OPTIMIZING RECURSIVE INFORMATION GATHERING PLANS

by

Eric M. Lambrecht

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 1998

OPTIMIZING RECURSIVE INFORMATION GATHERING PLANS

by

Eric M. Lambrecht

has been approved

September 1998

APPROVED:

______________________________ ,Chair

____

____

Supervisory Committee

ACCEPTED:

____

Department Chair

____

Dean, Graduate College

ABSTRACT

Research into information integration has shown how to create a single, uniform query interface to networked information sources. By building a world model over some domain a user is interested in and modelling information sources as materialized views on that model, it is relatively straightforward to create an information gathering plan to answer some query on the world model using the available information sources. These plans, while theoretically sound, are costly to execute directly.

In this thesis, the researcher considers the problem of optimizing information gathering plans. First the researcher looks at the general problem and how it fits in with traditional query optimization. Then the researcher specifically looks at information gathering on the Internet and list the available knowledge of information sources in this domain that can be applied to optimization. The researcher then defines the desirable characteristics of an information gathering plan that access only information sources available on the Internet. These are source completeness, source minimality, and bandwidth minimality.

The researcher then presents two methods for optimizing an information gathering plan that uses Internet information sources. First is a greedy method for removing unnecessary recursion and unnecessary calls to information sources in a plan without affecting the answer returned by the plan. The second is an adaptation of the "bound is easier" database evaluation heuristic that takes into account the way queries are answered by Internet information sources to reduce network traffic when executing a plan.

Finally, the researcher describes an implemented information gatherer, *Emerac*, that uses the algorithms and methods described in this thesis and evaluates the methods presented in this thesis.

To my parents, for getting me this far.

ACKNOWLEDGMENTS

Thanks to Dr. Kambhampati for his support, ideas, and criticism. Thanks to my parents for supporting me through school. Thanks to Erin for missing those valuable lunches with me so I could complete this. Thanks to Dr. Urban and Dr. Candan for being on my committee.

TABLE OF CONTENTS

Page

LIST OF FIGURES vii

CHAPTER

1 INTRODUCTION 1
- 1.1 Information Gathering 1
- 1.2 Optimizing Internet Information Gathering Plans 2
- 1.3 Contributions 3
- 1.4 Organization 3

2 INFORMATION GATHERING ON THE INTERNET 5
- 2.1 Introduction 5
- 2.2 Information Source and World Model Assumptions 5
- 2.3 Optimal IG Plans 9
- 2.4 Building Source Complete Information Gathering Plans 10
- 2.5 Summary and Related Work 12

3 REWRITING TECHNIQUES 13
- 3.1 Introduction 13
- 3.2 Preliminaries 13
- 3.3 Removing Overlapping Sources and Recursion 15
- 3.4 Summary and Related Work 21

4 EXECUTION OPTIMIZATION 22
- 4.1 Introduction 22
- 4.2 Using Knowledge of Query Capabilities to Reduce Network Traffic 22
- 4.3 Related Work 23

5 THE EMERAC SYSTEM 25
- 5.1 Architecture 25
- 5.2 Wrapper Interface 26
- 5.3 Evaluation 26
- 5.4 Summary and Related Work 27

REFERENCES 30

APPENDIX

A EMERAC IMPLEMENTATION DETAILS 32
- A.1 Interface 32
- A.2 Internals 35
- A.3 Tools 40

LIST OF FIGURES

Figure Page

1.1 The information gatherer acts as an intermediary between the user and information sources on the Internet. 4

1.2 The full process of query planning 4

3.1 Procedure to verify if predicate p can be removed from datalog program D . 17

3.2 Algorithm (from section 13.4 of [20]) to remove instances of predicate, p, from a datalog program. 17

3.3 Information gathering plan reduction algorithm for some plan P 18

4.1 Modified version of heuristic ordering algorithm (from figure 12.23 of [20]) 23

5.1 As more information sources are added, the cost of minimizing a plan with no initial recursion gets very expensive. 28

5.2 As more information sources are added, the minimizer successfully removes recursion from a plan and saves a lot of execution time. 29

CHAPTER 1

INTRODUCTION

1.1 Information Gathering

The explosive growth and popularity of the world-wide web has resulted in thousands of queryable structured information sources on the internet, and the promise of unprecedented information gathering capabilities to lay users. Unfortunately, the promise has not yet been completely 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* which take a user's query over a virtual database, and answer it using real information sources available over the network. This is illustrated in figure 1.1.

Several first steps have recently been taken towards the development of a theory of information gathering. The information gathering problem is typically modeled by building a global relational schema for the information that the user is interested in (also called the world model), and modelling the available information sources as materialized views on the global schema. When a query is posed on the global schema, the information gatherer constructs an *information gathering plan* for accessing available information sources, and executes it to answer the query.

Consider we have three relations that define a global relational schema we wish to allow users to query. The first relation, *student-class(S, C)*, represents every student S enrolled in class C. The second, *class-room(C, R)*, represents all the pairs of rooms and the classes held in them. The third, *student-class-grade(S, C, G)* represents the grades students have received for classes they are enrolled in.

Given REGISTRAR, which is an information source that provides information about classes that students are enrolled in, we might relate it to the global schema by defining it as the following materialized view:

$$\text{REGISTRAR}(C, S) \quad \rightarrow \quad \textit{class-student(C, S)}$$

We use "→" rather than ":-" when defining the view (for aesthetic reasons, which will be clear later), but the semantics are the same as in datalog. Further assume we have two more information sources that provide information on students and their grades, and classrooms:

$$\begin{aligned} \text{GRADELISTING}(C, S, G) &\rightarrow \textit{class-student(C, S)} \wedge \textit{student-class-grade(S, C, G)} \\ \text{DIRECTORY}(C, R) &\rightarrow \textit{class-room(C, R)} \end{aligned}$$

Recent research [8, 9] has resulted in a clean methodology for constructing *source complete* and sound information gathering plans for user queries posed in terms of the global schema using only information sources defined as materialized views on the global schema. An information gathering plan is a datalog program in which all EDB predicates are information source predicates. A plan is source complete if executing it results in all accessible answers to the query.

Given the following query

$$\textit{query(S, R)} \quad \text{:-} \quad \textit{class-student(C, S)} \wedge \textit{class-room(C, R)}$$

which asks for the rooms that every student has a class in, the algorithm presented in [8] would produce the following source complete information gathering plan to answer the query:

$$\begin{aligned} \textit{query(S, R)} &\quad \text{:-} \quad \textit{class-student(C, S)} \wedge \textit{class-room(C, R)} \\ \textit{class-student(C, S)} &\quad \text{:-} \quad \text{REGISTRAR}(C, S) \\ \textit{class-student(C, S)} &\quad \text{:-} \quad \text{GRADELISTING}(C, S, G) \\ \textit{class-room(C, R)} &\quad \text{:-} \quad \text{DIRECTORY}(C, R) \end{aligned}$$

Were this plan to be evaluated as a datalog program, it would generate all the available information that satisfies the user's query.

1.2 Optimizing Internet Information Gathering Plans

The process of building a query plan is just the first step in information gathering. The next step, which we focus on in this thesis, is query plan optimization. Traditional database query optimization highlights two important steps that must be performed during optimization: query plan *rewriting*, and *execution ordering*. Query plan rewriting is the logical rewriting of an inefficient or sub-optimal plan to make it more efficient. Execution ordering is the process of finding the best order in which to access information sources in order to minimize execution cost. The full process of query planning is shown in figure 1.2.

Consider the sample query plan above. What if the REGISTRAR information source contains all possible *class-student(C, S)* pairs? In this case, the reference to GRADELISTING in the plan above is unnecessary. An information gathering plan query optimizer should be able to rewrite the original query plan above into a logically equivalent plan, based on the knowledge of the completeness of the REGISTRAR information source, to achieve the following:

$$\begin{aligned} \textit{query(S, R)} &\ :\!-\ \ \textit{class-student(C, S)} \wedge \textit{class-room(C, R)} \\ \textit{class-student(C, S)} &\ :\!-\ \ \textsc{registrar}(\mathrm{C}, \mathrm{S}) \\ \textit{class-room(C, R)} &\ :\!-\ \ \textsc{directory}(\mathrm{C}, \mathrm{R}) \end{aligned}$$

This is an example of a query plan rewriting technique, in which inefficiencies in a plan can be reasoned out logically.

Now consider the order in which to execute the rewritten plan above. Ultimately the plan boils down to a simple join between the REGISTRAR and DIRECTORY information sources. When gathering information on the Internet, we cannot instruct the two sources to join with each other, because we assume they do not have that capability. It is necessary to select all the data from one source then iteratively query the second information source, or select all the data from both sources and perform the join locally. The choice of which method and which order to use in order to minimize cost is addressed during the execution ordering phase of query optimization.

1.3 Contributions

We make two contributions in this thesis. The first is a method for rewriting a source complete information gathering plan to remove unnecessary information sources and recursion in order to speed up execution of the plan. The second is a notation for easily informing the information gatherer of the relative querying costs of information sources, and an adaptation of the traditional "bound is easier" heuristic for query execution ordering that takes advantage of the notation.

1.4 Organization

The thesis is divided into four parts. In chapter 2 we discuss the knowledge and assumptions we have about information gathering on the Internet, the desired aspects of an optimal information gathering plan, and a method for building source complete information gathering plans. In chapter 3 we present an algorithm for rewriting an information gathering plan to minimize unnecessary information sources and recursion in the plan. Chapter 4 discusses how to order access to information sources using knowledge of their query costs in order to reduce network traffic. Chapter 5 discusses Emerac, the information gatherer we have implemented.

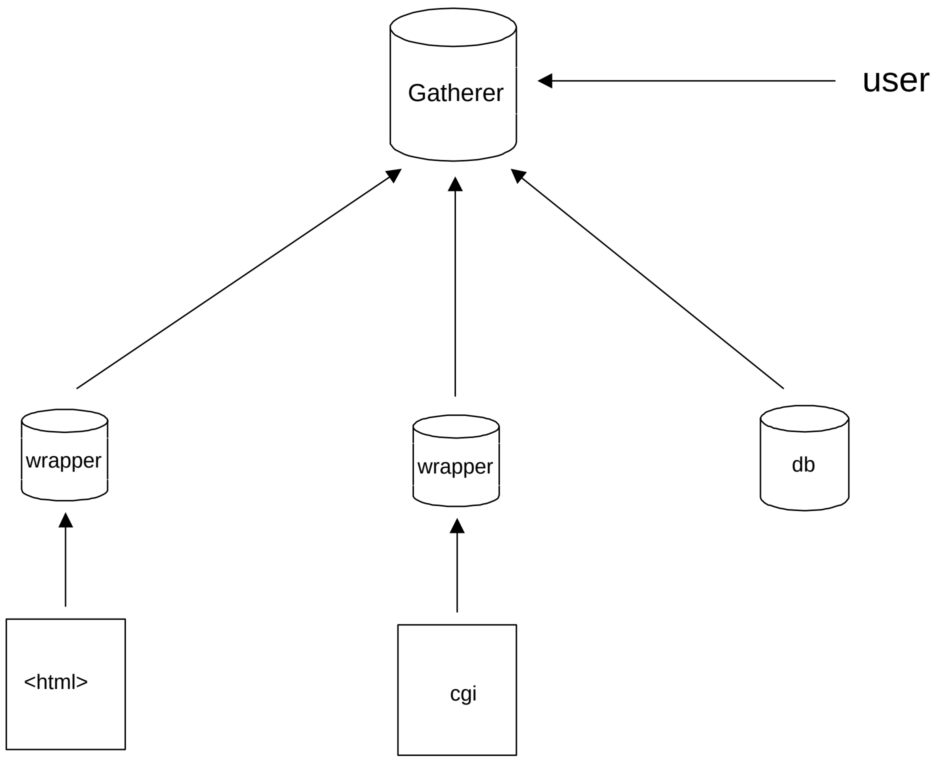

Figure 1.1 The information gatherer acts as an intermediary between the user and information sources on the Internet.

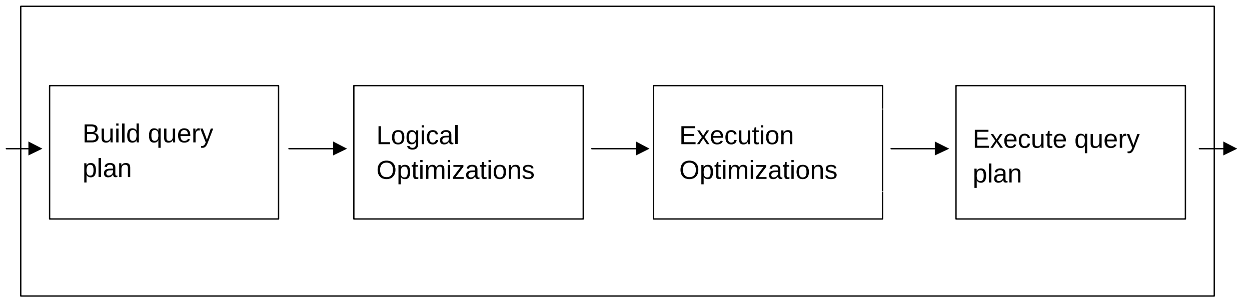

Figure 1.2 The full process of query planning

CHAPTER 2

INFORMATION GATHERING ON THE INTERNET

2.1 Introduction

In this chapter we describe the assumptions we make about information sources available on the Internet for information gathering. We describe the querying capabilities of the sources and the knowledge of their contents that is available to the information gatherer. Then we describe the desirable aspects of an optimal information gathering plan that gathers information from sources over the Internet. Finally, we show how a simple, unoptimized information gathering plan is created.

2.2 Information Source and World Model Assumptions

Current information gathering research makes some simplifying assumptions about the information gathering problem. The first is that information sources can all be modelled as relational databases, possibly with query binding constraints. This leads to the formulation of the world model as a relational schema and the mappings of the contents of the information sources to the world model as *materialized views* on the world model. Information sources are assumed to be able to answer simple relational queries that possibly require specific binding patterns to be satisfied.

We currently assume four pieces of information are available to the information gatherer for building and optimizing plans: descriptions of sources as materialized views, LCW statements about the guaranteed contents of sources, valid query patterns for each source, and relative query costs. Ultimately it would be nice if the owners of each information source would provide the information about the contents of the source, but it is more likely that the designer of the world model provides this information.

2.2.1 Views

We can relate our information sources to the world model by creating views that define the information sources' contents in terms of queries on the world model. Each information source is defined by a unique predicate that appears in the head of a view, and in the body is a query over the world model that would produce equivalent information as contained in the source. For example:

$$\text{WHITEPAGES}(\textit{Name, Area, Phone, Co}) \Leftrightarrow \begin{array}{l} \textit{name(Name)} \wedge \\ \textit{phone-of(Name, Area, Phone)} \wedge \\ \textit{phone(Area, Phone)} \wedge \\ \textit{phone-company(Name, Co)} \end{array}$$

In doing this, we are defining all the tuples that satisfy the world model relations in the body as being exactly those that exist in the information source named in the head.

It may be the case that our information sources do not map up exactly to a query in the world model. For instance, we might have a phone book information source which does not contain all name and phone number pairs that would be available in the world model, and we cannot define a specific enough (and reasonably small enough) query over the world model that exactly defines all the tuples contained in the source. In this case we will define a view where the body is over-general and describes more information than is possibly contained in the information source (just so we do not miss anything during information gathering). We will write the view definition with a "$\rightarrow$" rather than a "$\Leftrightarrow$" to denote that while all the information described in the head of the rule is contained in the information described in the body, it is not the case the all the information described in the body is contained in the head. Our new definition for our white pages database will be:

$$\text{WHITEPAGES}(\textit{Name, Area, Phone, Co}) \rightarrow \begin{array}{l} \textit{name(Name)} \wedge \\ \textit{phone-of(Name, Area, Phone)} \wedge \\ \textit{phone(Area, Phone)} \wedge \\ \textit{phone-company(Name, Co)} \end{array}$$

We are also now going to refer to the white pages database as being *incomplete*. An information source is incomplete if the information contained in it is less than the information that would be generated by executing the world query in its view on the conceptual database of all information that the world model describes. Conversely, if execution of the world query in the information source's view would generate exactly the same information that is in the information source, we say the information source is *complete*. Our notion of completeness, then, is defined as how accurately the information contained in some information source can be described in terms of the world model.

This notion of completeness is important, because the information gatherer should retrieve as much information described in the world query as possible, while at the same time minimizing the amount of redundant information gathered. If we do not state that some information sources are incomplete, the information gatherer might incorrectly reason that the information it has retrieved from some information sources completely answers a world query, when it actually does not. Also, the information gatherer might needlessly retrieve information from two sources where both of them contain exactly the same information.

2.2.2 LCW Statements

Materialized view definitions alone do not adequately provide enough information about our knowledge of information sources. We can more accurately describe the information sources by using *localized closed world (LCW)* statements[1] [10, 11]. Where a materialized view definition describes all possible information an information source might contain in terms of a global schema, an LCW statement describes what information the source is guaranteed to contain in terms of the global schema. Defining an LCW statement for an information source is similar to defining a view. We use the "$\leftarrow$" symbol to denote an LCW definition. Consider the following LCW definitions:

$$\begin{aligned} \textsc{source}\mathrm{A}(X, Y) &\leftarrow j(X, Y) \land k(Y, \text{“Duschka”}) \\ \textsc{source}\mathrm{B}(S, A) &\leftarrow j(X, Y) \end{aligned}$$

In this example, SOURCEB contains all possible instances of the *j* relation, and SOURCEA contains a subset of those instances.

The exact meaning of an LCW statement also depends on how the notion of time and change is taken into account. An information source with an LCW statement that proclaims to have all book titles published before 1910 is relatively straightforward. However, an information source with an LCW statement claiming to contain *all* book titles may be correct at the current point in time, but be incorrect within a month, when new books have been written but not added to the database. We assume that whatever LCW statements the planner is using at the current point in time are correct, and the completeness of the answers the planner returns will be dependant upon this assumption.

2.2.3 Binding Restrictions

Often a real-world information source cannot answer all arbitrary queries over the information it contains. In this case, we say that the information source has *query constraints* and is not, in database terminology, *fully relational*. One important class of restrictions are *binding restrictions*, where an information source requires that some attributes be bound in any query sent to it. Our white pages information source, for example, might require the *Name* attribute in any query posed to it be bound. So the following query is valid:

$$\textsc{whitepages}(Name, Area, Phone, Co) \land Name = \text{“Eric Lambrecht”}$$

but the next query is not, because the *Name* attribute is not bound to any value:

$$\textsc{whitepages}(Name, Area, Phone, Co) \land Co = \text{“USWest”}$$

By convention, we express these binding restrictions in the description of the information source in terms of the world view by putting a "\$" character in front of the attributes that

[1] In [6], the notion of a *conservative view* is equivalent to our LCW statements, and the notion of a *liberal view* is equivalent to our normal materialized view.

must be bound for a query. For example, we can express that the whitepages information source has the binding restriction on its *Name* attribute by defining its world view as:

$$\begin{aligned}\text{WHITEPAGES}(\$Name, Area, Phone, Co) \rightarrow\ & name(Name) \wedge \\ & phone\text{-}of(Name, Area, Phone) \wedge \\ & phone(Area, Phone) \wedge \\ & phone\text{-}company(Co)\end{aligned}$$

2.2.4 Relative Query Data Transfer Costs

The cost of answering specific query patterns (in terms of amount of data returned in response to the query) on different information sources on the Internet varies between the different types of information sources. All queries posed on wrapped web pages have the same cost: the entire web page must be transferred over the Internet. The amount of data returned from a query posed on a cgi script or relational database may change depending on how specific the query is. For instance, a cgi script that accepts a student's first name and, optionally, the student's major and returns a list of students matching the input would more likely return less data for a query with both a name and major specified than a query with only a name specified.

There is a simple way to inform the gatherer as to what types of queries on an information source might reduce the data it transfers over the network. When defining the unique predicate to represent an information source, for every argument of the predicate that, if bound, might reduce the amount of data transferred over the network in response to a query on the information source, we adorn with a "*%*".

Assume we have some fully relational database represented by the predicate RELATIONAL-SOURCE. We can adorn it as in the following:

RELATIONAL-SOURCE*(%X, %Y)*

The "*%*" on the *X* denotes that if the gatherer can bind *X* to some values before sending the query to RELATIONAL-SOURCE, we expect the amount of data to be transferred over the network in response to a query to be smaller than if we were to query RELATIONAL-SOURCE without binding X to any values. This also applies to the Y argument. Thus, we would expect the amount of data transferred over the network as a result of the query

RELATIONAL-SOURCE*(X, Y)*

to be larger than the amount of data due to the query

RELATIONAL-SOURCE*("Eric", Y)*

which in turn is expected to be larger than the amount of data due to query

RELATIONAL-SOURCE*("Eric", "Lambrecht")*

Consider that WEB-SOURCE represents a wrapped web page. We do not adorn WEB-SOURCE with any "%" signs. This is because binding values to the arguments of WEB-SOURCE before querying it have no effect on the amount of data transferred due to the query. That is, we do not expect the amount of data transferred over the network from the query

WEB-SOURCE*("Amol", Y)*

to be smaller than that transferred due to the query

WEB-SOURCE*(X, Y)*

Finally, assume that CGI-SCRIPT represents a cgi-script that accepts some input parameter *W* and returns a corresponding *X* or, if given no inputs, returns all *(W, X)* pairs. we annote CGI-SCRIPT as

CGI-SCRIPT*(%W, X)*

because we only expect that binding values to W will reduce the amount of data transferred over the network. Since CGI-SCRIPT does not accept bindings for *X* as part of the queries it will answer, any selections on *X* must be computed locally by the gatherer after querying CGI-SCRIPT for tuples with the given *W* bindings.

2.3 Optimal IG Plans

Now that we have described the information available to an Internet information gatherer, we can describe the desired aspects of an optimal Internet information gathering plan. There are three goals that we strive for: source completeness, source minimality, and bandwidth minimality.

Source complete means that there should be no other information gathering plan, given the same set of information sources and available data, that can generate more answers to the query.

For a plan to be *source minimal*, the plan should contain the fewest number of information sources in it needed to answer the query as completely as possible. If two sets of information sources conribute overlapping information to the query, the query planner should remove one of the sets so duplicate information is not sent over the network.

A *bandwidth minimal* plan means that execution of the plan transfers the smallest amount of data over the network during execution, without affecting the completeness of the plan. This goal is important because of the relatively large amount of time it takes to transfer data over the Internet.

2.4 Building Source Complete Information Gathering Plans

Building a source complete information gathering plan is quite easy, and is the initial step we take towards building a plan to satisfy all three goals. It is this plan that we will optimize in order to also achieve the goals of source and bandwidth minimality. In this section we review the algorithm for generating source complete plans.

Given a query on a some database schema and a set of materialized views defined over the same schema, it has been shown in [8] that a datalog program in which all EDB predicates are materialized views can be constructed to answer the query, if such a program exists. We call such a datalog program an *information gathering plan*. This method has been extended in [9] to deal with functional dependency information and information sources with query binding pattern requirements. The heart of the algorithm is a simple materialized view inversion algorithm.

Consider that we have the information source SOURCEA represented as a materialized view as follows:

$$\text{SOURCEA}(X, Y) \quad \rightarrow \quad j(X, Y) \wedge k(Y, Z)$$

Also, consider that we have the query:

$$query(X, Y) \quad :\text{-} \quad j(X, Y)$$

We can easily build an information gathering plan that will answer the query (at least partially) using only our materialized view above. This method works by *inverting* all materialized view definitions, then 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. All variables that appear in the head of the original rule are unchanged, but every variable in the body of the original view that does not appear in the head is replaced with a new function constant $f_N(X_1, \ldots, X_m)$. When we invert our definition above, we achieve

$$\begin{aligned} j(X, Y) \quad &:\text{-} \quad \text{SOURCEA}(X, Y) \\ k(Y, f_1(X, Y)) \quad &:\text{-} \quad \text{SOURCEA}(X, Y) \end{aligned}$$

When these rules are added to the original query, they effectively create a logic program to answer the original query. Notice that the global schema predicates that were considered EDB predicates are now IDB, and all EDB predicates are now materialized views.

Function constants in the constructed logic program can easily be removed through a flattening procedure, to convert it into a true datalog program. Note that function constants only appear through inverted rules, which are never recursive, so there will never be an infinite number of function constants in the plan. We can remove function constants through a simple derivative of bottom up evaluation. If bottom up evaluation of the logic program can yield a rule with a function constant in an IDB predicate, then a new rule is added with the corresponding function constant. Then, function constants that appear in IDB predicates are replaced with the arguments to the function constants, and the name of the IDB predicate is annotated to indicate this replacement. Thus, the rule

$$\textit{edge}(X, f(X, Y)) \quad \text{:-} \quad v(X, Y)$$

is replaced with

$$\textit{edge}^{<1,f(2,3)>}(X, X, Y) \quad \text{:-} \quad v(X, Y)$$

In this fashion, all function constants in the logic program are removed, and it becomes a true datalog program. This completes our creation of an information gathering plan to answer the original query.

The materialized view inversion algorithm can be modified in order to model databases that cannot answer arbitrary queries, and have binding pattern requirements. Consider that we have a second information source, SOURCEC that has a binding constraint on its first argument. Its view is as follows:

$$\text{SOURCEC}(\$X, Y) \quad \rightarrow \quad j(X, Y) \wedge k(Y, Z)$$

The inversion algorithm can be modified to handle this constraint as follows. When inverting a materialized view, for every argument X_n of the head that must be bound, the body of every rewrite rule produced for this materialized view must include the domain relation *dom*(X_n). Also, for every argument X_i that is not required to be bound in the head, SOURCE($X_1, \ldots, X_m$), of some materialized view we must create a rule for producing *dom* relations. The head of each domain relation producing rule is *dom*(X_i), and the body is the conjunction of the information source relation and a *dom*(X_n) relation for every variable X_n that is required to be bound. Thus, after inverting the materialized view definitions for the SOURCEC view and the SOURCEA view with the modified algorithm, we obtain

$$\begin{array}{rcl}
j(X, Y) & \text{:-} & \text{SOURCEA}(X, Y) \\
k(Y, f_1(X, Y)) & \text{:-} & \text{SOURCEA}(X, Y) \\
dom(X) & \text{:-} & \text{SOURCEA}(X, Y) \\
dom(Y) & \text{:-} & \text{SOURCEA}(X, Y) \\
j(X, Y) & \text{:-} & dom(X) \wedge \text{SOURCEC}(X, Y) \\
k(Y, f_2(X, Y)) & \text{:-} & dom(X) \wedge \text{SOURCEC}(X, Y) \\
dom(Y) & \text{:-} & dom(X) \wedge \text{SOURCEC}(X, Y)
\end{array}$$

What is interesting to note here is that the plan to solve a non-recursive query with no recursion might contain recursion as a result of the method used to model query constraints on information sources. In fact, if any information sources with binding pattern requirements are available and relevent to the user's query, then the plan that answers the query will contain recursion through the *dom* predicates [2].

2.5 Summary and Related Work

In this section, we have reviewed all the information we assume an information gatherer has about information sources on the Internet, and we have described that information gathering plans we wish to build should be source minimal, source complete, and bandwidth minimal. We have reviewed an algorithm for constructing source complete information gathering plans.

Other methods for building IG plans are presented in [17, 14, 22]. None of the algorithms effectively deal with binding pattern limitations or recursive queries. The algorithm in [22] might be easily extended to handle recursion, however.

In our system we only make use of LCW statements that describe sources in terms of the global schema. Information sources might also have pairwise or n-ary LCW statements between each other. That is, the completeness of one or more information sources could be defined in terms of one or more other information sources. Pairwise LCW statements are used in [11]. It should be possible to incorporate n-ary LCW statements into the minimization algorithm in this thesis. For example, the algorithm might be modified to take them into account during the uniform containment check, or they might be used as a second step to further minimize the plan produced with this algorithm.

The method we use for denoting binding restrictions assumes that all information sources have a fixed set of mandatory bindings and a fixed set of optional bindings. There are information sources whose binding restrictions are a little more complex (only one paper, [15], has addressed this). Rather than explicitly list all the possible legal binding patterns, we chose the simplicity of expressing binding patterns using the \$ notation. Adapting Duschka's inversion algorithm to invert view definitions with arbitrary binding patterns, however, is relatively straightforward.

While we have focused our optimization efforts on source minimality, bandwidth minimality, and source completeness, these constitute only a single optimization scenario out of many. For instance, information source latency, availability, and actual monetary cost might be taken into account and given more importance. Also, completeness of a plan might be considered less important than the time it takes to return an answer.

[2] Of course, we might use a specific *dom* type for each domain type, like *dom:integer* or *dom:string*, but in a reasonably sized domain this only delays the onset of recursion, rather than eliminate it

CHAPTER 3

REWRITING TECHNIQUES

3.1 Introduction

In this chapter, we discuss a method for rewriting an information gathering plan to remove unnecessary recursion and overlapping information sources to achieve our goal of source minimal and bandwidth minimal plans. We argue that provably maximally removing all unnecessary information sources and recursion is computationally hard. We then present a greedy algorithm guided by heuristics that makes use of LCW statements and an algorithm for reducing a datalog program under uniform equivalence to optimize an information gathering plan.

3.2 Preliminaries

3.2.1 LCW Reasoning & Rule Subsumption

Consider that we have two datalog rules, each of which has one or more information source predicates in its body that also have LCW statements, and we wish to determine if one rule subsumes the other. We cannot directly compare the two rules because information source predicates are unique and therefore incomparable. However, if we make use of the sources' view and LCW statements, we can determine if one rule subsumes the other [7, 11].

Given some rule, A, let $A[s \mapsto s \wedge v]$ be the result of replacing every information source predicate s_i that occurs in A with the conjunction of s_i and the body of its view. Also let $A[s \mapsto s \vee l]$ be the result of replacing every information source relation s_i that occurs in A with the disjunction of s_i and the body of its LCW statement. Given two rules, A and B, rule A subsumes B if there is a containment mapping from $A[s \mapsto s \vee l]$ to $B[s \mapsto s \wedge v]$ [8].

Consider the following rules:

r1: j(X, Y) :- *dom(X)* $\wedge$ SOURCEC*(X, Y)*
r2: j(X, Y) :- SOURCEB*(X, Y)*

We can prove that rule *r2* subsumes rule *r1* by showing a containment mapping from one of the rules from *r2*$[s \mapsto s \vee l]$, which is:

r3: j(X, Y) :- SOURCEB*(X, Y)*
r4: j(X, Y) :- *j(X, Y)*

to *r1*$[s \mapsto s \wedge v]$, which is

r5: j(X, Y) :- *dom(X)* $\wedge$ SOURCEC*(X, Y)*$\wedge$ *j(X, Y)* $\wedge$ *k(Y, Z)*

It is easy to see that there is indeed a containment mapping between *r4* and *r5*. This proves that *r2* subsumes *r1* and so *r1* can be removed from any plan with these two rules without affecting the answer

3.2.2 Uniform Equivalence

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 relations in the programs [18]. Checking equivalence is known to be undecidable. Two datalog programs are uniformly equivalent if they produce the same result for all possible assignments for all initial relations. Uniform equivalence is decideable, and implies equivalence. A method for minimizing a datalog program under uniform equivalence, from [18] is presented here, and later adapted for our information gathering plan minimization.

The technique for minimizing a datalog program under uniform equivalence involves removing rules from the program, one at a time, and checking to see if the remaining rules can produce the same data as the removed rule, given an initial assignment of relations. If the remaining rules cannot do so, then the removed rule is reinserted into the datalog program. The initial assignment of relations is built by instantiating the variables in the body of the removed rule. To instantiate the variables of the relation *foo(*$X_1 \ldots X_n$*)* means to create an instance of the relation *foo* with constants "X_1" … "X_n".

Consider that we have the following datalog program:

r1: p(X) :- *p(Y)* $\wedge$ *j(X, Y)*
r2: p(X) :- *s(Y)* $\wedge$ *j(X, Y)*
r3: s(X) :- *j(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 *j("Y")* we can assert *s("Y")* via rule *r3*. Then, given *s("Y")* and *j("X", "Y")*, we can assert *p("Y")* from rule *r2* . Thus the above program will produce the same results without rule *r1* in it.

3.3 Removing Overlapping Sources and Recursion

In this section we argue that provably removing all unnecessary recursion and information sources from an information gathering plan is difficult. Then we present a greedy minimization algorithm that makes use of LCW statements and Sagiv's algorithm for reducing a datalog program under uniform equivalence.

3.3.1 Argument for Greedy Minimization

Consider that an information gathering plan is a datalog plan, such that all information source predicates are essentially EDB predicates. It has been shown that checking for equivalence of two datalog program is undecideable [19]. Alternatively, consider if we could find all the non-recursive subsets of the original information gathering plan. It has been shown that checking for containment of a datalog program by a union of conjunctive queries takes EXPTIME [4].

Because the complexity of merely checking to see if a sub-plan contains or is equivalent to the original plan is so difficult (and impossible in some cases), our search for a plan minimization algorithm focused on greedy search based methods that can remove portions (individual rules) of a plan that are provably unnecessary.

3.3.2 Greedy Minimization

We can minimize an information gathering plan greedily using the algorithm for minimizing a datalog program under uniform equivalence and LCW statements. The general idea is to iteratively try to remove rules from the information gathering plan and check if the remaining rules subsume the removed rule, but make use of LCW and view statements for the subsumption check. Before applying this algorithm, however, it is important to remove useless IDB predicates to make it more effective. Also, the order in which we try to remove rules affects the final plan, so we use heuristics to guide rule removal.

IDB Predicate Removal

One artifact of the process of removing function constants from an information gathering plan is the large number of IDB predicates added to the plan. These extra predicates make it difficult to perform pairwise rule subsumption checks on rules in our information gathering plan. Recall that, in our example of function constant removal, we created a new predicate *edge*$^{<1,f(2,3)>}$ by flattening out an *edge* predicate with function constant arguments. This new predicate is incomparable with the $edge$ predicate, because they have different names and different arities, even though both refer to the same relation. Because of this mismatch, we try to eliminate the newly introduced predicates before attempting to minimize the information gathering plan. By reducing the number of IDB predicates in the plan to a minimum, pairwise subsumption checks work more effectively.

To further illustrate the problem, consider that we have the following information gathering program, where SOURCEA and SOURCEB are materialized view predicates:

$$\begin{array}{rcl} \textit{query(X)} & \text{:-} & j^{<1,f_1(2,3)>}\textit{(X, X, Y)} \\ \textit{query(X)} & \text{:-} & j^{<1,f_2(2,3)>}\textit{(X, X, Y)} \\ j^{<1,f_1(2,3)>}\textit{(X, X, Y)} & \text{:-} & \text{SOURCEA}\textit{(X, Y)} \\ j^{<1,f_2(2,3)>}\textit{(X, X, Y)} & \text{:-} & \text{SOURCEB}\textit{(X, Y)} \end{array}$$

There is no pair of rules for which we can build a containment mapping to prove subsumption, because each rule differs from all others by at least one predicate. However, if we remove the variants of the j predicate, we can obtain the equivalent program:

$$\begin{array}{rcl} \textit{query(X)} & \text{:-} & \text{SOURCEA}\textit{(X, Y)} \\ \textit{query(X)} & \text{:-} & \text{SOURCEB}\textit{(X, Y)} \end{array}$$

Since we know how to compare rules with materialized views using LCW and view statements, we can compare these rules to determine if one subsumes the other.

The IDB predicate removal algorithm works in two parts: first a search is performed to find predicates that can be safely removed without altering the meaning of the program, then the rules with those predicates in their head are removed and unified with the remaining rules. An IDB predicate can be safely removed if it does not appear as a subgoal of one of its own rules. A simple recursive method for testing if a predicate can be removed is shown in figure 3.1. Basically, we are trying to remove all non-recursive IDB predicates that do not unify with the query.

Once we have a list of all predicates that can be removed, we can replace references to those predicates in the information gathering program with the bodies of the rules that define the predicates. For some predicate p, an algorithm for such a task is offered in [20]. The algorithm is reproduced here in figure 3.2. After passing each predicate through this algorithm, we have successfully reduced the number of IDB predicates in our information gathering program to a minimum.

Greedy Minimization Algorithm

The basic procedure for reducing an information gathering plan runs as in the datalog minimization under uniform equivalence algorithm. We 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 algorithm 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

```
function **removeable($p$, $D$)** returns true if predicate $p$ can be removed from datalog
program $D$
        if $p$ is an EDB predicate
          then return false
        else
          return removeable-aux($p$, $\{\ \}$, $p$, $D$)

function **removeable-aux($c$, $visited$, $p$, $D$)**
        if $c$ is an EDB predicate
          then return true
        else if ($c$ = $p$) and (c $\in visited$)
          then return false
        else
          for each rule, $r$, in $D$ with head predicate matching $c$
                for each subgoal, $s$, of $r$ where $r \notin visited$
                        if removeable-aux($s$, $visited \cup \{\ c\ \}$, $p$, $D$) = false
                          then return false
        return true
```

Figure 3.1 Procedure to verify if predicate p can be removed from datalog program D

```
for each occurrence of $p$ in a subgoal $G$ of some rule $s$ do begin
    for each rule $r$ with head predicate $p$ do begin
        rename the variables of $r$ so $r$ shares no variable with $s$;
        let $\tau$ be the most general unifier of the head of $r$ and the subgoal $G$;
        create a new rule $s_r$ by taking $\tau(s)$ and replacing the subgoal $\tau(G)$ by
        the body of $\tau(r)$;
    end;
    delete rule $s$;
end;
delete the rules with $p$ at the head
```

Figure 3.2 Algorithm (from section 13.4 of [20]) to remove instances of predicate, p, from a datalog program.

```
**for** each IDB predicate, $p_i$ that occurs in $P$
    append *-idb* to $p_i$'s name
**repeat**
        let $r$ be a rule of $P$ that has not yet been considered
        let $\hat{P}$ be the program obtained by deleting rule $r$ from $P$
        let $\hat{P}'$ be $\hat{P}[s \mapsto s \vee l]$
        let $r'$ be $r[s \mapsto s \wedge v]$
        **if** there is a rule, $r_i$ in $r'$, such that $r_i$ is uniformly subsumed by $\hat{P}'$
          **then** replace $P$ with $\hat{P}$
**until** each rule has been considered once
```

Figure 3.3 Information gathering plan reduction algorithm for some plan P

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 3.3.

The process of transforming the candidate rule and the rest of the plan can best be described with an example. Consider the following problem. We have information sources described by the following materialized views and LCW statements:

$$\begin{aligned}
\text{ADVISORDB}(S, A) &\rightarrow \mathit{advisor}(S, A)\\
\text{ADVISORDB}(S, A) &\leftarrow \mathit{advisor}(S, A)\\
\text{CONSTRAINEDDB}(\$S, A) &\rightarrow \mathit{advisor}(S, A)\\
\text{CONSTRAINEDDB}(\$S, A) &\leftarrow \mathit{advisor}(S, A)
\end{aligned}$$

and our query is

$$\mathit{query}(X, Y) \quad \text{:-} \quad \mathit{advisor}(X, Y)$$

After rule inversion and addition of the query to the inverted rules, our information gathering plan is computed to be:

r1: query(X, Y)	:-	*advisor(X, Y)*
r2: advisor(S, A)	:-	ADVISORDB*(S, A)*
r3: advisor(S, A)	:-	*dom(S)* $\wedge$ CONSTRAINEDDB*(S, A)*
r4: dom(S)	:-	ADVISORDB*(S, A)*
r5: dom(A)	:-	ADVISORDB*(S, A)*
r6: dom(A)	:-	*dom(S)* $\wedge$ CONSTRAINEDDB*(S, A)*

Most of this plan is redundant. Only the rules *r1* (the query) and *r2* are needed to completely answer the query. The remaining rules are in the plan due to the constrained information source.

For our example, we'll try to prove that rule *r3* 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 information source could return. If we call our rule r, then we want to transform it to $r[s \mapsto s \wedge v]$. This produces:

advisor(S, A) :- *dom(S)* $\wedge$ CONSTRAINEDDB*(S, A)* $\wedge$ *advisor(S, A)*

There is a problem here that must be dealt with before this transformation. The *advisor* relation in the head of the rule no longer represents the same thing as the *advisor* relation in the body of the rule. That is, the *advisor* predicate in the head represents an IDB relation in the information gathering plan, while the *advisor* predicate in the body represents an EDB predicate. Before we replace an information source in some rule with its view or LCW, we need to rename the global schema predicates in the rule so they do not match the predicates from the views. For every world predicate named *predicate* that appears in the rule, we rename it *predicate-idb.* The correct transformation of r, then, is

advisor-idb(S, A) :- *dom(S)* $\wedge$ CONSTRAINEDDB*(S, A)* $\wedge$ *advisor(S, A)*

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

query(X, Y) :- *advisor-idb(X, Y)*
advisor-idb(S, A) :- ADVISORDB*(S, A)*
advisor-idb(S, A) :- *advisor(S, A)*
dom(S) :- ADVISORDB*(S, A)*
dom(S) :- *advisor(S, A)*
dom(A) :- ADVISORDB*(S, A)*
dom(A) :- *advisor(S, A)*
dom(A) :- *dom(S)* ∧ CONSTRAINEDDB*(S, A)*
dom(A) :- *dom(S)* ∧ *advisor(S, A)*

When we instantiate the body of the transformed removed rule, we get the following constants:

dom("S")
constrainedDB("S", "A")
advisor("S", "A")

After evaluating the remaining rules given with these constants, we find that we can derive *advisor-idb("S", "A")*, which means we can safely leave out the rule we have removed from our information gathering program.

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

The final information gathering plan that we end up with after executing this algorithm will depend on the order in which we remove the rules from the original plan. Consider if we tried to remove the *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, yet the final information gathering plan would contain the *dom* recursion in it, which greatly increases the execution cost of the plan.

Heuristics

Because of our assumption of large network delay, and our desire for minimal bandwith plans, we try to remove rules with *dom* predicates in their bodies first. The *dom* predicates cause recursion in the information gathering plan, so if we can remove any rules with recursion first, we can remove a potentially large execution time cost from the plan.

Alternatively, if the query optimizer knows about source access costs, it might use that information to try to select out rules with higher cost first. It might be possible to make use of available %-annotations for this purpose.

3.4 Summary and Related Work

We have presented an algorithm that effectively rewrites an information gathering plan to remove unnecessary recursion and redundant information sources. The algorithm is greedy, because we have found that merely testing sub-plans for equivalence or containment of the original plan is unfeasible and impossible in some cases. In chapter 5, we will see that the cost for the greedy algorithm turns out to be quite useful for information gathering plans with recursion.

Friedman and Weld [11] offer an efficient algorithm for reducing the size of a non-recursive rewritten query through the use of LCW statements. Their algorithm converts a non-recursive information gathering plan into a directed acyclic graph whose nodes represent relational operators and information sources. It then searches the graph for nodes that represent the relational operator *union*, and it attempts to minimize the union by performing pairwise subsumption checks (making use of LCW statements) on each subgraph leading from the union (which represent portions of the plan that are unioned together). It is because this algorithm only performs pairwise subsumption checks that it cannot handle recursion, since it does not make sense to compare a rule defined in terms of itself with another rule. The minimization algorithm in this thesis, on the other hand, can check if an information gathering plan subsumes a single rule. Thus while the algorithm presented here is more costly to execute, it can minimize a much greater range of plans.

Recently [21] has looked into keeping track of source overlap during plan execution. That is, they track the results of queries on information sources and attempt to record how similar the contents of two or more sources are. They propose that the query planner, given the choice of two or more sets of partially overlapping sources that can answer some portion of the query, use the overlap statistics to chose the least costly set. If there are numerous overlaps for different portions of the query, they propose building a query plan with the global least costly set of minimal-overlapping information sources. We have been able to modify our minimization algorithm to explore all the minimal plans discovered through the uniform completeness algorithm. This might be used to discover the portions of the plan that the overlap minimization algorithm can choose between.

CHAPTER 4

EXECUTION OPTIMIZATION

4.1 Introduction

In this chapter, we present a method for determining an execution order for an information gathering plan in an attempt to make execution of our plan bandwidth-minimal. This method is an adaptation of the database query processing "bound is easier" heuristic.

4.2 Using Knowledge of Query Capabilities to Reduce Network Traffic

A crucial practical choice we have to make during evaluation of an information gathering plan is the order in which predicates are evaluated. If we correctly order the queries to multiple sources, we expect to be able to use the results of earlier queries to reduce the size of results in future queries and facilitate their computation. In database literature, this is referred to as the "bound-is-easier" assumption [20].

Consider we have the following query, where each predicate represents a remote information source:

$$\textsc{source}\text{A}(\text{“}\textit{Lambrecht}\text{”}, X) \quad \wedge \quad \textsc{source}\text{B}(X, Y)$$

Which source should we query first? In the absence of any additional information, distributed database literature assumes both SOURCEA and SOURCEB are fully relational databases of similar size [16]. In this case, we would then query SOURCEA first, because we would expect the answer to this query to be smaller than retrieving the complete contents of SOURCEB. The results of the query on SOURCEA can then be sent to SOURCEB to complete the evaluation. Consider that if we were to query SOURCEB first, we would have to transfer the entire contents of SOURCEB over the network, then the values bound to X would have to be sent to SOURCEA to finally retrieve the answer at what would likely be a higher cost

Given our adorned information source descriptions, we can order our access to information sources in a rule according to the number of bound %-adorned arguments. Predicates should be ordered within a rule so that all predicates with no %-adorned arguments appear first in any order, followed by the remaining predicates ordered such that the number of

```
V := all variables bound by the head;
mark all subgoals ”unchosen”;
mark all subgoals with no %-adornments as ”chosen”;
for $i$ := 1 to $m$ do begin
    b := $-1$;
    for each unchosen subgoal G do begin
        find the bound and %-adorned arguments
        of $G$, given that $V$ is the set of bound variables;
        if there are $c > b$ bound %-adorned
        arguments of $G$ and with this binding
        pattern $G$ is permissible
        then begin
          $b$ := $c$;
          $H$ := $G$;
        end;
        if $b \neq -1$ then begin
          mark $H$ ”chosen”;
          add to $V$ all variables appearing in $H$;
        end
        else fail
    end
end
```

Figure 4.1 Modified version of heuristic ordering algorithm (from figure 12.23 of [20])

arguments both %-bound and adorned in subsequent predicates never decreases. This algorithm is the same as the one presented in [20], except that we compare the number of bound %-adorned arguments, rather than just the number of bound arguments in each predicate. The algorithm appears in Figure 4.1. If the algorithm fails, then subgoals are not reordered. If the algorithm succeeds, then the order in which to access subgoals is specified by *H*.

4.3 Related Work

While execution ordering is a well known problem in traditional database research [20], there is no discussion of it in information gathering research. There is also no research towards taking into account during query processing the costs of querying information sources using different binding patterns. Potentially a lot may be borrowed from multimedia database query processing research, where the cost of different query patterns are taken into account during query plan construction [3].

Sensing [2] tackles the issue of trying to minimize data transfers over the network by adding additional queries to the query plan that, when executed, derive data that can be used to prune portions of the query plan that will be executed after it. This can be viewed as an extreme example of execution ordering. Consider that the planner needs to retrieve and union together the contents of sources A and B. Sensing is the process of adding access to a third information source, C, that joins with and is executed before A and B, with the expectation that what is returned by C reduces the amount of information to be retrieved from A and B. Sensing has not been applied to datalog style query plans (in [2] they make use of description logics), but it should carry over.

CHAPTER 5

THE EMERAC SYSTEM

In this section we describe Emerac, a prototype information gatherer that we implemented using the ideas presented in this thesis. We give an overview of its architecture and its wrapper interface. We empirically evaluate the methods presented in this paper with Emerac. The implementation brought up many interesting points about real world information gathering, which are mentioned in the summary.

5.1 Architecture

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. 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 the algorithms from this thesis, but it can be replaced with alternate planners that, for instance, exhaust the plan search space to try to find an optimal plan. 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 [8]) and attempts to make it bandwidth minimal and source minimal using the rewriting algorithm presented in this thesis.

The optimized plan is passed to the plan executor, which transforms the plan into a relational operator graph. The plan executor makes use of %-adornments to determine the order to access each information source in a join of multiple sources, as described in this thesis. 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.

5.2 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 [12] 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.

Due to the difficulty in building wrappers (especially for cgi-scripts), we built artificial information sources for testing. The simulated sources returned fixed sets of data, and delayed answering each query for 2 seconds in order to simulate actual latency on the Internet.

5.3 Evaluation

We have implemented the minimization algorithm in Emerac, and found that the cost incurred for small plans (up to 8 information sources in them) is negligible (under a second), but quite useful in larger plans with recursion.

In our tests, we made use of our artificial information sources. We varied the number of duplicate information sources available to answer the plan to see how the planner and executor performed with and without minimizing the plan. The *no-minimziation* algorithm simply builds and executes source complete plans. The *minimization* method builds source complete plans, then applys the minimization algorithm described above before executing the plans. Both methods executed their plans in parallel. In all cases the *minimization* method was able to reason that accessing only one information source was sufficient to answer the query as completely as possible.

Figure 5.1 shows the extra cost incurred while minimizing an information gathering plan when there is no recursion in the original information gathering plan. Clearly the cost to minimize the information gathering plan becomes expensive for larger plans. However, in figure 5.2, where there is recursion in the original plan (due to query constraints), the minimization algorithm ultimately saves a large amount of time during execution. Because there are many information sources on the Internet, and many of them have query binding constraints which lead to recursion in query plans, we believe this shows that the minimization algorithm is very applicable to the Internet domain.

5.4 Summary and Related Work

Emerac is a fully implemented information gatherer that makes use of the algorithms presented in this thesis, yet is modular enough that new algorithms can be inserted to replace the existing ones.

During the construction of Emerac, we discovered numerous interesting details about practical information gathering that should be taken into account in future information gatherer implementations. First is source cacheing. Many sources contain information that rarely changes, so they might be cached locally to achieve a large speed improvement at the cost of a small chance of outdated information. Second is the difficulty of building and maintaining wrappers. Information source wrappers are often brittle, and can easily break when the contents of an information source change. When they do break, there must be some protocol for the wrapper to inform the gatherer that what was extracted from the source is no longer necessarily correct, so the gatherer can take that information into account. Related to this problem is that of dealing with failed information sources. The planner might remove an information source from the world model, due to its failure, then replan the original query (or keep track of alternate information sources during query planning, so a failure causes the planner to use a "backup plan" as in [11]). Finally, the information extracted from multiple information sources is often in a multitude of inconsistent formats that must be massaged to allow for proper joins between information sources.

Other information gatherers like Emerac have been implemented elsewhere. *Infomaster* was developed at Stanford, and uses the algorithm for building source complete plans from [8]. There is no discussion of how it optimizes the plans it produces. *SIMS* builds possibly suboptimal information gathering plans (represented differently than in this thesis) and iteratively rewrites them in order to improve the quality [1]. *Occam* [13] used AI planning techniques to generate plans, and *Razor* [11] used LCW reasoning to remove redundant plans. *TSIMMIS* [5] uses pattern matching to answer a query with a predefined query plan.

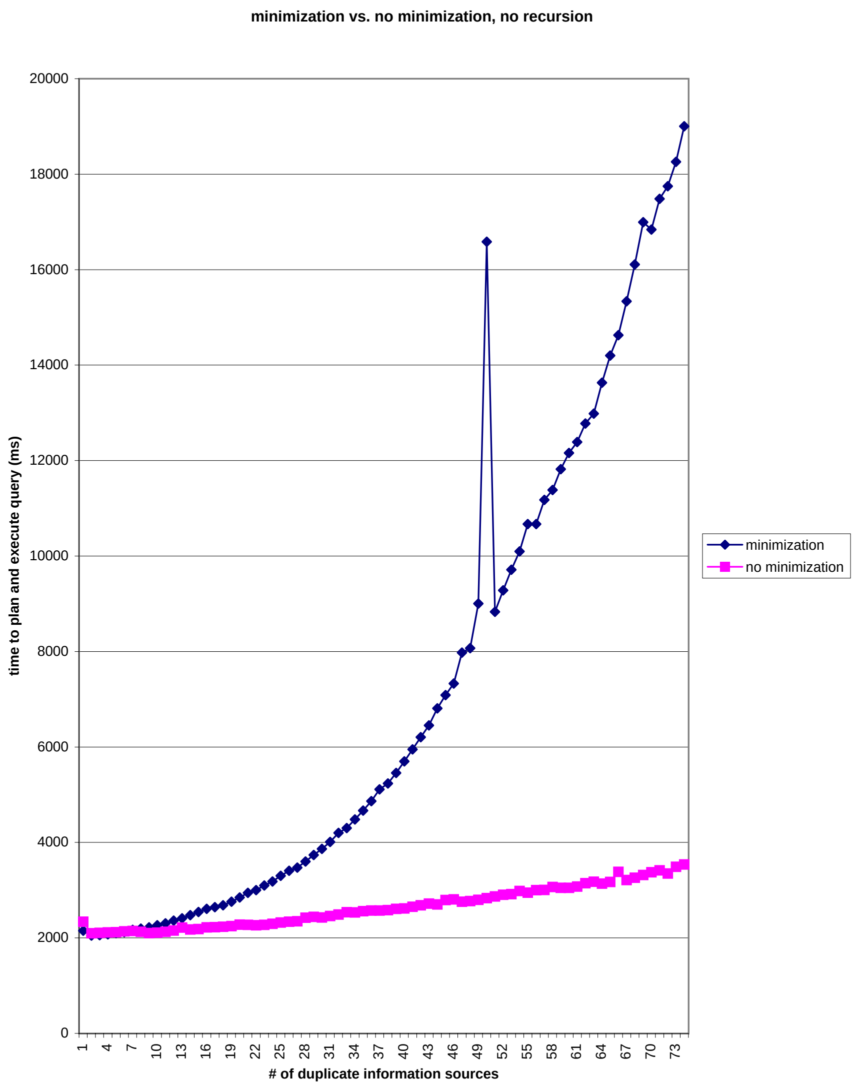

Figure 5.1 As more information sources are added, the cost of minimizing a plan with no initial recursion gets very expensive.

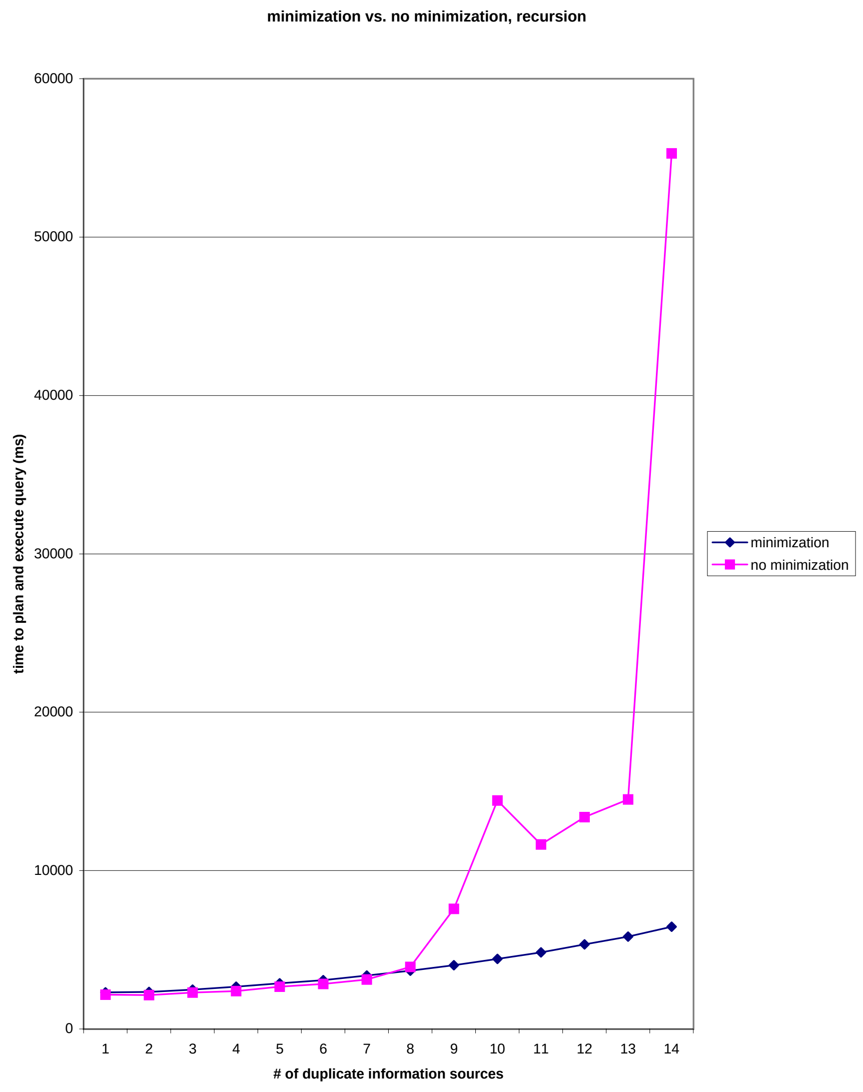

Figure 5.2 As more information sources are added, the minimizer successfully removes recursion from a plan and saves a lot of execution time.

REFERENCES

[1] Jose Luis Ambite and Craig A. Knoblock. Flexible and scalable query planning in distributed in distributed and heterogeneous environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, 1998.

[2] N. Ashish, C. A. Knoblock, and A. Levy. Information gathering plans with sensing actions. In *Proceedings of the Fourth European Conference on Planning*, 1997.

[3] Kasum Selcuk Candan and Yifeng Yang. Cost- and similarity-driven query optimization in the presence of user-defined functions. Technical Report TR-97-041, Arizona State University, 1998.

[4] Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. *Journal of Computer and System Sciences*, 41(1):61–78, February 1997.

[5] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The tsimmis project: Integration of heterogeneous information sources. In *Proceedings of the 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, October 1994.

[6] Oliver M. Duschka. Generating complete query plans given approximate descriptions of content providers. Technical report, Stanford University, 1996.

[7] Oliver M. Duschka. Query optimization using local completeness. In *Proceedings of AAAI*, 1997.

[8] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of AAAI*, 1997.

[9] Oliver M. Duschka and Alon Levy. Recursive plans for information gathering. In *Proceedings of IJCAI*, 1997.

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

[11] Marc Friedman and Daniel S. Weld. Efficiently executing information-gathering plans. In *Proceedings of IJCAI*, 1997.

[12] 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.

[13] Chung T. Kwok and Daniel S. Weld. Planning to gather information. Technical Report UW-CSE-96-01-04, University of Washington, 1996.

[14] Alon Y. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22nd VLDB Conference*, 1996.

[15] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference*, 1996.

[16] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[17] Xiaolei Qian. Query folding. In *Proceedings of the 12th International Conference on Data Engineering*, February 1996.

[18] Yehoshua Sagiv. *Optimizing Datalog Programs*. M. Kaufmann Publishers, 1988.

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

[20] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Volume II*. Computer Science Press, 1989.

[21] 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.

[22] W. Shen Y. Arens, C. A. Knoblock. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems*, 1996.

APPENDIX A

EMERAC IMPLEMENTATION DETAILS

This discussion of Emerac is split into three parts: the high level interface for making use of Emerac to do information gathering, the structure of the internals of Emerac for modifying it, and other related tools.

A.1 Interface

The interface to Emerac centers around the `InformationGatherer` interface definition[1]. After creating an instance of Emerac, the basic tasks that are performed with it are adding information sources, adding datalog rules (including materialized view definitions and lcw statements), executing queries, and receiving information that Emerac has gathered in response to a query.

A.1.1 Instantiating Emerac

The simplest way to create an instance of Emerac is to perform the following:

```
InformationGatherer emerac = new edu.asu.emerac.Emerac();
```

This creates an instance of emerac with a default planning and execution engine.

Emerac can be created with different planning and execution engines, which allow for testing of alternate planning techniques, by using the constructor that accepts the planner and executor that it should use:

```
QueryPlanner qp = new DuschkaPlanner();
PlanExecutor pe = new SagivExecutor();
InformationGatherer emerac =
                 new edu.asu.emerac.Emerac(qp, pe);
```

[1]The author initially created an interactive interface to Emerac, but this proved to be unwieldy for testing and experimental purposes and was removed. This can easily be re-inserted if the desire should arise.

A.1.2 Adding Information Sources

All information sources that Emerac makes use of must conform to the `InformationSource` interface. Basically, this interface forces the information source to appear as a source of relational data and accept arbitrary selection queries.

To make an information source known to Emerac, merely use the `registerInformationSource` method like so:

```
InformationSource msu = new edu.asu.gathertest.msu();
emerac.registerInformationSource(msu);
```

Once the information source has been registered to Emerac, it will never be used unless it is defined through a view. When referring to the information source in a rule, use the same string value returned by the information source's `name()` method. Defining a view to Emerac is discussed in the next section.

A.1.3 Adding and Removing Rules

Emerac has a simple rule interface that allows you to treat it almost exactly like a datalog evaluator. There are three types of rules that you can present to Emerac: materialized view definitions, lcw statements, and simple datalog rules.

All rules are given to Emerac as strings through the `addRule()` method. Each call to this method should have a single rule as an argument. Emerac uses the *javacc* parser to interpret rules, so it is quite robust. A `ParseException` (which contains useful debugging information) will be thrown if Emerac cannot parse the rule passed in. Emerac differentiates the types of rules through the string that separates the head of the rule from the body. Views use "`->`", lcw statements use "`<-`", and normal datalog rules use "`:-`".

A view defines the contents of an information source in terms of the global schema. Considering that we have the `msu` information source from the previous example, we can define its contents with the following:

```
 emerac.addRule("msu(name, position) -> " +
"faculty-position(name, position)," +
                "faculty-school(name, \"msu\"")");
```

Arguments to relations are considered variables, unless they are surrounded by quotes ("msu" in the example is a constant, while "name" and "position" are not)[2].

An lcw defines the portion of an information source that is complete with respect to the global schema. We can define an LCW statement for the "msu" information source:

```
emerac.addRule("msu(name, position) <- " +
               "faculty-school(name, \"msu\")");
```

[2]One word of caution: Emerac does not check to ensure that rules passed to it are *safe*. That is, it will not complain if there are variables in the head of the rule that do not appear in the body.

Likewise, we can pass arbitrary datalog rules to Emerac as well:

```
emerac.addRule("professors(name) :- " +
               "faculty-school(name, school)");
```

Rules cannot be invididividually deleted or modified, and can only be deleted completely. To remove all rules that have been given to Emerac, use the `removeAllRules()` method.

A.1.4 Querying Emerac

Emerac will answer queries that ask for instances of some global schema predicate. The answer to the queries are returned incrementally through an asynchronus interface. To pose a query to Emerac you call `query` method and give it the query (as a string) and a reference to an object that implements the `InformationAcceptor` interface. For example, we might query the global schema we have built up in the previous examples:

```
InformationAcceptor ia = new SimpleInformationAcceptor();
emerac.query("professors(name)", ia);
```

The call to `query` returns immediately, and processing of the query proceeds in a seperate thread. As information satisfying the query is retrieved, it is sent to the `InformationAcceptor` object.

The `InformationAcceptor` interface merely requires the presence of a method called `moreResults`, that accepts a table with some results from the query and a status code. The status code indicates if the plan to answer the query is still being executed and more results should appear, or the execution has been aborted, or the plan has completed execution.

The simplest implementation of an `InformationAcceptor` would be the following, which just prints out all the data returned by Emerac after it has completed execution:

```
public class GatherTest implements InformationAcceptor {
  public void moreResults(int status, Table results) {
    if (status == InformationAcceptor.FINAL_RESULTS)
      System.out.println("Answer: " + results);
    else if (status ==
  InformationAcceptor.EXECUTION_ABORTED)
      System.out.println("Execution Aborted! " +
                   "Partial Results: " + results);
  }
}
```

A.2 Internals

The internals to Emerac are fairly well documented, so this section only describes the common classes and major components of the system. First we describe the common classes that are used for storing and querying relational data, then we describe the classes used for logical reasoning. Second, we describe the architecture of Emerac and the jobs and interfaces of some of the major components.

A.2.1 Relational Classes

Because Emerac is largely a database system, there are classes for storing and dealing with relational data. The `Table` interface encapsulates dealing with the elements of a single relation (which is called a *table* in commercial literature). The `Query` class allows you to specify arbitrary selection queries on a table. The `JoinQuery` class allows you to specify joins and cross products between two `Tables`.

The `Table` interface encapsulates all the methods needed to deal with a table of data. The `InMemoryTable` class implements `Table`, and represents a table of information in memory. The `InformationSourceTable` class presents an information source to Emerac as a table. Emerac treats the information source just as another table, because the `InformationSourceTable` class translates all the operations on the "table" to calls on an `InformationSource` that it holds. For the following examples, we'll make use of the `InMemoryTable` class, but all the methods we use are applicable to all `Table` objects.

A table has a set of attributes (that should probably be called "columns" to stay consistent with the "table" metaphor) that each have a name (also referred to as their "type" in the source code). You can specify the attribute names at once when you create an `InMemoryTable`, or you can add the attributes one by one and give them your own names:

```
// create a table with three columns (attributes), given
// default names (types)
Table tableA = new InMemoryTable(3);

// create a table with two columns, called "one" and "two"
Table tableB = new InMemoryTable();
tableB.addAttribute("one");
tableB.addAttribute("two");
```

At any time you can remove all data and all attributes from a table and start over as if you had just created a new `Table` by using the `reset()` method:

```
// remove all attributes and data from tableA, then give it
// two cloumns, named "one" and "two"
tableA.reset();
tableA.addAttribute("one");
tableA.addAttribute("two");
```

`tableA` is now equivalent to `tableB` from the previous example.

A table holds tuples of data. To add a tuple, you create an `Object[]` array where each element of the array corresponds to an attribute in the table. Adding a tuple to the table is as follows:

```
Object[] tuple = new Object[2];
tuple[0] = "first attribute value";
tuple[1] = "second attribute value";
tableA.addTuple(tuple);
```

All tuples are inserted into the table sorted in order of their first attributes' `hashCode()` values, then by second attributes' `hashCode()`, and so on. The `Table` class has only been tested out with `String` objects as attribute values, but it should work for any object types.

Once a table has been populated with data, the number of tuples is returned by `size()`, an aribtrary tuple can be plucked out with `tupleAt()`, and a check to see if a specific tuple is in the table can be done with `tupleAt()`. In addition relational algebra operations are supported: *project*, *union*, *subtract*, *select*, and *join*. The interface to each operation is obvious, except for the *select* and *join* operations.

A select operation on a `Table` is specified using an instance of the `Query` class. After creating the `Query` object, select constraints have to be added to it, using the `addConstraint()` method. This method requires the index of the attribute that the select is being done on, along with a relational operator for the select (currently only "=" is supported), and a constant value as the argument for the relational operator. For example:

```
Query query = new Query();
query.addConstraint(1, "=", "value");

Table results = new InMemoryTable();
tableA.select(query, results);
```

In this example we're selecting out from `tableA` all the tuples whose first attribute is equal to the string `"value"`. Additional constraints can be added to make the selection more specific.

A join between two tables is specified via a `JoinQuery` object. After creating this object, the user just has to use the `addJoin()` call to specify the indexes of the attributes on each table that are to be joined. In the following example, we join the first attribute of `tableA` with the second of `tableB`:

```
JoinQuery jq = new JoinQuery();
jq.addJoin(1, 2);
```

```
Table results = new InMemoryTable();
tableA.join(jq, tableB, results);
```

If a `joinQuery` has no joins added to it, then a call to `Table.join()` results in a cross product between the two tables.

A.2.2 Logical Classes

In addition to classes for dealing with relational data, there are classes for encapsulating datalog style rules. These are the `Rule`, `Relation`, and `Argument` classes, which represent datalog rules, relations, and arguments to relations, respectively. The `RuleSet` class maintains unions of `Rule` objects. There is also a `BottomUpEvaluator` class which answers a query over some `RuleSet` using semi-naive bottom up evalution.

A `Rule` object contains `Relation`'s and has a type associated with it. The rule is either an lcw statement, materialized view, datalog rule, or rewrite rule (represented by `LCWSTATEMENT`, `SOURCEVIEW`, `WORLDVIEW`, and `REWRITE`, respectively). Likewise, a `Relation` holds one or more `Argument` objects, and is classified as a relation in the global schema, a domain relation, or an information source relation (`WORLDMODEL`, `DOMAIN`, or `INFORMATIONSOURCE`, respectively). The classification for each object is determined from its context when parsed in from a string.

A `RuleSet` object contains a set of `Rule` objects. As new rules are added to the `RuleSet` (via the `addRule()` method), a check is made to ensure the rule does not already exist in the `RuleSet`. Rules can be individually deleted, and the content of two `RuleSet`'s can be unioned together or subtracted.

The `BottomUpEvaluator` class is a fully functioning datalog evaluator. Given a `RuleSet`, a `Relation` that represents the query, constant values for the EDB relations,and a `Table` for results, the `BottomUpEvaluator` will fill the `Table` with all instances that match the query. Here is an example usage:

```
// construct a rule from a string
Parser parser = new Parser(new StringBufferInputStream(
               "query(X, Y) :- edb(X, Y)"));
Rule rule = parser.rule();

// stick the rule in a ruleSet
RuleSet program = new RuleSet();
program.addRule(rule);

// build the evaluator
BottomUpEvaluator bev = new BottomUpEvaluator();
bev.setRules(program);

// add some constant values for the "edb" relation
```

```
Object values[] = new Object[2];
values[0] = "first attribute";
values[1] = "second attribute";
bev.addConstant("edb", values);

// build the query
Relation query = new Relation("query",
      Relation.WORLDMODEL, 2);

// execute the query
Table results = new InMemoryTable();
bev.evaluate(query, results);

// results should now hold { {"first attribute",
//                            "second attribute"} }
System.out.println("answer: " + results);
```

Note that when adding constant values for some relation, the relation is specified using only the name of the relation (as opposed to the name plus the arity). The evaluator assumes that each relation has a unique name.

A.2.3 Information Gatherer

The `InformationGatherer` interface is just a front end for the two subsystems that perform the gathering: the `QueryPlanner` and the `PlanExecutor`. Basically, construction of the original plan and logical rewriting of it is all performed in the `QueryPlanner`, and run-time optimization and execution of the plan is performed in the `PlanExecutor`.

A.2.4 Query Planner

The input to the query planner are the set of rules the user has constructed, mappings from relation names to `InformationSource` objects, and the user's query. The basic protocol with a `QueryPlanner` object is like the following:

```
Hashtable sourceMappings = new Hashtable();
RuleSet rules = new RuleSet();

// ...
// Assume we put (source name, InformationSource object)
// pairs in ’sourceMappings’ and all the rules the user
// has defined into ’rules’
// ...
```

```
QueryPlanner qp = new BasicPlanner();
qp.setRules(rules);
qp.setSourceHash(sourceMappings);

// assume 'queryRelation' is the user's query (as a
// Relation object)
RuleSet plan = qp.plan(queryRelation);
```

The query planner may have the `plan()` method repeatedly called on to answer queries without having to call the `setRules()` and `setSourceHash()` calls again, provided that no rules have been added or removed, and no information sources have been added or removed.

Note that no inversion or processing of the user's rules has occurred before passing them to the query planner. It is the job of the query planner to interpret the rules passed in to the gatherer.

The example above used the `BasicPlanner`. This planner implements the ideas in this thesis. Other planners are: `NaivePlanner` which does basic rule inversion and no LCW reasoning, and `SystematicPlanner` which enumerates all minimal plans after doing inversion and selects the smallest one. The internals of each of these planners are well documented, and each uses the supporting logical and relational classes described above.

A.2.5 Plan Executor

The input to the plan executor is the plan (as a `RuleSet`) produced by the query planner, mappings from relation names to `InformationSource` objects, the user's query, and an `InformationAcceptor` object. The protocol with the `PlanExecutor` is as follows (assume this is occurring directly after the example interaction with the query planner, above):

```
pe.setSourceHash(sourceMappings);

// 'callback' is an object that implements the
// InformationAcceptor interface
pe.execute(queryRelation, plan, callback);
```

The plan executor is expected to begin execution in a seperate thread, and return from the `execute()` method immediately. Just as with the `QueryPlanner` object, if no information sources are added or removed the `setSourceHash()` method does not need to be called before every execution.

There are two classes that implement the `PlanExecutor` interface: `AnnotatedExecutor` and `BasicExecutor`. `BasicExecutor` merely executes the plan passed in to it, and each datalog rule is evaluated from left to right.

`AnnotatedExecutor` runs the same, but it makes use of the % annotations provided by information sources to reorder access to information sources.

The two implemented plan executors convert the information gathering plan passed in into a relational operator graph. Each node in the graph represents a relational operator, and contains links to other nodes that it requires data from. Once the graph is built, the plan is executed by traversing the graph, starting at the `DisplayNode`. When `UnionNode`'s are encountered, new threads of execution are spawned and run in parallel. A special extension of the `InMemoryTable` that enables thread locking, called the `InMemorySynchTable`, is used to pass data between nodes and prevent multiple threads from overwriting each other.

A.3 Tools

Arguably the most difficult part of building an information gatherer is wrapper construction. The ease the pain of performing this, I have created a tool to adapt wrappers generated by the *SoftMealy* algorithm [12] for use with Emerac.

The output of the `SoftMealy` learner is a set of files (with different extensions) that specify extraction rules, information about the page being learned, and test data. These can be converted into a "`.java`" file that can be compiled into an `InformationSource` object. To convert a group of files with a common prefix of, say, "wired" into a java class called "`WiredWrapper`" that belongs to the package "`edu.asu.gathertest`", do the following from the command line:

```
$ java edu.asu.emerac.wrapper.eml.WrapperBuilder wired
edu.asu.gathertest WiredWrapper
```

A file called "`WiredWrapper.java`" will be created that can be compiled. For testing purposes, the compiled class can be run on its own (it has a `main` function) and it will list all the data it can extract from the information source it wraps.