Optimizing Recursive Information Gathering Plans

Eric Lambrecht **Subbarao Kambhampati** **Senthil Gnanaprakasam**[*]
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
Email: {*eml, rao, gsenthil*}*@asu.edu* URL: rakaposhi.eas.asu.edu/yochan.html

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, and 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.

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. 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 answers requires rewriting the query such that all the extensional (EDB) predicates in the rewrite correspond to the materialized view predicates that represent information sources. Several researchers from AI and database communities have addressed this rewriting problem [13, 15, 10]. Recent research by Duschka and his co-workers [5, 6] 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.

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. The plans produced by Dushka's methodology, while complete, in that they will retrieve all (accessible) answers to the query, tend to be highly redundant in that they access any information source that may be remotely relevant to the query. They need to be minimized first by removing redundant sources before being executed. The execution phase also presents several challenges since traditional execution optimization models and techniques [3] do not apply in the context of information gathering on Internet.

In this paper we describe the query optimization techniques that we have developed in the context of *Emerac*, a prototype information gathering system that we are developing. Figure 1 provides a schematic illustration of the query planning and optimization process in *Emerac*. It contains two steps: logical optimization and execution optimization. For logical optimization, we describe a technique that operates on the recursive plans generated by Dushka's algorithm and greedily minimizes them 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 other sources. Our techniques are based on an adaptation of Sagiv's [16] method for minimizing datalog programs under uniform equivalence. Al-

[*]This research is supported in part by NSF young investigator award (NYI) IRI-9457634, ARPA/Rome Laboratory planning initiative grant F30602-95-C-0247, Army AASERT grant DAAH04-96-1-0247, AFOSR grant F20602-98-1-0182 and NSF grant IRI-9801676. We thank Selçuk Candan for many helpful comments.

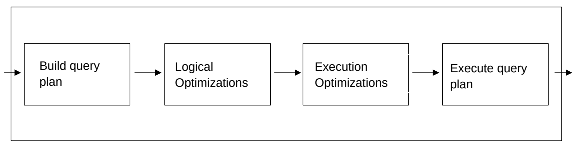


Figure 1: Query planning phases in *Emerac*

though there exists some previous research on minimizing information gathering plans using LCW statements [4, 8], 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 [5]. 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 execution optimization boils down to doing joins between the sources efficiently. This problem differs significantly from the traditional database query optimization problem, as sources on the Internet have a variety of access limitations (aka “source capabilities”) and the execution cost in information gathering is affected both by network traffic and by the connection setup costs. Our second contribution is a way of representing the access capabilities of sources, and a greedy algorithm for ordering source calls that respects source limitations. The algorithm also takes both access costs and traffic costs into account, without requring full source statistics.

The paper starts with a brief review of Dushka’s query plan formation methodology in Section 2. Section 3 presents plan minimization preliminaries, and Section 4 presents our plan minimization algorithm. Section 5 explains how source accesses in the minimized plan can be ordered using the knowledge of sourece access capabilities. Section 6 describes empirical evaluation of our ideas in the context of *Emerac* system. Section 7 discusses the related work and Section 8 presents our conclusions.

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)* $\rightarrow$ *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)* $\wedge$ *A = “Weld”*

Dushcka et. al. [5, 6] 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[1] program whose execution produces all the tuples satisfying the query.

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)* $\rightarrow$ *advisor(S, A)*

The ‘$’ notation denotes that S must be bound 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)* $\wedge$ *A=“Weld”*
r2: advisor(S, A)	:-	ADDB*(S, A)*
r3: advisor(S, A)	:-	*dom(S)* $\wedge$ CONDB*(S, A)*
r4: dom(S)	:-	ADDB*(S, A)*
r5: dom(A)	:-	ADDB*(S, A)*
r6: dom(A)	:-	*dom(S)* $\wedge$ 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.

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

[1]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.

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 [7]. 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”*

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 [4] 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.[2]

Minimization under uniform equivalence: Pair-wise rule subsumption checks alone are enough to detect redundancy in non-recursive plans [12, 8], 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 [17, pp. 908]. We instead base our minimization process on the notion of uniform containment for datalog programs, presented in [16]. 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 [16]. 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 [16] 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 source calls. This is done by removing global schema predicates, and replacing them with bodies of inversion rules that define those predicates (see [17, Sec. 13.4]).[3] Our example plan $\mathcal{P}$, from Section 2, after this transformation with the LCW statements in Section 3 looks as follows:

r2: query(S, A)	:-	*adDB(S , A)* ∧ *A=“Weld”*
r3: query(S, A)	:-	*dom(S)* ∧ CONDB*(S, A)* ∧ *A=“Weld”*
r4: dom(S)	:-	ADDB*(S, A)*
r5: dom(A)	:-	ADDB*(S, A)*
r6: dom(A)	:-	*dom(S)* ∧ 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 2.

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, 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 infor-

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

[3]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.

```
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 2: The greedy plan minimization algorithm

mation 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, $\mathcal{P} - r_3$, we transform them into $\mathcal{P}' = (\mathcal{P} - r_3)[s \mapsto s \vee l]$, 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 $\mathcal{P}'$ 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.

Heuristics for ordering rules for removal: 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 above, suppose we had another LCW statement:

CONDB*(S, A)* $\leftarrow$ *advisor(S, A)*

In such a case, we could have removed *r2* from the original information gathering plan $\mathcal{P}$, instead of removing *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*. This is significant, since in this case, a plan with rule *r3* in it is much costlier to execute than the one with rule *r2* in it. 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). We thus consider for elimination rules containing non-recursive predicates before those containing recursive predicates (such as *dom* terms). Beyond this, we also consider any gathered statistics about the access costs of the sources (such as contact time, response time, probability of access etc.) to break ties [11].

Complexity of Minimization: The complexity of the minimization algorithm in Figure 2 is dominated by the cost of uniform containment checks. As Sagiv [16] 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 since 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. Nevertheless, the exponential complexity justifies our greedy approach for minimization, as finding a globally minimal plan would require considering all possible rule-removal orders.

5 Ordering source calls during Execution

After the minimization phase, the information gathering plan is ready for execution. A crucial practical choice we have to make during the execution of the minimized plans (datalog programs) is the order in which predicates are evaluated. Ultimately plan execution in our context largely boils down to doing joins between the sources efficiently. Although there is a large body of work on join-ordering [3], most of it assumes that all data sources are fully relational databases, ignores source access costs (concentrating only on the traffic costs), and assumes the availability of elaborate source statistics. Such approaches are not particularly suited for *Emerac*. In the information gathering domain, the assumption that information sources are fully relational databases is rarely valid, as sources tend to have a variety of access limitations. Source access costs (connection set up costs etc.) can outweigh the traffic costs. Finally, due to the decentralized nature of Internet, full statistics about sources are rarely available. We now discuss how *Emerac* represents the source limiations, and provide a greedy algorithm for ordering sources that uses this representation to reduce both traffic and access costs during execution.

5.1 Representing source limitations

On Internet, an information source may 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.

In *Emerac*, we use 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 variables that must be bound, and "%" annotation to identify unselectable attributes (i.e., those that must not be bound). Thus a fully relational source would be adorned $source(X, Y)$, a form interfaced web-page 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 as $source(\$X, Y, \%Z)$.

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 S_1 (they must be filtered locally). Thus the call S_1^{bbf} must be executed as S_1^{bff} filtered locally with the binding on Y. Finally, given two binding patterns α and β for a source S, S^α is said to be more general than S^β (written $S^\alpha \succ_g S^\beta$, if every selectable (non "%"-annotated) variable that is free in β is also free in α, but not *vice versa*. Finally, we define $\#(\alpha)$ as as the number of bound variables in α that are not %-annotated. Notice that $\succ_g$ holds only between binding patterns of the same source while $\#(.)$ can be used to relate binding patterns of different sources.

5.2 A greedy algorithm for ordering source calls

The normal heuristic for ordering subgoals in a datalog program is to use "bound is easier" assumption [17], and call sources with more specific binding patterns before those with more general ones. The idea is to reduce costs associated with data transfer (number of tuples transferred). It turns out that bound-is-easier assumption can wind up increasing the connection and 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 leads to increased access costs.

Emerac source-call ordering method considers the connection costs to be of primary importance and the network traffic costs to be of secondary importance. To reduce connection costs, we attempt to access sources with the most general feasible binding patterns. To take traffic costs into account, we also maintain a table HTBP of least general (w.r.t. "$\succ_g$") source binding patterns that are still known to be high-traffic producing. Our algorithm, shown in Figure 3 attempts to pick, for each source, the most general feasible binding pattern that is neither equal to, nor more general than any binding pattern for that source listed in HTBP. An assumption motivating 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.

If all of the feasible binding patterns of all sources are found to be in HTBP in a given step, then the algorithm selects the source with the binding pattern containing most number of bound variables that are not %-annotated (adopting the "bound-is-easier" assumption). This selection then gives rise to further bound variables (enlarges V in the algorithm above), and makes low traffic binding patterns feasible at the next step.

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

```
Inputs: FBP: table of forbidden binding patterns
  HTBP: table of high traffic binding patterns
  V := all variables bound by the head;
  $C[1 \cdots m]$: Array where $C[i]$ lists sources chosen at $i^{th}$ stage;
  $P[1 \cdots m]$: Array where $P[i]$ lists sources postponed at $i^{th}$ stage
for $i$ := 1 to $m$ (where $m$ is the number of subgoals) do begin
  $C[i] := \emptyset$; $P[i] := \emptyset$;
  for each unchosen subgoal $S$ do begin
    $\mathcal{B}$ := All feasible binding patterns for $S$ w.r.t. $V$ and FBP
          sorted using "$\succ_g$" relation.
    for each $\beta \in \mathcal{B}$ do begin
      if $\nexists_{\beta' \in HTBP}$ s.t.$(\beta = \beta') \vee (\beta \succ_g \beta')$
      then begin
          Push $S$ with binding pattern $\beta$ into $C[i]$;
          Mark $S$ as "chosen";
          add to $V$ all variables appearing in $S$;
      end
  end
  if $\mathcal{B} \neq \emptyset$ and $S$ is not chosen
    then Push $S^\gamma$ into $P[i]$, where
      $\gamma \in \mathcal{B}$ has the maximum $\#(.)$ value;
  end
  if $C[i] = \emptyset$ and $P[i] \neq \emptyset$
    then begin
      Take the source $S^\beta \in P[i]$ with maximum $\#(.)$ value
      and push it into $C[i]$;
      add to $V$ all variables appearing in $S$;
    else fail
  end
  Return the array $C[1..i]$.
```

Figure 3: A greedy source call ordering algorithm that considers both access costs and traffic costs.

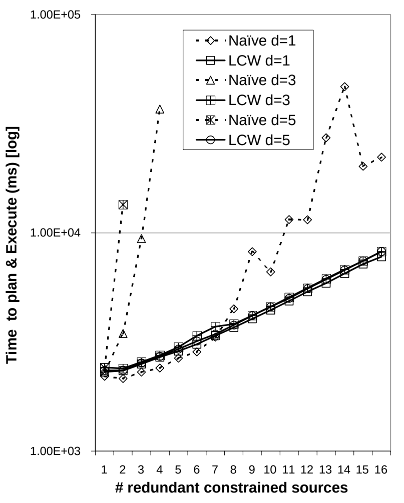
involves issuing calls to sources with the specified binding pattern; where each bound variable in the binding pattern is instantiated to all values of that variable collected upto that point during execution. If the bound variable is a %-annotated variable, then the call is issued without variable instantiation, and the filtering on the variable values is done locally. The tuples returned by the source calls in each stage are locally joined.

Notice that each element of C is a (possibly non-singleton) set of source calls with associated binding patterns (rather than a single source call). This parallelism supports "bushy join trees" [3] and cuts down the overall time wasted during connection delays. The complexity of our ordering algorithm is $O(n^2)$ where n is the length of the rule.

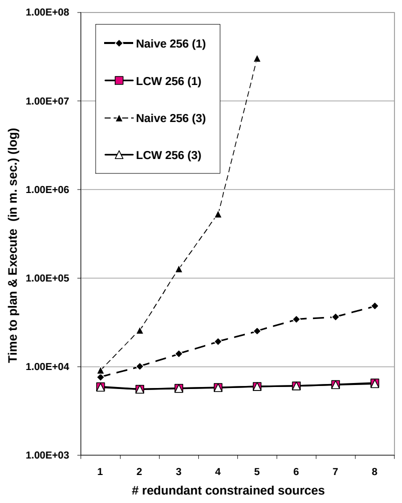
It is worth noting that the behavior of our algorithm depends on the contents of the HTBP table. When HTBP contains no binding patterns, the algorithm essentially concentrates on reducing the source accesses (similar to [14]). When all source binding patterns are listed in HTBP, the algorithm winds up focusing on the network traffic, and reduces to a variant of conjunct ordering by bound-is-easier assumption [17].

6 Implementation and Evaluation

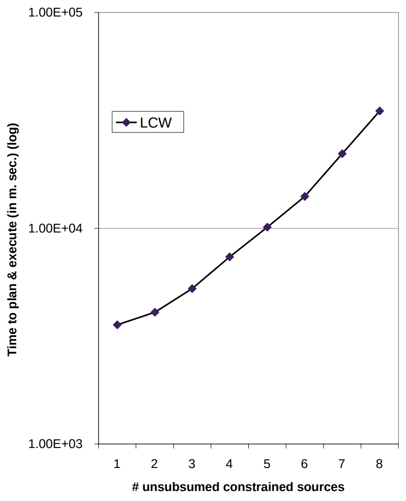
Emerac is a prototype information gathering system under development that implements the ideas in this paper. It is written in Java, and is intended to be a library used by applications that need a uniform interface to multiple information sources. *Emerac* is internally split into two parts: the query planner and the plan executor. The default planner uses Duschka's



(a) Cumulative costs of LCW vs. Naive (artificial sources)



(b) Cumulative costs of LCW vs. Naive (artificial sources)



(c) Effect of unsubsumed constrained sources on LCW

Figure 4: Results characterizing utility of minimization algorithm.

[5] plan generation techniques coupled with our plan minimization techniques. The plan is executed by traversing the relational operator graph [17] corresponding to the minimized plan. 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. The executor uses the algorithm in Section 5 to determine the order to access each information source in a join of multiple sources, as described in Section 5. Recursion in the relational operator graph is controlled by using a depth-limit.

We used the prototype implementation of *Emerac* to evaluate the effectiveness of the optimization techniques proposed in this paper. We used two sets of experimental data. The first were a set of small artificial sources containing 5 tuples each. Our second data set was derived from the University of Trier's Database and Logic Programming (DBLP) online database, which contains bibliographical information on database-related publications. Individual sources used in the experiments corresponded to different subsets of DBLP data (ranging from 128 to 2048 tuples). In each case, some of the sources are unconstrained, while others have binding restrictions (leading to recursive plans). To normalize for differences caused by individual source implementations, we extracted the data into tables which we stored on disk as Java serialized data. All experiments were conducted using a simple wrapper (written in compiled Java) to return the contents of the serialized tables.

The sources delay 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.

Utility of minimization: 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, and compared the total time taken for optimization (if any) and exection. Given that the minimization step involves an exponential "uniform containment" check, it is important to ensure that the time spent in minimization is made up in improved execution cost. Notice that we are looking at only the execution time, and ignoring other costs (such as access cost for premium sources), which also can be reduced significantly with the minimization step. The naive method simply builds and executes source complete plans. The "LCW" method builds source complete plans, then applies the minimization algorithm described in Section 4 before executing the plans. For both methods, we support fully parallel execution at the union nodes in the r/g graph. Since in practice, recursive plans are handled with depth bounded recursion, we experimented with a variety of depth limits (i.e., the number of times a node is executed in the rule-goal graph), starting from 1 (which in essence prunes the recursion completely).

The plots in Figure 4 show the results of our experiments. Plot a is for the artificial sources, and shows the relative time performances of LCW against the naive algorithm when the number of redundant constrained sources is increased. In this set of experiments, LCW statements allow us to prove all constrained sources to be redundant, and the minimization algorithm prunes them. The y-axis shows the cumulative time taken for minimization and execution. We note that the time taken by the LCW algorithm remains fairly independent of recursion depth as well as number of constrained sources. The naive algorithm, in contrast, worsens exponentially with increasing number of constrained sources. The degradation is more pronounced for higher recursion depths, with the LCW method outperforming the naive one when there are two or more redundant constrained sources. Plot b repeats the same experiment, but with the sources derived from the DBLP data. The sources are such that the experimental query returns upto

256 tuples. The experiment is conducted for recursion depth limits 1 and 3. We note once again, that LCW method remains fairly unaffected by the presence of redundant constrained sources, while the naive method degrades exponentially. Plot c considers DBLP data sources in a scenario where some constrained sources are left unsubsumed after the minimization. As expected, LCW performance degrades gracefully with increased number of constrained sources. Naive algorithm would not have shown such graceful degradation no sources would be removed through subsumption.

Although we have not completed a formal evaluation of the source ordering strategy described in Section 5, informal experiments with artificial sources indicate that the technique produces plans with better cumulative access and traffic costs than those offered by ordering based on bound-is-easier assumption.

7 Related Work

Early work on optimizing information gathering plans (c.f. [10, 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 [5, 6] 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 [8] offer an efficient algorithm for minimizing a non-recursive query plan through the use of 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 [4] also suffers from similar problems as it is based on the idea of conjunctive (un)foldings of a query in terms of source relations [15]. 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 [18] 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.

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*. Our current directions involve integrating the minimization and source-call ordering phases more tightly, 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.

References

[1] Adali, S., Candan, K.S., Papakonstantinou, Y., and Subrahmanian, V.S., Query caching and optimization in distributed mediator systems. In *Proc. SIGMOD-96*, pp. 137–148, 1996.

[2] Arens, Y., Knoblock, C., and Shen, W-M., Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems*, 6(2/3):99–130, June 1996.

[3] Chaudhuri, S. An overview of query optimization in relational systems. In *Proc. PODS-98*, pp. 34–43, 1998.

[4] Duschka, O. Query optimization using local completeness. In *Proc. AAAI-97*, pp. 249–255.

[5] Duschka, O. and Genesereth, M. Answering recursive queries using views. In *Proc. PODS-97*, pp. 109 – 116.

[6] Duschka, O and Levy, A. Recursive plans for information gathering. In *Proc. IJCAI-97*. 1997.

[7] Etzioni, O., Golden, K., and Weld, D. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89(1–2):113–148, January 1997.

[8] Friedman, M. and Weld, D. Efficiently executing information-gathering plans. In *Proc. IJCAI-97*. 1997.

[9] Kambhampati, S. and Gnanaprakasam, S. Optimizing source-call ordering in information gathering plans. In *Proc. IJCAI-99 workshop on Intelligent Information Integration.* 1999.

[10] Kwok, C., and Weld, D. Planning to gather information. In *Proc. AAAI-96*, 1996.

[11] Lambrecht, E. and Kambhampati, S. Optimization strategies for information gathering plans. ASU CSE TR 98-018. 1998.

[12] Levy, A. Obtaining complete answers from incomplete databases. In *Proc. 22nd VLDB*, pp. 402–412. 1996.

[13] Levy, A., Rajaraman, A., and Ordille, J. Querying heterogeneous information sources using source descriptions. In *Proc. 22nd VLDB*, pp. 251–262. 1996.

[14] Yerneni, R., and Li, C. Optimizing large join queries in mediation systems. In *Proc. Intl. Conf. on Database Theory*, 1999.

[15] Qian, X. Query folding. In *Proc. 12th Intl. Conf. on Data Engineering*, pp. 48–55. 1996.

[16] Sagiv, Y. *Optimizing Datalog Programs*, in *Foundations of Deductive Databases and Logic Programming*, chapter 17. M. Kaufmann Publishers, 1988.

[17] Ullman, J. *Principles of Database and Knowledgebase Systems*, volume 2. Computer Science Press, 1989.

[18] Zhu, Q., and Larson, P-A. Developing regression cost models for multidatabase systems. In *In Proc. of PDIS*, 1996.