Warehousing and Incremental Evaluation for Web Site Management

Mary Fernandez*
mff@research.att.com

Daniela Florescu
Daniela.Florescu@inria.fr

Alon Levy
alon@cs.washington.edu

Dan Suciu
suciu@research.att.com

February 21, 1998

Abstract

Recently, several systems have been proposed for building Web sites from declarative specifications. The salient feature of these systems is that they model both the structure and the content of a Web site as a view over some existing raw data. In this "Web site as view" paradigm, a critical issue is when to compute the site. One approach is to materialize the site completely before the users browse it: the disadvantages of this approach are high maintenance costs and stale data. Moreover, this approach is not applicable if the Web site has forms, because then the queries defining the Web site depends on user input. The other extreme approach is to precompute only the root(s) of the Web site, and then compute dynamically (at "click time") the queries that retrieves the information required to display the next page.

In this paper we address the problem of choosing the optimal configuration for run-time management of a declaratively specified Web site, trading off the benefits of precomputation and incremental evaluation. First, we identify several novel optimizations that need to be considered in addition to optimizations obtained by precomputing views. These optimizations include simplifying queries based on contexts that can be derived from the browsing patterns, and lookahead computation. Second, we describe an algorithm for cost based search of a space of configurations which include the above optimizations. Finally, given a configuration we show how to reformulate our Web-site definition to exploit the views and data being cached.

1 Introduction

The World-Wide Web (WWW) is a prime vehicle for disseminating information. Consequently, sites are growing in size, have increasingly complex structure, and often serve information derived from multiple data sources. Managing the *content* and *structure* of such Web sites is a novel data management problem. Consider the tasks that a Web-site builder must address: (1) choosing and accessing the data that will be displayed at the site, (2) designing the site's structure, i.e., specifying the data contained within each page and the links between pages, and (3) designing the graphical presentation of pages. In existing Web-site management tools, these tasks are usually interdependent. Without a site-creation tool, a site builder writes HTML files by hand or writes programs to produce them and must focus simultaneously on a page's content, its relationship to other pages, and its graphical presentation. As a result, important tasks, such as automatically updating a site, restructuring a site, or enforcing integrity constraints on a site's structure, are tedious

*Contact author

to perform. Our desire to support these tasks naturally leads us to view Web-site management as a data-management problem.

An important observation is that by specifying a Web site's content and structure *declaratively*, instead of in the ad hoc manner of current practice, solutions to the above problems become possible. Several architectures have been proposed for building Web-site management systems from declarative specifications [8, 2, 5, **?**]. A common feature of these proposals is that they model the graph that represents a Web site as a *view* of some existing raw data. The proposals' data models and languages for specifying views differ.

In this "Web site as view" paradigm, a critical issue is when to compute the view. One approach is to materialize the view completely, i.e., compute the complete site before users browse it. This approach, however, has problems similar to those of data warehousing, such as highly prohibitive maintenance costs and presentation of stale data. Unlike in data warehousing, completely materializing a Web site is often impossible, because Web pages that depend on user input, e.g., from forms, cannot be materialized statically, but must be created dynamically.

Another approach is to precompute the root(s) of a Web site, then compute at *click time* the query that obtains the information required to display the next page. In the proposed formalisms, it is possible to analyze the site specification and, for each page in the Web site, compute a set of click-time queries that obtain the data corresponding to that page. Consider a naive incremental evaluation algorithm that repeatedly computes these click-time queries for each page accessed. Such an algorithm would be adequate for sites whose raw data sources are small and whose structure is relatively simple. For sites with large underlying data sources, however, the algorithm could result in click-time queries that are prohibitively expensive. In particular, a naive algorithm often results in repeated evaluation of the same or closely related queries and does not exploit browsing patterns to help optimize click-time queries.

In this paper, we describe algorithms that are the basis for a Web-site management tool that evaluates various *configurations* for incrementally evaluating a site-definition query. We discuss several types of optimizations: (1) precomputing a set of views, (2) memoizing the results of certain queries for later use, (3) performing lookahead computation, i.e., computing more data than is immediately needed, but which may be useful in subsequent queries, and (4) simplifying click-time queries using preconditions that can be inferred from possible browsing paths. We formalize the content of a warehouse for a Web site as a *configuration,* which includes a set of views to materialize, memoizations, and lookahead computations. Specifically, this paper makes the following contributions:

- We define the space of possible configurations based on the novel optimizations listed above. Hence, the design of a warehouse supporting a Web site can be formulated as a search problem.

- We present an algorithm for searching the space of configurations and for assisting the Web site administrator to choose among them. The algorithm is designed to search only a restricted portion of the search space, but is still guaranteed to find the optimal configuration under certain assumptions.

- We describe an algorithm that, given a configuration, reformulates the site-definition query such that the click-time queries maximally exploit preconditions and the cached data in the configuration. The algorithm is a solution to the problem of rewriting queries using views, that is tailored to the needs of our application.

- As a result of the above algorithms, we obtain a run-time algorithm for incrementally computing a Web site that balances the tradeoffs of static and dynamic computation.

The declarative specification of a site's content and structure make the optimizations we describe possible. Consider current practice, in which a specialized program, e.g., a CGI-bin script, computes each page in an incrementally evaluated site. In that model, it is practically impossible to detect automatically repeated computations. In fact, a Web site builder usually discovers such optimizations and manually hardwires them into the code. Using our techniques, a Web site builder would *never* have to write a CGI-bin script, because our algorithms would derive all these programs automatically. Moreover, our techniques allow the site builder to consider many possible configurations, which combine static and dynamic computation of pages in a site; such exploration is infeasible in current practice.

We describe our algorithms in the context of the STRUDEL Web-site management system [8], but note that our solutions could be adopted by any system that declaratively represents site structure. We argue that this is the direction Web-site management systems will follow in the future. Database vendors already made a step in this direction by providing tools that serve data from databases in Web sites, often by embedding queries in HTML pages. We expect these tools will become more sophisticated and eventually support definition of complex sites.

2 Web Site Management Architecture

We begin by describing the general architecture of declarative Web-site management systems. Our work is done in the context of the STRUDEL system, but we note that the problems we address and the solutions we propose are not specific to STRUDEL. To illustrate the generality, we adopt a formalism that captures that commonalities of several other recent proposals [2, 5].

2.1 Overview

First, we describe the steps of building a Web site using STRUDEL. Building a complex Web site often involves displaying data that may exist in several external repositories, such as databases or structured files. We refer to this data as the *raw* data. Using STRUDEL, a site builder declaratively describes the *content* and the *structure* of the site serving the data. The declarative description specifies (1) the pages in the site and the links between them, and (2) what data is displayed in each page.

The Web site's content and structure are declaratively specified in the STRUQL language. Applying a STRUQL query to an instance of the raw data produces a *site graph*. The site graph contains a node for every Web page in the site and the links between the Web pages (modeled as edges). In addition, each node modeling a Web page has a set of data values. We can consider the site graph as a *view* over the raw data. A key characteristic that distinguishes STRUQL from other view-definition languages is that a STRUQL query produces a labeled directed graph, not a relation.

We note that a site graph *does not* specify the graphical presentation of pages, therefore the last step is to define the graphical presentation of pages and generate the browsable Web site. The graphical presentation is specified by a set of *HTML templates*, which are HTML files with variables. Every node in a site graph has a corresponding HTML template, which may be unique to the node, but commonly is shared by a collection of related nodes. Given a node in a site graph, the corresponding HTML template is instantiated by replacing variables in the template with the appropriate values from the node thereby producing an HTML file. The browsable Web site is constructed by instantiating the appropriate HTML template for each node in the site graph.

Before continuing, we mention two important issues in the design of a Web-site management system that we do not discuss in this paper. The first is data integration. Since the raw data served at the site may come from several external data sources, the system requires a mediator component. Such a component provides the Web-site designer a uniform view and interface to all the underlying data sources. Any of the solutions to the data integration problem would be applicable in our context [4, 13, 1, 9, 10, 17]. The second issue is the management of semistructured data. In [8], we argue that the semistructured data model (i.e., a labeled directed graph) is appropriate for Web-site management, because data often comes from multiple sources (some of which are not databases) and is often irregular and because a Web site's schema evolves rapidly. Another advantage of semistructured data is that it models uniformly the input data and the resulting site graph. To simplify our discussion in this paper, we assume that the raw data is stored in one relational database system.

2.2 Site Specification Language

In this section, we define the language for declaratively defining a Web site. We illustrate the language by an example, showing how to build a fragment of the internal AT&T Research Web site. Although we have built a version of that site using STRUDEL, we restrict our example of a fragment of that site.

Figure 1 gives the schema of the raw data for the example site, and a small fragment of the corresponding database. There are two types of organizations (denoted by the attribute Orgtype): labs and departments, which are suborganizations of labs. The SuperorgId of a department denotes the id of the lab to which it belongs. Labs have a NULL value for the attribute SuperOrgId. Other than organizations and their members, the schema also models the publications of its members.

A fragment of the Web site we are building is shown in Figure 2. From the root, we can browse lab pages. Every lab page has a link to each of its departments and a link to a person who is a designated "contact person". Every department has a link to each of its members' homepages, and members' pages have a link back to their departments. Each member's homepage points to its publications, which are grouped by year.

The site graph shown in Figure 2 is the result of applying the specification shown in Figure 3 to the database shown in Figure 1. The query is composed of several blocks, which may be nested. Each block includes a *data extraction* component and a *graph creation* component. The extraction component produces a relation, which provide bindings for a set of variables. The creation component uses the bindings to create the nodes and links in the resulting graph.

The extraction component can be expressed in any query language that produces a relation. In the STRUQL language, this component is adapted to the semistructured data model (i.e, includes querying schema and stating regular path expressions over the data). Here we use the notation of conjunctive queries. Note that an underscore represents an existential variable.

The creation part of the query specifies the links and nodes in the graph. Each link clause is either of the form $F(\bar{X}) \rightarrow L \rightarrow G(\bar{Y})$ or $F(\bar{X}) \rightarrow L \rightarrow V$, where F and G are *Skolem functions*, V is a variable, $\bar{X}$ and $\bar{Y}$ are tuples of variables defined in the extraction part of the same block, and L is a string or a variable.[1] Intuitively, the first kind of link specifies links between pages in

[1] To keep the presentation simple, we restricted L to be only a string in all examples in this paper.

```
Orgs(OrgId, Name, SuperorgId, Orgtype, SecrId)
    (o1, ``Information Systems'', null, ``Lab'', p2)
    (o2, ``Artificial Intelligence'', o1, ``Dept'', null)
    (o3, ``Databases'',o1, ``Dept'', null)

Person(PersonId, Name,  OrgId, Email)
    (p1, ``Brian Smith'', o2, ``brian@ai.com'')
    (p2, ``Anita Young'', o3, ``anita@db.com'')

Paper(PaperId, Title, Year, HowPublished)
   (a1, ``Programs With Common Sense'', 1996, ``AAAI'')
   (a2, ``Neural Net Databases'', 1997, ``ACM SIGMOD'')

PaperPerson(PaperId, PersonId)
   (a1, p1)
   (a2, p1)
   (a2, p2)
```

Figure 1: A fragment of the organization database

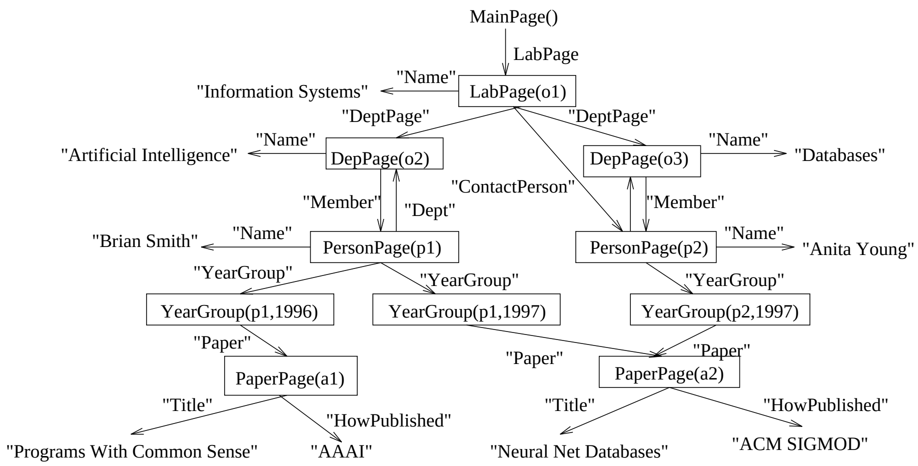

Figure 2: The site graph

the Web site, while the second kind specifies the data associated with each page.

For every Skolem term of the form $F(\bar{X})$ that appears in the query and for every binding $\bar{a}$ for the variables $\bar{X}$, the result graph includes a *single* node identified by $F(\bar{a})$. The meaning of a link clause of the form $F(\bar{X}) \rightarrow L \rightarrow G(\bar{Y})$ is that for every binding $\bar{a}$ for $\bar{X}$ and binding $\bar{b}$ for $\bar{Y}$, the graph has a link labeled L from the node $F(\bar{a})$ to the node $G(\bar{b})$. In the case $F(\bar{X}) \rightarrow L \rightarrow V$, there is an edge labeled L from $F(\bar{a})$ to v, where v ranges over all bindings for V.

In the query in Figure 3, the first block creates a root page for the site and links it to all the lab pages; that is, there are as many `"Labpage"` links as there are bindings for the variable `L`. It also links the lab pages to their names and contact persons. Note that the name is a value, while the contact person is another page. The second block connects each lab page to all its department pages, etc.

This example highlights the advantages of declarative specification. In particular, the tasks of inspecting, modifying, and updating the site's content or structure are restricted to a declarative

```
// Q1: Create Main Page, link it to the Labs Pages
{WHERE Orgs(L, N, _, "Lab", P) /* Query Q1(L,N,P) */
 LINK  MainPage() -> "Labpage" -> LabPage(L),
       LabPage(L) -> "Name" -> N,
       LabPage(L) -> "ContactPerson" -> PersonPage(P)
}
// Q2: Link Labs pages to their Depts Pages
{WHERE Orgs(D, M, L, "Dept", _) /* Query Q2(D,M,L) */
 LINK  DeptPage(D) -> "Name" -> M,
       LabPage(L) -> "DeptPage" -> DeptPage(D)
}
// Q3: Associate People with Depts
{WHERE Orgs(D, _, _, "Dept", _), Person(P, S, D, _) /* Query Q3(D,P,S) */
 LINK  DeptPage(D) -> "Member" -> PersonPage(P),
       PersonPage(P) -> "Dept" -> DeptPage(D),
       PersonPage(P) -> "Name" -> S
}
// Q4: Associate Papers with People
{WHERE Person(P, _, _, _), PaperPerson(P, Q), Paper(Q, T, Y, H) /* Query Q4(P,Q,T,Y,H) */
 LINK PersonPage(P) -> "YearGroup" -> YearGroup(P,Y),
      YearGroup(P,Y) -> "Paper" -> PaperPage(Q),
      PaperPage(Q) -> "Title" -> T
      PaperPage(Q) -> "HowPublished" -> H
}
```

Figure 3: Fragment of the definition of a corporate research Web site

specification, which clearly reflects the site's content and structure. In the current AT&T Research site, as in many other complex Web sites, groups of link clauses are implemented by a special program, e.g., a CGI bin script. In this setting, maintenance and updating tasks require rewriting many individual programs, which is tedious and error prone.

2.3 Site Schemas

It is often convenient to analyze the site specification using the *site schema*. Given a site specification Q, the site schema associated with Q is an equivalent reformulation of Q in terms of a graph, which provides an abstraction of the structure of the Web site generated by Q.

The site schema for a site specification Q is formally defined to be the labeled graph G_Q. G_Q has one node N_F for every Skolem function symbol F in the query. The graph has one edge $N_F \rightarrow N_G$ for every link expression $F(\bar{X}) \rightarrow L \rightarrow G(\bar{Y})$ in Q, and one edge emanating from N_F for every link expression $F(\bar{X}) \rightarrow L \rightarrow V$ (with V a variable). Each edge in G_Q that corresponds to a link expression $F(\bar{X}) \rightarrow L \rightarrow G(\bar{Y})$ is labeled with $(Q, L, \bar{X}, \bar{Y})$, where Q is the query in the where-clause associated with that link expression.

Note that no information is lost by the site schema, i.e., we can recover the site specification from the site schema. For many practical queries, including the example in Fig. 3, it is possible to describe schemas in a more intuitive formalism. Namely, if all occurrences of F in the query use the same variables $\bar{X}$, then we represent a node N_F by $F(\bar{X})$. Moreover, we simplify the edge labels from $(Q, L, \bar{X}, \bar{Y})$ to (Q, L), since the variables $\bar{X}$ and $\bar{Y}$ can be read from the source and destination node. In the rest of the paper we ignore the data links emanating from nodes in the site graph because their treatment is similar to that of the other edges. The site schema that

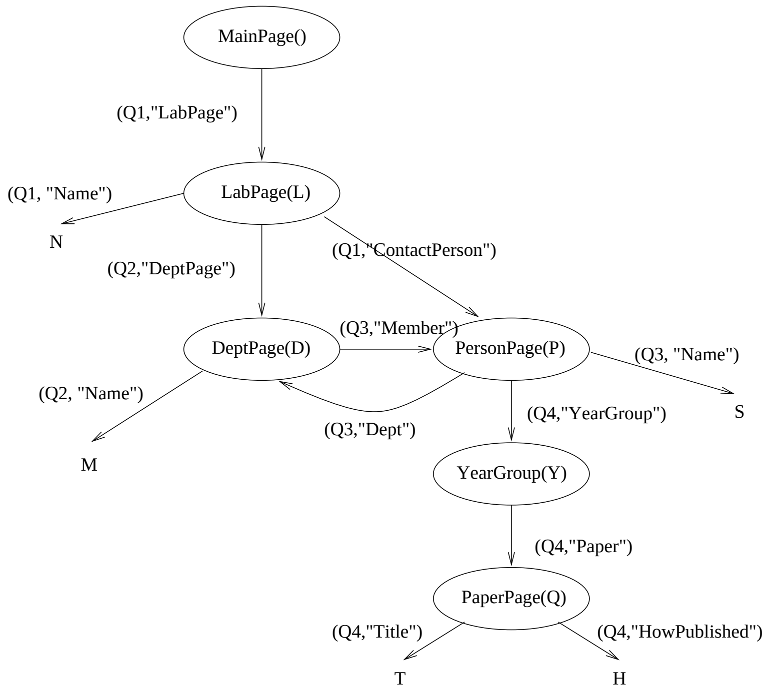

Figure 4: Site Schema for the Example 3

corresponds to the site specification in Figure 3 is shown in Figure 4.

3 The Problem of Incremental Evaluation

One of the most critical issues that arises in Web-site management is the question of when the site graph is computed. One option is to completely materialize the site graph, i.e., compute it before the users start browsing. However, this solution has obvious disadvantages. When changes to the raw data occur, they need to be propagated to the materialized site graph, and this incurs high maintenance costs. To avoid these maintenance costs, one solution is to sacrifice the freshness of the site's data, which would be unacceptable for certain sites, e.g., consider the CNN Web site. The tradeoffs involved are very similar to those of maintaining data warehouses (e.g., [**?**, **?**]). Unlike in data warehousing, it may not be possible to completely materialize a Web site, because Web pages are created dynamically from user input, e.g. via forms.

In practice, achieving acceptable performance requires a combination of precomputation and dynamic computation of data. In this section, we outline our approach that considers the spectrum of the different degrees of materialization. We begin in Section 3.1 by describing an extreme approach in which all the data is computed at run-time and no caching is performed. We then explain the components needed in order to pursue a more flexible approach. In Section 3.2, we describe the kinds of data, including precomputed views and results cached from previous computations, that are useful in a warehouse that supports a Web site, and formally define Web site warehouse configurations. In Section 3.4 we explain how a configuration can be used to reformulate a site specification, yielding an improved incremental evaluation algorithm. The two key steps of our algorithm, i.e., choosing an optimal configuration and reformulating the site specification are then described in details in Sections 4 and 5, respectively.

3.1 Naive Incremental Evaluation Algorithm

The pure dynamic approach to computing Web sites executes at *click time* the query that computes the information necessary to display the next page.[2] Recall that each page in the Web site is of the form $\mathtt{F}(\bar{\mathtt{a}})$, where F is a skolem function and $\bar{\mathtt{a}}$ is a particular binding. To display the page $\mathtt{F}(\bar{\mathtt{a}})$ the algorithm only needs to compute the outgoing edges of that node.

```
ALGORITHM NaiveIncrEval
INPUT: A node F(a) in the site graph to be displayed
OUTPUT: All successors of F(a)
METHOD:

/* compute the outgoing edges for F(a) */
  FOR EVERY outgoing edge F -> G labeled (Q, L, X, Y) in the site schema DO
    Compute the query I(L, Y) :- Q[a/X]
    FOR EVERY (l,b) in the answer of I DO
       OUTPUT edge F(a) -> l -> G(b)
```

Figure 5: Naive Incremental Evaluation

Figure 5 shows a *naive algorithm* for incremental evaluation. The input to the algorithm is a node $\mathtt{F}(\bar{\mathtt{a}})$ to be displayed. To create the HTML corresponding to the node $\mathtt{F}(\bar{\mathtt{a}})$, we need to compute the outgoing edges of this node: i.e., all the pairs of the form $(\mathtt{l}, \mathtt{G}(\bar{\mathtt{b}}))$, where $\mathtt{F}(\bar{\mathtt{a}}) \rightarrow \mathtt{l} \rightarrow \mathtt{G}(\bar{\mathtt{b}})$ is a link in the site graph. These pairs are obtained by computing queries of the form $\mathtt{I}(\mathtt{l}, \bar{\mathtt{b}})$, (called *click-time queries*) that are derived from the site schema as follows. Note that every edge of the form $\mathtt{F}(\bar{\mathtt{a}}) \rightarrow \mathtt{l} \rightarrow \mathtt{G}(\bar{\mathtt{b}})$ in the site graph is an instantiation of an edge between N_F and N_G in the site schema. Hence, for the current node $\mathtt{F}(\bar{\mathtt{a}})$, we consider all the outgoing edges from N_F. For each edge labeled $(Q, L, \bar{X}, \bar{Y})$ there will be a click-time query obtained from Q by substituting $\bar{X}$ with $\bar{a}$, and projecting the result on the attributes L and $\bar{Y}$. Henceforth, the tuple $\bar{\mathtt{a}}$ is called the *input* to the click-time queries applied in computing $\mathtt{F}(\bar{\mathtt{a}})$.

To illustrate, consider a node of the form `PersonPage(p)` in the site graph, where p is a constant. To compute its outgoing edges, we consider the three successors of the node `PersonPage(P)` in the site schema (Fig. 4); their corresponding queries are `Q3`, `Q3`, and `Q4` respectively, and their successors are given by the queries `I1, I2, I3`:

```
I1("Dept",D) :- Orgs(D, _, _, "Dept", _), Person(p, S, D, _)   /* Q3(D,p,S) */
I2("Name",N) :- Orgs(D, _, _, "Dept", _), Person(p, S, D, _)   /* Q3(D,p,S) */
I3("YearGroup",p,Y) :-  Person(p, _, _, _), PaperPerson(p, Q), Paper(Q, T, Y, H) /* Q4(p,Q,T,Y,H) */
```

Together, these three queries specify all successors of the node `PersonPage(P)`, e.g., for every tuple ("`YearGroup`", `p`, `y`) in `I3`, there is an edge `PersonPage(p)`$->$ "`YearGroup`"$->$ `YearGroup(p, y)`.

We emphasize that the *declarative* specification of a Web site allows us to provide a method for automatically deciding which query to compute at click time. In current practice, site designers, who build a Web site with a set of CGI bin scripts, must "compute" click-time queries manually and write programs that correspond to manually optimize queries.

[2]For the purpose of this paper we make the simplifying assumption that there is a one-to-one correspondence between nodes in the site graph and web pages.

3.2 Data Warehouse design for Web sites

In many cases, the naive algorithm may be of practical interest, for example, when the underlying databases are small, or the structure of the Web site is simple (e.g., every node corresponds to a simple selection query on the database). In other cases, however, the naive algorithm may result in click-time queries that are prohibitively expensive to compute. In practice, obtaining acceptable run-time performance requires maintaining additional data. Some of the data may be statically precomputed as in traditional data warehouses. In addition, we may store data that is obtained by computations executed during browsing. In this section, we first describe the kinds of data we may want to store in a data warehouse supporting a Web site, and then formally define the content of a warehouse as a *configuration*.

Precompute views: A natural optimization is to choose a set of materialized views over the raw data and reformulate the click-time queries to exploit these views. The problem of choosing views to materialize to support a set of predefined queries is known as the *data warehouse design* problem and has been addressed in several works (e.g., [?, ?, ?, ?]). As in data warehousing, computing views trades off the evaluation cost of click-time queries with space and view-maintenance cost. The key difference between our setting and data warehouse design is that our queries may depend on user inputs (in web sites with forms) and cannot be precomputed.

Memoization: One improvement over the naive algorithm is to memoize (cache) the results of the click-time queries. Once we have computed the query `I(L,`$\bar{\mathtt{Y}}$`)` associated with a node `F(`$\bar{\mathtt{a}}$`)`, we cache the query's results (i.e. store all tuples ($\bar{\mathtt{a}}$, `l`, $\bar{\mathtt{b}}$), for all (`l`, $\bar{\mathtt{b}}$) in the query's result). Later, if the same query needs to be evaluated on the same constants $\bar{\mathtt{a}}$, we can retrieve the results from the cache. To illustrate, suppose we have computed the `PersonPage(p)`, computing, among other things, all years of its publications, i.e. the links to pages of the form `YearGroup(p,y)` (query `I3` above). As a side effect, we can store the mapping from `p` to `y`. If the user reaches the same person page `PersonPage(p)`, possibly through a different path, we can reuse the result.

Memoization differs from browser caching in that it stores results on a semantic basis. A browser is only able to cache a static Web page (i.e., one that is not created on the fly). In contrast, our memoization is able to take advantage of partial changes, and invalidate only the parts of the cache for which the raw data has changed.

Context saving: An improvement to memoization is to cache not only the result of a click-time queries, but also the values of other variables that are projected out, and may be useful for later clicks. Consider, for example, query `I3`:

```
I3("YearGroup",p,Y) :-  Person(p, _, _, _), PaperPerson(p, Q), Paper(Q, T, Y, H) /* Q4(p,Q,T,Y,H) */
```

This particular click query only the values of the variables p, Y are needed, and therefore the variable `Q` would not be memoized. At a later stage, however, when requesting the display of the node `YearGroup(p,y)`, where `(p,y)` is one of the tuples in `I3`'s answer, the click-time query would be

```
I5("Paper",Q) :-  Person(p, _, _, _), PaperPerson(p, Q), Paper(Q, T,y, H) /* Q4(p,Q,T,y,H) */
```

Computing $I5$ would redo much of the work done by I3. With context saving we could project in $I3$ the values of *all* `p`, `Y` and `Q`, thereby transforming $I5$ into a simple lookup in the cache.

Node	Naive	PreC	Views	Cxt	Look
MainPage	438				
LabPage	906	866			
DeptPage	1250	846	450	383	1651
PersonPage	3106	1942	722	1018	122
YearGroup	1670	672	682	200	
PaperPage	1404	677	625	23	

Table 1: Results of preliminary experiments. Mean execution time in *ms* per object.

Lookahead computation: often it pays off not only to store additional context associated with a click-time query, but also to compute *in advance* information which is only necessary later. For example, recall the display of a department page, `DeptPage(d)`, requiring, among other things, the computation of all its members:

```
I4("Member",P) :-  Person(P, S, D, _)
```

At this point it may be beneficial to also compute the member's publications and publication years, that is, to compute `I4'("Member",P,Q,Y) :- Person(P, S, D, _)` $⟗$ `(PaperPerson(P, Q)` $\bowtie$ Paper(Q,_,Y,_))

Note that the symbol $⟗$ denotes an outerjoin. The result of this extra computation will be useful in following three types of edges: from the department pages to their members, from each person page to the page corresponding to their publications in a certain year, and from the year page to specific publication pages. It is important to note that the lookahead computation usually involves an outerjoin between the data needed for the current node and the data needed for subsequent nodes. In our example, we need to compute an outerjoin because the department page also displays persons without publications.

Example benefits: We ran several tests on the AT&T Research Web site in order to gauge the potential impact of our techniques. The experiments were run on an Intel 130MHz Pentium PC under Linux, using the current version of STRUDEL, which is written in Java and uses JDK 1.1.1. Table 1 reports the mean execution times in *ms* per object for each technique. **N**aive is the naive algorithm with no optimizations performed; **V**iews uses precomputed views to compute each query; **C**xt uses the contexts computed by previous queries and saves contexts for subsequent queries; **L**ook uses the lookahead values cached by previous queries and computes lookahead values for subsequent queries. **P**reC optimizes each query under known preconditions (an additional optimization technique that is used in the reformulation step). Both the **C**xt and **L**ook results include optimizing under known preconditions.

Not surprisingly, materialized views reduce execution time significantly, but suffer the disadvantages of precomputation. The context saving technique (**C**xt), however, is even more effective than precomputing views in three cases. For example, context saving reduced the average execution time for computing `DeptPage` from 1.3sec to 0.38sec (Table 1): a three-fold improvement. Both `YearGroup` and `PaperPage` benefit from the context saved during computation of `PersonPage`. For `YearGroup`, context saving yielded an eight-fold improvement and for `PaperPage`, a 60-fold improvement. Even for `PersonPage`, in which views were more effective than context saving, context saving reduced execution time from 3.1sec to 1.0sec: a three-fold improvement. Recall that

PersonPage is computed by the incremental queries `I1,I2,` and `I3`. An important observation is that context saving reduces the execution time of incremental queries to 1sec or less, which makes it feasible to evaluate and display them dynamically at "click time". In the lookahead column we can see that the extra computation in the department pages increased the time by 30%, but this speeded up significantly the computation of instances of PersonPage.

3.3 Configurations

The general problem is to inspect the site definition query and propose a set of views, memoizations and lookahead computations which minimize a global cost function. To formalize the search problem, we introduce the notion of a *configuration* which models the content of Web Site Warehouse.

In this paper we consider views and functions defined by algebraic expressions constructed from datalog atoms using the operators select, project, join and full outerjoin[3]. A *subquery* of a query of the form $A_1 \bowtie \ldots \bowtie A_n$, where the A_i's are atoms is a query of form $A_{i_1} \bowtie \ldots \bowtie A_{i_m}$, where $i_l < i_{l+1} \leq n$.

Definition 3.1: *A Web site warehouse configuration is a pair* $(\mathcal{V}, \mathcal{F})$*, where* $\mathcal{V}$ *is a set of views to materialize, each view being defined by an expression, and* $\mathcal{F}$ *is a set of triples* (V, In, N)*, where* V *is a parameterized expression, whose parameters are the subset* In *of its distinguished variables, and* N *is the set of nodes in the site-schema where the expression will be computed.*

The meaning of a configuration is the following. Each of the views in $\mathcal{V}$ is computed statically (i.e. before the users start browsing), like in a classical warehouse environment. A memoized function can be viewed as storing a subset of the tuples of a view and it is dynamically computed (i.e., while the users browse the site). If the triple (V, In, N) is in $\mathcal{F}$, then the system will maintain the mapping from some values of In to the tuples of the result of evaluating V with inputs In. New values are computed for (V, In, N) when the user browses nodes that are instances of elements of N.

For a simple illustration, the memoization function

$$(\pi_{\{\mathtt{D,P,Y,Q}\}}(\mathtt{Person(P,S,D,\_)} ⟗ (\mathtt{PaperPerson(P,Q)} \bowtie \mathtt{Paper(Q,\_,Y,\_)})), \{\mathtt{D}\}, \{\mathtt{N_{DeptPage}}\})$$

represents the lookahead in query $\mathtt{I4}'$ of Sec. 3.2. The node $\mathtt{N_{DeptPage}}$ is where the mapping is computed: once it is computed, it may later be used at different nodes (for example, at $\mathtt{N_{PersonPage}}$).

Both views and memoization mappings contain cached data. However, there are three significant differences between them, which change the way we treat them in our algorithm: (1) when are they computed, (2) whether they are complete, and (3) whether the updates to the raw data are automatically propagated. As for the first two issues: views are considered to be statically computed and complete (i.e., contain *all* the tuples defined by their defining expression), while memoized functions are dynamically computed, and by nature, partial. These differences affect the way the run-time algorithms makes use of views and mappings. Finally, another difference concerns the policy of maintaining freshness. Whereas views are assumed to be automatically maintained by propagating updates, (i.e., the warehouse guarantees certain measures of freshness), updates are not propagated into memoized functions. If updates occur that invalidate a value in a memoized function, the value is simply dropped.

[3]Recall that we need to consider outerjoins because of the lookahead computations.

The Equal Access Cost Assumption: In this paper we make the assumption that the raw data used in building the Web site and the cached data (views and mappings) are accessible at run-time with the *same* cost. Except for being a reasonable assumption about current practice in managing Web sites[4], this assumption enables us to focus our discussion on the novel issues of designing a data warehouse. In particular, we will not consider the distinction between local data and remote data, and we will not consider maintaining additional views *only* for the purpose of self-maintainability of other views.

3.4 Reformulating a Site using a Configuration

Given a configuration for the warehouse, we need to reformulate the site definition queries so they make use of the materialized views and the memoization functions. Since the memoized functions are not necessarily complete and therefore may not contain the tuple we need at a given point, it is not enough to consider a single rewriting for every click-time query in the site definition. Instead, we need to consider every combination of memoization functions.

Definition 3.2: *Given a site-schema that includes the click-time queries $Q_1, \ldots, Q_n$, and a configuration $C = (\mathcal{V}, \mathcal{F})$, a reformulation of the site-schema is a set of queries $\mathcal{Q}_i$ for every $1 \leq i \leq n$, such that each query $Q \in \mathcal{Q}_i$ is a rewriting of Q_i using the views, the memoized functions and the raw data, and such that*

- *there is exactly one Q in each $\mathcal{Q}_i$, called the* default query*, that does not use any of the memoized functions. Rewritings that do use memoized functions are called* contingent queries*.*
- *there is no pair of queries Q', Q'' in $\mathcal{Q}_i$ that use exactly the same set of views and functions.*

Obviously, there are many reformulations of the site schema that satisfy the above conditions. With every reformulation we associate a cost, which provides a estimate of the utility of the rewritten site. The cost function estimates the cost of evaluating the reformulated click-time queries, and takes into consideration the different browsing patterns, the maintenance costs and the costs of managing the memoized functions (i.e., computing and refreshing their values). To goal of the reformulation step is to find a reformulation of the site schema that minimizes the cost function.

The problem of reformulating a site schema is closely related to the problem of rewriting queries using views [18, 3, 12, 15, 6]. The major differences between previous works and the problem we solve here are: (1) we need to consider queries with outerjoins, (2) memoized functions can only be used in restricted ways in the rewritings, and (3) we need to also consider the possibility of exploiting known preconditions on the constants appearing in the queries. The last point is very unique to the Web site context. Recall that the click-time queries will only be evaluated on nodes $\mathtt{F}(\bar{\mathtt{a}})$ which were constructed previously by other click-time queries. Hence, by tracing back the path in the site graph leading to $\mathtt{F}(\bar{\mathtt{a}})$, we can derive certain preconditions which are known to hold on the constant(s) $\bar{\mathtt{a}}$. Thus, we can rewrite the click-time query at $\mathtt{F}(\bar{\mathtt{a}})$ based on these preconditions, yielding a more efficient query. In practice this often results in important query optimization, because the preconditions and the incremental query often share common subgoals. Section 5 describes how we deal with these problems.

[4]Note that if an organization does create a data warehouse solely for supporting its Web site, then this warehouse is considered the raw data.

3.5 New Incremental Evaluation Algorithm

Given a configuration and a rewriting of the site schema based on the configuration, we obtain a natural run-time algorithm, shown in Figure 6. Recall that the reformulation of each click-time query in the site specification results in a single default query and in a set of contingent queries. Clearly, the expected cost of executing a contingent query is lower than that of the default query. A contingent query `Q` is applicable at a given node with input $\bar{a}$ if for each function `F` used in the definition of `Q`, the data cached for `F` contains the tuples which are needed for the current computation of `Q` given $\bar{a}$. Hence, the execution of the click-time query in the naive algorithm is replaced by an attempt to find an applicable contingent query, and if none is found, the default query is executed.

A final difference from the naive algorithm is that in addition to computing the outgoing edges from $\mathtt{F}(\bar{a})$ at a given node, we also check whether the configuration implies that we need to update the cache by computing values for any of the memoized functions with the input $\bar{a}$.

```
ALGORITHM IncrEval
INPUT: some node F(a) in the site graph to be displayed
OUTPUT: all successors of F(a)
SIDE EFFECT on the cached data for the memoized functions
METHOD:

/* side effect of the content of the memoized functions */
   FOR EVERY memoization function (Q, In, N), if N includes the current node DO
     compute Q[a/In] and add all the tuples result to the cache

/* compute the outgoing edges for F(a) */
  FOR EVERY outgoing edge F -> G labeled (Q, L, X, Y) in the site schema DO
    LET S be the set of rewritten queries for Q
    LET Q' be the default query for Q
    LOOK for a contingent query Q'' in S such that Q''  is applicable with input a
    IF found
        THEN  compute the query I(L, Y) :- Q''[a/X]
        ELSE compute the query I(L, Y) :- Q'[a/X] /* the default query */
    FOR EVERY (l,b) in the answer of I DO
       OUTPUT edge F(a) -> l -> G(b)
```

Figure 6: Incremental Evaluation Algorithm

4 Searching the Space of Configurations

The problem we consider in this section is the following. Given a site schema, which includes a set of click-time queries $\mathcal{Q} = Q_1, \ldots, Q_n$, find a configuration whose cost is minimal. A naive approach of searching through the set of *all* possible views and memoization functions that can be expressed over the raw data would be impractical. In this section we describe a transformation based search in a space of possible Web site configurations. The algorithm is distinguished in that it searches only a subset of the possible configuration whose size is tractable, and is still guaranteed to find the optimal configuration given the equal access cost assumption.

Before we begin, it is instructive to note the differences between our problem and the traditional problem of data warehouse design [**?**, **?**, **?**]. In that work a warehouse configuration includes only a set of views to materialize while in our context, as we showed in the previous section, it is important to consider memoizing functions. Second, as a consequence of the equal access cost assumption, in our context it is not necessary to consider configurations that contain data useful only for self-maintainability of other views. We can restrict the search to configurations containing only data useful in some rewritings. Finally, we note that since we can access the raw data, we consider rewritings that involve both the raw data and the materialized views, while in traditional data warehouses it is often the case that only the views can be used in rewritings.

The search algorithm proceeds as follows. The initial state is the empty configuration, i.e., no views and no functions to memoize. The algorithm uses a set of transformations `T1`, ..., `T5` to generate the space. The application of the transformations is governed by the following rule: do not apply a transformation T_i to a configuration C if a transformation T_j, where $j > i$ has been applied to obtain C. Note that this rule significantly reduces our search by avoiding cycles in the search space and reducing the number of configurations we consider multiple times.

The transformation rules are the following. Below we denote the current configuration by $C = (\mathcal{V}, \mathcal{F})$, and we denote views by V, V_1 and V_2.

T1. **Add a new view V to $\mathcal{V}$**, where V is a subquery of one of the Q_i's that does not involve selections or projections.

T2. **Replace two views V_1 and V_2 in $\mathcal{V}$ by $V_1 \text{⟗} V_2$.**

T3. **Replace $V \in \mathcal{V}$ by $\sigma_c(V)$.** We consider selections c constructed as follows. Let $\mathcal{S}$ be the set of all selections appearing in $\mathcal{Q}$. We consider only selections that are constructed by formulas involving conjunction, disjunction and elements of $\mathcal{S}$.

T4. **Replace $V \in \mathcal{V}$ by $\pi_{\bar{A}}(V)$**, where $\bar{A}$ is a subset of the variables of V.

T5. **Relegate a view from $\mathcal{V}$ to $\mathcal{F}$**, i.e., remove a view V from $\mathcal{V}$ and add to $\mathcal{F}$ a memoization function of the form (V, In, N), where In a subset of the distinguished variables of V, and N is a subset of the nodes in the site schema.

The following gives a syntactic characterization of the configurations that our algorithm will consider. This characterization is the basis for showing that our algorithm is guaranteed to find the optimal configuration.

Property 4.1: *Our algorithm generates a configuration if and only if all the views and memoization functions in the configuration are defined by expressions of the following form:*

$$\pi_{\bar{X}}(\sigma_{\bar{C}}(\tau_1 \text{⟗} \tau_2 \text{⟗} \ldots \text{⟗} \tau_n))$$

where τ_i is of the form $A_1 \bowtie \ldots \bowtie A_m$, $\bar{X}$ is a set of variables to be projected, $\bar{C}$ is a set of selection conditions, and each τ_i is a subquery of one of the Q_j's.

We show that our algorithm is guaranteed to find the optimal configuration by showing that the configurations that are ignored are redundant. The equal access cost assumption enables us to define the set of *redundant configurations* to be configurations that store data that is not used in any rewriting of the queries in $\mathcal{Q}$. Formally, we define redundant configurations as follows.

Definition 4.1: *A configuration $(\mathcal{V}, \mathcal{F})$ is redundant if there exists a non identical configuration $(\mathcal{V}', \mathcal{F}')$ such that*

- *for every $V \in \mathcal{V}$ there exists a $V' \in \mathcal{V}'$, such that V' can be obtained from V by a combination of selections and projections,*
- *for every $F \in \mathcal{F}$ there exists an $F' \in \mathcal{F}'$ or a view $V \in \mathcal{V}$, such that F' can be obtained from either F or V by a combination of selections and projections,*
- *The cost of the configuration $(\mathcal{V}', \mathcal{F}')$ is less or equal to the cost of the configuration $(\mathcal{V}, \mathcal{F})$.*

Clearly, the optimal configuration is not redundant. The following theorem entails that our algorithm is guaranteed to find the optimal configuration.

Theorem 4.1: *If a configuration $C = (\mathcal{V}, \mathcal{F})$ is not redundant, then each expression used in defining views or functions in C is equivalent to an expression of the form described in Property 4.1.*

The proof of the theorem is based on analysis of the different possible forms for expressions used in the configuration. In the first step we consider expressions without selections or projections and we show that expressions that are *not* defined as outerjoins of subqueries of the Q_i's cannot be used in rewritings. In the second step we generalize this claim to expressions involving selections and projections.

Improvements on the search algorithm: Theorem 4.1 shows that our algorithm searches a *superset* of the set of nonredundant configurations, that is, it may still examine redundant configurations. It is possible to refine the transformation rules by adding conditions on their application and thereby reduce the number of redundant configurations that the algorithm will consider. For example, consider the transformation `T2`. Assume that V_1 and V_2 are generated by rule T1, i.e., each of them is a subquery of one of the Q_i's. The additional condition states that we only replace V_1 and V_2 by their outerjoin if there exist queries Q_1, Q_2, Q_3 in $\mathcal{Q}$ such that V_1 is a subquery of Q_1, V_2 is a subquery of Q_2, $V_1 \bowtie V_2$ is a subquery of Q_3, but not of Q_1 or Q_2. The motivation is that from the outerjoin of V_1 and V_2 we can obtain through selections or projections each of V_1, V_2 and $V_1 \bowtie V_2$. If at least one of these three (V_1, V_2, $V_1 \bowtie V_2$) is *not* used independently in a rewriting, this means that the configuration containing the outerjoin of V_1 and V_2 is redundant, therefore being non-optimal. This condition naturally generalize to the case when V_1 and V_2 are expressions generated by T2. Similar conditions can be formulated for the other transformation rules.

5 Reformulating the Web Site

Given a configuration, i.e., a set of views to materialize and functions to memoize, we need to reformulate the queries in the site definition to use the views and functions. Broadly speaking, the reformulation step can be considered as an instance of the problem of *rewriting queries using views*, which has been considered in several works [18, 3, 12, 15, 16, 6]. However, there are several issues in our context that introduce additional difficulties and provide opportunities for special optimizations. In this section we describe the differences and and describe an algorithm for reformulation that is especially appropriate in this context.

5.1 Differences from the Traditional Setting

There are three main differences between the view rewriting problem we need to consider here and previous work.

Rewriting with functions: both the views and the functions in a configuration are defined by query expressions. However, we cannot treat the functions in the exact same way as the views, since the functions are not necessarily complete and are also *directional* (from the input variables to the other variables). Recall that a click-time query is given an input. In considering a contingent query Q at run-time, we need to be able to determine whether the answers obtained by Q are complete *before* applying Q (by checking whether the appropriate values exist in the caches of memoized functions used by Q). This restricts the way we can use memoized functions in rewritings in the following way: in a rewriting of a click-time query Q the input variables of a function F need to be a subset of the input variables of Q. The following example illustrates a possible pitfall in using functions.

```
{ WHERE R(X,Z)                  /* Q0(X,Z) */
  LINK RootPage() -> "Start" -> Page1(X)
}
{ WHERE R(X,Z), T(X,W)          /* Q1(X,Z,W) */
  LINK  Page1(X)-> "Related" -> Page2(X)
}
{ WHERE R(X,U), S(U,V), R(V,Y)  /* Q2(X,U,V,Y) */
     Page2(X) -> "Next" -> Page1(Y)
}
```

Assume we chose to maintain the function $\mathtt{F(X,Z) :- Q1(X,Z,W)}$ whose input variable is X: the intent is to compute the function at the nodes Page1(X) and use the cached values at the nodes Page2(X). The click-time query for Page2(X) is:

$\mathtt{I(\text{"}Next\text{"},Y) :- Q2(X,U,V,Y)}$

where X is expected to be bound. The best rewriting of the query using the function F is

$\mathtt{I'(\text{"}Next\text{"},Y) :- F(X,U),S(U,V),R(V,Y)}$.

That is, we cannot further rewrite R(V, Y) into F(V, Y), because we would then be using the function F with bindings for the variable V, which are not given as input to I. Hence, we could not determine ahead of time whether the rewriting would return the complete answer. Note that the restriction on the use of X here is simpler than that of limited access pattern to tuples: query rewriting with limited access patterns was considered in previous work [15, 13, 11, 7]. In those works the above rewriting would still yield the complete answer.

Rewriting under preconditions: In our setting a click-time query is not arbitrary; a query can only result by following some *path* beginning at an entry point of the Web site. This observation may enable us to further optimize the click-time query, because some of the previous queries on the path may render some subgoals redundant. As an example, consider again the query in Fig. 3, and assume we want to display the node DeptPage(d). One of the click-time queries to evaluate is I("Member", P):

```
I("Member",P) :- Orgs(d, _, _, "Dept", _), Person(P, S, d, _)
```

Depending on how we reached node DeptPage(d), our precondition could be either Q2 or Q3: assume that it is Q3, i.e., we reached that department by traversing a link "Dept" from some other person's page. Then, we know that d satisfies the following precondition:

```
C(d) :- Orgs(d, _, _, "Dept", _), Person(P', S', d, _)
```

Hence, we can reformulate the click-time query to be:

```
I("Member",P) :-  Person(P, S, d, _).
```

Rewriting with outerjoins: As described in the previous section, we may store views in our configuration that involve outerjoins of conjunctive queries. Thus, we need to extend methods for rewriting queries using views to views with outerjoins. Fortunately, it is easy to observe that given an outerjoin $\pi(\sigma(\mathtt{F_1} ⟗ \mathtt{F_2}))$, it is easy to obtain from it the expressions $\mathtt{F_1}, \mathtt{F_2}$ and $\mathtt{F_1} \bowtie \mathtt{F_2}$ via selections and projections, and conversely, the outerjoin can be obtained from these three relations (assuming we can use the union operator). Hence, it can be shown that the problem of query rewriting with outerjoins is reducible to query rewriting with conjunctive queries.

5.2 Site Reformulation Algorithm

We now describe an algorithm for efficiently finding rewritings of the site-definition queries using the views and memoized functions. Recall that a view is simply a function whose set of input variables is empty. To simplify the discussion, our algorithm makes three assumptions:

- at most one memoized function F is used in the rewriting of a particular query Q in the site definition,
- the function F will occur at most once in the rewriting of Q, and
- the set of input variables $\bar{\mathtt{X}}$ of F is the same as the inputs of Q.

The first and third assumptions can be relaxed and are only made in order to simplify the discussion. The second assumption is crucial for the complexity of the algorithm we describe, but captures many commonly occurring cases in practice. Furthermore, it follows from [12] that even if this assumption is made, the complexity of rewriting queries using views is still NP-complete.

Formally, we are given a function $\mathtt{F}(\bar{\mathtt{X}}, \bar{\mathtt{Z}})$ and a click-time query $\mathtt{Q}(\bar{\mathtt{X}}, \bar{\mathtt{Y}})$ from the site specification, where $\bar{\mathtt{X}}$ are the inputs of both F and Q. Our goal is to find an equivalent rewriting of the form

$\mathtt{S}(\bar{\mathtt{X}}, \bar{\mathtt{Y}}) : -\mathtt{F}(\bar{\mathtt{X}}, \bar{\mathtt{U}}), \mathtt{Q}'(\bar{\mathtt{X}}, \bar{\mathtt{Y}}).$

where $\mathtt{Q}'(\bar{\mathtt{X}}, \bar{\mathtt{Y}})$ is a subset of Q. The variables $\bar{\mathtt{U}}$ are taken either from the variables of Q or are fresh variables. We require that $\mathtt{S}(\bar{\mathtt{X}}, \bar{\mathtt{Y}})$ is equivalent to the query

$\mathtt{T}(\bar{\mathtt{X}}, \bar{\mathtt{Y}}) : -\mathtt{F}(\bar{\mathtt{X}}, \bar{\mathtt{Z}}), \mathtt{Q}(\bar{\mathtt{X}}, \bar{\mathtt{Y}}).$

Note that $\mathtt{F}(\bar{\mathtt{X}}, \bar{\mathtt{Z}})$ appears in $\mathtt{T}(\bar{\mathtt{X}}, \bar{\mathtt{Y}})$ because F can also be viewed as a precondition on $\bar{\mathtt{X}}$, and our goal is to find a rewriting that is equivalent under the preconditions.

The main problem faced in our context is that our search algorithm needs to apply the rewriting algorithm at every state in its search space. However, we observe that the reformulation problems that arise in adjacent states share several of their inputs, and therefore significant performance improvements can be obtained if the reformulation algorithm can exploit the commonalities between successive applications. In what follows we describe an algorithm based on this idea. Specifically, given a STRUQL query with n WHERE clauses $\mathtt{Q_1}, \ldots, \mathtt{Q_n}$, our method proceeds in two steps. Step 1 preprocesses the n queries and constructs an auxiliary data structure. Step 2 consists of a fast rewriting algorithm for a rewriting problem F, Q where both F and Q are (sub)queries of $\mathtt{Q_1}, \ldots, \mathtt{Q_n}$: this step is invoked at every state in the search space. We describe both steps in detail next.

Step 1: Preprocessing the site definition. The auxiliary data structure we construct is a collection of n^2 *partial variable mappings*, $\mathtt{f_{ij}}$ from $\mathtt{Q_i}$ to $\mathtt{Q_j}$, $\mathtt{i}, \mathtt{j} = \mathtt{1}, \ldots, \mathtt{n}$. Each partial variable mapping consists of a subset of the body of $\mathtt{Q_i}$, called *domain* and denoted with $\mathtt{dom(f_{ij})}$, and a variable mapping from its domain to $\mathtt{Q_j}$. To be more precise, $\mathtt{f_{ij}}$ maps the variables occurring in $\mathtt{dom(f_{ij})}$ to $\mathtt{Q_j}$'s variables, s.t. for each goal $\mathtt{G}$ in $\mathtt{dom(f_{ij})}$, $\mathtt{f_{ij}(G)}$ is a goal in $\mathtt{Q_j}$. There is no restriction on how to choose the subset of goals $\mathtt{dom(f_{ij})}$: that is, $\mathtt{dom(f_{ij})}$ may share common variables with other goals in $\mathtt{Q_i}$. Any of the $\mathtt{f_{ij}}$'s may be empty: but, as expected, making an effort to discover as large partial mappings as possible pays off in Step 2, with better rewritings. Note that we retain only a *single* partial containment mapping for each pair of queries, although, in general, several may exists.

Step 2: Fast rewriting algorithm. In this step we are given a rewriting problem $\mathtt{F(\bar{X}, \bar{Z}), Q(\bar{X}, \bar{Y})}$, where both queries are either one of the original n queries in the site definition or are a subset of such a query[5]. Hence, from Step 1 we have a pair of partial mappings $\mathtt{f} : \mathtt{Q} \to \mathtt{F}$ and $\mathtt{g} : \mathtt{F} \to \mathtt{Q}$. Our approach to solving the rewriting problem efficiently is to restrict the search only to those rewritings which are "witnessed" by $\mathtt{f}$ and $\mathtt{g}$. Formally, our algorithm searches for a minimal subquery $\mathtt{Q'}$ of $\mathtt{Q}$ such that

1. there exists $\mathtt{\bar{U}}$ s.t. for each variable $\mathtt{Z_i} \in \bar{Z}$ either $\mathtt{U_i} = \mathtt{Z_i}$, or $\mathtt{f(Z_i)} = \mathtt{U_i}$ and $\mathtt{g(U_i)} = \mathtt{Z_i}$,
2. there exists a containment mapping $\varphi : \mathtt{F(\bar{X}, \bar{U}), Q'(\bar{X}, \bar{Y})} \to \mathtt{F(\bar{X}, \bar{Z}), Q(\bar{X}, \bar{Y})}$, which is the identity on $\mathtt{\bar{X}, \bar{Y}}$, and s.t. for any variable $\mathtt{V}$ either $\varphi(\mathtt{V}) = \mathtt{f(V)}$ or $\varphi(\mathtt{V}) = \mathtt{V}$, and
3. there exists a containment mapping $\gamma : \mathtt{F(\bar{X}, \bar{Z}), Q(\bar{X}, \bar{Y})} \to \mathtt{F(\bar{X}, \bar{U}), Q'(\bar{X}, \bar{Y})}$, which is the identity on $\mathtt{\bar{X}, \bar{Y}}$, s.t. for any variable $\mathtt{V}$, either $\gamma(\mathtt{V}) = \mathtt{g(V)}$ or $\gamma(\mathtt{V}) = \mathtt{V}$.

We show in the sequel that, under these restrictions on the possible rewritings, the maximal rewriting can be found efficiently. We do so by reducing this problem to the 2SAT problem [14], which has an efficient PTIME solution.

The reduction of the rewriting problem to a 2SAT problem proceeds by defining a number of propositional variables encoding the decisions we have to take in (1), (2), and (3) above. The propositional variables are described in Fig. 7. In the table, $\mathtt{C(Z)}$ encodes the choice of the variables $\mathtt{\bar{U}}$, while $\mathtt{A(V)}, \mathtt{A'(V)}$ encode the decisions leading to the choices of mappings φ and γ. In addition to encoding these mappings we introduce some variables associated to goals. First note that it suffices to restrict the mappings to $\varphi : \mathtt{F(\bar{X}, \bar{U})} \to \mathtt{F(\bar{X}, \bar{Z}), Q(\bar{X}, \bar{Y})}$ and $\gamma : \mathtt{Q(\bar{X}, \bar{Y})} \to \mathtt{F(\bar{X}, \bar{U}), Q'}$: then, the propositional variables $\mathtt{B(G)}$ and $\mathtt{B'(G)}$ encode where each goal $\mathtt{G}$ on the left hand side is mapped on the right hand side.

The propositional variables are subject to the conditions given in Figure 8. Here $\mathtt{G}$ ranges over goals and $\mathtt{V} \in \mathtt{var(G)}, \mathtt{V} \notin \mathtt{\bar{X}, \bar{Y}}$. For example the first condition states that all goals $\mathtt{G}$ in $\mathtt{F(\bar{X}, \bar{U})}$ which are not in $\mathtt{dom(f)}$ have to be mapped by φ to themselves (not to $\mathtt{Q(\bar{X}, \bar{Y})}$). The second says that if $\varphi(\mathtt{V}) = \mathtt{f(V)}$ then $\mathtt{G}$ is mapped by φ to $\mathtt{Q(\bar{X}, \bar{Y})}$, etc. Our problem is to find truth assignemnts to all propositional variables s.t. all these conditions are satisfied.

There is a 1-1 correspondence between truth assignments to the satisfaction problem and solutions to the rewriting problem satisfying our restriction. Furthermore, our rewriting goal is to minimize $\mathtt{Q'}$, which translates into maximizing the number of variables $\mathtt{B'(G)}$ assigned the value `true`. However the satisfaction problem is not 2SAT, because of the clause before last in Figure 8.

[5]Recall that we are treating outerjoins separately.

Problem	Parameter	Propositional Variable	Meaning
find $\bar{\text{U}}$	$\text{Z} \in \bar{\text{Z}}$	C(Z)	true iff Z is substituted by f(Z) false iff Z is substituted by itself
find φ	$\text{V} \in \text{var(F)}, \text{V} \notin \bar{\text{X}}$	A(V)	true iff $\varphi(\text{V}) = \text{f(V)}$ false iff $\varphi(\text{V}) = \text{V}$
	$\text{G} \in \text{body}(\text{F}(\bar{\text{X}}, \bar{\text{U}}))$	B(G)	true iff $\varphi(\text{G}) \in \text{body}(\text{Q}(\bar{\text{X}}, \bar{\text{Y}}))$ false iff $\varphi(\text{G}) \in \text{body}(\text{F}(\bar{\text{X}}, \bar{\text{Z}}))$
find γ	$\text{V} \in \text{var(Q)}, \text{V} \notin \bar{\text{X}}, \bar{\text{Y}}$	$\text{A}'\text{(V)}$	true iff $\varphi(\text{V}) = \text{g(V)}$ false iff $\varphi(\text{V}) = \text{V}$
	$\text{G} \in \text{body}(\text{Q}(\bar{\text{X}}, \bar{\text{Y}}))$	$\text{B}'\text{(G)}$	true iff $\varphi(\text{G}) \in \text{body}(\text{F}(\bar{\text{X}}, \bar{\text{U}}))$ false iff $\varphi(\text{G}) \in \text{body}(\text{Q}'(\bar{\text{X}}))$

Figure 7: Definition of propositional variables

Problem	Parameters	Boolean Clause
find φ and $\bar{\text{U}}$	$\text{G} \notin \text{dom(f)}$	$\neg\text{B(G)}$
($SAT(\varphi)$)		$\text{A(V)} => \text{B(G)}$
		$\text{C(Z)} => \text{A(Z)}$
		$\text{B(G)} => \text{A(V)}$
	$\text{Z} \in \bar{\text{Z}}, \text{g(f(Z))} \neq \text{Z}$	$\neg\text{C(Z)}$
find γ and $\bar{\text{U}}$	$\text{G} \notin \text{dom(g)}$	$\neg\text{B}'\text{(G)}$
($SAT(\gamma)$)		$\text{A}'\text{(V)} => \text{B}'\text{(G)}$
	$\text{g(V)} \notin \bar{\text{Z}}$ or $\text{f(g(V))} \neq \text{V}$	$\text{B}'\text{(G)} => \text{A}'\text{(V)}$
	$\text{g(V)} = \text{Z} \in \bar{\text{Z}}$ and $\text{f(Z)} = \text{V}$	$\text{B}'\text{(G)} \wedge \neg\text{C(Z)} => \text{A}'\text{(V)}$
	$\text{g(V)} = \text{Z} \in \bar{\text{Z}}$ and $\text{f(Z)} = \text{V}$	$\text{A}'\text{(V)} => \neg\text{C(Z)}$

Figure 8: Conditions imposed on the propositional variables

Therefore we split the problem into two: (a) the SAT problem for φ, call it $SAT(\varphi)$, and (b) the SAT problem for γ, call it $SAT(\gamma)$. $SAT(\varphi)$ is not only 2SAT, but is also a Horn satisfaction problem, HORNSAT [14]: this implies that it has a maximal solution (i.e. for which the set of true variables is maximal among all solutions). Moreover, given any solution to $SAT(\varphi)$, we obtain a 2SAT problem for $SAT(\gamma)$ by substituting the appropriate values of all propositional variables C(Z). Of course, we don't want solve $SAT(\gamma)$ for *every* solution to $SAT(\varphi)$: instead we will solve it only for the maximal solution. The following lemma shows that this is sufficient:

Lemma 5.1: *Consider a solution S1 to the combined problem* $SAT(\varphi), SAT(\gamma)$*, and some solution S2 to* $SAT(\varphi)$ *s.t. for every variable* C(Z)*, if p is its value in S1 and q its value in S2, then* $p \longrightarrow q$*. Then there exists a solution S2' to the combined problem, extending S2, s.t. for any propositional variable* $\text{B}'\text{(G)}$ *its value in S2' is the same as its value in S1.*

Proof: Note that there exists a one to one correspondence between the variables C(Z) and $\text{A}'\text{(V)}$ which appear in a common clause in $SAT(\gamma)$. Then, by changing C(Z) from false to true, we may be forced to change its associated $\text{A}'\text{(V)}$ from true to false (clause $\text{A}'\text{(V)} => \neg\text{C(Z)}$): the other clause connecting them, $\text{B}'\text{(G)} \wedge \neg\text{C(Z)} => \text{A}'\text{(V)}$ becomes automatically true. The changed $\text{A}'\text{(V)}$'s may further only occur in rules of the form $\text{A}'\text{(V)} => \text{B(G)}$, which are obviously true.

Finally, once we substituted the maximal solution of $SAT(\varphi)$ in $SAT(\gamma)$, the latter is again a HORNSAT: hence it admits a maximal solution (recall: we want to maximize the $\text{B}'\text{(G)}$'s) which can be found efficiently. This shows:

Theorem 5.2: *Under the given restriction, the query rewriting problem can be solved efficiently and in polynomial time.*

Finally, we note that the algorithm easily extends to the case of multiple functions $\mathtt{F}_1, \mathtt{F}_2, \ldots$, by incorporating them in the rewritings successively: that is, $\mathtt{Q}$ is first rewritten as $\mathtt{F}_1, \mathtt{Q}'$, then as $\mathtt{F}_1, \mathtt{F}_2, \mathtt{Q}''$, etc.

6 Conclusions

A critical issue in building Web sites from declarative specifications is when to compute the Web site: completely in advance of browsing or dynamically at click-time. We described an approach for automatically selecting a Web-site warehouse configuration that optimally trades off precomputation and dynamic evaluation. We identified a novel set of optimizations that can be incorporated into the design of a warehouse supporting a Web site, and formulated the warehouse design problem as a search in a space of configurations. We presented an algorithm for searching the space of configurations and for assisting the Web site administrator to choose among them. The algorithm is designed to search only a restricted portion of the search space, but is still guaranteed to find the optimal configuration under certain assumptions. Finally, we described an efficient algorithm for reformulating the Web site specification given a configuration. The algorithm represents a novel solution to the problem of rewriting queries using views that is tailored to the needs of our application.

Several systems have been proposed lately for building web sites declaratively. The ARANEUS project [2] is based on importing a Web site in a relational database management system, processing it using a relational query language, then reconverting it back into Web hypertexts. The proposal described in [5] is based on the YAT data conversion algorithm to convert data in any format into HTML, while [**?**] describes the language WebOQL that is able to restructure Web data. With the exception of [5] that is designed to compute the HTML on the fly (corresponding to our naive algorithm), all the proposals completely precompute the Web site, and none of them have discussed the issue of trading off precomputation and dynamic computation.

References

[1] S. Adali, K. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of SIGMOD-96*, 1996.

[2] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. To weave the web. In *Proceedings of VLDB*, 1997.

[3] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of International Conference on Data Engineering*, 1995.

[4] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogenous information sources. In proceedings of IPSJ, Tokyo, Japan, October 1994.

[5] Sophie Cluet, Claude Delobel, Jerome Simeon, and Katarzyna Smaga. Your mediators need data conversion. In *To appear in Proceedings of SIGMOD*, 1998.

[6] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona.*, 1997.

[7] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.

[8] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with strudel: experience with a web-site management system. In *To appear in Proceedings of SIGMOD*, 1998.

[9] D. Florescu, L. Raschid, and P. Valduriez. A methodology for query reformulation in cis using semantic knowledge. *Int. Journal of Intelligent & Cooperative Information Systems, special issue on Formal Methods in Cooperative Information Systems*, 5(4), 1996.

[10] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd VLDB Conference, Athens, Greece*, 1997.

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

[12] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Jose, CA*, 1995.

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

[14] Christos Papadimitriou. *Computational Complexity.* Addison Wesley Publishing Company, 1994.

[15] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Jose, CA*, 1995.

[16] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101–118, 1996.

[17] Darrel Woelk, Paul Attie, Phil Cannata, Greg Meredith, Amit Seth, Munindar Sing, and Christine Tomlinson. Task scheduling using intertask dependencies in Carnot. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 491–494, 1993.

[18] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *Proceedings of the 13th International VLDB Conference*, pages 245–254, 1987.