Efficient Execution of Recursive Integration Plans

Snehal Thakkar and Craig A. Knoblock
University of Southern California
Information Sciences Institute & Department of Computer Science
4676 Admiralty Way
Marina del Rey, CA 90292
{thakkar, knoblock}@isi.edu

Abstract

Information integration systems provide a uniform query interface to a set of sources. One of the key challenges for an information integration system is to provide maximally complete answers to user queries and to execute user queries efficiently. We describe an approach to map recursive datalog programs into a streaming, dataflow execution system. We show that our method can be used in conjunction with the Inverse Rules algorithm to create a new information integration system that can provide maximally complete answers to user queries and efficiently execute those queries. Our preliminary results show that in addition to generating maximally complete answers, we obtain performance improvements ranging from 8% to 24.3% over datalog execution.

1 Introduction

Information integration systems, such as, InfoMaster [Duschka, 1997], Ariadne [Knoblock et al., 2001], and Information Manifold [Levy et al., 1996] provide users with uniform access to various sources. One of the challenges for the information integration systems is to reformulate user queries into a query plan containing a set of source queries that provide maximally complete answers to the user query and execute the source queries as efficiently as possible. We show that the state of the art information integration systems can only provide either efficient execution or maximally complete answers to the user queries. In this paper, we describe an information integration framework that utilizes the Inverse Rules algorithm [Duschka, 1997] to reformulate user queries into a set of source queries, maps the resulting set of source queries to integration plans that can be executed by a streaming dataflow style execution system called Theseus [Barish and Knoblock, 2002], and executes the integration plans efficiently using the Theseus execution engine. We show that the information integration framework described in this paper can provide maximally complete answers to the user queries and execute the user query as efficiently as possible. The key contributions of this paper are (1) mapping the set of source queries produced by the Inverse Rules algorithm into an integration plan for a streaming dataflow style execution system, and (2) utilizing the mapping procedure in conjunction with the Inverse Rules algorithm and the Theseus execution engine to create a new information integration system.

The remainder of this paper is organized as follows. Section 2 describes the problem of translating datalog programs to query plans and applying the resulting method to the information integration domain. Section 3 provides a brief overview of a state of the art query reformulation algorithm called the Inverse Rules algorithm and a streaming dataflow execution system termed Theseus. Section 4 describes the process of translating datalog programs to Theseus plans. Section 5 provides initial experimental results of the resulting information integration system. Section 6 describes the related work and Section 7 concludes the paper by discussing our contributions and future work.

2 Problem Description

Datalog has an extensive set of features that can be used to represent different type of query plans. However, datalog execution engines cannot execute several datalog predicates in parallel or stream data between datalog predicates. In this paper we describe a method to translate datalog programs to the programs that can be executed by a streaming dataflow execution engine. A key application of such a method is in the area of information integration. Information integration systems have three key components: (1) query reformulation, (2) query optimization, and (3) query execution. A major challenge for an information integration system is to reformulate user queries into set of source queries that provide maximally complete answers to the user query and execute the source queries as efficiently as possible.

In order to meet this challenge, the query reformulation engine must reformulate the user query into a set of source queries that can be executed by the query execution engine. The limitations of the state of the art algorithms for query reformulation algorithms force the in-

formation integration system developers to pick either maximally complete answers or efficient execution. The MiniCon algorithm [Pottinger and Levy, 2000] (based on the earlier Bucket algorithm [Levy et al., 1996]) and the Inverse Rules algorithm [Duschka, 1997] are two state-of-the-art algorithms to reformulate user queries into a set of queries on the source relations.

The source queries generated by the MiniCon algorithm can be executed efficiently using a streaming dataflow execution system, such as, Tukwila. The Tukwila execution system uses flexible scheduling of queries, pipelining, and convergent query processing to execute the source queries efficiently [Ives et al., 1999]. However, the MiniCon algorithm does not support recursion. Therefore, if the information integration system utilizes the MiniCon algorithm, it may not be able to answer recursive user queries and may lose completeness of the results in the presence of recursive queries. Furthermore, an information integration system may have to reformulate the user query into recursive queries on the sources to satisfy binding patterns. An information integration system that utilizes the Minicon algorithm would not support such queries.

The Inverse Rules algorithm [Duschka, 1997] can reformulate recursive user queries. The Inverse Rules algorithm produces a datalog program, which can be executed using any datalog execution engine. However, datalog execution engines do not have the ability to both execute multiple operations in parallel and stream data between operations. Therefore, at present information integration system developers face a dilemma of picking between the completeness of query results or highly parallel, streaming execution.

In this paper, we describe how we solve this problem by translating datalog programs to information integration plans that can be executed efficiently using the Theseus execution engine [Barish and Knoblock, 2002]. We show that by using our system in conjunction with the Inverse Rules algorithm and the Theseus execution engine, one can develop information integration system that can handle recursive user queries and efficiently execute the queries on the source relations.

2.1 Motivating Example

To better understand the problem addressed in this paper, consider the following scenario. An information integration system for flight schedules can access the following source relation.[1]

NonstopSrc(dep^b, arr^b, airline^f, num^f, deptime^f, arrtime^f, depgate^f, arrgate^f)

NonstopSrc is a source that provides airline, flight number, departure time, and arrival time information about direct flights between any two airports. The *dep* and the *arr* columns in the *NonstoSrc* relation have binding constraints, i.e. the user must provide values for those columns to query the *NonstopSrc* relation. The information integration system allows the user to query the flight information using the following global relations.

```
Flights(dep, arr, airline, num, deptime, arrtime)
GateInfo(airline, num, depgate, arrgate)
Allflights(dep, arr, airline, deptime, arrtime, depgate, arrgate)
```

Here the *Flights* relation provides information about direct flights between any two airports while the *GateInfo* relation provides information about the departure and arrival gates for a given flight. *Allflights* relation provides the information about all direct and connecting flights between any two airports. The information integration system defines the following view definition for the source relation *NonstopSrc*:

```
NonstopSrc(dep, arr, airline, num, deptime, arrtime, depgate,
        arrgate) :-
    Flights(dep, arr, airline, num,deptime, arrtime)^
    GateInfo(airline, num, depgate, arrgate).
```

The Allflights global relation is defined as a part of domain rules to make user queries easier. Here is the definition of the *Allflights* relation:

```
Allflights(dep, arr, airline, deptime, arrtime, depgate, arrgate) :-
    Flights(dep, arr, airline, num, deptime, arrtime)^
    GateInfo(airline, num, depgate, arrgate).

Allflights(dep, arr, airline, deptime, arrtime, depgate, arrgate) :-
    Flights(dep, conarr, airline, num, deptime, conarrtime)^
    GateInfo(airline, num, depgate, conarrgate)^
    Allflights(conarr, arr, airline, condeptime, arrtime, condep-
        gate, arrgate)^
    condeptime > conarrtime.
```

The user can pose queries, such as, find flight information for all flights operated by the 'American Airlines' between the 'LAX' airport and the 'JFK' airport.

```
Q1(deptime, arrtime, depgate, arrgate) :-
    Allflights('LAX', 'JFK', 'AA', deptime, arrtime, depgate,
        arrgate).
```

Using the MiniCon algorithm [Pottinger and Levy, 2000], there is no way to write a recursive query plan to get all direct and indirect flights between the 'LAX' airport and the 'JFK' airport. Using the Inverse Rules algorithm [Duschka, 1997], the user can obtain a datalog program to query all direct and connecting flights between those airports. In this paper, we describe a system that utilizes the Inverse Rules algorithm [Duschka, 1997] to reformulate user queries into queries on the source relations, translate the datalog program produced by the In-

[1] The flight schedule example in the paper is simplified for exposition. Actual flight scheduling system may take into account lots of other factors, such as, the flight frequency, and airfare.

verse Rules algorithm into a Theseus plan, and execute the generated Theseus plan using the Theseus execution engine [Barish and Knoblock, 2002].

3 Background Work

Research described in this paper builds on the Inverse Rules algorithm to reformulate the user queries into queries on the source relations and the Theseus execution engine to execute the information integration plan.

3.1 Inverse Rules algorithm

The Inverse Rules algorithm was utilized by the InfoMaster information integration system [Duschka, 1997]. The key advantages of the Inverse Rules algorithm are the ability to handle recursive user queries, functional dependencies, and access pattern limitations. The information integration systems that use the inverse view algorithm utilize the Local-as-view model [Levy, 2000], i.e. they define the source relations as a view over the global relations. The first step in the Inverse Rules algorithm is to invert the view definitions to obtain definitions for all global relations as view(s) over the source relations. For every view definition, $V(X)$:- $S_1(X_1),..,S_n(X_n)$, where X, X_i refer to set of attributes in the corresponding view or relation, the Inverse Rules algorithm generates n inverse rules, for i = 1,..,n, $S_i(X'_i)$:- V(X), where if $X_i \varepsilon X$, X'_i is the same as X_i else X_i is replaced by a function symbol [Duschka, 1997]. For the given example, the Inverse Rules algorithm analyzes the view definitions and generates the rules R1 through R4.

The second step to reformulate the user query is to union the inverse rules with the query to produce set of datalog rules to answer the user query. The resulting program for our example is shown in Figure 1.[2]

3.2 Theseus Execution Engine

Theseus is based on a streaming dataflow architecture [Barish and Knoblock, 2002]. The tuples of different relations in a Theseus plan are streamed between the operators. Theseus has variety of operators ranging from data access operators to allow easy access to different types of data sources, such as, databases and web pages to data management operators, such as, select, project, and join. Theseus is also unique in its support for recursion among the streaming dataflow systems.

Most operators in Theseus accept one or more relations as inputs and produce one or more relations as outputs. For example, the select operator accepts a relation as input and a select condition and generates a new relation with tuples that satisfy the select condition.

Figure 2 shows graphical representation of the Theseus plan for the rule R4 in the example datalog program. The generated Theseus plan consists of a retrieval operation to extract the flight data from the *NonstopSrc* relation (1). Next, flights from the extracted flights that reach the destination are added to the list of flights that reach the users destination (4a). In parallel, Theseus finds all flights that reach at an intermediate airport, i.e. all flights that may have connecting flights from the intermediate airport to the user's destination (4b). From the set of possible indirect flights, flights that fly to the previously visited airports are filtered out using a select operation (5b). If there are no possible indirect flights that fly to previously unvisited airports, then the list of flights that reach the users destination are routed to a project operator (7a). The output of the project operator is the query result. If there are some possible indirect flights, then the plan calls itself with the indirect flights and a new set of results as new inputs (7b). Theseus can execute multiple operations in parallel, for example operations (4a) and (4b) are executed in parallel. Furthermore, Theseus also streams tuples between operators, e.g. if there were several flights retrieved from the *NonstopSrc* relation, Theseus would stream them one tuple at a time to the select operation (2) and as soon as the select operator was done processing one flight that flight would be passed to the next operation [Barish and Knoblock, 2002].

Schema:

Nonstopsrc(dep, arr, airline, num, deptime, arrtime, depgate, arrgate)

Rules:

R1: Flights(dep, arr, airline, num, deptime, arrtime) :-
NonstopSrc(dep, arr, airline, num, deptime, arrtime, depgate, arrgate).

R2: GateInfo(airline, num, depgate, arrgate) :-
NonstopSrc(dep, arr, airline, num, deptime, arrtime, depgate, arrgate).

R3: Allflights(dep, arr, airline, deptime, arrtime, depgate, arrgate) :-
Flights(dep, arr, airline, num, deptime, arrtime)^
GateInfo(airline, num, depgate, arrgate).

R4: Allflights(dep, arr, airline, deptime, arrtime, depgate, arrgate) :-
Flights(dep, conarr, airline, num, deptime, conarrtime)^
GateInfo(airline, num, depgate, conarrgate)^
Allflights(conarr, arr, airline, condeptime, arrtime, condepgate, arrgate).

Q1(deptime, arrtime, depgate, arrgate) :-
Allflights(LAX, JFK, AA, deptime, arrtime, depgate, arrgate).

Figure 1. Example Datalog Program

4 Translating Datalog Program

The Theseus execution engine can execute the given query much more efficiently compared to any datalog execution engine. However, the result of the Inverse Rules algorithm is a datalog program and Theseus cannot execute datalog programs. In this section we show how the datalog programs generated by the Inverse Rules al-

[2] If any function symbols are generated while inverting the view definition, we can use a flattening procedure described in [Duschka, 1997], to eliminate the function symbols.

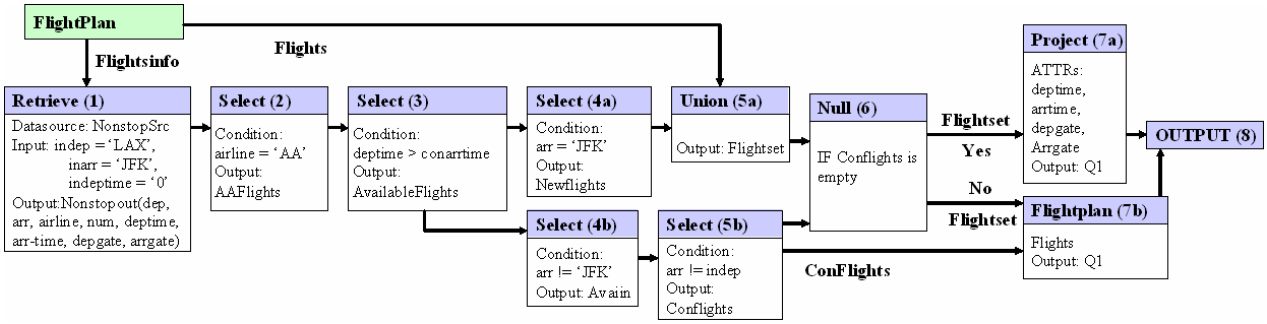

Figure 2: Example Theseus Plan

gorithm can be automatically translated to the plans that can be executed by Theseus.

The process of translating a given datalog program into a Theseus plan starts by validating the given datalog program. The datalog program is first parsed to check the syntactical validity of the program. Next, we check that the queries specified in the queries section can be answered by the rules given in the datalog program. After the validation, each datalog rule is translated into a Theseus subplan. Finally, the system generates a master plan that calls the subplans corresponding to the datalog rules.

4.1 Mapping Datalog Rules to Theseus

In the absence of recursion, datalog rules can be directly mapped into Theseus subplans. The Theseus subplans may accept one or more input relations and generate one of more output relations. Each datalog rule consists of a head and a body. The body of the rule consists of a set of datalog predicates. To translate a datalog rule to the Theseus plan, we map each datalog predicate to a Theseus operation. The rest of this section describes how different datalog predicates are mapped to the corresponding Theseus operators.

Data access: Data access predicates are translated to a retrieval operation in a Theseus plan. For example, *NonstopSrc(dep, arr, airline, num, deptime, arrtime, depgate, arrgate)* is translated to a retrieval operation. A data access predicate may include constants in the attribute list for a relation. Data access statement containing a relation with a constant value for an attribute having a binding constraint, is translated to a retrieval operation with the constant as the input parameter value. For example, *NonstopSrc('LAX', 'JFK', airline, num, deptime, arrtime, depgate, arrgate)*, is translated to a retrieval call with inputs *dep* = 'LAX' and *arr* = 'JFK' (operation 1).

If the attribute list of the relation in the data access predicate contains a constant for a free attribute, then the data access statement is replaced by a retrieval operation followed by a select operation. In our example, data access predicate *NonstopSrc('LAX', 'JFK', 'AA', num, deptime, arrtime, depgate, arrgate)* is replaced by retrieval operation (1) and select operation (2) in the example Theseus plan shown in Figure 2.

Select: Order constraints, such as (x = 5) or (condeptime > conarrtime) are translated into a select operations. The Select operation accepts a relation and a select condition and provides a new relation that contains tuples that satisfy the selection condition. In our example, select predicate (condeptime > conarrtime) is translated to a select operation (operation 3) in the example Theseus plan shown in Figure 2.

Project: A projection in datalog is translated to a project operation. The Project operation accepts a relation and attributes to be projected and provides a new relation consisting of tuples with the specified attributes. In the example query, Q1(deptime, arrtime, depgate, arrgate) is translated to a project operation (7) in Figure 2.

Join: A datalog statement containing two relations with one or more common attribute name specifies a join. If the common attribute name in the join is a free attribute in both relations, then the join is replaced by a join operation. A join operation accepts two relations and a join condition, and outputs a joined relation.

If the common attribute in the join has a binding constraint in one of the relations, then the join is translated into a dependency between two operations in Theseus. For example, the datalog statement *SigmodPapers(title, year)^AuthorInfo(title^b, author, institution)* is translated to a retrieval from *SigmodPapers* followed by a retrieval from *AuthorInfo* data source using the *title* attribute from the *SigmodPapers* data source.

4.2 Generating the Master Plan

Generating a master plan involves both analyzing rules for union predicates and adding calls to subplans corresponding to individual rules. If a datalog program contains one or more unions, then the system generates the Theseus subplans to perform a union operation.

Union: In datalog two statements having the same target relation represent a union. The union is translated to a union operation in Theseus. The union operation accepts two relations as input and provides one output relation that is the union of the two relations. The subplan corresponding to the union datalog predicate contains one of more union operations to union a set of input relations.

Once the subplans corresponding to all the rules are generated, the system generates a master plan to call the generated subplans. The master plan is executed using the Theseus execution engine to obtain the query results.

4.3 Mapping Recursive Rules

The recursive datalog rules are translated to recursive Theseus plans. The recursive Theseus plans are typically divided in five parts: (1) data processing, (2) query results update, (3) loop detection, (4) termination check, and (5) recursive callback.

The first part of a recursive Theseus plan is data processing. Data processing in a recursive Theseus plan may involve accessing data from a data source and processing the data. In the example Theseus plan, operations 1, 2, and 3 perform data processing. This part typically corresponds to the non-recursive part of the datalog statement and is translated in the same manner as the non-recursive datalog statements.

The second part of the recursive Theseus plan is the update of the query results. Recursive Theseus plans may need to keep track of all results that have been acquired through recursion. The query results update segment of the recursive Theseus plan contains statements to update the cumulative result set. In our example, this part is responsible for adding qualifying tuples to the relation *Q1* using a union operation as shown in the example plan in Figure 2.

The third part of the recursive plan is loop detection. In datalog, the interpreter is responsible for handling loops. Therefore, the datalog programs do not require explicit statements to perform loop detection. Theseus does not automatically handle loop detection. Therefore, when translating datalog programs to Theseus plans, we must add Theseus operations to handle loop detection. Intuitively, recursion can be viewed as a graph traversal problem, where each recursive step is to follow an edge from one node in the graph to the other. We handle loop detection in recursive plans by keeping track of all visited nodes in the graph and in each recursive step only follow the edges that lead to unvisited node. The attribute corresponding to the nodes in the graph is the attribute involved in the recursive join condition. In the given example, *conarr* attribute in the *Flights* relation corresponds to nodes in the graph. Next, a select to filter out previously visited nodes is added to the plan. In the given example, a select operation (5b) is used to filter out tuples with previously visited destination airports.

The fourth part of the recursive Theseus plan is to check for the termination condition. When translating datalog programs to Theseus plans, the termination condition is satisfied when no new input tuples can be found. In the example plan, null operation (operation 6) checks for the termination condition. Finally, the last part of the recursive Theseus plan is a recursive call if the termination condition is not satisfied.

5 Experimental Results

The goal of our initial experiments was to demonstrate that our system provides maximally complete answers to user queries and provides better response time to answer the user queries compared to a datalog execution. To facilitate experimental evaluation, we used flight schedules from American Airlines, Northwest Airlines and Delta Airlines consisting of about 15,000 nonstop flights between over 250 different airports. The flight schedule and airports data was stored in Microsoft Access databases with the same schema as the example shown in Section 2.1. We implemented two mediator systems with the example global schema. Both mediator systems utilized the Inverse Rules algorithm to reformulate user queries into source queries. One mediator system translated the datalog programs generated by the Inverse Rules algorithm to plans for the Theseus execution engine and executed the resulting plans using Theseus. The other mediator system executed the datalog program generated by the Inverse Rules algorithm using the Theseus execution engine without streaming and parallel execution.

Both mediator systems executed following three queries: (1) Find all direct flights between the given airports, (2) Find all flights between the given airports, and (3) Find all airports that can be reached from the given airport. The MiniCon algorithm can only reformulate the first query into source queries. The MiniCon algorithm cannot reformulate user queries (2) and (3) as it does not support recursion. As shown in Table 1, the mediator with the Theseus execution system executes all queries more efficiently compared to the simulated datalog execution engine. In fact, the example plan described in this paper does not benefit a lot from parallel execution as there is only one data source. We plan to perform more experiments with plans that more closely model the way data sources on the web would be accessed.

6 Related Work

Work on parallel execution strategies for logic programs execution has mainly focused on either assigning different operators to different processors or assigning different data to different processors [Cacace et al., 1993]. In addition, parallel execution strategies described in the survey do not support recursion. Using our methods, we can map both recursive and non-recursive datalog to plans for highly parallel streaming execution engine.

In [Kambhampati et al., 2003], the authors describe strategies to optimize the recursive and non-recursive datalog programs generated by the Inverse Rules algorithm. The research focus of their work is to remove redundant information sources and to order access to different sources to reduce the query execution time. The focus of our work is to execute the given datalog program using a highly parallel, streaming execution engine. In that sense, the two works are complementary to each other as we can optimize the datalog programs using the algorithms described in [Kambhampati et al., 2003]. The resulting optimized programs can be translated to Theseus plans using the technique described in this paper.

In [Ullman, 1988], the author describes various techniques, such as, semi-naïve execution, to optimize the execution of datalog programs. However, none of the techniques can execute multiple datalog predicates in

Table 1 Experimental Results

Mediator/ Query	Datalog execution (ms)	Theseus execution (ms)	% Improvement
Find flight information for all direct flights between airport a and airport b	12	11	8.33%
Find flight information for all flights between airport a and airport b	32	27	15.63%
Find all airports that can be reached from	37	28	24.32%

parallel or stream tuples between different predicates. There has been some work on streaming query execution in the data integration community [Barish and Knoblock, 2002; Hellerstein et al., 2000; Ives et al., 1999; Naughton et al., 2001]. However, these streaming query execution engines cannot directly execute datalog programs.

7 Conclusions and Future Work

In this paper, we describe our research on mapping datalog programs to programs for streaming dataflow style execution engines. We described a mediator system that utilizes the Inverse Rules algorithm to reformulate user queries into a datalog program consisting of a set of source queries. The resulting datalog program is translated to an integration plan for the Theseus execution engine using our technique to map datalog program to the integration plan for the Theseus execution engine. The resulting execution plan is executed in highly parallel and streaming manner using the Theseus execution engine. We show that we get significant performance improvements compared to the mediator that executes the datalog program using a simulated datalog execution engine.

Acknowledgements

We would like to thank Dr. Jose Luis Ambite for his comments on various issues in this paper. This material is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract/agreement numbers F30602-01-C-0197 and F30602-00-1-0504, in part by the Air Force Office of Scientific Research under grant numbers F49620-01-1-0053 and F49620-02-1-0270, in part by the United States Air Force under contract number F49620-02-C-0103, and in part by a gift from the Microsoft Corporation.

The U.S.Government is authorized to reproduce and distribute report for Governmental purposes notwithstanding any copy right annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[Barish and Knoblock, 2002] G. Barish and C. A. Knoblock. *An Expressive and Efficient Language for Information Gathering on the Web*. Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-2002) Workshop: Is There Life Beyond Operator Sequencing? - Exploring Real-World Planning, Toulouse, France, 2002.

[Cacace et al., 1993] F. Cacace, S. Ceri and M. A. W. Houtsma. A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs. *Distributed and Parallel Databases* **1**(4): 337-382, 1993.

[Duschka, 1997] O. M. Duschka. Query Planning and Optimization in Information Integration. Ph.D. Thesis, Computer Science, Stanford University, 1997.

[Hellerstein et al., 2000] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman and M. A. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 2000.

[Ives et al., 1999] Z. G. Ives, D. Florescu, M. Friedman, A. Levy and D. S. Weld. *An Adaptive Query Execution System for Data Integration*. ACM SIGMOD Conference, 1999.

[Kambhampati et al., 2003] S. Kambhampati, E. Lambrecht, U. Nambiar, Z. Nie and S. Gnanaprakasam. Optimizing Recursive Information Gathering Plans in EMERAC. *To appear in Journal of Intelligent Information Systems*, 2003.

[Knoblock et al., 2001] C. Knoblock, S. Minton, J. L. Ambite, N. Ashish, I. Muslea, A. Philpot and S. Tejada. The ARIADNE Approach to Web-Based Information Integration. *International Journal on Intelligent Cooperative Information Systems (IJCIS)* **10**(1-2): 145-169, 2001.

[Levy, 2000] A. Levy. Logic-Based Techniques in Data Integration. Logic Based Artificial Intelligence. J. Minker, Kluwer Publishers, 2000.

[Levy et al., 1996] A. Y. Levy, A. Rajaraman and J. J. Ordille. *Querying Heterogeneous Information Sources Using Source Descriptions*. Proceedings of the 22nd VLDB Conference, Bombay, India, 1996.

[Naughton et al., 2001] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin* **24**(2): 27-33, 2001.

[Pottinger and Levy, 2000] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries Using Views. *VLDB Journal*: 484-495, 2000.

[Ullman, 1988] J. Ullman. *Principles of Data and Knowledge-Base Systems*. New York, Computer Science Press, 1988.