls to sources in the specified binding pattern; where each bound variable in the binding pattern is instantiated to all values of that variable collected upto that point during execution. If the bound variable is a %-annotated variable, then the call is issued without variable instantiation, and the filtering on the variable values is done locally.

Notice that each element of C is a set of source calls with associated binding patterns (rather than a single source call)–thus supporting bushy joins. Thus, $C[i]$ may be non-empty for only a prefix of values from 1 to m. This parallelism cuts down the overall time wasted during connection delays.

The complexity of this ordering algorithm is $O(n^2)$ where n is the length of the rule. Calls with binding patterns that involve bindings to %-annotated variables are augmented with local selection tests.

6 Implementation and Evaluation

6.1 Architecture of *Emerac*

Emerac is written in the Java programming language, and is intended to be a library used by applications that need a uniform interface to multiple information sources. Full details of *Emerac* system are available in [16]. *Emerac* presents a simple interface for posing queries and defining a global schema. *Emerac* is internally split into two parts: the query planner and the plan executor. The default planner uses algorithms discussed in this paper, but it can be replaced with alternate planners. The plan executor can likewise be replaced, and the current implementation attempts to execute an information gathering plan in parallel after transforming it into a relational operator graph.

The query planner accepts and parses datalog rules, materialized view definitions of sources, and LCW statements about sources. Given a query, the query planner builds a source complete information gathering plan (using the method from [6]) and attempts to minimize it using the minimization algorithm presented in Section 4.

The optimized plan is passed to the plan executor, which transforms the plan into a relational operator graph. The plan executor makes use of "$" and "%"-adornments to determine the order to access each information source in a join of multiple sources, as described in this paper. The plan is executed by traversing the relational operator graph. When a *union* node is encountered during traversal, new threads of execution are created to traverse the children of the node in parallel. Use of separate threads also allows us to return answers to the user asynchronously.

Handling Recursion during execution: Since information gathering plans can contain recursion, handling recursive plans during execution becomes an important issue. Since each recursive call to a node in the r/g graph [23] can potentially generate an access call to a remote source, evaluating a program until it reaches fix point can get prohibitively expensive. Currently, we take a practical solution to this problem involving depth-limited recursion. Specifically, we keep a counter on each node in the r/g graph to record how many times the node has been executed. When the counter reaches a pre-specified depth-limit, the node would not be executed, and an empty set will be returned to represent the result of executing the node.

Wrapper Interface: *Emerac* assumes that all information sources contain tuples of information with a fixed set of attributes, and can only answer simple *select* queries. To interface an information source with *Emerac*, a Java class needs to be developed that implements a simple standard interface for accessing it. The information source is able to identify itself so as to provide a mapping between references to it in materialized view and LCW definitions and its code.

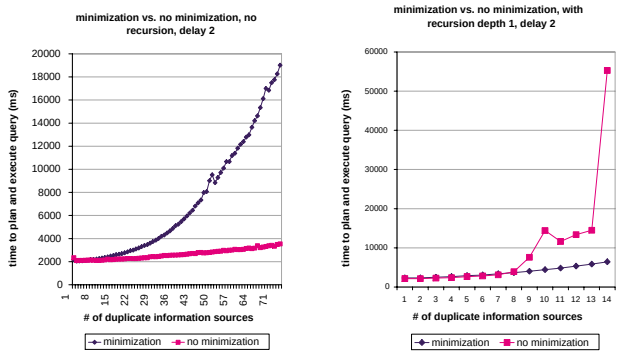
In order to facilitate construction of wrappers for web pages, a tool was created to convert the finite state machine based wrappers created by SoftMealy [14] into Java source code that can be compiled into information sources usable by *Emerac*. We have successfully adapted 28 computer science faculty listing web pages wrapped with SoftMealy into information sources usable by *Emerac*.

6.2 Empirical Evaluation

We used the prototype implementation of *Emerac* to evaluate the effectiveness of the optimization techniques proposed in this paper. To ensure experimental control, we built and used artificial information sources for our empirical study. The simulated sources delayed answering each query for a set period of time in order to simulate actual latency on the Internet. In all our experiments, this delay was set to 2 seconds, which is quite reasonable in the context of current day Internet sources.

To see how the planner and executor performed with and without minimization, we varied the number of duplicate information sources available and relevant to the query. The "*no-minimization*" method simply builds and executes source complete plans. The "*minimization*" method builds source complete plans, then applies the minimization algorithm described above before executing the plans. Both methods executed their plans in parallel where possible. The test cases were such that *minimization* method was able to reason that accessing only one information source was sufficient to answer the query as completely as possible.

We start by noting that the minimization algorithms can in theory be shown to result in potentially unbounded improvements in execution cost, depending on the way the



(a) Non-recursive plans

(b) Recursive plans (recursion depth=1)

Figure 5: Results characterizing the *lower-bound* on the improvements feasible through minimization.

costs are defined, and the way plans are executed. Specifically, the minimization routine can be considered effective even if it removes access to one source, when the removed source has a sufficiently high cost of access. Moreover, if the source that we removed is a source with binding restrictions, then potentially unbounded improvements in execution time can be shown by simply increasing the recursion depth (i.e., the number of times a node is executed in the r/g graph).

Since based on the experimental set up any potential improvements can be shown, we decided to concentrate instead on characterizing a lower-bound on the improvements in terms of execution time (*ignoring all other costs*) we can get. Accordingly, in our experiments, we supported fully parallel execution at the union nodes in the r/g graph, and experimented with a variety of depth limits, starting from 1 (which in essence prunes the recursion completely).

The plots in Figures 5 and 6 show the results of our experiments. The plots in Figure 5 consider the worst case scenarios for minimization algorithm. Plot **a** shows the effect of minimization when there is no recursion in the plan being minimized. Since we are allowing parallel access to all sources, and are ignoring the traffic costs, execution time cannot worsen because of redundant sources. Not surprisingly, the results show that the extra cost incurred in minimization is not recouped in terms of execution cost. Plot **b** shows the situation where the query plans have recursion (due to the presence of sources with binding restrictions). In our experiments, all such sources happen to be redundant, and the minimization algorithm prunes them. The recursion depth limit is set to 1 (which is the worst scenario for the minimization algorithm since no recursion is allowed on the dom predicates and there is little time that can be recouped during execution by removing the constrained sources). The results show that even in this case, the minimization algorithm starts breaking-even when there are about 8 duplicate sources, and the time improvements from then on grow exponentially.

The plots in Figure 6 consider cases that are a bit more favorable to the minimization algorithm. Specifically plot **a** repeats the experiment in Figure 5(b), but with recursion depth limit set to 2 (that is two recursive calls are allowed on any node in the r/g graph), while plots **b** and **c** do the same with recursion depth limit set to 3 and 5 respectively. We see that minimization algorithm starts breaking even with 1 or 2 duplicate sources in these scenarios. Given that recursion depths between 2-5 are quite reasonable in executing recursive programs, these experiments attest to the savings in execution time afforded by the minimization algorithm.

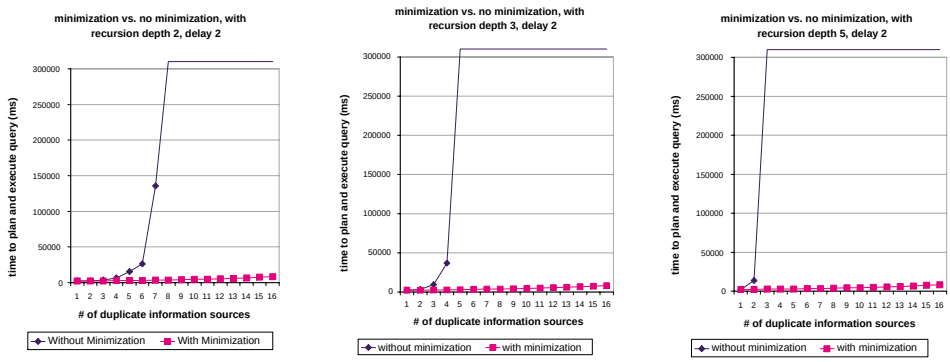
Before concluding this section, we would like to remind the reader that the experiments above only consider the cost improvements in terms of execution time. When other costs such as access costs and network traffic costs are factored in, minimization could be useful even when the plans being minimized do not contain any recursion.

At the time of this writing, we are in the process of completing a set of experiments with more realistic data sources. Specifically, we are experimenting with a set of sources derived from the DBLP bibliography sever data. These experiments will evaluate both the redundancy elimination algorithm and the source-call ordering algorithm.

7 Related Work

Early work on optimizing information gathering plans (c.f. [15, 2]) combined the phases of query plan generation and optimization and posed the whole thing as a problem of search through the space of different executable plans. By starting with Duschka's work [6, 7] which gives a maximally contained plan in polynomial time, and then optimizing it, we make a clean separation between generation and optimization phases.

Friedman and Weld [11] offer an efficient algorithm for minimizing a non-recursive query plan through the use of



(a) Recursion depth=2 (b) Recursion depth=3 (c) Recursion depth=5

Figure 6: Results characterizing the effect of moderate recursion depth limit on the utility of minimization. We set a "patience limit" of 5 minutes (300000 msec), and aborted runs after that.

LCW statements. Their algorithm is based on pair-wise subsumption checks on conjunctive rules. Recursive rules correspond to infinite unions of conjunctive queries, and trying to prove subsumption through pair-wise conjunctive rule containment checks will not be decidable. The approach in Duschka [5] also suffers from similar problems as it is based on the idea of conjunctive (un)foldings of a query in terms of source relations [21]. In the case of recursive queries or sources with binding restrictions, the number of such foldings is infinite. In contrast, our minimization algorithm is based on the notion of uniform containment for recursive datalog programs. This approach can check if sets of rules subsume a single rule. Thus it can minimize a much greater range of plans.

Our source-access ordering technique assumes that statistics regarding source relations are not easily available, and thus traditional join-ordering strategies are not applicable. An interesting alternative is to try and learn the source statistics through experience. Zhu and Larson [27] describe techniques for developing regression cost models for multi-database systems by selective querying. Adali et. al [1] discuss how keeping track of rudimentary access statistics can help in doing cost-based optimizations.

Concurrent with our work, several research efforts [13, 20, 12, 25, 26] have considered the issue of optimizing queries in the presence of sources with limited query capabilities. In contrast to our approach of using "$" and "%" annotations to represent query capabilities of sources, TSIMMIS system [20, 26] lists all "legal" binding patterns entertained by a source, while GARLIC [13] system provides the query processing capabilities of sources in the form of a set of rules. More elaborate languages for representing source access capabilities are proposed in [12, 25]. While some of these systems [26, 12] provide more rigorous optimality guarantees than our greedy approach, their primary focus is on optimization of conjunctive query plans. They thus do not address recursive query plans or the issue of source redundancy. Moreover, to our knowledge, none of these systems consider the source access costs and traffic costs together in doing optimization.

8 Conclusion

In this paper, we considered the query optimization problem for information gathering plans, and presented two novel techniques. The first technique makes use of LCW statements about information sources to prune unnecessary information sources from a plan. For this purpose, we have modified an existing method for minimizing datalog programs under uniform containment, so that it can minimize *recursive* information gathering plans with the help of source subsumption information. The second technique is a greedy algorithm for ordering source calls that respects source limitations, and takes both access costs and traffic costs into account, without requring full source statistics. We have then discussed the status of a prototype implementation system based on these ideas called *Emerac*, and presented an evaluation of the effectiveness of the optimization strategies in the context of *Emerac*. We have related our work to other research efforts and argued that our approach is the first to consider end-to-end the issues of redundancy elimination and optimization in recursive information gathering plans.

Our current directions involve explicitly modeling and exploiting cost/quality tradeoffs, dealing with runtime exceptions such as sources that become inaccessible, as well as run-time opportunities such as the use of caches [1]. We are also exploring the utility of learning rudimentary source models by keeping track of time and solution quality statistics, and the utility of probabilistic characterizations of coverage and overlaps between sources [9, 24]. Finally, we are working towards extending our current greedy plan generation methods, so as to search a larger space of feasible plans

and to provide bounds on generated plans [26]. Part of this involves integrating the logical minimization and source-call ordering phases, so as to get a more globally optimal plan.

References

[1] Sibel Adalı, Kasim S. Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the ACM Sigmod International Conference on Management of Data*, pages 137–148, 1996.

[2] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems*, 6(2/3):99–130, June 1996.

[3] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 98*, pages 34–43, 1998.

[4] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseak Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, IEEE Comput. Soc. Press, pages 190–200, Los Alamitos, CA, 1995.

[5] Oliver M. Duschka. Query optimization using local completeness. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI-97*, pages 249–255, Providence, RI, July 1997.

[6] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '97*, pages 109 – 116, Tucson, AZ, May 1997.

[7] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, Nagoya, Japan, August 1997.

[8] Oren Etzioni, Keith Golden, and Daniel Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89(1–2):113–148, January 1997.

[9] Daniela Florescu, Daphne Koller, Alon Y. Levy, and Avi Pfeffer. Using probabilistic information in data integration. In *Proceedings of VLDB-97*, 1997.

[10] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for world-wide web: A survey. *SIGMOD Record*, September 1998.

[11] Marc Friedman and Daniel S. Weld. Efficiently executing information-gathering plans. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, Nagoya, Japan, August 1997.

[12] Hector Garcia-Molina, Wilburt Labio, and Ramana Yerneni. Capability sensitive query processing on internet sources. In *Proc. ICDE*, 1999.

[13] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. VLDB*, 1997.

[14] Chun-Nan Hsu. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. In *Proceedings of the AAAI Workshop on AI and Information Integration*, pages 66–73, 1998.

[15] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, 1996.

[16] Eric Lambrecht. Optimizing recursive information gathering plans. Master's thesis, Arizona State University, August 1998.

[17] Eric Lambrecht and Subbarao Kambhampati. Planning for information gathering: A tutorial survey. Technical Report ASU CSE TR 97-017, Arizona State University, 1997. rakaposhi.eas.asu.edu/ig-tr.ps.

[18] Alon Y. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 402–412, Bombay, India, 1996.

[19] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 251–262, Bombay, India, 1996.

[20] Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey Ullman, and Murty Valiveti. Capability based mediation in tsimmis. In *Proc. SIGMOD*, 1998.

[21] Xiaolei Qian. Query folding. In *Proceedings of the 12th International Conference on Data Engineering*, pages 48–55, New Orleans, LA, February 1996.

[22] Yehoshua Sagiv. *Optimizing Datalog Programs*, chapter 17. M. Kaufmann Publishers, 1988.

[23] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 2. Computer Science Press, 1989.

[24] Vasiis Vassalos and Yannis Papakonstantinou. Using knowledge of redundancy for query optimization in mediators. In *Proceedings of the AAAI Workshop on AI and Information Integration*, pages 29–35, 1998.

[25] Vasilis Vassalos and Yannis Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *Proc. VLDB*, 1997.

[26] Ramana Yerneni and Chen Li. Optimizing large join queries in mediation systems. In *Proc. International Conference on Database Theory*, 1999.

[27] Qiang Zhu and Per-Ake Larson. Developing regression cost models for multidatabase systems. In *In Proceedings of PDIS*, 1996.