Querying Web-Sources within a Data Federation

Lynn Wu
Aykut Firat
Tarik Alatovic
Stuart Madnick

Working Paper CISL# 2006-09

August 2006

Composite Information Systems Laboratory (CISL)
Sloan School of Management, Room E53-320
Massachusetts Institute of Technology
Cambridge, MA 02142

Querying Web-Sources within a Data Federation

Web-based Information Systems and Applications

Lynn Wu
Massachusetts Institute of Technology
Cambridge, MA
USA
lynnwu@mit.edu

Aykut Firat
Northeastern University
Boston, MA
USA
a.firat@neu.edu

Tarik Alatovic
Massachusetts Institute of Technology
Cambridge, MA
USA
talatovic@yahoo.com

Stuart Madnick
Massachusetts Institute of Technology
Cambridge, MA
USA
smadnick@mit.edu

Abstract

The web is undoubtedly the largest and most diverse repository of data, but it was not designed to offer the capabilities of traditional data base management systems – which is unfortunate. In a true data federation, all types of data sources, such as relational databases and semi-structured websites, could be used together. IBM WebSphere uses the "request-reply-compensate" protocol to communicate with wrappers in a data federation. This protocol expects wrappers to reply to query requests by indicating the portion of the queries they can answer. While this provides a very generic approach to data federation, it also requires the wrapper developer to deal with some of the complexities of capability considerations through custom coding. Alternative approaches based on declarative capability restrictions have been proposed in the literature, but they have not found their way into commercial systems, perhaps due to their complexity. We offer a practical middle-ground solution to querying web-sources, using IBM's data federation system as an example. In lieu of a two-layered architecture consisting of wrapper and source layers, we propose to move the capability declaration from the wrapper layer to a single component between the wrapper and the native data source. The advantage of this three-layered architecture is that each new web-source only needs to register its capability with the capability-declaration component once, which saves the work of writing a new wrapper each time. Thus the inclusion of web-sources through this mechanism can be accelerated in a way that doesn't require a change in existing data federation technology.

Keywords: federated data, web data sources, capabilities, query handling

1. Introduction

The web is undoubtedly the largest and most diverse repository of data, but it was not designed to offer the capabilities of traditional database management systems – which is unfortunate. In a true data federation, all types of data sources, such as relational data bases and semistructured websites, could be used together.

Commercial database systems such as Oracle, DB2, and SQL Server can incorporate heterogeneous data sources with varying capabilities such as flat files, XML, Web Services, and functional sources (e.g., BLAST) into a data federation so that they can be queried as if they are part of a single large database. Incorporating a dynamically generated website (e.g., Yahoo! Finance[1]) into a data federation, however, is still a relatively complicated task requiring manual intervention. For example, it is not possible to retrieve all stock quotes in Yahoo! Finance, because it requires specific ticker symbols to retrieve stock quotes.

Although IBM DB2, for example, comes with built-in wrappers for flat files, static XML pages, Web Services, and BLAST, there is no general-purpose wrapper that supports querying dynamically generated web pages. This is a difficult problem because these web pages can impose arbitrary restrictions before returning any data. Using IBM DB2, which has one of the most advanced tools in this area, one is expected to code a wrapper each time a new website needs to be added into a data federation. This is a daunting task considering the vast number of websites on the internet. Using an alternative approach, a single versatile wrapper that can deal with a multitude of websites can be coded by parameterizing site addresses, extraction patterns of data (e.g., regular expressions), and capability restrictions (e.g., bound variables, operator restrictions). This wrapper, however, still has to be intelligent enough to do some non-trivial planning, optimization and execution when the federated data planner pushes down a query request involving multiple web-sources with varying capabilities to this wrapper.

The core difficulty in this case is that websites can have arbitrary capability restrictions. Commercial products, such as IBM DB2's "request-reply-compensate" protocol (RRC), resolve this problem by pushing it to the individual wrappers. In this case, RRC expects wrappers to reply to query requests by indicating the portion of the queries they can answer, and then tries to compensate for the limitations of the native data sources. However, this approach fails for some websites. For example, Yahoo! Finance will not return the closing prices of more than 50 company symbols at a time. As a result, the wrapper has to deal with the restriction of the native data source by itself. It should also be noted that virtual[2] Web Services cannot solve this problem. Alternative approaches using context-free grammars to express capability restrictions of non-relational data sources have been proposed in the literature, but they have not found their way into commercial systems, perhaps due to their complexity.

We offer a systematic middle-ground solution to querying dynamic web-sources within a data federation system under the existing data federation technology. In lieu of a two-layered architecture consisting of wrapper and source layers, we propose to move the capability declaration from the wrapper layer to a single component between the wrapper and the native data source. Capability restrictions are no longer buried inside the wrapper code, but turned into simple declarative expressions that both the wrapper and the data federation engine can use to resolve the query. The advantage of this three-layered architecture is that wrapper code need not be modified each time a website with a new restriction joins the federation. Thus, the generic inclusion of web-sources through this mechanism can be accelerated in a way that doesn't require a change in existing data federation technology.

The overall architecture of the system is presented in Figure 1. The diagram on the left of the arrow illustrates how the current technology, such as IBM DB2, incorporates web-sources into a data federation. For example, as illustrated in the diagram, if a new source (s4) is to be added into a data federation, a new wrapper for the source needs to be coded. This new wrapper not only has to deal with capability restriction on the source but also data extraction from the web page as well. This can be a very cumbersome process when adding a large number of websites into the data federation. However, the majority of capability restrictions can be captured using simple declarative expressions. Taking this approach means that instead of dealing with each specific capability restriction, the wrapper would simply need to handle a set of generic ones. Motivated by this observation, we extract the capability handling component out of the wrapper, as illustrated in the new architecture, depicted on the right side of Figure 1. When a new website is added to the data federation, one would only need to declare a capability record

[1] http://finance.yahoo.com/

[2] A web service that provides the WSDL and code to transform a third-party web page into a web service.

describing the source. For example, when adding Yahoo! Finance into a data federation, only a simple capability record that describes the source restriction is needed (Yahoo! Finance requires one or more ticker symbols, but not more than 50, in order to retrieve any stock quotes.) Since the wrapper already knows how to handle generic binding restrictions, adding the Yahoo! Finance source requires no changes to the underlying wrapper[3]. Once the capability record and the extraction rules for the native source are declared, the data federation engine can resolve any query to the native source as long as it can satisfy the capability restriction. The detail of this process is illustrated in Section 4. Using this approach, adding new web-sources into a data federation is tremendously expedited.

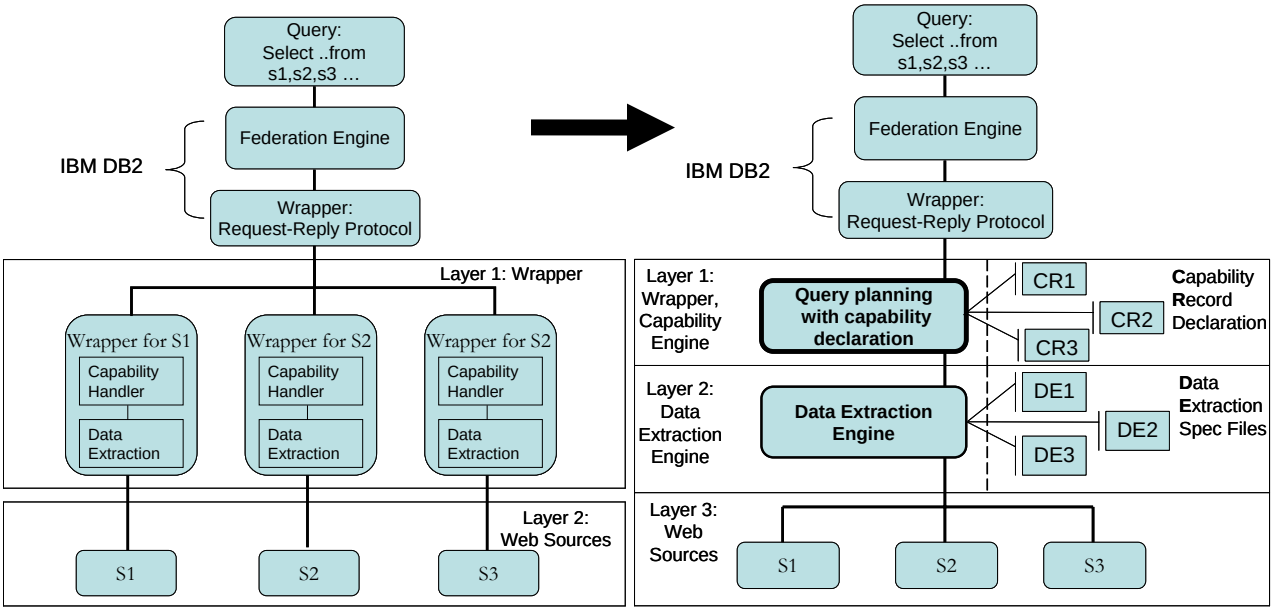

Figure 1. The overall architecture: (a) the left figure is the traditional two layered approach; (b) the right figure is the proposed three-layered architecture with added capability declaration.

In the new architecture shown on the right side of the arrow in Figure 1, we use IBM DB2 as the data federation engine and the Cameleon web wrapper for data extraction purposes. Similar to our generic wrapper that can handle capability records, Cameleon is also a general purpose data extraction engine that uses declarative specification files to extract data from the web. Using this new architecture, the query is first sent to the IBM data federation engine, which uses the request-reply-compensate (RRC) protocol to communicate with our generic wrapper that handles the capability restriction. Together with the RRC, the generic wrapper creates a query execution plan and accesses the Cameleon wrapper for data extraction. The Cameleon wrapper then uses declarative specification rules to extract data from the native websites and return the results to the wrapper.

This paper makes two important contributions, which are summarized below:

First, this paper provides a complete solution that can expedite the process of incorporating the web into existing data federation systems. We take a practical middle ground approach that takes advantage of the flexibility of existing federation systems while incorporating the ability to describe source restrictions advocated by the context-free grammar approach. Using this approach, adding a new source can be as simple as creating a capability record, as long as there is already a data extraction mechanism for the source, such as Cameleon. This approach significantly reduces the amount of coding necessary to access the vast majority of new web pages.

Second, this paper describes a simple capability record that is used by both the federation engine and the wrapper. Previously, access to the capability records was buried inside the wrapper; however, by abstracting this component out of the wrapper, the data federation engine can also use it to create an optimal query plan. The format of the

[3] This is assuming the data extraction for Yahoo! Finance has already been provided. Our Cameleon web wrapper provides tools that can easily extract data from web pages using a specification file [Firat, 2000].

capability record is very simple but powerful enough to describe the key limitations of many web-sources. Using these simple capability records allows our generic wrapper to handle query planning for most websites.

In the rest of the paper, we describe the architecture of such a generic wrapper interacting with a data extraction system, and the algorithms used to perform planning and optimization with simple capability records. Using this approach, we preserve the generic solution offered by commercial data federation systems, while easing the burden of the wrapper developers dealing with complex capability issues. First we describe the related work in the next section. In Section 3, we first describe the problem we are addressing in more detail. Then we provide background on data federation with non-relational data sources, and the data extraction engine we use to convert websites into limited relational data sources. In section 4, we describe our proposed solution. We end with a discussion of future work and conclusions in sections 5 and 6.

2. Related Work

This work is built upon two streams of research: web wrapping technology and query planning with source capabilities. Many wrapping technologies have focused on data extraction from a website. Usually a wrapper is provided with a specification file that describes the general extraction rule for a class of web pages. Our research makes use of the data extraction technology provided by this class of research, specifically Cameleon [Firat 2000], for data retrieval purposes. Instead, we focus on query planning for data sources with arbitrary capability restrictions. In the past, handling of capability restrictions was buried within the wrapper code. However, by providing a simple and declarative capability record, we are able to bring the capability restriction layer out of the wrapper. By creating a generic wrapper that can handle capability restrictions described by our simple capability record, we allow rapid incorporation of web-sources into data federation without modifying any existing code.

There are a number of efforts to develop languages for describing the capability restrictions imposed by sources. There are two types of approaches in the existing literature: using the wrapper to deal specifically with each source, and using complex descriptive languages to describe capability restrictions. The IBM Websphere project follows the first approach. It relies on Request-Reply-Compensate protocol to communicate with the wrapper. Handling all capability restrictions is pushed down to the wrapper layer. Although this is a generic framework to incorporate many different sources, coding a different wrapper every time for a website with different capability restrictions can be extremely wasteful, since most of the code between these wrappers will be common.

There are many projects that follow the other approach by describing capability restrictions with complex descriptive language. These systems are generally very powerful and are designed specifically to solve the capability restriction problem. As the result, it is often very hard to incorporate them into existing data federation technologies. In order to take advantage of these systems, the existing data federation technologies would need to be modified significantly. Two projects, TSIMMIS [Chawathe 1994] and DISCO [Tomasic 1998] are examples of this approach that uses context-free grammar. Both attempt to provide a general solution to describe capability restrictions of a source. Although context-free grammar is the most expressive in describing capabilities, this technique reduces the efficiency of capabilities-based rewriting because it treats queries as "strings." Li [Li 2000] builds on top of TSIMMIS by focusing on using sources that are not directly mentioned in the query to retrieve results. Using this approach, their system is able to answer queries that do not satisfy capability constraints. However, it suffers from the fact the entire data federation engine may need to be rewritten to take advantage of their algorithm. The Garlic [Roth 1997] and Information Manifold [Levy 1996] projects incorporate capabilities using query rewriting. Although they do not use context-free grammar, their capability descriptions are significantly more complex than our approach. Incorporating them into existing federation system is still a challenging task.

Each approach has its own shortcoming: complex descriptive languages are general solutions but are too complex for existing data federation engines, while writing different wrappers for each website is too labor-intensive. We use a simple and declarative capability records that can describe most of the capability restriction imposed by the web - source. Although our capability records are by no way exhaustive, it significantly cuts down the work required for query planning while reducing the complexity of incorporating them into existing data federation solutions.

3. Background

The goal of a data federation system is to allow clients to access diverse and distributed data sources, regardless of location, format, or access language, from a single interface. While data federation may have a slower access performance compared to data consolidation (as in data warehousing), it has the benefits of (i) reduced

implementation and maintenance costs, (ii) access to current data from the source of record, and (iii) combining traditional data with mixed format data [IBM]. A sample data federation architecture, based on IBM DB2, is shown in Figure 2.

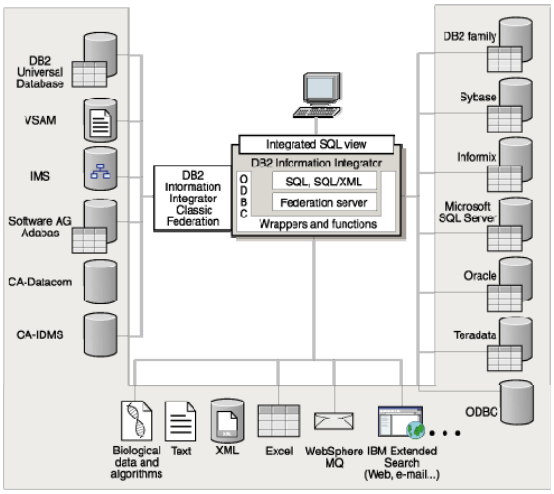

Figure 2. IBM DB2 Data federation Architecture (Adapted from [IBM])

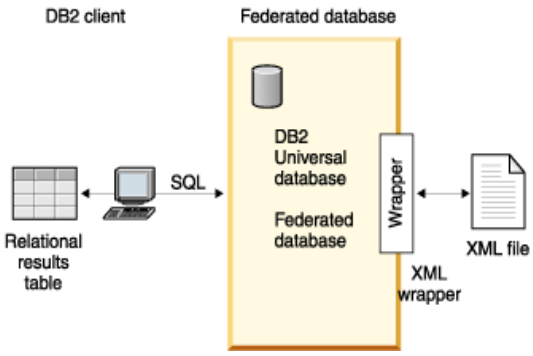

Figure 3. DB2 XML Wrapper (adapted from [IBM])

Data federation systems use wrappers to access non-relational data sources such as flat files, XML pages, and Web Services. Wrappers are programs that accept relational query requests, communicate with the native data source through an API to satisfy the request, and return the data in relational format. In Figure 3, the XML wrapper's interaction with the IBM DB2 is illustrated.

When a user submits a query, the federated server designs access plans for evaluating the query in collaboration with wrappers for each different data source. Such a plan might call for parts of the query to be processed by the wrappers, by the federated server, or partly by the wrappers and partly by the federated server. The federated server chooses among the plans primarily on the basis of cost. Upon receiving requests from the federated server, the wrappers retrieve data through the API of the native data sources and return the appropriate data to the server. After receiving all data from the wrappers, the federated server will also compensate for work that data sources were unable to perform.

The first part of our architecture uses data federation. Although any commercial data federation technology could be used, we find IBM DB2 to be the most useful tool on the market for our purposes.

3.1 Request-Reply-Compensate Protocol (RRC)

IBM DB2 uses the RRC protocol to communicate between the federated server and wrappers. During query planning, the federated server generates query fragments which are sub-pieces of the original query submitted by the user. A query fragment can contain tables, predicates, and head expressions. Head expressions are expressions found in the SELECT clause of a query. The optimizer then submits each query fragment to a wrapper in a request.

Upon receiving the request, the wrapper indicates which sub-pieces of the fragment it can evaluate, and puts this information in the reply to the request. Typically, a reply object can contain accepted relations, predicate or any head expressions. Request properties such as cost, cardinality and ordering properties can also be included. For a typical request, a wrapper could return zero, one or more reply objects. Each reply represents a different accepted fragment.

By the end of query planning, the federated server will weigh all the cost estimations and determine a query execution plan incorporating some set of the accepted fragments offered up by the wrapper in response to requests. During query execution, the federated server will ask the wrapper to execute these query fragments. The federated

server will also compensate for any query fragments which have not been accepted. Examples of this include a complex predicate or a sort that is beyond the capability of the data source in question. This includes all cross-source joins as well as any other expressions (such as function invocations) that mix data from multiple sources since each fragment is single-source.

An example situation is illustrated in Figure 4. The query fragment (`SELECT Name, Rate + Tax FROM Hotels WHERE Stars=3 and Rate < 120`) is passed to the wrapper as a request by indicating the head expressions, table name, and the predicates. The wrapper cannot handle the complete request as it cannot do the `Rate + Tax calculation` and cannot do two predicates at a time, so replies with two separate parts, which when combined in the federated server answers the original query.

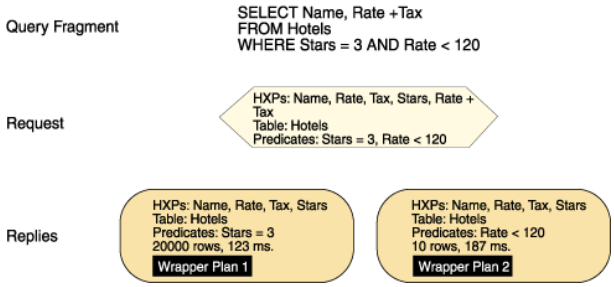

Figure 4. Request-Reply-Compensate protocol example
Adopted from [IBM]

The request-reply-compensate protocol offers a generic framework allowing the federated server to communicate with non-relational data sources through a black box wrapper. Among the built in wrappers that comes with IBM DB2, however, there is none that can deal with dynamically generated websites; therefore, such a wrapper has to be coded before such sites can be included in the data federation. This is particularly important since the web has become an enormously important and extensive repository of semi-structured information. It is not efficient, however, to code a new wrapper for each website to be included in the data federation even though most of the code between these wrappers will be common. Instead, we propose using a declarative "web wrapper" engine that can turn a website into a limited relational data source. This declarative web wrapper has two major components: capability handler and data extraction engine. The detail of the capability handler is described in section 4, while the data extraction engine works by simply defining extraction rules for a given site. Cameleon web wrapper is one such engine that can be used for data extraction purpose.

3.2 Cameleon Web Wrapper

Cameleon is a general-purpose web wrapper engine that extracts data from web pages using declarative specification files, which contain traversal rules to navigate to and through a website, input parameters expected from the user, extraction patterns for the data to be extracted, and advanced handling of cookies, javascript, etc. An example specification file, used by Cameleon, is shown in Figure 5 corresponding to the Expedia website. Creating that file is enough to treat Expedia as a limited relational source that provides price and airline information given the arrival and departure dates and locations. More detail on Cameleon can be found in [Firat 2000].

.

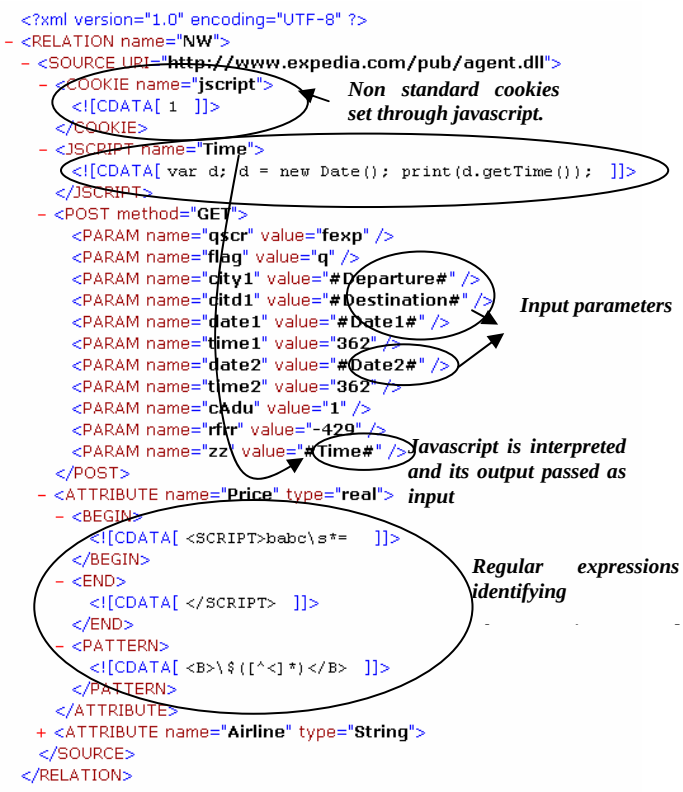

Figure 5: A Spec File Example for Cameleon

Although using Cameleon for "wrapping" websites saves a significant amount of work, there are still important problems related to capability restrictions in fully incorporating web-sources into a data federation. As a simple example, consider a finance website that provides the closing price and headline news for given companies and has been turned into a relation using Cameleon: **Quicken (Ticker, Close, News).** This website, however, only accepts one ticker at a time. So our wrapper without any extra coding cannot answer a simple query like:

```
SELECT *
FROM Quicken
WHERE Ticker IN ('IBM', 'ORCL', 'MSFT');
```

In fact, this query needs to be turned into three separate queries, each involving a single ticker and executed in parallel. Currently, there is no known way to communicate even such a simple restriction within the Request-Reply-Compensate protocol. A more complicated example would be a query as follows:

```
SELECT *
FROM Yahoo
WHERE Yahoo.Ticker  IN
(SELECT Ticker FROM NASDAQ);
```

In this example, **yahoo** is a Cameleon source that accepts at most 50 ticker symbols at a time, and **NASDAQ** is another Cameleon source that returns all the tickers of companies that are in **NASDAQ**. This query should be run in chunks of 50 tickers at a time. Yet again, this restriction cannot be communicated within the Request-Reply-Compensate protocol.

4. Our Solution

We propose and implemented a solution that allows websites with capability restrictions to be easily incorporated into a data federation system. This solution involves a secondary mini query planner/optimizer/execution component based on source capability records [Alatovic, 2001], [Fynn, 1997]. Unlike the planner of the federated data server, this mini planner is limited in scope and is designed specifically to work with capability records describing web-sources. The solution does not require any changes in the existing data federation technologies, but is still able to offer a systematic and accelerated approach to querying web-sources in a data federation system.

Using this system, incorporating a new web-source is as easy as adding a nickname to a federated database. The nickname declaration has a capability-record option which is used by the wrapper to understand the limitation of the web-source. There is no need to modify any existing wrapper code at all as long as it already has data extraction rules defined for the Cameleon web wrapper. Because nicknames can be used in conjunction with other nicknames, multiple websites can now be used together in a query. For example, the query below allows users to request stock quotes of all the companies in a specific industry.

```
select yahoo.ticker, yahoo.lasttrade
from ("select companyticker from companytable where industry='Biotechnology'") AS x, yahoo where
yahoo.ticker = x.companyticker
```

This query would have been unanswerable by a Cameleon wrapper designed only to handle queries of a single web-source. However, with the help of data federation, answering queries involving multiple sources become possible.

In order to answer this query, the federated engine needs to access two separate websites—**Yahoo! Finance** (http://finance.yahoo.com/) and a website describing all companies in a specific industry (http://biz.yahoo.com/p/industries.html). The federated engine can gain use of these websites by adding a nickname along with the capability records[4] for each of the website. Yahoo! Finance allows the ticker to bind to 50 company names at a time while company-names website requires the industry name to take a single value. By leveraging query semantics capability of the IBM federated engine, our systems understands that company names must be retrieved before getting the stock quotes. Using the capability records, our system can effectively retrieve the stock quotes by binding up to fifty company values to the ticker at a time.

4.1 System Overview

The challenge for this mini query planning engine is to create a plan that can efficiently retrieve remote data while satisfying query restrictions. Generally, a query planning engine needs to decompose the original query into component subqueries (CSQ), such that each CSQ can be answered using a single data source [Alatovic, 2001], [Flynn, 1997]. Our mini query planning engine does not need to perform the decomposition since the IBM federated engine already divides the original query into small query fragments, known as requests, where each request can be processed by a single data source. In the following sections we will refer to request objects as CSQs. In addition to query decomposition, a query planning engine also needs to maintain the CSQ execution order. Typically, independent CSQs are executed first, followed by dependent CSQs that can be answered using prior results. Thus, understanding the dependencies among the CSQs is important. Our query planning engine uses both the IBM federated engine and capability records to analyze CSQ dependencies. When the CSQ dependency can be determined using query semantics, our query planning engine uses the IBM federated engine. However, when a CSQ does not meet all the capability restrictions based on the capability record, the query planning engine will determine if information from other parts of the query can be used to satisfy the capability restriction. If the restriction can be satisfied, the CSQ will be modified with the required information so that it can be answered by the native data source.

[4] This assumes that the Cameleon web wrapper already knows how to answer queries for the individual website already. In this case, Cameleon knows how to answer queries to **Yahoo! Finance** and also queries to the website containing company names.

4.2 Query Restrictions

Because many data-sources on the web cannot provide full relational capabilities, the CSQs must be designed to obey these restrictions. For example, the relation **olsen** in Figure 6 is a currency conversion provided by the website Oanda[5] (from Olsen & Associates). This website requires the attributes **expressed, exchanged**, and **date** to be bound and attribute **rate** to be free. These restrictions are necessary because the native source requires the values of **expressed, exchanged** and **date** to retrieve a web page with the desired exchange rate.

A capability record describes restrictions a query must obey when querying the relation. It is represented by a binding restriction component and an operator restriction component. The binding restriction component is a list of all possible binding combinations of the attributes in the relation. The restriction on each attribute is then described using binding specifiers. Specifically, the binding specifier "b" indicates that the attribute has to be bound; "b(N)" indicates that the attribute can be bound with up to N values, which is also known as N keys-at-a-time binding restriction; "f" indicates that the attribute must be free; and "?" indicates that the attribute can be either bound or free. In the example shown in Figure 6, the binding restriction describes that attributes **exchanged, expressed,** and **date** must have one binding value while the attribute **rate** must remain free. Typically, the restriction list only contains one binding combination, but it is possible to have more than one. For example, the **olsen** relation could have another binding combination where attributes **exchanged** and **expressed** both need just one binding value, the attribute **date** can have two binding values and the attribute **rate** must remain free: ([[b(1),b(1),f,b(2)]].

The second component of a capability record is the operator restrictions. It is a list of operators that cannot be used when querying the relation. The example in Figure 6 shows that the **olsen** relation cannot handle queries containing the following operators: '<', '>', '<>', '<=', and '>=' (i.e., it can only handle the '=' operator.)

Capability records are very important for querying web-sources because they can describe key-at-a-time restrictions which are quite common among web-wrapped relations. A key-at-a-time restriction specifies how many values an attribute can take to process a query. For example, as shown in Figure 6, the key-at-a-time restriction requires that all binding restrictions to take one parameter at a time, but binding more than one key at a time (N>1) is also common. An example of this is a stock quote server like finance.yahoo.com, which allows up to 50 stock quote symbols to be entered at a time. Although capability records can describe many general restrictions on a website, it should be noted that they do not at all exhaust all the possible capability restrictions present on any website. The goal of this capability record is to have simple expression describing the capability restrictions and yet general enough that it covers the majority of the restrictions.

```
relation(cameleon,
        olsen,
        [['Exchanged',string],['Expressed',string],['Rate',number], ['Date',string]],
        cap([[b(1),b(1),f,b(1)]], ['<','>','<>','<=','>=']]).
```

A relation captures schema information and its capability record. A capability record has two components: binding restriction (cap([[b(1),b(1),f,b(1)]]) and operator restriction (['<','>','<>','<=','>=']).

Figure 6: The schema of the olsen relation and its capability record.

4.3 Query Execution Plan

Query execution plan (QEP) is the core of our system. It uses knowledge of query fragment dependency, capability restriction and other measures to specify an execution order of the CSQs in a query. Typically, CSQs with no dependencies are executed first, followed by those with data dependencies that can be satisfied using prior results.

[5] http://www.oanda.com/

The simplest case for a query execution plan is when all the CSQs can be executed independently. This happens when all of them meet the capability restriction imposed by their native data sources, so that all the query fragments can be executed independently and in parallel. In this case, the IBM federated engine simply decomposes the original query into CSQs and sends them to the native sources through wrappers. After receiving all processed row sets from the native sources, the IBM federated engine aggregates the data and returns the final result.

However, when a CSQ cannot be executed by itself, it is necessary to determine if the CSQ can still be processed using results from other CSQs. Two procedures are used to determine the dependency: the first method relies on the IBM federated engine to detect dependencies using query semantics; the second method employs the capability records to meet any unsatisfied restrictions using information from other processed CSQs. The next two sections describe in detail how the two procedures work and how they compensate for each other.

4.4 Dependencies Maintained by IBM Federated Engine

The IBM federated engine has no notion of capability restrictions that are imposed by the native data source. However, it uses query semantics to determine if any dependencies exist between the CSQs. Typically the IBM federated engine searches for the presence of any unbound parameters in the CSQ and determines if any other CSQ can supply the missing value.

Figure 7 shows an example where the IBM federated engine alone can determine the CSQ dependencies. In this example, the original query is decomposed into an inner-select CSQ, which can be executed independently, and an outer-select CSQ, which depends on the data returned by the inner-select CSQ. The IBM federated engine detects the dependency because the attribute **ticker** in the dependent CSQ needs to be bound dynamically at runtime. The engine then tags this attribute with a type—unbounded kind—to signal to the wrapper that the binding values would be available after the inner-select CSQ is executed. Once the result from the inner-select CSQ is returned, the wrapper creates a new set of CSQs by replacing the "unbound type" tag in the original CSQ with the returned result. In this example, illustrated in Figure 8, since four company names are returned from the inner-select CSQ (ACAD, ACAM, ACOR, ACEL), a set of four new CSQs are created after binding each company name to the **ticker** attribute. The query planning engine would then send this new set of CSQs to the native data source. Once the native sources have processed the CSQs, the wrapper assembles the results and returns them to the user. In this example, the wrapper sends the four queries to **Yahoo** and retrieves the last-traded stock values for the user.

SELECT YAHOO.TICKER, YAHOO.LASTTRADE FROM ("SELECT COMPANYTICKER FROM COMPANYTABLE WHERE INDUSTRY='BIOTECHNOLOGY' AND COMPANYTICKER < AD") AS X, YAHOO WHERE YAHOO.TICKER = X.COMPANYTICKER

Independent query fragment	**SELECT COMPANYTICKER, INDUSTRY FROM COMPANYTABLE WHERE INDUSTRY = BIOTECHNOLOGY AND COMPANYTICKER < AD**
Depends on the previous query fragment	**SELECT LASTTRADE, TICKER FROM YAHOO WHERE TICKER = [<unbound kind>]**

Figure 7: An example of query dependencies that the IBM federated engine knows how to handle.

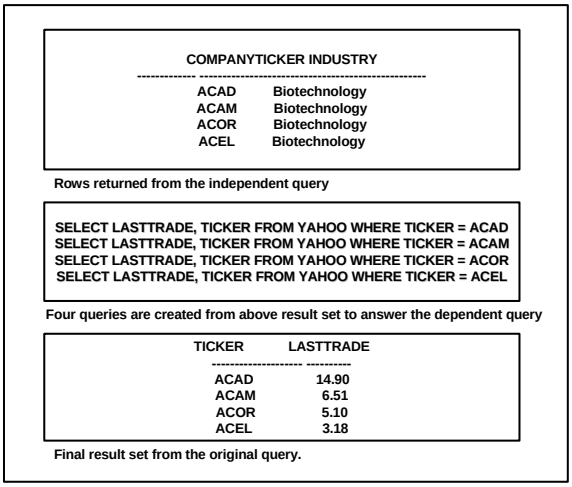

Figure 8: Procedure to process a dependent query.

4.5 Dependencies determined by capability records

Although the IBM federated engine leverages the query semantics to process dependent CSQs, it is still limited because it does not use capability records to mitigate any unsatisfied requirement in a CSQ. For example, the IBM federated engine would not be able to answer the query shown in Figure 9, which asks for all the exchange rates

between the USD and all other currencies on 01/05/94. To process this query, the query planning engine needs to invoke the **currency_map2** relation to retrieve all the currencies in the world (currency_map2.char3_currency), and then pass them to the **olsen** relation to obtain the exchange rates. In order to answer this query, the IBM federated engine decomposes the "from" clause of the query into two CSQs (**currency_map2** and **olsen).** Since none of the CSQs has unbound parameters, the IBM federated engine simply assumes they can be executed independently by using the native data source. However, the **olsen** CSQ (Figure 10) can not be executed by itself. The capability record for the **olsen** relation (Figure 6) indicates that the **exchange** attribute, whose required binding is missing in the CSQ, must have a binding value. In this case, the **olsen** CSQ can not be processed, and therefore, the IBM federated engine fails to answer the query.

```
select olsen.rate,
        from
        (select char3_currency,
         from   currency_map
         where  char3_currency <> 'USD') currency_map2,
        (select exchanged, 'USD', rate, '01/05/94'
         from   olsen
         where  expressed='USD'
         and    date='01/05/94') olsen,
where
and    currency_map2.char3_currency = olsen.exchanged
and    currency_map2.char3_currency <> 'USD'
```

Figure 9: an example of a query that relies on capability records for processing. Two CSQs, listed in bold are created: currency_map2, and olsen. The olsen relation lacks the binding value for the exchanged attribute, but this value can be found using one of the join conditions.

However, by using capability records, it is possible to process the **olsen** CSQ by finding the missing information from other parts of the query. The join condition "**currency_map2.char3_currency = olsen.exchanged**" can be used to supply the binding for the **exchange** attribute. Using this join condition, the system will rewrite the CSQ in Figure 10 into a CSQ in Figure 11, by providing the values for attribute **exchanged** from the **currency_map2** relation. With this added condition, the **olsen** CSQ satisfies the capability restriction and thus can be processed by the native source. Although this modified **olsen** CSQ depends on the result from the **currency_map2** CSQ, this dependency can be easily resolved by the IBM federated engine using query semantics (see Figure 12)

```
(select exchanged, 'USD', rate, '01/05/94'
 from    olsen
 where expressed='USD'
 and     date='01/05/94') olsen
```

Figure 10: The binding value for the exchanged Attribute is missing.

```
(select exchanged, 'USD', rate, '01/05/94'
 from   olsen
 where  expressed='USD'
 and    date='01/05/94'
 and exchanged in (select char3_currency
from currency_map2 where
char3_currency<>'USD')) olsen
```

Figure 11: The olsen CSQ with added Binding from Join Relation with currency_map2

Independent CSQ	select char3_currency from currency_map2 where char3_currency<>'USD'
Depends on the previous CSQ above	select exchanged, 'USD', rate, '01/05/94' from olsen where expressed='USD' and date='01/05/94' and exchanged = **[<unbound kind>]**

Figure 12. QEP with the modified olsen CSQ.

The basic algorithm, which uses capability records to process CSQs, is presented in Fig. 13. The algorithm is based on finding independently executable CSQs in the query and processing them before any dependent CSQs. In most cases, the CSQs that cannot be executed independently lack at least one binding restriction. Once such a CSQ is detected, the algorithm determines if the CSQ can still be executed by searching for the missing binding from other CSQs. If the algorithm finds the missing binding, it is incorporated into the CSQ so that it can be processed by the native source. In this example, the algorithm discovers that the **olsen** CSQ can be executed by binding its **exchanged** attribute to the **char3_currency** attribute from the **currency_map2** CSQ. The algorithm then adds the binding to the **olsen** CSQ to satisfy its capability requirement.

```
Input: Single Query q
Output: Query Execution Plan (QEP)

QEP Generation Algorithm:
1.  initialize set S to an empty set
2.  for all CSQs c in S
3.      if c is independently executable
4.      add c to set S
5.      add entry 0: c to QEP
6.  repeat until no more CSQs are added to S
7.      for all CSQs c outside of S
8.          if CSQ c can be executed using bindings from CSQs in S
9.              add an entry for c to QEP including all join bindings of c
10.             add CSQ c to set S
11.  if S does not contain all CSQs in a query
12.      throw exception "query cannot be executed"
13. return QEP
```

Figure 13. QEP Generation Algorithm Supporting Binding Query Restrictions

There are two non-trivial steps in this algorithm: a) determining if a CSQ can be independently executed (step 3), and b) deciding whether a CSQ can be processed using join bindings from a set of executed CSQs (step 8). The details of these two steps are illustrated in the following sections.

4.5.1 Determining independently executable CSQs

Figure 14 shows the algorithm for determining whether a CSQ is independently executable. The algorithm uses the capability records to detect any missing binding in the CSQ, and if they exist, the algorithm determines if these binding conditions can still be satisfied.

The first relation in Figure 15 illustrates the simplest case, in which any CSQ querying the relation can be executed as-is. The capability record for **currrency_map** has only one binding specifier "[?, ?]". Since the "?" attribute specifier is not binding, any CSQ querying this relation can be processed. In this case, the algorithm does not find any unsatisfied binding, and correctly concludes that the **currency_map2** CSQ is independently executable.

```
Input: CSQ C
Output: true if CSQ C is independently executable
        false otherwise

Independently Executable CSQ:
1. for each binding specifiers B in C's underlying relation R
2.    for each attribute specifiers A in B
3.       if A is of type bound and there is no binding in CSQ C for corresponding attribute
4.          continue 1
5.       else
6.          continue 2
7.    return true
8. return false
```

Figure 14. Algorithm for determining whether a CSQ is independently executable.

Figure 15b presents a case where the CSQ cannot be executed based on its capability record. In this example, the capability record in **olsen** has just one set of binding specifier, [b(1),b(1),f,b(1)], which states that the attributes **exchanged**, **expressed** and **date** need exactly one binding value. However, the CSQ has no binding value for the attribute **exchanged**. The algorithm, therefore, detects the missing binding in step 3 and tries to satisfy the restriction from using other sets of binding specifiers. Since [b(1),b(1),f,b(1)] is the only set of binding specifiers for **olsen**, the algorithm correctly determines that **olsen** CSQ is not independently executable.

```
relation( oracle,
          'currency_map',
          [['CHAR3_CURRENCY',string],
          ['CHAR2_CURRENCY',string]],
          cap([[?,?]], []]).
```

(a) Any CSQ querying against this relation can be executed independently.

```
select    exchanged, 'USD', rate, '01/05/94'
from olsen
where     expressed='USD' and date='01/05/94'
```

```
relation(cameleon,
          olsen,
          [['Exchanged',string],
          ['Expressed',string],
          ['Rate',number],
          ['Date',string]],
          cap([[b(1),b(1),f,b(1)]]
```

(b) The CSQ can not be executed independently with the defined relation.

Figure 15. Examples of a CSQ that can independently executed vs. a CSQ that depends on information from other CSQs.

4.5.2 Determining whether a CSQ is executable given a set of executed CSQs

The algorithm for determining whether a CSQ is executable, given a set of CSQs that have already been executed, is depicted in Fig. 16. Now consider the earlier example of **olsen**. Although the **olsen** CSQ cannot be executed independently, it can still be processed by finding the missing binding through the use of join conditions in the query. This algorithm detects this class of CSQs that are missing bindings but can still be executed using information made available through executing other parts of the query. For the specific example of **olsen**, upon finding the attribute **exchanged** to be unbound, the algorithm discovers a joint binding, **currency_map2.char3_currency=olsen.exchanged**, that can provide the missing values to the attribute **exchanged**. After modifying the **olsen** CSQ with the new join binding, the **olsen** CSQ can be executed.

```
Input:    set of executed CSQs S, new CSQ n
Output:  if n cannot be executed given join bindings from CSQs in S
              returns null
            else
              returns list of join bindings for CSQ n

CSQ Executable:
1.   for each binding specifiers B of CSQ n
2.        initialize list of join bindings to an empty list jbl
3.        for each attribute specifier A of B
4.             if A is of type bound and CSQ n does not contain binding for attribute matching as
5.                  if there is a join binding jb from n's attribute matching A to one of CSQs in S
6.                  add jb to jbl
7.                  continue 3
8.             else
9.                  continue 1
10.       return jbl
11. return null
```

Figure 16. Algorithm for determining whether a CSQ is executable given a set of executed CSQs

4.6 Handling Query Operator Restrictions

In some cases, the query planning engine can also process operators that are forbidden in the capability records. This processing usually applies for queries that have extra filtering clauses in addition to the binding requirements. To process such a query, the wrapper will simply ignore any expression containing the forbidden operator and pass the legal parts of the query to the native data source as long as the query still satisfies the capability restrictions. After getting the result back from the wrapper, the IBM federated engine will then compensate for the native source by processing the forbidden operator itself.

In the example shown in Figure 17a, the **COMPANYTABLE** relation can not handle the "<" operator on the attribute **COMPANYTICKER**. To process the query in Figure 17b, the wrapper would need to only answer a query without the predicate containing the forbidden operator as shown in Figure 17c, and return a super set of the desired result. The IBM federated engine will then compensates the unprocessed predicate by pruning all the **COMPANYTICKER**s that are less than "AD" lexicographically.

```
relation (cameleon
          COMPANYTABLE
          [['COMPANYTICKER', string],
           ['INDUSTRY', string]]
          Cap([[b(1), f]],
          ['<','>','<>','<=','>=']))
```

(a) The COMPANYTABLE relation with its capability records.

```
SELECT COMPANYTICKER, INDUSTRY
FROM COMPANYTABLE
WHERE INDUSTRY = BIOTECHNOLOGY
AND  COMPANYTICKER < AD
```

(b) The original query with the forbidden operator

```
SELECT COMPANYTICKER, INDUSTRY
FROM COMPANYTABLE
WHERE INDUSTRY = BIOTECHNOLOGY
```

(c) The query accepted by the native

Figure 17. Processing queries with forbidden operators. The query planner will simply send the parts of the query that the native source can answer and then process the forbidden operator locally after getting results back from the native source.

```
select price
from Quicken
where ticker in ('GE', 'IBM', 'MSFT');
```

(a) A query that violates the key-at-a-time restriction.

```
relation(cameleon,
        'Quicken',
        [['TICKER', string],
         ['PRICE',   number]],
        cap([[b(1), f]],
```

(b) The Yahoo relation with its capability records.

```
(select price from Quicken
 where ticker= 'GE')
union
(select price
 from yahoo
 where ticker= 'IBM')
union
(select price
 from yahoo
 where ticker= 'MSFT')
```

(c) Modified query to handle relaxed

Figure 18. Examples of relaxing key-at-a-time restrictions. When the number of values for a binding parameter in a query exceeds what is allowed, the engine rewrites the original query into a union of several subqueries, where each subquery processes only what is allowed by the native source.

4.7 Handling Key-at-a-Time Query Restriction

For some queries, the key-at-a-time restriction can also be relaxed, especially when a query can be executed by performing unions of results from subqueries. An example is illustrated in Figure 18a. Based on the capability record in Figure 18b, the Cameleon wrapper cannot handle this query directly because the **Quicken** relation has a one-key-at-a-time query restriction for the attribute **ticker.** In addition, it cannot handle queries with the 'in' operator[6]. In order to answer the query, the engine needs to change the query into a union of three one-key-at-a-time queries, and perform the union operation locally as shown in Figure 18c. Figure 19 shows the algorithm that rewrites the query to relax the key-at-a-time query restrictions. The algorithm separates the query into a few sub-queries that maximize the key-at-a-time restriction. Finally, the algorithm returns the result by performing the union operator on the results of all the sub-queries.

KAT (key-at-a-time) rewriting algorithm:
1. for every key-at-a-time-restricted attribute in CSQ
2. if attribute is bound to more values than allowed by query restriction then
3. rewrite the CSQ as union of 2 KAT queries:
4. first KAT query with the attribute bound to the maximum allowed number of values
5. second KAT query with the attribute bound to the remaining values
6. call KAT algorithm recursively on both KAT queries

Figure 19: Algorithm to relax key-at-a-time query restrictions.

[6] Note that queries containing 'in' operator are common because when CSQ c1 is joined remotely with CSQ c2, executioner feeds results of CSQ c1 into CSQ c2 through 'in' operator (e.g. ticker in (select ticker from Ticker_Lookup)).

4.8 Limitations

In the previous sections, we have shown how to use capability records to process queries that were otherwise unanswerable by both the IBM data federation engine and the Cameleon web wrapper. However, our algorithms are by no mean the solution to answer all queries against any website. For example, our system could not answer queries

```
select yahoo.ticker, yahoo.lasttrade
from ("select companyticker from NYSE) AS x, yahoo where yahoo.ticker = x.companyticker
```

such as "SELECT * FROM YAHOO", because it is impossible for the query to know all the company names stored in the **Yahoo! Finance** website. However, if we knew that **Yahoo! Finance** contains only the company tickers for NYSE, then we could extract all the stock prices stored in **Yahoo! Finance** by using the following query. In this case, we invoke another source **NYSE** to retrieve all the company names in the source, and use them to retrieve information from **Yahoo! Finance**.

Obviously, we also can not answer queries that violate the capability restriction. For example, as illustrated by the following query, it is impossible to retrieve all the country pairs whose currency translation rate is less than 2.0 on January 5, 1994. Since the **olsen** relation requires that the **exchanged** and **expressed** attributes to have exactly one value each, the system simply could not provide an answer from information given by the query.

```
select exchanged, expressed, rate from olsen where rate <2.0 and date = '01/05/94'
```

5. Future work

Although our generic web wrapping solution can already incorporate many existing types of web-sources, a number of improvements can still be made, especially in areas of handling query restrictions. Our current capability records can handle the most common cases of query restrictions, such as binding restrictions, operator restrictions, and key-at-a-time restrictions. However, it would be important to handle other query restrictions as well, such as retrieving a batch of tuples at a time.

Some web-sources impose a restriction on the number of tuples that can be returned when querying a relation (where tuples are a binding pair of values). Search engines typically exhibit this behavior as they only return a pre-set number of results at the time for each query. For example, querying www.google.com for "context mediation" results in 280 records but only 10 records are returned per page. To retrieve all 280 records, the wrapper needs to query www.google.com 28 times fetching 10 tuples at a time. To conform to this restriction, the capability records need to carry the information on how many tuples can be returned from the source. Wrappers would also need to take additional input parameters such as starting record number. Extending SQL with the "startat" keyword allows the query to specify the record number from which to start counting the batch of tuples. This process makes it easy to retrieve only the results after the 31st tuple, by issuing the following query.

```
select search_results
from google
where search_keyword='context mediation'
and startat= 31;
```

6. Conclusion

We present a three-layered architecture for querying web-sources. In contrast to the traditional two-layered architecture where capability restrictions are buried inside the wrapper code, we demonstrate the versatility of our system by converting capability restrictions into simple declarative expressions. Using this three-layered architecture, we show that our system can answer complex queries involving multiple web-sources. It uses IBM federated engine to decompose a complex query into component subqueries (CSQs) and determine the CSQ

execution order using both query semantics and capability records. The system relies on the IBM federated engine for query semantic analysis. However, a CSQ with no semantic dependencies could still not be executed because it may not satisfy restrictions imposed by the native source. Through understanding the capability records, our system can compensate the CSQ by finding the missing requirement from other parts of the query. Moreover, the system can also relax the key-at-time restriction by rewriting the original query into several pieces that conforms to the capability record.

Not only can our system deal with complex queries using the web, the main advantage of this three-layered architecture is that the wrapper code needs not be modified each time a website with a new restriction joins the data federation. Instead, creating a capability record for the data source is most often enough. Using this approach, we preserve the generic solutions offered by commercial data federation systems, while easing the burden on wrapper developers from dealing with complex capability issues. Thus, the inclusion of web-sources through this mechanism can be accelerated in a way that doesn't require a change in existing data federation technology.

References

Alatovic, T. (2001) Capabilities Aware, Planner, Optimizer, Executioner for Context Interchange Project. Thesis (S.M.) M.I.T, Dept. of EE & CS.

Chan, C. (2000) OLE DB for the Context Interchange System. Thesis (S.M.) M.I.T, Dept. of EE & CS.

S. S. Chawathe, H. **Garcia**-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom (1994). The TSIMMIS project: Integration of heterogeneous information sources. IPSJ, pages 7–18.

Chuang, S.W. (2004) A Taxonomy and Analysis of Web Wrapping Technologies. Thesis (S.M.) M.I.T, Technology and Policy Program.

Firat, A., Siegal, M.(2000) "The Caméléon Web Wrapper Engine", Proceedings of the VLDB2000 Workshop on Technologies for E-Services, [SWP #4128, CISL #00-03].

Firat, A. (2003) Information Integration Using Contextual Knowledge and Ontology Merging Thesis (Ph.D.) M.I.T.

Fynn, K. (1997) A Planner/Optimizer/Executioner for Context Mediated Queries, MIT Masters Thesis, Electrical Engineering and Computer Science.

IBM, Developing Wrappers—Overview, DB2 Universal Database, IBM Corporation. March, 2006 < http://publib.boulder.ibm.com/infocenter/db2luw/v8//index.jsp>

Knoblock, C., Lerman, K., Minton, S., Muslea, I. (2000) IEEE Data Engineering Bulletin, 23(4):33--41, December 2000.

Laender, A., Ribeiro-Neto, B., Silva, A. and Teixeira, J. (2002) A Brief Survey of Web Data Extraction Tools, SIGMOD Record, Volume 31, Number 2.

Li, C. Chang, E. (2000) Query planning with limited source capabilities, ICDE 2000

A. Y. Levy, A. Rajaraman, and J. J. Ordille. (1996) Querying heterogeneous information sources using source descriptions. Very Large Data Bases (VLDB) Conference, pages 251–262,

Papakonstantinou, Y., Gupta A., Haas L. (1996) Capabilities-based query rewriting in mediator systems. To appear in Fourth International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida.

M. T. Roth and P. M. Schwarz (1997) Don't scrap it, wrap it! A wrapper architecture for legacy data sources. Very Large Data Bases (VLDB) Conference, pages 266–275.

A. Tomasic, L. Raschid, and P. Valduriez (1998) Scaling access to heterogeneous data sources with DISCO. IEEE Transactions on Knowledge and Data Engineering, 10(5):808–823.