Optimization of Run-time Management of Data Intensive Web Sites

Daniela Florescu
INRIA Rocquencourt, France
Daniela.Florescu@inria.fr

Alon Levy
University of Washington, Seattle
alon@cs.washington.edu

Dan Suciu
AT&T Labs – Research
suciu@research.att.com

Khaled Yagoub
PRISM Versailles, France
khaled.yagoub@prism.uvsq.fr

Abstract

An increasing number of web sites have their data extracted from relational databases. Several commercial products and research prototypes have been moving in the direction of declarative specification of the sites' structure and content. Specifically, the entire site is specified using a collection of queries describing the site's nodes (corresponding to web pages and the data contained in them) and edges (corresponding to the hyperlinks). Given this paradigm, an important issue is *when* to compute the site's pages. Two extreme approaches, with obvious drawbacks, are (1) to precompute the entire site in advance, and (2) to evaluate on demand all the queries necessary to construct a given page. We consider the problem of automatically optimizing the run-time management of declaratively specified web sites. In our approach, given a declarative site specification and constraints on the application, an efficient run-time evaluation policy is automatically derived. An evaluation policy specifies which data to compute at a given browser request. We describe several optimizations that can be used in run-time policies, focusing mostly on optimizations that exploit the *structure* of the web site. We evaluate experimentally the impact of these optimizations on a web site derived from the TPC/D database. Finally, we describe a heuristic-based optimization algorithm which compiles a declarative site specification into a run-time policy that incorporates the proposed optimizations.

1 Introduction

The World Wide Web (WWW) has been proven to be an excellent medium for businesses to disseminate information. As a result, the ability to populate web sites with content derived from large databases has become the key to building enterprise web sites. Tools addressing this problem range from low-level CGI-bin scripts to more sophisticated tools provided by most major DBMS vendors, that enable embedding SQL queries in HTML templates.

In parallel, a new paradigm for building and maintaining web sites based on declarative specifications has arisen in the research community [9, 3, 7, 2, 20]. Two main features underlie this paradigm. First, a declarative specification is based on a logical model of the web site, that captures the content and structure of the web site and is meant to be independent of its graphical presentation. Second, the logical model of the site is defined as a view, in some declarative language, over the data underlying the site. Web-site management systems based on declarative representations have been shown to provide good support for common tasks which are otherwise tedious to perform, such as automatic site updates, site restructuring, creation of multiple versions of a site from the same data, and specification and enforcement of integrity constraints [10].

An example of this paradigm, which is the focus of this paper, is the case where web sites' content is derived from large relational databases. We model web sites as graphs whose nodes represent pages in the web site or data items associated with pages, and the links in the graph represent either hyperlinks between pages or association of data with pages in the site. The structure of the web site is defined *intensionally* by a site schema, which can be regarded as a hyperlink view defined over the relational database.

A critical issue that arises when sites' contents are populated from large databases is when to compute the pages in the site [21] and/or the corresponding nodes in the logical model. One approach is to materialize the site completely, i.e., evaluate all the database queries in the site definition, and compute the complete site before users browse it. Unfortunately, this approach has several obvious drawbacks. First, precomputation cannot be applied to sites with forms (since the inputs are only known at run-time). Second, materializing the site would imply an important space overhead, often even greater than duplicating the entire database, since the same information in the database

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

can appear in multiple web pages. Finally, propagating updates from the database to the web site is costly once the site has been materialized.

A second extreme approach (deployed by commercial tools for extracting content of web sites from databases) is to precompute only the root(s) of a web site, and when a page is requested, to issue to the database a set of parameterized queries that extract the necessary data. The main disadvantage of this approach is that some queries may be too expensive to evaluate at run-time, which is unacceptable given the interactive nature of web access. Furthermore, evaluating queries at run-time may result in repeated computation. An obvious repetition occurs when multiple browsers request the same page. A second, more interesting observation, is that successive queries issued by a single browser share much of their computation.

Multiple requests for the same web page could conceivably be treated by web caching techniques. However, these solutions have two problems. First, current caching techniques do not cache dynamically generated pages. Second, even if caching techniques are extended (e.g., by server-side caching for dynamically generated pages), the granularity of an entire HTML page is too coarse. Clearly, in order to develop optimizations based on sharing of computation in query sequences, a deeper semantic analysis of the web-site structure is required.

Currently, since response time is the main priority, web site builders end up hardwiring optimizations into the design of their sites. Such hardwiring is a labor intensive task which needs to be repeated whenever changes are made to the site's structure. This paper considers the problem of automatically optimizing the run-time behavior of the dynamic evaluation of declarative web sites. We describe a framework, where a declarative specification is compiled into a run-time policy. The policy decides which actions to perform and which queries to evaluate depending on the browsing history. Run-time policies are able to express several traditional optimizations, such as view materialization and data caching, and novel optimizations that depend on the structure of the web site, such as optimization under preconditions and lookahead computation. In a sense, the distinction between the declarative specification of the web site and the run-time policy is analogous to the distinction between a declarative query and a query execution plan in a traditional database. As in the latter context, we automatically compile the declarative specification into an "optimal" run-time policy which is "equivalent" to the declarative specification, using a global cost model, statistics on the database and browsing patterns.

As a first cut, a possible approach to our problem is to consider the set of parameterized queries that are executed against the database as a particular workload, and to apply some of the existing techniques aimed at optimizing a given workload. Such techniques have been considered in various contexts, such as view materialization [24, 13, 12, 14, 6], index selection, data caching [16, 15, 8], multiple query optimization [23], and reuse of query invariants [17, 22].

However, none of the above techniques exploit a key aspect of our context, namely the structure of the web site. The structure of a web site imposes a *topology* over the possible navigational paths through the site and therefore on the set of queries in the workload. More precisely, at each point in the site, while issuing new queries to the database, we have an additional valuable information about the *past* queries issued to the database (which we call the browsing context), as well as extra information about the likelihood of possible *future* queries that may be executed. In this paper, we show that exploiting this structure leads to significant savings over and above the application of the known techniques mentioned above. As a consequence, our techniques are also useful beyond web-site management, for contexts in which the application imposes an analyzable topology on the workload of queries (e.g. SQL queries embedded in programming languages or trigger chains).

In summary, we make the following contributions.

- We describe a framework for automatic compilation of web-site specifications. The framework distinguishes between a declarative specification of the structure and content of a web site, and a run-time policy governing the computation of the web site. The formalism for describing run-time policies can encompass traditional optimizations as well as novel ones specific to our context.
- We describe several optimization techniques for speeding up the run-time behavior of web sites. One class of optimizations includes precomputing a set of views and caching results of certain computations. The second class of optimizations exploits the structure of the web site and includes (1) simplification of queries based on known preconditions, and (2) lookahead computation, i.e., computing more data than is immediately needed for use in nodes that are likely to be visited subsequently. We evaluate the impact of these optimization techniques on a web site derived from the TPC/D data, and show that each of them, even in isolation, yields significant speedups.
- Based on our experiments, we describe a set of guidelines for constructing an algorithm for compiling declarative specifications into run-time policies. We show that applying these guidelines in our experimental setting produced high-quality run-time policies.
- We describe the implementation of STRUDEL-R system[1], which embodies the ideas described in the paper.

The paper is organized as follows. Section 2 describes declarative web-site management systems and different run-time management techniques. Section 3 formally defines the problem we consider in the paper. Section 4 describes several optimization techniques and evaluates their impact. Section 5 formally defines run-time policies, and Section 6 describes the compilation methodology. Finally, Section 7 describes the implementation of STRUDEL-R, and then we conclude with related work.

2 Declarative specification of web sites

We begin by describing the general architecture of declarative web-site management systems, as embodied in the STRUDEL-R system. We note that the key architectural aspects of the STRUDEL-R are common to other systems for declarative web-site management [2, 3, 20, 7]. STRUDEL-R is based on a logical representation of a web site, called a *site graph*, which is independent of its graphical presentation or of the underlying data management systems. The site graph models the pages in the web site, the links between them, and the data associated with each page. A

[1] STRUDEL-R is a derivative of the STRUDEL system [9] where the content is derived from a single relational database system, as opposed to multiple external semi-structured data sources.

site graph in STRUDEL-R is defined intensionally, via a *site schema*. Applying the site schema to a particular instance of the database results in a site graph. The site graph computed by the above procedure can be converted into a browsable web site by applying HTML templates to each of the nodes in the graph.

The STRUDEL-R system contains two components. The site graph generator applies the intentional definition of the site schema to the underlying data and produces (fragments of) the site graph. The HTML generator applies HTML templates to nodes in the site graph, resulting in browsable HTML pages. In the rest of this section we describe site graphs and site schemas. The details of the HTML templates [9] are not relevant to our discussion.

2.1 Site graphs

A site graph is a directed, rooted, labeled graph. There are two types of nodes in the site graph: internal nodes corresponding to web pages, and leaf nodes corresponding to data values. [2] An edge between two internal nodes, called a *ref arc*, models a hyperlink, or the nesting of page components; an edge from an internal node to a leaf, called a *data arc*, models data values to be displayed on the page. Every arc l in the site graph is labeled with a string *label*(l), and with a string *anchor*(l): *label*(l) is the name of the relationship between the two nodes (e.g., "Region"), while *anchor*(l) is the string shown on the HTML link corresponding to the arc (e.g., the name of the region "Europe").

In STRUDEL-R pages are classified into a small number of relatively homogeneous collections [9]: for example nodes corresponding to customers form a collection, while those corresponding to suppliers another. We refer to the collections of pages in a web site as *site collections*. Each internal node can be uniquely identified by a term of the form $F(a_1, \ldots, a_n)$, where F is the collection's name, and $a_1, \ldots, a_n$ are data items from the database: such an expression is called a Skolem term, and $n \geq 0$ is the collection's arity. We can always model a highly specialized node as a collection with one member, e.g., the root collection *Root* is of arity 0 and has a single member: *Root*().

2.2 Site schemas

A site schema is a directed, rooted, labeled graph G, whose nodes are partitioned into internal nodes and leaf nodes. There is one internal node for each site collection F, and that node is labeled by a Skolem term of the form $F(X_1, \ldots, X_n)$ ($F(\bar{X})$, in short), where F is a site collection name and $X_1, \ldots, X_n$ are variables. The root is labeled by a 0-arity Skolem function: in this paper it will always be *Root*(). Leaf nodes are labeled with single variables and correspond to data items. As before we classify edges into ref arcs and data arcs.

A ref arc between two internal nodes $F_1(\bar{X}_1)$ and $F_2(\bar{X}_2)$ in the site schema is labeled by a query specifying the conditions needed for the existence of an arc between instances of F_1 and F_2 in the site graph. Similarly a data arc between $F_1(\bar{X}_1)$ and Y has a query specifying the conditions needed for the existence of a corresponding arc in the site graph. In this paper we use the notation of conjunctive queries (corresponding to select-project-join queries in SQL) of the form:

$$q(\bar{X}) : -e_1(\bar{X}_1), \ldots, e_m(\bar{X}_m),$$

where $e_1, \ldots, e_m$ are relations in the database and $\bar{X}, \bar{X}_1, \ldots, \bar{X}_m$ are tuples of variables or constants. We denote all variables in q by $Vars(q)$, and call the variables in $\bar{X}$ *distinguished* variables. Thus, arcs in the site schema are labeled as follows.

- **Ref Arcs:** an arc from $F_1(\bar{X}_1)$ to $F_2(\bar{X}_2)$ is labeled by a triple: $(q, anchor, label)$, where: (a) q is a conjunctive query whose distinguished variables are $\bar{X}_1 \cup \bar{X}_2$, (b) *anchor* is either a string or one of the distinguished variables of q, and (c) *label* is a constant string associated with the arc.
- **Data arcs:** an arc from $F(\bar{X})$ to Y is labeled by a pair: $(q, label)$, where q and *label* have the same meaning as above and $\bar{X} \cup \{Y\}$ are distinguished variables of q.

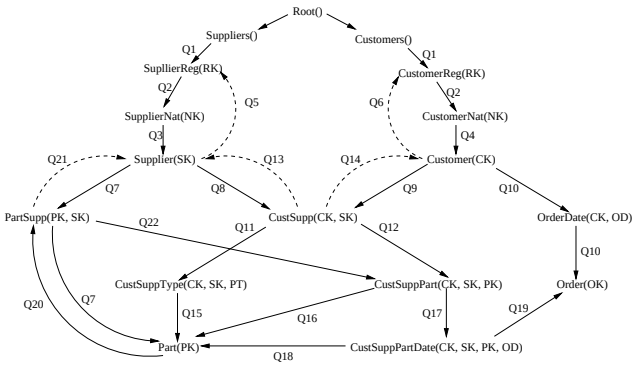

Figure 1: The site schema for the TPC/D example. For clarity, we omitted the data arcs, the anchors and the labels.

Example 2.1 We use the following example throughout the paper and in our experiments. Suppose we want to produce a browsable version of the data contained in the TPC/D benchmark. The database contains information about products, customers, and orders. A simplified version of the TPC/D schema is given below.

Part(partkey, name, brand, type, size)
Supplier(suppkey, name, address, nationkey, phone)
PartSupp(partkey, suppkey, availqty, supplycost, comment)
Customer(custkey, name, address, nationkey, phone)
Nation(nationkey, name, regionkey, comment)
Region(regionkey, name, comment)
Lineitem(orderkey, linenumber, partkey, suppkey, quantity)
Order(orderkey, custkey, orderstatus, totalprice, orderdate)

The site schema shown in Figure 1 provides the following organization of the data. The root node has two links to suppliers (Suppliers()) and customers (Customers()). Both suppliers and customers are grouped by geographical region (e.g., SupplierReg(RK)), and inside each region by nationality (e.g., SupplierNat(NK)). Suppliers and customers have further links to detailed information about orders. Specifically there is one page for each customer-supplier pair (CustSupp(CK,SK)) where the customer ordered from the supplier: that page can be accessed from both the supplier and customer pages. From here there are further links

[2] To simplify the exposition, our discussion does not include the formalisms needed to model forms in HTML pages, as well as the internal structure within a page. However, we note that extending site specifications to include the above features is relatively straightforward.

to pages detailing orders placed by that customer to that supplier. Of course, in designing the web site, we also add links back to facilitate navigation. The definitions of the queries in the site schema are given in Figure 2.

2.3 Semantics of site schemas

A site schema G and a database instance D define a unique site graph $G(D)$ as follows.

- **Create ref arcs:** let $l = (q, anchor, label)$ be an arc from $F_1(\bar{X}_1)$ to $F_2(\bar{X}_2)$. Let $q(D)$ be the result of evaluating q over the database D. For each tuple $\bar{a} \in q(D)$, we define $\bar{a}_1$ and $\bar{a}_2$ as restrictions of $\bar{a}$ to the variables $\bar{X}_1$ and $\bar{X}_2$, respectively. Then, $G(D)$ contains a link between $F_1(\bar{a}_1)$ and $F_2(\bar{a}_2)$, labeled *label* and whose anchor is the value of the variable *anchor* in the tuple $\bar{a}$. We note that if the nodes $F_1(\bar{a}_1)$ and $F_2(\bar{a}_2)$ were not in the site graph, then they are added as a side effect of inserting the arc.
- **Create data arcs:** let l be a data arc in G between the nodes $F(\bar{X})$ and Y, labeled by $(q, label)$. For each $\bar{a} \in q(D)$ we define $\bar{a}_1$ and a_2 as projections of $\bar{a}$ on $\bar{X}$ and Y, respectively. Then, $G(D)$ contains a link between $F(\bar{a}_1)$ and a_2, labeled *label*.
- **Root node:** the root node in $G(D)$ is $Root()$.
- **Eliminate unreachable nodes:** any node in the site graph that is not reachable from the root is removed.

2.4 Strategy for site graph evaluation

There are many strategies for computing the site graph. The semantics described above provide a natural method to compute the entire site graph in advance of browsing. We refer to it as the *static* evaluation strategy.

An alternative strategy is to expand the site graph *dynamically*, starting from the root and computing the nodes in the site graph only upon request. We recall that each web page corresponds to a node $F(\bar{a})$ is the site graph. Therefore, an HTTP request for a given page translates into a request for a node of the form $F(\bar{a})$. To construct that page we need to compute all the data appearing in the page as well as all the outgoing HTML links. Formally, this translates into the following procedure.

Given a site schema G and a database instance D, in order to produce the node $F(\bar{a})$, the dynamic algorithm proceeds as follows:

- **Create ref arcs:** let $l(q, anchor, label)$ be an arc from $F(\bar{X})$ to $F_1(\bar{X}_1)$. Let $q(D)$ be the result of evaluating $q \wedge (\bar{X} = \bar{a})$ over the database D. For each tuple $\bar{b} \in q(D)$, we define $\bar{b}_1$ to be the projection of $\bar{b}$ on $\bar{X}_1$. Then, $G(D)$ contains a link between $F(\bar{a})$ and $F_1(\bar{b}_1)$, labeled *label* and whose anchor is the value of the variable *anchor* in the tuple $\bar{b}$.
- **Create data arcs:** let $l = (q, label)$ be a data arc from $F(\bar{X})$ to Y. For each tuple $\bar{b} \in q(D)$ we define b_1 to be the projection of $\bar{b}$ on Y. Then, $G(D)$ contains a link between $F(\bar{a})$ and b_1, labeled *label*.

Starting at $Root()$ and applied repeatedly (e.g., in depth first order), this procedure eventually computes the entire site graph. It is important to note that this graph is provably isomorphic to the site graph given by the static computation, assuming that the database is not changing during the computation.

3 Problem definition

The static and the dynamic evaluation algorithms represent two extreme strategies, with obvious advantages and disadvantages. The goal of our work is to automatically find an optimal intermediate strategy for a given web site, that combines pre-computation, caching and dynamic evaluation of the requested data. The optimal strategy is expressed as a *run-time policy*, which specifies which data to precompute or cache and which actions to execute at each page request, depending on the history of the browsing.

In this section we set up a general framework for studying run-time policies and formally define the optimization problem we consider.

3.1 Inputs to the optimization problem

3.1.1 Statistics on browsing patterns

To evaluate a particular run-time policy, it is necessary to know the characteristics of the browsing patterns. Therefore, we assume that we have access to the following statistics:

- **Node probability distribution:** let $F_1, \ldots, F_n$ be the set of internal nodes in the site schema. We assume the availability of the probability distribution $(p_1, \ldots, p_n)$, where p_i is the probability that a request for a page on the site (from any user) will be for an instance of F_i.
- **Arc probability distribution**: for internal node F in the site schema, with the set of successors $F_1, \ldots, F_m$, we assume the availability of the probability distribution $(l_0, l_1, \ldots, l_m)$, where l_i is the probability that a user will request a page of type F_i after viewing a page of type F, and l_0 is the probability that a user does not follow one of F's children (i.e., either stops browsing or goes back to a predecessor page).
- **Value probability distribution**: for each internal node F in the site schema, let $(F(\bar{a}_1), \ldots, F(\bar{a}_s))$ be its instances in the site graph. We assume that we have the probability distribution $(r_1, \ldots, r_s)$, where r_j is the probability that a request for a page of F will be for $F(\bar{a}_j)$.
- **Context probability distribution**: since in our framework the actions to evaluate a specific node depend on the browsing history leading to that point, we assume that there exists an integer k, such that for every internal node F and and a path $P = F_1, \ldots, F_l$ in the site schema where $F_l = F$ and $l \leq k$, we can obtain the probability that, given a request for an instance of F, it was made after following the path P.

The statistics above can be obtained in several ways. One possibility is to analyze the web site log and another is for the web site administrator to estimate them based on knowledge of the application. It is important to emphasize that, since these statistics (except for the value probability distribution) concern the site schema, they are independent of database updates.

3.1.2 Application constraints

Clearly, the choice of an optimal run-time strategy depends on specific constraints of the given application. In our framework, we identify the following measures associated with a given web site:

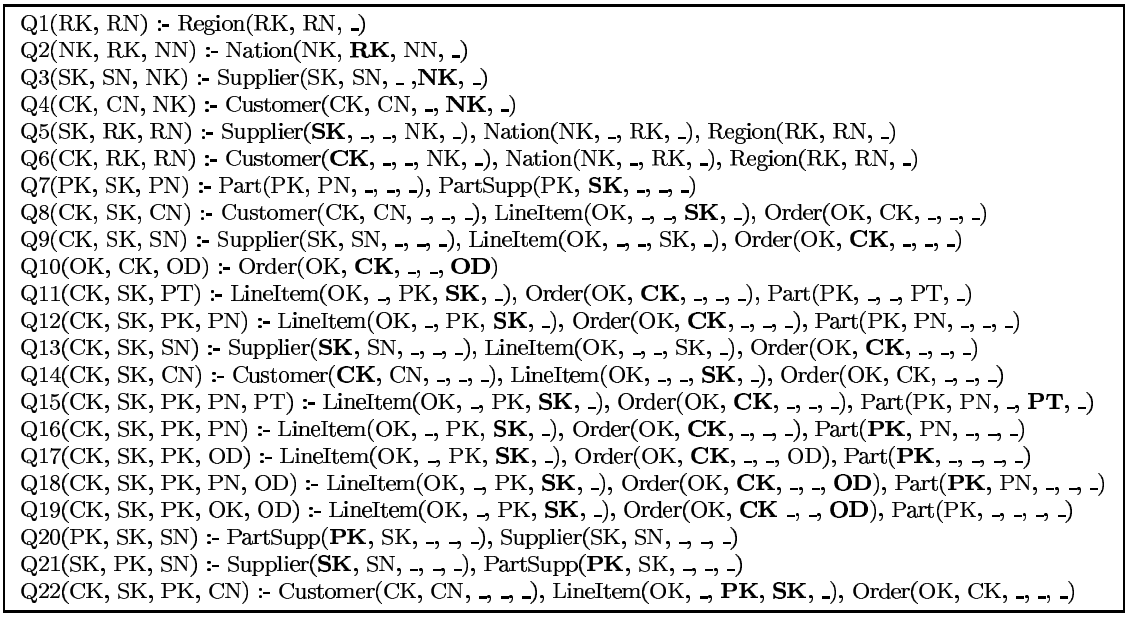

Figure 2: Queries labeling the arcs in the site schema in Figure 1. The attribute names in boldface are bound variables in the dynamic evaluation.

- *size(WS)*: the size of the (possibly) materialized HTML pages plus the size of the (possibly) precomputed or cached data;
- *age(WS)*: every data item I shown in the web site depends on a set of data items $dep(I)$ in the database. The age of a web site denotes the maximum difference between the timestamp of a data item I in the web site and the timestamp of a data item in $dep(I)$.
- *wait(WS)*: the maximum estimated cost of all the database operations needed to compute a web page.

We assume that a given web site has a set of given parameters (S, A, W) such that we have the following constraints: $(size(WS) < S, age(WS) < A, wait(WS) < W)$, specifying that we should not exceed space S, the maximum waiting time should be at most W, and the web site freshness should be at least A.

3.2 Cost model

Among the evaluation strategies that satisfy the above constraints, our goal is to find the strategy minimizing the waiting time for expanding an instance of a node in the site schema, weighted by the probability of requesting that node. Formally, we denote by $wait_{RP}(F(\bar{X}))$ the average time for executing the queries needed for expanding a node of type $F(\bar{X})$ in a run-time policy RP. Let $F_1, \ldots, F_n$ be the set of internal nodes in the site schema. The cost formula that we use to estimate the efficiency of a specific run-time policy RP for a web site is:

$$cost(RP) = \sum_{i=1}^{n} p_i \times wait_{RP}(F_i) \qquad (1)$$

3.3 Equivalence of run-time policies

Ideally, our optimization algorithm should choose among *equivalent* run-time policies, i.e., policies that produce identical site graphs. However, equivalence of site graphs is tricky to define when the underlying data is updated concurrently with the site graph expansion. In this work, we consider a weak equivalence condition, by imposing an age constraint of k time units on the site. In this case, we are assured that all the data associated with a given page is computed on snapshots within k time units from one another.

4 Optimization techniques for web-site management

In order to develop a meaningful formalism for specifying run-time policies, we first need to consider which optimizations such a formalism should capture. In this section we describe several techniques for optimizing the dynamic evaluation of web sites, and validate their utility. The first class of optimizations includes precomputation of materialized views and dynamic caching of data. The second class is more specialized for our context, and exploits the structure of the web site in order to reformulate the queries in the site definition and to determine useful caching policies.

We evaluate the impact of our optimizations on the STRUDEL-R system. Our experiments were performed on a web site derived from the TPC/D benchmark. The experiments were run on a TPC/D database at scale factor 1, resulting in a database of 1.84GB. We used the Oracle DBMS Version 7.3.2 and a dedicated Ultra Sparc I machine (143 MHz and 128MB of RAM), running SunOS Release 5.5. The indexes on the database were manually tuned for performance before applying our optimizations.

Our experiments measure the average time for the database operations needed to expand a node in the site graph. The numbers are generated as a result of running 100 independent browsing sequences of length at most 20. The browsing sequences are generated by choosing the next web page randomly using a uniform distribution over the emanating links. Note, however, that since our experi-

ments report speedups per node in the site schema, as opposed to the global utility of a run-time policy, the uniform distribution does not bias the results. The experiments report only the running times for the nodes affected by the proposed optimization and are all presented on a logarithmic scale.

4.1 Query simplification under preconditions

The first optimization we consider is a query rewriting technique that exploits the knowledge about the path used to reach a given node. When evaluating a parameterized query with a particular input, we can often simplify the query if we know which previous query produced the input. For example, assume the user requests the node $F_2(\bar{a}_2)$ after visiting $F_1(\bar{a}_1)$, and q_0 is the query on the corresponding arc between F_1 and F_2 in the site schema. According to the semantics, the tuple $(\bar{a}_1, \bar{a}_2)$ is in the result set of q_0. In order to expand $F_2(\bar{a}_2)$ we have to evaluate all the queries labeling the outgoing arcs from the node F_2 in the site schema, with the additional selection $\bar{X}_2 = \bar{a}_2$. Let q be one of those queries. In some cases the query $q \wedge (\bar{X}_2 = \bar{a}_2)$ can be simplified given that we know that the tuple $(\bar{a}_1, \bar{a}_2)$ is in the result set of q_0 (i.e., some conjuncts will be removed from the query). The following example illustrates this optimization, which we call *simplification under preconditions*.

Example 4.1 Consider a request for an instance of the node CustSuppPart(CK,SK,PK) in Figure 1. In order to expand this node we have to compute the following query. In the rest of the paper we note in bold the variables which are bound in the evaluation of the queries.

```
Q16(CK,SK,PK,PN) :- Order(OK,CK,_,_,_),
      Part(PK,PN,_,_,_), LineItem(OK,_,PK,SK,_)
```

We observe that one way we could have reached this node is from CustSupp(CK,SK) via the edge Q12. To be more precise, the values binding the variables CK, SK, PK should be in the answer set of the query:

```
Q12(CK,SK,PK,PN) :- Part(PK,PN,_,_,_),
      Order(OK,CK,_,_,_), LineItem(OK,_,PK,SK,_)
```

Based on this knowledge, it is possible to expand the instance of the CustSuppPart(CK,SK,PK) node by computing the following simpler query:

```
Q16'(CK,SK,PK,PN) :- Part(PK,PN,_,_,_)
```

Query simplification under preconditions is a form of query rewrite. Unlike traditional query rewriting techniques, this rewriting cannot be done manually by the person writing the queries for the site specification. For example, the user cannot manually replace Q16 in Figure 1 with Q16' for several reasons. First, this query is not safe for the static evaluation (since some variables in the head do not occur in the body). Second, we may not use this query even during dynamic evaluation if the time between page requests exceeds the age limit of the site. In that case we need to use the original query Q16. Third, the correctness of this rewrite depends on the user's browsing context. When there are multiple paths to a node in the site schema, we obtain different rewritings of the query depending on the path traversed.

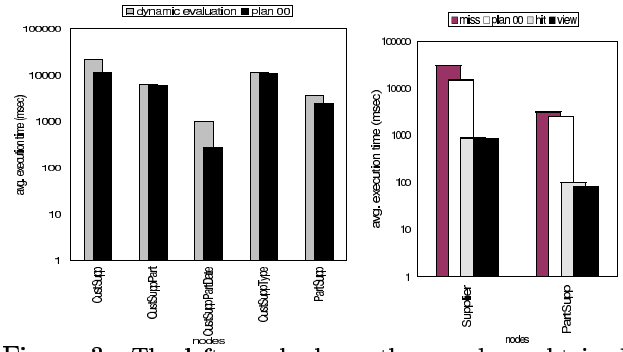

Figure 3: The left graph shows the speedups obtained from query simplification under preconditions. The right graph shows the running times of data caching (hits or misses), view materialization and the original queries.

When query simplification under preconditions modifies the query, it always reduces the running time. Figure 3 (left) shows the running times for the naive dynamic run-time policy versus the policy where all queries are simplified under preconditions (this plan is referred to as Plan 00 in the figures). As we can see, we obtain up to a 4 fold speedup in performance (for the node CustSuppPartDate). The figure shows all the nodes that benefited from query simplification under preconditions. In subsequent experiments, we always compare the additional optimizations to the plan obtained after applying query simplification under preconditions.

4.2 View materialization

Another way to speed up the web-site's performance is to precompute materialized views. The problem is to decide which set of views $\mathcal{V}$ to materialize in order to optimize the evaluation of the parametrized queries involved in the run-time of the web site. The problem of choosing a set of materialized views for a given query workload has received significant attention in the recent literature [24, 13, 12, 14, 6].

Two issues are important when deciding which views to materialize. First, it is essential to choose simultaneously the views *and* their respective indexes. Second, we need to consider views with outerjoins. In order to simultaneously optimize two queries q_1 and q_2 which have a common subquery q_3 (i.e., $q_1 = q_3 \wedge q_1'$ and $q_2 = q_3 \wedge q_2'$), it is attractive to materialize the outerjoin of the queries q_3, q_1' and q_2', i.e., materialize the expression $(q_3 \bowtie\!\sqsubset q_1') \bowtie\!\sqsubset q_2'$. In this case, the materialized result can be reused in both q_1 and q_2[3].

Example 4.2 As an example, assume we decide to materialize the following view with an index on the attribute SK:

```
V(CK,SK,PK,CN) :- Order(OK,CK,_,_,_),
      LineItem(OK,_,PK,SK,_), Customer(CK,CN,_,_,_)
```

The view V can be used in answering the queries Q14, Q8 and Q22. We measured considerable speedup rates for the

[3]We note that algorithms for rewriting queries using views are actually simpler when joins are replaced by outerjoins.

respective nodes: 32 for PartSupp nodes and 18 for Supplier nodes. However, the additional space needed for the view and the indexes is around 600M (27% of the size of the original database).

4.3 Data caching

View materialization decreases query response time, but comes at the expense of significant space overhead and high maintenance costs. An alternative strategy is to cache at run-time the result of parameterized queries executed so far [8, 16] and reuse the result if the same computation is requested again. In this way we can store less data and still obtain significant speedups in certain cases. Furthermore, it is often cheaper to periodically invalidate data in a cache than to pay the cost of view maintenance.

Therefore, our optimization algorithm stores the result of certain parameterized queries in particular relations called *cache functions*. Formally, a cache function f is a pair $(q_f, input(f))$, where q_f is a conjunctive query and the input variables $input(f)$ are a subset of the distinguished variables of q_f. The function encodes a mapping $\bar{a} \mapsto S$, where $\bar{a}$ is a binding for $input(f)$, and S is the set of tuples in the answer of q_f whose projection on $input(f)$ is $\bar{a}$.

In our system, cache functions are implemented as tables in the DBMS, with the same schema as their corresponding view q_f. At run-time, the corresponding table is initialized to be empty, and tuples from q_f are inserted whenever q_f is evaluated with new bindings for the input variables. We impose the following invariant on the contents of a cache function: *for any constant $\bar{a}$, either the cache function does not contain any tuples from q_f whose projections on $input(f)$ is $\bar{a}$, or it contains all such tuples.*

An important question is how functions are used at run-time. Assume that we have a cache function $f = (q_f, input(f))$ stored in a table T and a query q to be executed. Let q' be the equivalent reformulation of q that uses the view q_f and let q'' be obtained by replacing the occurrence of q_f in q' by T. The result of the queries q and q'' are identical if and only if all the needed values for computing q'' are cached in T. Therefore, before computing q' we first need to check whether the needed values are cached, and if not, we compute them before submitting q''. In order to guarantee that we can perform this check, we limit the ways in which functions can be used. Specifically, we require that for any occurrence of T in q'', the variables corresponding to input variables of the cache function are also bound variables in q (and consequently in q''). If at least one of the queries that have to be evaluated in order to expand a certain type of node can benefit from a cache function, we say that the cache is used in this node.

Finally, an important difference between materialized views and cache functions concerns their maintenance policy. Here we assume that views are periodically updated, while functions are not. Instead, expired or invalidated tuples are simply dropped from a function.

Example 4.3 As we saw in Example 4.2, the view V significantly improved performance but at the price of high space overhead. Suppose that instead of V we want to maintain a semantically equivalent cache function, updated while expanding an instance of a node Supplier(SK) and used in the node PartSupp(PK,SK). For simplicity, we mark in bold the input variables of the cache function.

F(SK,CK,PK,CN) :- Customer(CK,CN,_,_,_),
LineItem(OK,_,PK,**SK**,_), Order(OK,CK,_,_,_)

Assume we store the content of the cache function in the table T. The query Q22' bellow is equivalent to the query Q22, in the case where the binding for the variable SK given in Q22 is cached in T.

Q22'(CK,SK,PK,CN) :- T(**SK**,CK,**PK**,CN)

At run-time, when we compute Q22', we first check to see if the given value for SK occurs in the cached input values in T. If we have a hit, we return the set of associated values for the variables PK,CK,CN, from which we selected the ones corresponding to the desired value of PK. In the case of a miss, we first compute the function's body with the additional binding for SK, insert the result in the table T, and then proceed as before.

The utility of caching

Figure 3 (right) illustrates the utility of caching. For each node in the figure we compare the average cost of computing the node in four cases: (1) using a view for one of the outgoing arcs, (2) using an equivalent cache function and assuming a hit, (3) similar to (2), but assuming a miss, and (4) no views or functions. Clearly, the time for case (1) is the lowest because no checks are needed. Case (2) provides speedup factors of 25 and 17 compared to case (4). Most interestingly, the overhead of case (3) compared to case (4) is relatively low (a slowdown of 2%) due to the extra cache check and update.

Choosing which functions to cache and how much memory to allot to each cache is an optimization problem with two constraints: (1) the size of the cache should be sufficiently large so that the hit rate guarantees better performance than no caching at all, and (2) the size of the cache should be much less than the size of the materialized view as to make caching the more attractive option.

Given estimates on the costs of evaluating the query in each of the cases described above, we can use the value probability distribution to estimate the minimal cache size that will yield savings. Specifically, suppose we denote the cost of evaluating a query with no caching by $c_{regular}$, the cost of evaluating a query with a cache hit by $c_{hit}(\mathbf{f})$, and the cost of evaluating a query with a cache miss by $c_{miss}(\mathbf{f})$. In the first step, we use the following formula to derive the minimum value of the hit ratio $\tau(\mathbf{f})$ that will yield savings for the cache $\mathbf{f}$:

$$\tau(\mathbf{f}) \times c_{hit}(\mathbf{f}) + (1 - \tau(\mathbf{f})) \times c_{miss}(\mathbf{f}) < c_{regular} \qquad (2)$$

Given the minimum value of $\tau(\mathbf{f})$ and the value probability distribution (see Section 3), we can derive the minimum amount of memory M such that if we allot to the cache less than M we are guaranteed that we cannot achieve the required hit ratio. We assume that there exists a module in the system responsible for periodically removing items from the cache such that the hit ratio is maintained above the necessary threshold, the size of the cache does not exceed the limit and the age constraints are satisfied. The key for such a module is the use of the value probability distribution.

Up to this point we have only considered caching local to a particular node, i.e., a cache is updated in the same node in which it is used. In addition, the cache functions always

concerned one of the queries on the arcs in its entirety. In the next section we extend the idea of caching to exploit the structure of the web-site definition. In particular, (1) a cache function can be updated in one node in the site and the result can be used in multiple nodes, and (2) a cache can be defined as a subquery or a superquery of a query appearing on an arc.

4.4 Lookahead computation

The key idea behind lookahead computation is to modify the definition of cache functions such that a query computed in a node F can be used later in one or more of F's descendants in the site schema. We describe two types of lookahead computations: *conservative* and *optimistic* lookahead. Intuitively, conservative lookahead represents the minimal amount of work that would have been done anyway at F and can be reused as much as possible in subsequent requests. In contrast, optimistic lookahead introduces additional computation that would not be needed at F, but is deemed to be useful for future nodes.

Conservative lookahead

Consider the expansion of an instance of the node CustSuppPart(CK,SK,PK) in our example, where we need to compute the following query:

Q17(CK,SK,PK,OD) :- Order(OK,**CK**,_,_,OD),
LineItem(OK,_,**PK**,**SK**,_), Part(**PK**,_,_,_,_)

In a subsequent click of the same user, we might have to expand an instance of a node CustSuppPartDate(CK,SK,PK,OD), with the same bindings for the variables CK,SK,PK. In order to do this, we need to compute the query Q19:

Q19(CK,SK,PK,OK,**OD**) :- Order(OK,**CK**,_,_,**OD**),
LineItem(OK,_,**PK**,**SK**,_), Part(**PK**,_,_,_,_)

Assume we updated a cache function for Q17 with inputs CK, SK and PK. As we can see, much of the computation performed for the function for Q17 is also useful for Q19. However, if we simply cache the result of Q17, we cannot use it unchanged for Q19 because Q17 projected out the attribute OK. Conservative lookahead would define a function with the same subgoals (since the subgoals of Q17 and Q19 are identical) and whose head includes all the attributes needed for both Q17 and Q19.

More generally, consider two consecutive arcs in the site schema, $F_1(\bar{X}_1) \longrightarrow F_2(\bar{X}_2) \longrightarrow F_3(\bar{X}_3)$, where the arcs are labeled with the queries q and q', respectively. We want to define a function in the first node and use it in the second. We want to update a cache while expanding the node F_1 and use it while expanding F_2. The cache f will have as body the intersection: $body(f) = body(q) \cap body(q')$. The distinguished variables of f include (1) all variables in f which are distinguished in q or q', and (2) all variables in f which also occur in $q - f$ or in $q' - f$, where the difference denotes the set difference of the subgoals of the respective queries. $input(f)$ are defined to be those variables of f that occur in $\bar{X}_1$. The cache f will be updated while expanding $F_1(\bar{X}_1)$. It can be used at node $F_2(\bar{X}_2)$ only if $input(f) \subseteq \bar{X}_2$; otherwise we cannot use it (because of the constraint we imposed on cache usability in Section 4.3).

The previous technique can be extended to a set of arcs that form a tree in the site schema. In this way, a cache updated at the root of the tree can be used in its descendants. By applying this technique to the set of nodes CustSupp, CustSuppType, CustSuppPart and CustSuppDate, we obtain the following cache function, updated in the node CustSupp and used in all the others.

F(CK,SK,PK,OD,OK,PN,PT) :- Order(OK,**CK**,_,_,OD),
LineItem(OK,_,PK,**SK**,_), Part(PK,PN,_,PT,_)

Optimistic lookahead

Optimistic lookahead performs while expanding a certain node an additional computation that may be usable for expanding later nodes. For example, consider the expansion of an instance of a node Customer(CK), where we need to compute the following query:

Q9(CK,SK,SN) :- Supplier(SK,SN,_,_,_), Order(OK,**CK**,_,_,OD),
LineItem(OK,_,PK,SK,_)

In a subsequent request, we might need to expand an instance of the node CustSupp(CK,SK). In order to do so, we need to compute the query:

Q12(CK,SK,PK,PN) :- Part(PK,PN,_,_,_), Order(OK,**CK**,_,_,_),
LineItem(OK,_,PK,**SK**,_)

Suppose we want to use a cache function for Q9 that also performs all the necessary computation for query Q12. To do this, we define a function that includes the common subgoals of Q9 and Q12, but also performs an outerjoin with the other subgoals of Q9 and Q12 that are not in the intersection. Specifically, we would define a cache function as follows:

F(CK,SK,PK,PN,SN) :- ((Order(OK,**CK**,_,_,_) ⋈
LineItem(OK,_,PK,SK,_)) ⟕ Part(PK,PN,_,_,_)) ⟕
Supplier(SK,SN,_,_,_)

This cache is defined in the node Customer but can also be used in the rewriting of one of the queries of the node CustSupp. Note that in node Customer we do a join with Part that is not necessary there, but that will drastically reduce the cost of computing CustSupp.

More generally, consider two consecutive arcs in the site schema, $F_1(\bar{X}_1) \longrightarrow F_2(\bar{X}_2) \longrightarrow F_3(\bar{X}_3)$, where the arcs are labeled with the queries q and q', respectively. We want to update a cache in the first node that also performs the computation necessary for the second node. Let q_0 be the intersection of the bodies of q and q'. The cache f will have as body the expression $(q_0 \text{⟕} (q - q_0)) \text{⟕} (q' - q_0)$, where the difference denotes the set difference of the subgoals of the respective queries.. The distinguished variables are the union of the distinguished variables of q and q'. $input(f)$ are defined to be those variables of f that occur in $\bar{X}_1$. The cache f will be updated at node $F_1(\bar{X}_1)$. It can be used at node $F_2(\bar{X}_2)$ only if $input(f) \subseteq \bar{X}_2$: otherwise we cannot use it.

As with conservative lookahead, we can generalize optimistic lookahead to trees in the site schema. For example, the cache function shown above can also be used in the evaluation of the queries needed for the nodes CustSuppType, CustSuppPartDate, CustSuppPart and CustSupp.

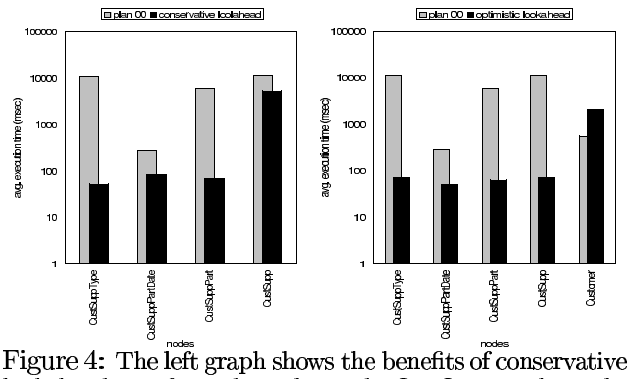

Figure 4: The left graph shows the benefits of conservative lookaheads performed in the node CustSupp. The right graph shows the benefits of optimistic lookahead performed at the node Customer.

The utility of lookaheads

Figure 4 shows the experimental results concerning lookahead computations. The first graph shows the speedups obtained by using conservative lookahead. We observe that the cost of computing CustSupp was not affected, while the speedups obtained for its descendants ranged from factors of 3 to 210. The second graph shows the results for optimistic lookahead. We observe that the node Customer that has a more expensive computation was slowed down by a factor of 3.7, while the speedups for its descendants varied from 5 to 161.

We end this section by noting that lookahead computations benefit from specific patterns in the structure of web sites. However, these patterns occur quite frequently in web sites because they correspond to a natural hierarchical organization of data.

5 Run-time management

After the discussion of possible optimizations in the previous section, we are now in a position to formally define run-time policies that encompass the different optimizations that we presented so far. To define run-time policies we first describe run-time schemas, which are the sets of views and caches over which the run-time policies are expressed.

5.1 Run-time schema

The run-time schema consists of a set of precomputed views $\mathcal{V}$ and a set of dynamically maintained functions $\mathcal{F}$, formally defined as follows.

- $\mathcal{V}$ is a set of view specifications, where a view specification is formally defined as a quadruple (N_V, Q_V, I, age_V), where N_V is the name of the database table storing the view, Q_V is a select-project-join-outerjoin expression defining the view, I is a set of indices on the view N_V and age_V is the maximum allowed difference between a data item in the view and the raw data.
- $\mathcal{F}$ is a set of cache function specifications, where a function specification is formally defined as a quintuple $(N_F, Q_F, Input_F, max\_size_F, min\_hit_F, age_F)$, where N_F is the name of the database table storing the function, Q_F is a select-project-join-outerjoin expression defining the function, $Input_F$ is a set of distinguished variables of Q_F which are inputs to the function, max_size_F is the maximum allowed size for the dynamically maintained table, age_F is the maximum allowed difference between a data item in the function and the raw data, and min_hit_F is the minimum hit ratio acceptable for the function.

5.2 Run-time policy

The run-time policy tells the system what to compute at every page request, i.e., how to use the run-time schema and data in order to compute the requested HTML page. There are several points to note about run-time policies. First, the action that the policy specifies does not depend only on the origin and destination of the hyperlink being followed, but may also take into consideration the path (or parts thereof) used to get to the origin. Hence, the actions in a run-time policy are parameterized by *contexts*, which we define below. The second point to note is that there are two types of possible actions: *query actions*, which specify how to obtain the data needed, and *update actions*, specifying when to update the dynamically maintained functions, and with which inputs.

Contexts are used to formalize the dependence of actions in the run-time policy on the previously visited nodes in the site graph. Formally, a path $[F_1, \ldots, F_n]$ in the site schema G is called the *context* of a request for a node $F(\bar{a})$ if $F = F_n$ and the previously requested nodes of the same user were of the form $[F_1(\bar{a}_1), \ldots, F_{n-1}(\bar{a}_{n-1})]$. A run-time policy fixes the maximum length of the contexts that are maintained at run-time. An action parameterized by a specific context is called a *rule*.

A run-time policy P_G for a site schema G is a directed graph, isomorphic to the graph of G, and whose nodes are labeled with the same Skolem terms as in G. In addition, nodes are labeled with set of update rules, and the arcs are labeled with sets of query rules. An *update rule* associated with a node F is a triple (H, f, ψ), where H is a possible context for the node F, f is the name of a given function and $\psi : Input_f \rightarrow \bar{X}$ is a mapping from the input variables of the function to the set of variable $\bar{X}$, which describes how to obtain input values for the function from the current binding of $\bar{X}$. A *query rule* associated with an arc $F_1(\bar{X}_1) \rightarrow F_2(\bar{X}_2)$ is a pair (H, q), where H is a possible context for the node F_1, and q is the parametrized query to be executed in order to obtain all the outgoing links of a node of type F_1 to nodes of the type F_2.

In order to facilitate inspection and manual construction of run-time policies in our work, we developed a language for describing run-time schemas and policies. We illustrate run-time policies with this language [11] in Figure 5.

5.3 Run-time algorithm

The execution engine of the web-site management system interprets the run-time policy. Execution proceeds in a similar fashion to the dynamic approach, with a few notable differences. Suppose the user requests an instance of the node $F(\bar{X})$ with a binding $\bar{X} = \bar{a}$ and a context $[F_1, \ldots, F_k]$ where $F_k = F$ (k is a constant depending on the run-time policy). We proceed in two steps:

1. Execute any update action (H, f, ψ) associated with the node F whose history H is a suffix of $[F_1, \ldots, F_k]$.

```
/* Run-time schema definition */
define view V as
   SELECT o.custkey l.suppkey, l.partkey,
   FROM LineItem l, Order o
   WHERE l.orderkey=o.orderkey
 max age = 2 hours
 define index on suppkey
define cache function T as
   SELECT o.custkey,l.suppkey,p.partkey,p.name,s.name
   FROM LineItem l, Order o, Supplier s
   WHERE l.orderkey=o.orderkey and s.suppkey=l.suppkey
 input custkey
 max size = 1M
 min hit ratio=0.3
 max age = 20min
/* Run-time policy for the node CustSupp */
 Node CustSupp(CK, SK)
   /* check and (eventually) update the cache */
   if context [Customer,CustSupp]
      update T with custkey → CK
 Link to CustPartType(CK,SK,PT)
   if context [Customer,CustSupp] compute
      SELECT f.custkey, f.suppkey, p.name, p.type
      FROM T f, Part p
      WHERE f.custkey=CK and f.suppkey=SK and
            f.partkey=p.partkey
   else compute
      SELECT v.custkey, v.suppkey, p.name, p.type
      FROM V v, Part p
      WHERE v.custkey=CK and v.suppkey=SK and
            v.partkey=p.partkey
```

Figure 5: Fragments of a run-time schema and policy for our running example.

Specifically, if $\psi(\bar{a})$ is not in the cache of f, we compute q_f with bindings $\bar{a} \circ \psi$ and add the result to the cache.

2. For each arc l outgoing from F we select the rule (H, q) with the most specific context matching $[F_1, \ldots, F_k]$ (i.e., for which there is no longer suffix of $[F_1, \ldots, F_k]$ matching another rule). We evaluate the query q with the binding $\bar{a}$.

5.4 Correctness of a run-time policy

Clearly, we need to impose constraints on run-time policies in order for them to be faithful to the declarative site definition. As we discussed earlier, updates to the database complicate the notion of correctness of a web site. We aim to formalize a minimal notion of correctness here: *given that the materialized views and cache functions are taken from the same snapshot of the database, then applying the dynamic evaluation strategy to that snapshot will produce the same result as invoking the run-time policy.*

The conditions are the following. Consider a link in the site definition $F_1(\bar{X}_1) \rightarrow F_2(\bar{X}_2)$ labeled with a query q. Suppose the corresponding link in the run-time policy is labeled by the pairs $(h_1, q_1), \ldots, (h_n, q_n)$, where the h_i's are contexts. The following conditions have to be satisfied:

- For each i, $1 \leq i \leq n$, q_i is an equivalent rewriting of q using the views and the functions under the preconditions implied by h_i (note that in this definition functions are used as view definitions).
- If one of the q_i's uses a cache function f, then the node F in the run-time policy includes a update action for f. The update action in the node F does not necessarily imply that the appropriate values are computed at F. Indeed, they may be computed elsewhere in the site (e.g., using lookahead computations), but the check is still necessary.
- For each possible context H for a request for a node of type F_1 and for each outgoing arc from the node F_1, there it exists a rule (h_i, q_i) labeling this arc such that h_i is a suffix of the context H.

Finally, it should be noted that given the probability distributions on contexts (see Section 3) and estimates on the cost of evaluating SQL queries, it is possible to compute the average waiting time for a request for an instance of a node F in the site schema for a given run-time policy. Hence, we can now compute the global cost of a run-time policy according to Formula 1 in Section 3.

6 Compiling site definitions

The ultimate goal of our work is to automatically compile a declarative site definition into an efficient run-time policy. We have shown that various optimizations can significantly improve the behavior of a web site. In section 5 we showed how to formalize the compilation problem as a search in a space of run-time policies. An important observation is that, in order to obtain the optimal run-time policy, it suffices to consider a finite number of policies.[4]

Given the number of parameters involved and the size of the resulting search problem, finding a compilation algorithm that is both efficient and produces high quality run-time policies is a problem in its own right. We now describe a set of heuristics to partition the search problem into manageable steps that are each relatively well understood. The steps that we describe are inspired by the results of our experiments. In our experiments, applying these heuristics provided significant improvement over the naive dynamic evaluation approach. Hence, we argue that our steps (which can be embodied by a collection of algorithms) provide a proof of the viability of automatic compiling of web-site specifications.

The steps are the following:

1. Apply query simplification under preconditions to all the nodes in the site schema.
2. Detect the set of *sensitive* nodes in the site schema: (1) the nodes whose average cost is above the acceptable limit on waiting time, and (2) the nodes with relatively high cost and probability of access. Let $q_1, \ldots, q_n$ be the parameterized queries on the arcs outgoing from the chosen nodes.
3. Apply a view materialization selection algorithm to $q_1, \ldots, q_n$, with the size and freshness constraints imposed by the web site. This step results in a set of views to materialize. In this step we can apply an exhaustive transformational algorithm similar to the one described in [24].
4. If a view V was a good candidate for improving performance in the previous step but was not chosen because of space or freshness constraints, consider including in the

[4]The crux of the claim is that it suffices to consider only a finite number of run-time schemas because there are only a finite number of views or functions that can be maintained and still be useful in a run-time policy.

run-time schema functions of the form (V, Inp), where Inp is a subset of the arguments of V.

5. For each such function which is usable for expanding a node F, consider applying conservative and optimistic lookahead optimizations, for all the subtrees rooted at F. The decision on which subtrees to consider should take into account the probability of visiting the descendants, given that the user visited F.

Figure 6 shows the results of applying these steps in two scenarios. In the upper figure we alloted enough space for the web site to be able to materialize a sizeable view (we allowed an additional 1GB to the original size of the database). In the lower figure we only alloted an additional 10MB. In the first case the run-time schema included the materialized join between Order and LineItem with 4 indexed columns. The view is used in almost all the nodes, and as a result, all the queries in the site ran in less than 8 seconds, and all but three in less than 400 milliseconds. In the second case the run-time policy includes a conservative lookahead in the node Supplier (which benefits the node PartSupp), and an optimistic lookahead computation in the node Customer which benefits the nodes CustSupp, CustSuppType, CustSuppPart and CustSuppDate. The running times of all the other nodes are comparable with that of the previous run-time policy. This example highlights the savings obtained purely by exploiting the structure of the web site, with very little memory overhead.

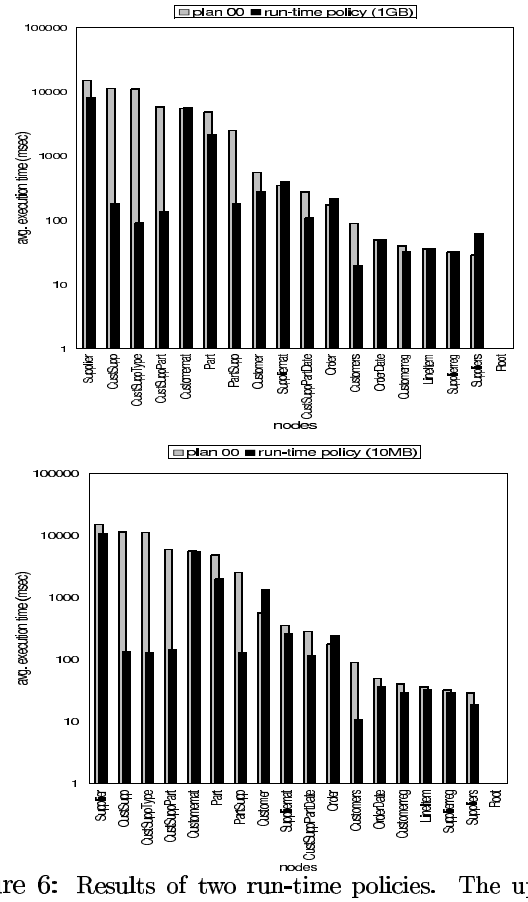

Figure 6: Results of two run-time policies. The upper graph shows a run-time policy in which 1GB of additional memory were provided, and the lower graph shows a policy when only 10MB was provided.

7 Implementation

The STRUDEL-R system[11] is implemented and fully operational, though the compiler from declarative specifications to run-time policies is relatively simple. The queries in the site definitions are given in SQL, and are allowed to contain selections, projections, joins, and outerjoins. Run-time policies are expressed in the language described in the previous section [11]. We note that it is also possible for a web site administrator to directly specify a particular desired run-time policy, bypassing STRUDEL-R's compiler.

A browsing session starts with a simple request for a root of the web site, which is precomputed. In order to employ our run-time policies, when an HTML page is served to the browser, the outgoing links (within the same site) are implemented as calls to a CGI-bin script. The script take as input the node in the site schema, the bindings for the variables associated with the node and the browsing context. It first calls the HTML generator, which in turn calls STRUDEL-R's execution engine with the same parameters. The execution engine follows the specification of the run-time policy. In doing so, certain functions can be updated. Finally, the result (the data contained in the page and information about the outgoing links) is sent back to the HTML generator which delivers the final page. The request, as well as all the statistics associated with it (utility of caches, response time, cardinality of resulting data, etc) are recorded in the web site trace.

In order to perform our experiments we also implemented a browser simulator. The input to this module is a set of probability distributions, as described in section 3. The simulator bypasses the HTML generator and calls the execution engine directly. Given a node in the site graph, the simulator randomly chooses the next node to request, according to the given probability distribution.

The system is implemented in Java, and all the database connections are done through Oracle's JDBC driver.

8 Conclusions and related work

Commercial products for constructing web sites from large databases and recent research prototypes are clearly moving in the direction of declarative specification of the structure and content of web sites. A critical issue that immediately arises is when to compute parts of the site. Currently, web site designers manually optimize site design in order to achieve reasonable performance, and this is a very labor intensive activity.

This paper described several techniques for optimizing the run-time behavior of web sites, and a framework in which declarative site specifications can be automatically compiled into run-time policies which incorporate these optimizations. Broadly speaking, many optimizations are easy to achieve if we have unlimited space. However, we have shown that, even with limited additional space, we can obtain order-of-magnitude speedups by exploiting the structure of the web site. We described a heuristic based algorithm for compiling a declarative web site definition into a run-time policy, which already yielded much better performance in our experiments. The problem of developing compilation algorithms that are both efficient and produce high-quality run-time policies clearly deserves significant

further research. Finally, another important note about our framework and implementation is that they were purposely designed to be built on top of an existing database system and did not require modifying any of its internals. In fact, our prototype can be deployed on top of any JDBC compliant database.

To begin our discussion of related work, several other systems have considered web-site management based on declarative representations [7, 2, 4, 20, 3, 25] but none considered the problem of run-time management of the site. The work of [25] considers the problem of decomposing a site specification to produce an entire tree of HTML pages into smaller chunks which are dynamically invoked when pages are requested. This decomposition can also result in our version of lookahead computation, though their decomposition is at the level of HTML pages and not the underlying data. Furthermore, they do not perform their decomposition w.r.t. a cost function.

A large body of work is concerned with caching web documents (e.g., [5]). The work in [19] extends the idea to prefetching of pages based on statistics on web site browsing patterns. However, this work considers caches at the level of HTML pages, as opposed to the underlying content. The performance improvements and the added flexibility achieved in our work were obtained by analyzing the database queries that produce the content of HTML pages.

In database systems, caching the result of parameterized computations has also been considered in several contexts such as data integration [1], nested correlated queries (implemented in commercial databases), caching for expensive methods [16, 15]. Our work takes the idea of caching further into the context of web-site management: our decisions of what to cache are based on cost estimates, and we do not necessarily cache exactly the computation specified by the parameterized input, but possibly only parts of it or larger computations. In addition, our caching decisions are based on the structure of the web site.

As stated early on, there has been a significant amount of work that tries to optimize workloads of queries on DBMS. This work took the form of selecting views to materialize (and their indexes) (e.g., [24, 13, 12, 14, 6]), multiple query optimization [23] and index selection. All these techniques are of course applicable to our context, since a dynamically generated web site can be viewed as a workload of parameterized queries. However, in our context we can perform additional optimizations because of the known structure of the web site. Also, our application is different in that it has new age and time limit constraints, and because queries in the workload are considered in succession, not in parallel.

Finally, a related body of work uses invariants for query execution [22] (in the context of nested correlated queries) and [17] in the context of optimizing recursive trigger calls. In the latter work, the authors compile the code of the triggers depending on the context of the calls, which are similar to our simplification under preconditions.

Acknowledgments

The authors are thankful to Zack Ives, Ioana Manolescu, Rachel Pottinger, Eric Simon, Ken Ross and Alain Pirotte for many insightful comments on this paper.

References

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

[2] G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, databases and Webs. In *ICDE*, 1998.

[3] P. Atzeni, G. Mecca, and P. Merialdo. To weave the Web. In *VLDB*, 1997.

[4] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive Web sites. In *EDBT*, 1998.

[5] M.-L. S. B. Chidlovskii, C. Roncancio. Semantic cache mechanism for heterogeneous Web querying. In *WWW8*, 1999.

[6] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, 1997.

[7] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion. In *SIGMOD*, 1998.

[8] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[9] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a Web-site management system. In *SIGMOD*, 1998.

[10] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the World Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[11] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Run-time management of data intensive Web-sites. INRIA Technical report RR-3684, 1999.

[12] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, 1997.

[13] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *ICDE*, 1997.

[14] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, 1999.

[15] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *SIGMOD*, 1996.

[16] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases: Design, Realization, and Evaluation. *TKDE*, 6(4):587–608, 1994.

[17] F. Llirbat, F. Fabret, and E. Simon. Eliminating costly redundant computations from SQL trigger executions. In *SIGMOD*, 1997.

[18] T. Nguyen and V. Srinivasan. Accessing relational databases from the World Wide Web. In *SIGMOD*, 1996.

[19] T. Palpanas and A. Mendelzon. Web prefetching using partial match prediction. Technical report CSRG-376. Departement of Computer Science, University of Toronto, 1998.

[20] P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable Web applications. In *EDBT*, 1998.

[21] B. Proll, W. Retschitzegger, H. Sighart, and H. Starck. Ready for prime time - pre-generation of Web pages in tiscover. In *WebDB Workshop, in conj. with SIGMOD*, 1999.

[22] J. Rao and K. A. Ross. Reusing invariants: A new strategy for correlated queries. In *SIGMOD*, 1998.

[23] T. K. Sellis. Multiple-query optimization. *TODS*, 1988.

[24] D. Theodoratos and T. Sellis. Data warehouse design. In *VLDB*, 1997.

[25] M. Toyama and T. Nagafuji. Dynamic and structured presentation of database contents on the Web. In *EDBT*, 1998.