Run-time Management of Data Intensive Web-sites

Daniela Florescu, Alon Levy, Dan Suciu, Khaled Yagoub

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Run-time Management of Data Intensive Web-sites

Daniela Florescu - Alon Levy - Dan Suciu - Khaled Yagoub

No 3684

March 1999

THÈME 3

ISSN 0249-6399

INRIA
ROCQUENCOURT

Run-time Management of Data Intensive Web-sites

Daniela Florescu - Alon Levy - Dan Suciu - Khaled Yagoub

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Rodin

Rapport de recherche n ° 3684 — March 1999 — 41 pages

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 structure and content of sites. 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. In one extreme approach, the site is precomputed in advance, while in the other extreme, the queries necessary to construct a given page are computed on demand. Both approaches have their obvious drawbacks: large space and maintenance overhead in the first approach, and poor run-time performance and unnecessary repeated computations in the second.

In this paper 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 derived automatically. 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 of the web site definition. 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 our novel optimizations.

Key-words: web site, declarative site specification, optimization, materialized views, caching.

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Gestion de Sites Web à Usage Intensif de Données

Résumé : Un nombre considérable et grandissant de sites web publient des données provenant de bases de données relationnelles. Récemment, plusieurs produits commerciaux et prototypes de recherche s'orientent vers une spécification déclarative de la structure et du contenu des sites. Plus particulièrement, un site web est spécifié en utilisant une collection de requêtes décrivant ses nœuds (correspondant aux pages HTML et à leur contenu) et ses arcs (correspondant aux liens hypertextes entre les pages). Une des plus importantes questions à considérer concerne le moment de la construction des pages HTML constituant le site. Dans une première approche extrême, l'ensemble des pages HTML est construit à l'avance, avant même que l'utilisateur ne commence à naviguer. Dans l'autre approche extrême, les pages sont construites sur demande et les requêtes nécessaires à la construction d'une page donnée sont donc évaluées dynamiquement. Les deux approches souffrent de problèmes majeurs : consommation d'espace et coût de maintenance très élevé pour la première et problèmes de performances et calculs répétitifs non nécessaires pour la deuxième.

Dans ce papier nous considérons le problème d'optimisation automatique des systèmes de gestion de sites web spécifiés d'une façon déclarative. Dans notre approche, un plan d'évaluation optimal est généré automatiquement à partir d'une spécification déclarative et de l'ensemble des contraintes d'utilisation du site spécifique à l'application. Le plan d'évaluation généré décrit l'algorithme qu'il faut appliquer lorsqu'un utilisateur demande une nouvelle page. Nous décrivons plusieurs types d'optimisations qui exploitent la structure du site afin d'améliorer les performances. Ensuite, nous évaluons expérimentalement l'impact de ces optimisations sur un site construit à partir de la base de données TPC/D. Finalement, nous décrivons un algorithme à base d'heuristiques qui compile une spécification déclarative et génère un plan d'évaluation incorporant les nouvelles optimisations que nous considérons.

Mots-clé : site web, spécification déclarative, optimisation, vues matérialisées, bufferisation.

1 Introduction

The World Wide Web (WWW) has been proven to be an excellent medium for businesses to disseminate information, both internally and externally. As a result, the ability to populate web sites with content derived from large databases has become 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 [10, 3, 8, 2, 21]. Two main features underly this paradigm. First, a declarative specification is based on a *logical* model of the web site, as opposed to direct manipulation of HTML files. The logical model 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.

As an example of this paradigm, which we consider in this paper, we look at sites whose 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. Rather than defining this graph extensionally, one node (page) at a time, we assume that we have an *intensional* definition, called the *site schema*, which logically defines sets of nodes (pages) in the graph. Hence, the site schema includes a set of database queries that define which pages exist in the site, which data is associated with every page, and which links exist between the pages. In this paradigm, the web site can be regarded as hyperlink view defined over a relational database.

A critical issue that arises when sites' contents are populated from large databases is *when* to compute the pages in the site 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 (i.e. where the values of the input variables 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 can appear

in multiple 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. While this approach has the advantage of always presenting fresh data, it also has significant disadvantages. First, some queries may be too expensive to evaluate at run-time which is unacceptable in the interactive nature of web access. Second, 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, which is the focus of this paper, is that successive queries issued while browsing a site share much of their computation (e.g., they either share many of the subgoals in their body, or they have identical subgoal and differ only in the distinguished variables).

The simple case of 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-grained, since a change to any of the data elements on a page would force the recomputation of the entire page. Clearly, exploit the optimization opportunities raised by the second observation above a deeper semantic analysis of the site definition is required.

Since a first priority of web site managers today is to ensure reasonable response time, they end up hardwiring optimizations into the design of their sites, and doing so 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 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*, which 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. 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; note however that a run-time policy still contains declarative queries that are optimized in the traditional fashion. As in the latter context, we would like to automatically compile the declarative specification into

an "optimal" run-time policy which is "equivalent" to the declarative specification, using a global cost model and statistics on the database and browsing patterns.

One obvious solution 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 proposed in the literature to optimize a given workload. Such techniques have been considered in various contexts, such as view materialization [14, 24, 11, 13, 12, 6], index selection [7], function caching [17, 15, 9], multiple query optimization [23] and reusing query invariants [18, 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 possible *future* queries that may be executed, and their respective probabilities. In this paper we show that exploiting this structure leads to significant savings over and above the application of the known techniques mentioned above.

In summary, this paper makes the following contributions.

1. 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 is chosen such that it can encompass traditional optimizations as well as novel ones specific to our context.

2. We describe several optimization techniques for speeding up the run-time behavior of web sites. These methods include: (1) precomputing a set of views, (2) simplification of queries based on known preconditions, (3) memoizing the results of certain computations for later use, (4) performing lookahead computation, i.e., computing more data than is immediately needed, but which may be useful in subsequent queries, and (5) using the browsing context of a user in order to more efficiently compute the data needed for a certain web page. 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.

3. Based on our experiments, we describe a set of guidelines for constructing an algorithm for compiling declarative specifications into run-time policies. Applying these guidelines in our experimental setting produced high quality run-time policies. We

remark that the problem of f finding a compilation algorithm that both is efficient and produces high quality run-time policies is a challenge in its own right, but prove the viability of automatic compilation of web site specifications.

4. Finally, we describe the implementation of STRUDEL-R, which embodies the ideas described in the paper and exposes several implementation choices that have significant impact on the web site's performance.

A few remarks are in order before we proceed. First, our solution is enabled by the declarative specification of a site's content and structure. Consider current practice, in which CGI-bin scripts compute each page in a site on demand. 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. Second, we note that even though our work is described in the context of STRUDEL-R web site management system, our solutions can be applied in a broader context. For example, one can extract a site specification from a collection of HTML templates with embedded SQL queries. Finally, we focus on the issues concerning the optimization of the run-time behavior of web sites, and do not discuss other issues that are important for web site management, such as integration of data from multiple sources and management of semi-structured data [10].

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.[1] We note that the key architectural aspects of the STRUDEL-R (see Figure 8 for the architecture of the system) are common to other systems for declarative web site management [2, 3, 21, 8]. STRUDEL-R is based on a *logical* representation of a web site, called a *site graph*, which is independent of its graphical

[1] STRUDEL-R is a derivative of the STRUDEL system [10] where the content is derived from a single relational database system, as opposed to multiple external semi-structured data sources.

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 site graph in STRUDEL-R is defined intensionally, via a *site schema*, rather than extensionally, one page at a time. Applying the site schema to a particular instance of the database results in (fragments of) a site graph.

Finally to create a web site, the system contains a HTML generator which applies HTML templates to nodes in the site graph, resulting in browsable HTML pages. In the rest of this section we describe site graphs, site schemas. The details of the HTML templates (see [10]) are not relevant to our discussion. We note that some HTML template languages also enable including behavioral characteristics (e.g., using Java scripts). In this paper we are only concerned with the underlying data processing computations.

2.1 Site graphs

A site graph is a directed labeled graph with some nodes labeled as roots. There are two types of nodes in the site graph: internal nodes corresponding to web pages or their subcomponents, and leaf nodes corresponding to data values. Links between pages are modeled as arcs between the internal nodes representing them in the graph; we will call these *ref arcs*. The site graph also associates with every web page the data that is displayed on the page. This association is represented by *data arcs* from internal nodes to leaves. With every arc l in the site graph we associate a string valued attribute, $attribute(l)$, and a string valued anchor, $anchor(l)$. The attribute of a link is the name of the relationship between the two nodes (e.g. "client"), while the anchor is the string shown on the HTML link corresponding to the arc (e.g., the name of the client "Joe Smith"). Note that in order to apply the HTML template we must compute all the attributes and anchors emanating from a given page.

Declarative modeling of web sites is especially effective when the pages in the site can be classified into a small number of relatively homogeneous collections [10] (e.g., nodes corresponding to pages of customers or suppliers). We refer to the collections of pages in a web site as *site collections*. Note that we can always model a highly specialized node (e.g., the root) as a collection with one member. Nodes in the site graph (and hence in the web-site) can be uniquely identified by their collection and a set of items from the underlying data. Formally, we associate a unique identifier with each internal node of the form $C(X_1, \ldots, X_n)$, where C is a site collection name and $X_1, \ldots, X_n$ are a list of data items. Note that collections have a fixed arity.

2.2 Site schemas

A site schema G is a labeled directed graph with a unique root. The graph G contains an internal node for every site collection. The node for a site collection F of arity n is labeled by an atom of the form $F(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are variables. The leaves of the site schema are labeled with single variables and correspond to data items.

An arc between a pair of 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. In this paper we use the notation of conjunctive queries (corresponding to select-project-join queries in SQL) to specify the conditions on nodes and arcs. A conjunctive query has 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 the variables of a query q by $Vars(q)$, and the variables in $\bar{X}$ are called the *distinguished* variables of q, representing the variables selected in the result of the query. When the head of the query is omitted, we assume that all the variables are distinguished. The arcs in the site schema are labeled as follows.

- **Ref Arcs:** a ref arc in the site schema between $F_1(\bar{X}_1)$ and $F_2(\bar{X}_2)$ is labeled by a triple: $(q, anchor, label)$, where: (a) q is a conjunctive query denoting the condition for the existence of the arc in the site graph, and whose distinguished variables include $\bar{X}_1 \cup \bar{X}_2$ (b) *anchor* is either a string or one of the distinguished variables of q determining which value will appear on the HTML anchor associated with the link, and (c) *label* is a string denoting the name of the attribute resulting in the site graph.
- **Data arcs:** a data arc in the site schema between $F_1(\bar{X}_1)$ and 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.

To simplify the exposition, our discussion does not include the formalisms needed to model forms in HTML pages. However, we note that extending site specifications to include forms does not complicate our algorithms.[2] A site schema and a database instance define a unique site graph, which is defined as the result of the static evaluation procedure described below.

[2] In particular, accommodating forms requires that we mark some of the variables in the queries of the site graph as bound.

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 [26]. The database contains information about products, customers and client 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, shipdate)
Order(orderkey, custkey, orderstatus, totalprice,orderdate, orderpriority)

The site schema shown in Figure 1 provides the following organization of the data. There is a root page with two links to suppliers (the node labeled `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 the orders. Specifically there is one page for each customer-supplier pair (`CustSupp(CK,SK)`) where the customer ordered from the supplier: that page is accessed both from the supplier and customer pages. From here there are further links to pages detailing orders placed by that customer to the that supplier. Of course, in designing the web site we also add sufficient links back to facilitate navigation. The definitions of the queries in the site schema are given in the appendix A.

2.3 Static and dynamic evaluation of the site graph

As a basis for the subsequent discussion, we identify two extreme approaches to computing the web site: a static and a dynamic approach. When the database does not change during browsing, the two approaches will result in identical site graphs. We explain the two approaches below and highlight their disadvantages.

In the static approach, 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 in G between the nodes $F_1(\bar{X}_1)$ and $F_2(\bar{X}_2)$. Let $q(D)$ be the result of evaluating q over the database D. For each tuple $\bar{a}q(D)$, 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.

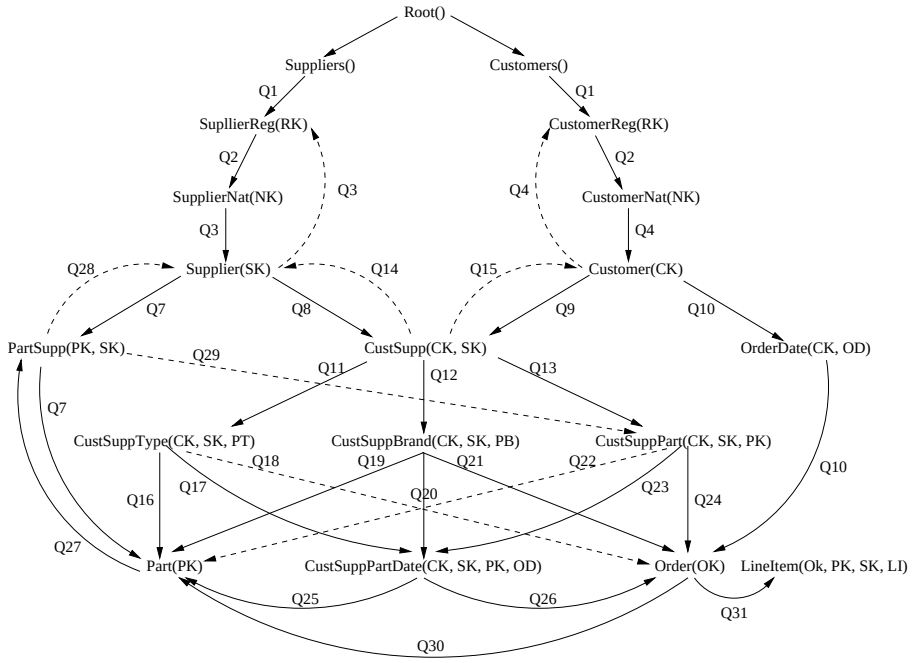

Figure 1: The site schema for the TPC/D example. For brevity, we omitted the data arcs.

Then, $G(D)$ contains a link between $F_1(\bar{a}_1)$ and $F_2(\bar{a}_2)$, labeled *label* 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*.
- **Eliminate unreachable subgraphs:** the roots of the site graph are defined to be all the nodes of the form $F(\bar{a})$, where F is the root of the site schema. Any node that is not reachable from some root of the site graph is removed.

The site graph computed by the above procedure can be converted into a browsable web site by applying the HTML templates to each of the nodes in the graph.

In the second approach the nodes in the site graph are computed only on demand. An HTTP request for a given page translates into a request for a node of the form $F(\bar{a})$. In order

to produce the page corresponding to $F(\bar{a})$, we need to compute all the data appearing in this page (i.e., all the data arcs emanating from $F(\bar{a})$ in the site graph) and all the outgoing HTML links (i.e., all ref links). 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 in G between the nodes $F(\bar{X})$ and $F_1(\bar{X}_1)$. 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 in G between the nodes $F(\bar{X})$ and Y. For each tuple $\bar{b} \in q(D)$ we define b_1 to be the projection of $\bar{b}$ to Y. Then, $G(D)$ contains a link between $F(\bar{a})$ and b_1, labeled *label*.

It is interesting to note that the tools available from relational vendors for embedding SQL queries in HTML files can be viewed as implementing the dynamic approach.

Both of the aforementioned approaches have significant disadvantages. The static approach cannot be applied to sites with forms (i.e., where the values of the input variables are only known at run-time), and incurs significant space overhead of duplicating the database (which can even be greater than duplicating the entire database, since the same information may appear in multiple pages). In addition, propagating updates from the database to the web site is more costly once the site has been materialized. In the dynamic approach, some queries may be too expensive to evaluate at run-time, and when multiple browsers request the same page this may result in wasted computation. A more important phenomenon on which we focus in this paper is that successive queries issued while browsing a site share much of their computation: (1) they either share many of the subgoals in their body, or (2) they have identical subgoal and differ only in the distinguished variables. Hence, exploiting the structure of the site and the similarity between the queries can lead to significant optimizations.

3 Problem Definition

This paper develops a framework for compiling a declarative specification of a web site into an optimal evaluation strategy expressed as a *run-time policy*. The run-time policy specifies *which* data to precompute or cache, *which* actions to execute at each page request, depending on the history of the browsing. To formally specify our problem we begin by describing the

inputs to the compilation problem, and the cost that our compilation algorithm is trying to minimize.

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. Ideally, given the browsing history of a user, the compilation algorithm should have the probability distribution on the successors of a web page. With such information, the compilation algorithm can estimate the frequency of execution of each of the queries on the DBMS and each possible sequence of queries. In reality, it is unlikely that we can obtain and manage such fine-grained statistics. Instead, we assume that we have access to the following higher-level statistics

- Node probability distribution: let $F_1, \ldots, F_n$ be the set of internal nodes in the site schema. We assume that we have 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 that we have 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 graph, let $(F(val_1), \ldots, F(val_s))$ be its instances in the web site. 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 $F(val_j)$.
- 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 provide useful estimates also when the database changes.

The node probability distribution and the context probability distribution are used directly in our formula for evaluating the cost of a run-time policy. In addition, the node probability distribution is used to guide our optimization algorithm to the most frequently requested pages. The value probability distribution is used to decide which input values occur with high frequency, and therefore which parameterized queries to cache (see Section 4.3). Finally, the arc probability distribution is to decide on lookahead computations, where actions are chosen based on likely expected futures (see Section 4.4).

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:

- *size(WS)*: the size of the (eventually) materialized HTML pages plus the size of the (eventually) precomputed or cached data;
- *age(WS)*: every data item I shown in the web site depends on a set of values $dep(I)$ 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[3]: $(size(WS) < S, age(WS) < A, wait(WS) < W)$, 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. We consider only run-time policies that satisfy these conditions.

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 a page, weighted by the probability of accessing

[3]We might imagine giving different *age* constraints for different data items, and different *wait* constraints for different types of nodes, but we use this simpler model for ease of exposition.

that page. We calculate this as follows. We denote by $wait_{RP}(F(\bar{X}))$ the average time for executing the queries needed for generating a page 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:

$$cost(RP) = \sum_i p_i \times wait_{RP}(F_i) \quad (1)$$

3.3 Equivalence of run-time policies

Ideally, our optimization algorithm should choose among *equivalent* run-time policies, i.e., policies that produce identical web sites. However, equivalence of web sites is tricky to define when the underlying data may change. Consider for example the dynamic approach. If all the queries for a given page are executed within a single transaction, then the data on the page represents a consistent snapshot of the database. Even then, data across pages may not be consistent. The problem is further complicated when, as we discuss later on, we also materialize views and cache previous results. In that case, since the data on a single page is computed from both the database and the views, inconsistencies may appear even within a single page.

One simple (but too restrictive) method to achieve consistency for a given page is to require that the age of all the data relevant to the page be 0. In this work we consider a weaker condition by imposing an age constraint of k time units on the site. In this case, we are assured that all the data for a given page are based on snapshots within k time units from one another. A more general approach that is beyond the scope of this paper is to require dependencies between the timestamps of the sources of different data items, i.e., require that all the data for a given set of pages is taken from the *same* snapshot of the database.

4 Optimization techniques for web-site management

In order to develop a meaningful formalism for specifying run-time policies this section considers several techniques for optimizing the dynamic evaluation of web sites. We explore several kinds of optimizations such as precomputation of materialized views, dynamically caching of data, and query rewriting optimizations that exploit the structure of the web site, such as query simplification under preconditions and lookahead computations.

We evaluate the impact of our optimizations on the STRUDEL-R system (described in Section 7). Our experiments were performed on a web site derived from the TPC/D benchmark,

whose site schema is shown in Figure 1. The experiments were run on TPC/D databases at scale factors 0.1 and 1, resulting in a database of 196MB and 1.84GB respectively. 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. One disk was used to hold the databases and another for Oracle's binaries. It should be noted that all queries have been implemented in the best possible way before the application of our optimizations techniques. This involved a significant amount of manual tuning like adding indexes on several tables for primary and non-primary keys in order to help the optimizer of the DBMS to generate acceptable queries execution plans and consequently, to get reasonable response times.

Our experiments measure the average time for the database operations needed to compute a request for a page in the site schema. The graphs are generated as a result of running 100 independent browsing sequences of length at most 20. The browsing sequences are generated based on choosing the next web page randomly based on a uniform distribution over the emanating links. Note, however, that since our experiments report speedups per node in the site schema, as opposed to the global utility of a run-time policy, the browsing patterns do 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.

Since the experiments that we present consider a single web site (albeit a natural design of a web site for TPC/D data) we do not mean to draw general conclusions based on them. Our purpose is to validate the potential impact of our proposed optimizations, and to highlight the tradeoffs involved with each optimization. Furthermore, experiments performed in different settings (e.g., the DB&LP web site, different organizations of the TPC/D data, and using the TPC/D large database) show similar trends and tradeoffs.[4]

4.1 Query simplification under preconditions

The first optimization we consider is a query rewriting technique that exploits the knowledge about the path(s) 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 clicks on a page $F_2(\bar{a}_2)$ when 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. Clearly, the tuple $(\bar{a}_1, \bar{a}_2)$ is in the result set of q_0. In order to display $F_2(\bar{a}_2)$ 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 those queries. In some cases the query $q \wedge (\bar{X}_2 = \bar{a}_2)$ can be simplified (i.e., some conjuncts will be removed from the query) given that we know that the

[4] We eagerly await the publication of the TPC/W benchmark!

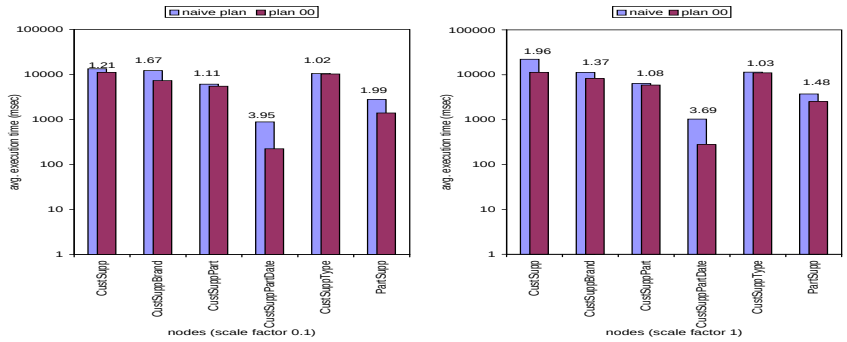

Figure 2: The graphs show the savings obtained from query simplification under preconditions at scale factors 0.1 and 1.

tuple $(\bar{a}_1, \bar{a}_2)$ is in the result set of q_0. The following example illustrates this optimization, which we call *simplification under preconditions.*

Example 4.1: Consider the node `CustSuppPart(ck,sk,pk)` in Figure 1. Here `ck,sk,pk` are values for the variables `CK,SK,PK` respectively. To expand the node we have to compute:

```
Q22(CK,SK,PK,PN) :- Order(OK,CK,_,_,_,_), Lineltem(OK,_,PK,SK,_,_),
                    Part(PK,PN,_,_,_), CK=ck, SK=sk, PK=pk
```

Before computing the join we observe that we could have reached this node from the `CustSupp(CK,SK)` node via the edge `Q13`. To be precise, the constants `ck,sk,pk` are in the answer set of the query:

```
Q13(CK,SK,PK,PN) :- Order(OK,CK,_,_,_,_), Lineltem(OK,_,PK,SK,_,_), Part(PK,PN,_,_,_)
```

Hence, it is possible to expand the node `CustSuppPart(ck,sk,pk)` by computing the following simpler query: `Q22'(CK,SK,PK,PN) :- Part(PK,PN,_,_,_), CK=ck, SK=sk, PK=pk`

Query simplification under preconditions is a form of query rewrite. It is interesting to note that unlike traditional query rewriting techniques (e.g., predicate pushdown), query simplification cannot be done manually by the person writing the queries for the site specification. For example, the user cannot manually replace Q22 in Figure 1 with:

```
Q22'(CK,SK,PK,PN) :- Part(PK,PN,_,_,_)
```

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 `Q22`. 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.

When query simplification under preconditions modifies the query, it always reduces the running time. Figure 2 shows the the running times for the naive dynamic evaluation versus the policy where all queries are simplified under preconditions. 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 at scale factors 0.1 and 1. In subsequent experiments, we always compare the additional optimizations to the plan obtained after applying query simplification under preconditions (this plan is referred to as Plan 00 in the figures).

4.2 View materialization

Another way to speed up the web site's performance is to precompute materialized views. This is especially attractive when multiple queries in the site share similar computations. The problem we face is to decide which set of views $\mathcal{V}$ to materialize in order to optimize the behavior of the site. This problem of choosing a set of materialized views for a given query workload has received significant attention in recent literature [14, 24, 11, 13, 12, 6]. It is important to note that in our context it becomes even more important to consider selecting views *and* indexes on them at the same time.[5] Since the queries in our site definition are always parameterized, there are obvious heuristics for choosing the appropriate indexes. In addition we need to consider the issues of view freshness and views with outer-joins.

As discussed earlier, we associate freshness constraints with a web site. These constraints need to be propagated to corresponding constraints on freshness of the materialized views. Many methods have been proposed for incremental maintenance of materialized views. The only aspect that is relevant to our system is whether there exists a method that guarantees the required freshness constraints. When not, this excludes the possibility of using certain views.

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 outer-join 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. We note that algorithms for rewriting queries using views can be straightforwardly extended to handle outerjoins.

[5] In fact, our experiments showed that precomputing views without creating appropriate indexes can slow down performance by a factor of 20.

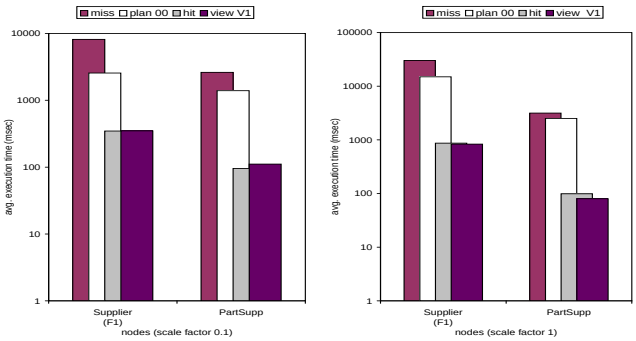

Figure 3: Both graphs compare running times of function caching (hits or misses), view materialization and the original queries.

Example 4.2: Assume we decide to materialize the following view with an index on the attribute SK:

V1(CK,SK,PK,CN) :- Customer(CK,CN,_,_,_),Lineltem(OK,_,PK,SK,_,_),Order(OK,CK,_,_,_,_)

The view `V1` can be used in answering both `Q8` and `Q29`. We measured considerable speedup rates for the rewritten queries (Figure 3): 13 for `PartSupp` nodes and 8 for `Supplier` nodes at scale factor 0.1 and respectively, 32 and 18 at scale factor 1. However, the additional space needed for the view and the indexes is around 50M at scale factor 0.1 and 600M at scale factor 1 (27% of the size of the original databases).

4.3 Data caching

View materialization always speeds up 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 only the result of parameterized queries executed so far [9]. In this way we can store less data and still obtain significant speedups. Furthermore, it is often cheaper to periodically invalidate data in a cache than to pay the cost of view maintenance.

Our caching optimization saves the results of parameterized queries in special relations called *functions*. Formally, a 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 whose projection on $input(f)$ is $\bar{a}$. At runtime, f 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 function:

> For any constant $\bar{a}$, either f does not contain any tuples from q_f whose projections on $input(f)$ is $\bar{a}$, or it contains *all* such tuples.

In our system functions are implemented as tables in the DBMS, with the same schema as their corresponding view. We maintain an additional attribute to keep track of input values for which there is not tuple in q_f. We always construct an index on the attributes of the table corresponding to the function's input variables.

An important question is how functions are used at run-time. Assume that we have a function $f = (q_f, input(f))$ stored in a table `F` and a query q to be executed; we use f in q by rewriting q into a an equivalent query q' that refers to `F`. We cannot treat `F` in the same way we treat a materialized view, because treating the function as a view would be incorrect when the needed values for computing q' are not cached. 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 f in q', the variables corresponding to input variables of f are also bound variables in q'.

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

Example 4.3: As we saw in Example 4.2 the view V1 significantly improved performance but at the price of space overhead. Suppose that instead of V1 we want a semantically equivalent function F1, defined in the node `Supplier(sk)` and reused in the node `PartSupp(pk, sk)`.

```
F1(SK,CK,PK,CN) :- Customer(CK,CN,_,_,_), Lineltem(OK,_,PK,SK,_,_), Order(OK,CK,_,_,_,_),
                   input SK
```

We can use F1 to rewrite queries `Q8` and `Q29` as follows:

```
Q8_F1(CK, SK, CN) :- F1(SK,CK,_,CN), SK=sk
Q29_F1(CK, SK, PK, CN) :- F1(SK,CK,PK,CN), PK=pk, SK=sk
```

At run-time, when we compute $Q8_{F1}$, we first check to see if `sk` occurs in the domain of `F1`. If we have a hit, we return the set of associated pairs `(ck, cn)`. Otherwise we compute the function's body with the additional binding `SK = sk`, insert the result in the function, and return the result. Query `Q29` is handled similarly.

The utility of caching

Figure 3 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 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 14 and 7 at scale factor 0.1 and respectively 25 and 17 at scale factor 1 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 materialization 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 $cost_{regular}(\mathbf{F})$, the cost of evaluating a query with a cache hit by $cost_{hit}(\mathbf{F})$, and the cost of evaluating a query with a cache miss by $cost_{miss}(\mathbf{F})$. In the first step, we use the following formula to derive the minimum value for the required hit ratio on the cache, $\tau(\mathbf{F})$:

$$\tau(\mathbf{F}) \times cost_{hit}(\mathbf{F}) + (1 - \tau(\mathbf{F})) \times cost_{miss}(\mathbf{F}) < cost_{regular}(\mathbf{F}) \quad (2)$$

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. In this paper 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 function is updated in the same node in which it is used. In addition, the cached 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 can be updated in one node in the site and the result can be used in multiple nodes, and (2) a cache can include a subquery of a query appearing on the arc.

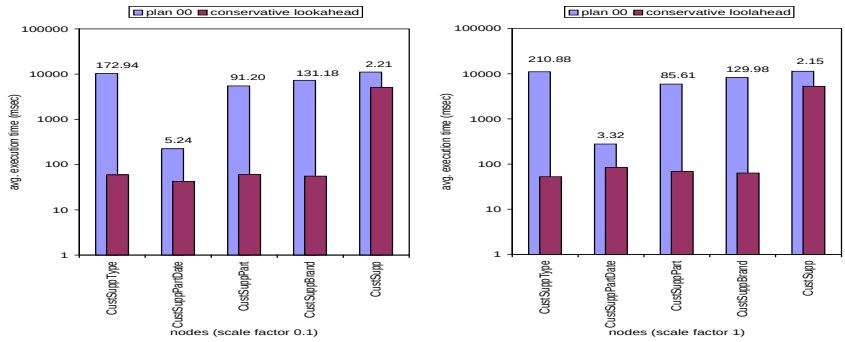

Figure 4: The graphs show the benefits of conservative lookaheads performed in the node `CustSupp`.

4.4 Lookahead computation

The key idea behind lookahead computation is to modify the definition of cached functions such that a function 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 performed at F, but is deemed to be useful for future nodes.

Conservative lookahead

Consider the node `CustSuppPart(CK,SK,PK)` in our example, where we need to compute the following

```
Q23(CK,SK,PK,OD) :- Order(OK,CK,_,_,OD,_), Lineltem(OK,_,PK,SK,_,_), Part(PK,_,_,_,_),
                    CK=ck, SK=sk, PK=pk
```

In the next node, `CustSuppPartDate(CK,SK,PK,OD)` we need to compute query `Q26`:

```
Q26(CK,SK,PK,OK,OD) :- Order(OK,CK,_,_,OD,_), Lineltem(OK,_,PK,SK,_,_), Part(PK,_,_,_,_),
                       CK=ck, SK=sk, PK=pk, OD=od
```

Assume we define a function for `Q23` with inputs CK, SK and PK. As we can see, much of the computation performed for the function for `Q23` is also useful for `Q26`. However, if we

simply cache the result of `Q23`, we cannot use unchanged for `Q26` because `Q23` projected out the attribute OK. Conservative lookahead would define a function with the same subgoals (since the subgoals of `Q23` and `Q26` are identical) and whose head includes all the attributes needed for both `Q23` and `Q26`.

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. The function f will have as body the intersection: $body(f) = body(q) \cap body(q')$. The distinguished variables of f include (1) the distinguished of q and q', and (2) the variables that are either in q or q' but not in the body of f. $input(f)$ are defined to be those variables of f that occur in $\bar{X}_1$. The function f will be defined 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 (because of the constraint we imposed on using functions 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 function computed at the root of the tree can be used in its descendants. By applying this technique to the following set of nodes `CustSupp, CustSuppType, CustSuppBrand, CustSuppPart` and `CustSuppDate` we obtain the following function which is computed in the node `CustSupp` and used in all the others.

```
F(CK,SK,PK,OD,OK,PN,PB,PT) :- Order(OK,CK,_,_,OD,_), Lineltem(OK,_,PK,SK,_,_),
                              Part(PK,PN,PB,PT,_), Input CK, SK
```

Optimistic lookahead

Optimistic lookahead performs additional computation that may be usable in later nodes. For example, consider the node `Customer(CK)`, where we need to compute the following

```
Q9(CK,SK,SN) :- Supplier(SK,SN,_,_,_), Order(OK,CK_,_,_,_), Lineltem(OK,_,_,SK,_,_),
                CK=ck
```

In the next node, `CustSupp(CK,SK)` we need to compute query `Q13`:

```
Q13(CK,SK,PK,PN) :- Order(OK,CK,_,_,_,_), Lineltem(OK,_,PK,SK,_,_), Part(PK,PN,_,_,_),
                    CK=ck, SK=sk
```

Suppose we want to define a function for `Q9` that also performs all the necessary computation for the query `Q13`. To do this, we define a function that includes the common subgoals, but also performs an outer-join with the subgoals of both `Q9` and `Q13` not included in the intersection. Specifically, we would define the function

```
F(CK,SK,PK,PN,SN) :- ((Order(OK,CK,_,_,_,_) ⋈ Lineltem(OK,_,PK,SK,_,_)) ⋉⊏
                      Part(PK,PN,_,_,_)) ⋉⊏ Supplier(SK,SN,_,_,_), Input CK
```

This function 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 the node `Customer` we do a join with Part that is not necessary there, but 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 define a function 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 function f will have as body the expression $(q_0 \ltimes\sqsubset (body(q) - q_0)) \ltimes\sqsubset (body(q') - q_0)$. 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 function f will be defined 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 function shown above can also be used in the rewritings of the queries of nodes `CustSuppType, CustSuppBrand, CustSuppPartDate, CustSuppPart` and `CustSupp`.

The utility of lookaheads

Figure 4 shows the experimental results concerning conservative lookahead computations. The graphs show the speedups obtained for the TPC/D databases with scale factors 0.1 and 1. We observe that the cost of computing `CustSupp` was not affected, while the speedups obtained for its descendants ranged from factors of 5 to 172 in the left graph and from 3 to 210 in the right one. `CustSupp` is a particular node because it has three outgoing edges with similar queries (`Q11, Q12 and Q13`) that benefit from the function caching done at this node. Figure 5 shows the results for optimistic lookahead. We observe that the `Customer` node, which has an additional computation, was slowed down by a factor of 3.7, while the speedups for its descendants varied from 4 to 139 at scale factor 0.1 and from 5 to 161 at scale factor 1. We notice that in both cases, we get more improvement at scale factor 1 than at scale factor 0.1.

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 because they correspond to a natural organization of data in a hierarchical fashion.

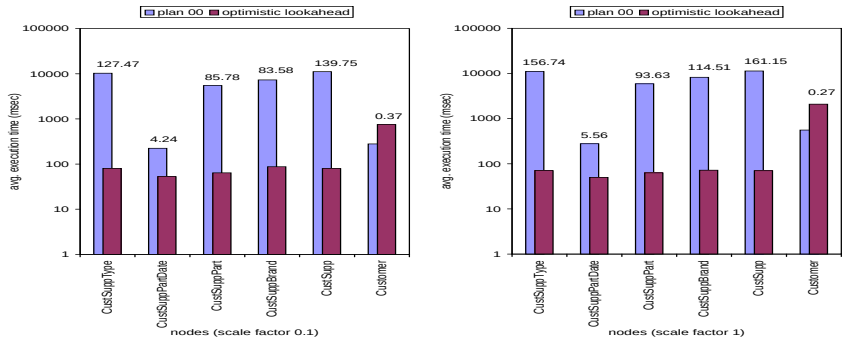

Figure 5: The graphs show the benefits of optimistic lookahead performed at the node `Customer`.

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. To define run-time policies we first describe run-time schemas, which are the sets of relations and functions 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 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.

Note that in principle we may decide to maintain several run-time schemas for a web-site (and accompanying run-time policies) for different classes of users. For simplicity, we restrict our discussion to a single schema and run-time policy.

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 of 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. An action parameterized by a specific context is called a *rule*.[6]

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 page $F(\bar{a})$ if $F = F_n$ and the previously requested pages of the same user where 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.

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 query to be executed in order to obtain all the outgoing links of a page of type F_1 to pages of the type F_2.

[6]Sometimes users jump back in their browsing. At these points, we can use the context of the previous visit to the page as the current context.

```
/* 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 function F 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 F with custkey → CK
      Link to CustSuppType(CK,SK,PT)
            if context Customer.CustSupp compute
                SELECT f.custkey, f.suppkey, p.name, p.type
                FROM F 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.partkey=p.partkey and v.custkey=CK and v.suppkey=SK
      Link to CustSuppPart(CK,SK,PK)
            if context Customer.CustSupp compute
                SELECT f.custkey, f.suppkey, p.partkey, p.name
                FROM F f, Part p
                WHERE f.custkey=CK and f.suppkey=SK and f.partkey=p.partkey
            else compute
                SELECT v.custkey, v.suppkey, p.partkey, p.name
                FROM V v, Part p
                WHERE v.partkey=p.partkey and v.custkey=CK and v.suppkey=SK
```

Figure 6: Fragments of a run-time schema and policy for our running example.

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 (the complete BNF specification for the run-time schema and policy language is given in Appendixes D and E). We illustrate run-time policies with this language in Figure 6.

5.3 Run-time management 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 $F_k = F$, and 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 $[F_1, \ldots, F_k]$. Specifically, if $\psi(\bar{a})$ is not in the cache, we compute $f(\psi(\bar{a}))$ and add it 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 $q \wedge (\bar{X} = \bar{a})$.

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 cached 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 $F(\bar{X}) \rightarrow G(\bar{Y})$ 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. Then, 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 than in this definition functions are used as view definitions).

- If one of the q_i's uses a function f, then the node F in the run-time policy includes a update action for f. The intuition for this condition is that it is only correct to use a function in a query if we are guaranteed that the required values are already cached in the function. Note that an 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 here is necessary in order to guarantee the correctness of the result.

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 declarative 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 the previous section 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.[7]

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 steps 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].

[7] 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.

4. If a view V was a good candidate for improving run-time performance in the previous step but was not chosen because of space or freshness constraints, consider including in the run-time schema functions of the form (Inp, V), where Inp is a subset of the arguments of V.

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

Figure 7 shows the results of applying these steps in two scenarios. In the first 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 second 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 and a function defined in the `Part` node. 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`), an optimistic lookahead computation in the node `Customer` which benefits the nodes `CustSupp, CustSuppType, CustSuppBrand, CustSuppPart` and `CustSuppDate` and, as in the first case, a function in the node `Part`. Except for the node `Supplier`, 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 practically no memory overhead.

7 Implementation

The the STRUDEL-R system is implemented and fully operational, though the compiler from declarative specifications to run-time policies is relatively simple. The queries in the site definition 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. 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. This feature was also particularly useful during the performance evaluation stage, where hundreds of run-time policies were manually generated and tested in order to isolate the positive or negative effects of particular techniques that we explored.

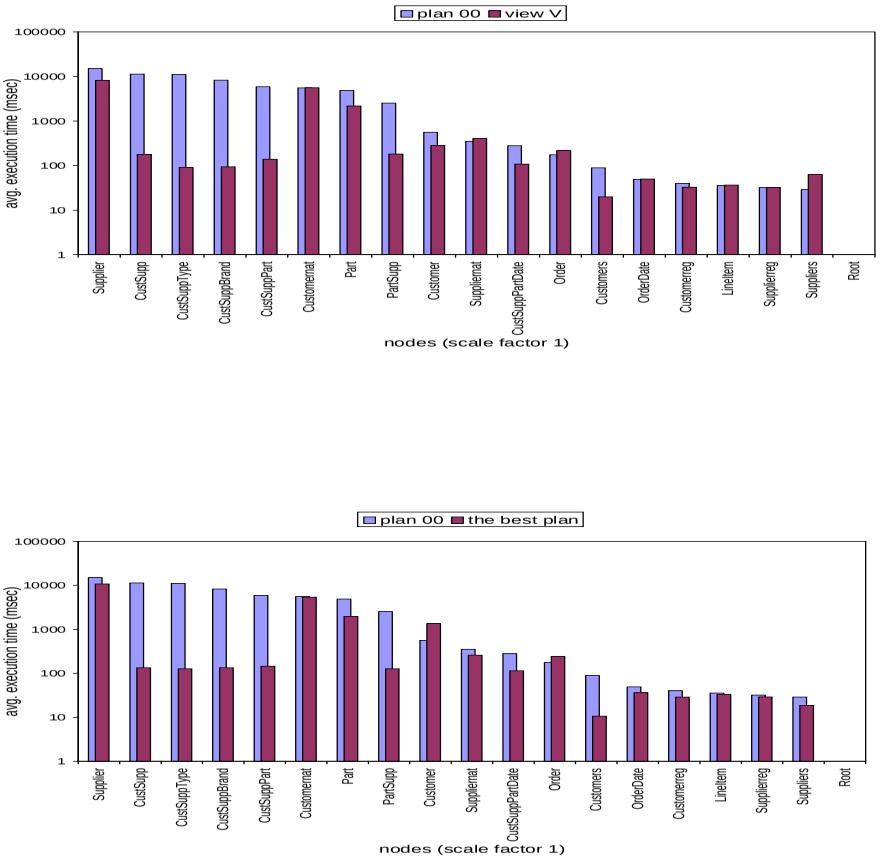

Figure 7: Results of two run-time policies. The upper graph shows a run-time policy in which 1GB of additional memory was provided, and the lower graph shows a policy when only 10MB were provided.

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 CGI-bin script takes as input the node in the site schema, the bindings for the

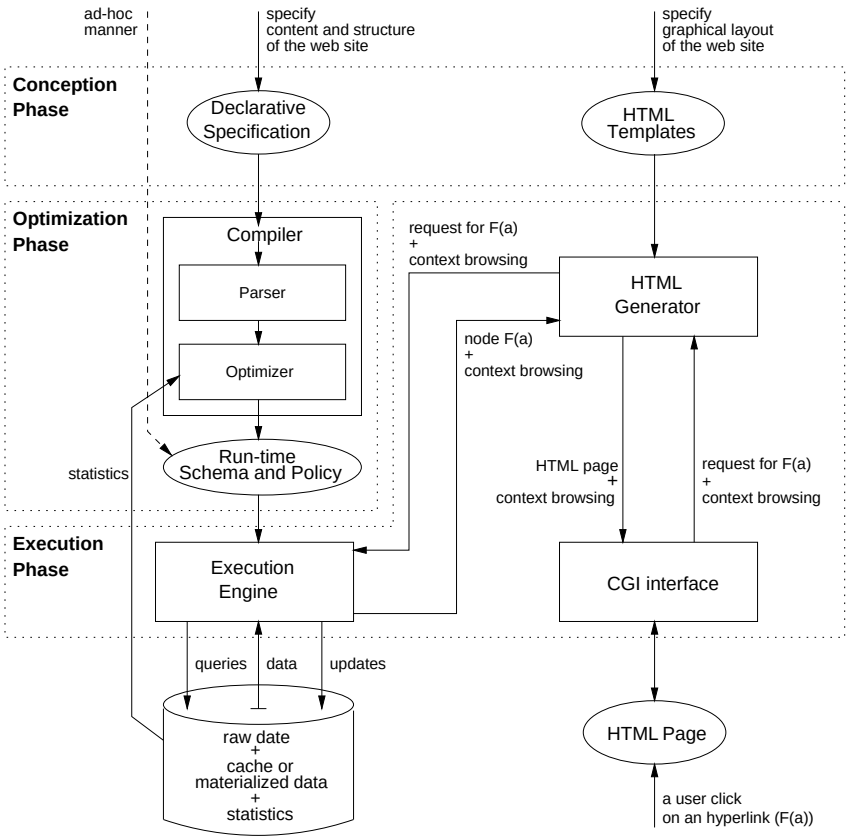

Figure 8: STRUDEL-R architecture. Dotted boxes refer to the main conceptual phases during the web site design. Solid boxes indicate the main components of the system.

variables associated with the node and the browsing context. The script calls the HTML generator, which in turn calls STRUDEL-R's execution engine with the same parameters. The execution engine follows the run-time policy, and the result (the data contained in the page and information about the outgoing links) is sent back to the HTML generator in order to compute an HTML page for the browser. 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.

Two important implementation choices regarding the functions were learned from our experience. First, functions should be implemented as temporary tables, thus avoiding the overhead of logging operations. Second, it is beneficial to duplicate the domains of the cached functions outside the DBMS, thereby saving a database access when accessing functions.

The system is implemented in Java over using the Oracle DBMS Version 7.3.2. All the database connections are done through Oracle's JDBC Thin 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 significant 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 is clearly a problem deserving 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 [8, 2, 4, 21, 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 [20] 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 [17, 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., [14, 24, 11, 13, 12, 6]), multiple query optimization [23], index selection [7], and parameterized query optimization [16]. 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, the constraints on our application our different such as the age and time limit constraints, and the fact that queries in the workload appear in succession, not in parallel.

Finally, a related body of work is that on using invariants for query execution [22] (in the context of nested correlated queries) and [18] 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.

References

[1] S. Adali, K. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.

[2] Gustavo Arocena and Alberto Mendelzon. WebOQL: Restructuring documents, databases and webs. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Orlando, Florida, 1998.

[3] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. To weave the web. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1997.

[4] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. Design and maintenance of data-intensive web sites. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.

[5] M-L. Schneider B. Chidlovskii, C. Roncancio. Semantic cache mechanism for heterogeneous web querying. In *8th International World Wide Web Conference, Toronto, Canada*, 1999.

[6] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized views selection in a multidimensional database. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 156–165, 1997.

[7] S. Chaudhuri and V. R. Narasayya. Microsoft index tuning wizard forSQL Server 7.0. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 553–554, 1998.

[8] Sophie Cluet, Claude Delobel, Jerome Simeon, and Katarzyna Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.

[9] Shaul Dar, Michael J. Franklin, Björn Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 330–341, 1996.

[10] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.

[11] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 208–219, 1997.

[12] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, pages 453–470, 1999.

[13] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[14] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 205–216, 1996.

[15] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 423–434, 1996.

[16] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. *VLDB Journal*, 6(2):132–151, 1997.

[17] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases: Design, Realization, and Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):587–608, 1994.

[18] F. Llirbat, F. Fabret, and E. Simon. Eliminating costly redundant computations from sql trigger executions. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 428–439, 1997.

[19] Tam Nguyen and V. Srinivasan. Accessing relational databases from the world wide web. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.

[20] T. Palpanas and Alberto Mendelzon. Web prefetching using partial match prediction. Technical report CSRG-376. Departement of Computer Science, University of Toronto, 1998. Available from `http://www.dblab.ece.ntua.gr/ thpalpa`.

[21] P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, 1998.

[22] J. Rao and K. A. Ross. Reusing invariants: A new strategy for correlated queries. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 37–48, 1998.

[23] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[24] Dimitri Theodoratos and Timos Sellis. Data warehouse design. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[25] Motomichi Toyama and T. Nagafuji. Dynamic and structured presentation of database contents on the web. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.

[26] TPCD. Benchmark. Available from `http:// www.tpc.org`.

A The definitions of the site schema Queries

The following are the queries labeling the arcs in the site schema in Figure 1. The attribute names in boldface are bound variables in the dynamic evaluation.

Q1(RK, RN) :- Region(RK, RN, _)

Q2(NK, RK, NN) :- Nation(NK, **RK**, NN, _)

Q3(SK, SN, NK) :- Supplier(SK, SN, _ ,**NK**, _)

Q4(CK, CN, NK) :- Customer(CK, CN, _, **NK**, _)

Q5(SK, RK, RN) :- Supplier(**SK**, _, _, NK, _), Nation(NK, _, RK, _), Region(RK, RN, _)

Q6(CK, RK, RN) :- Customer(**CK**, _, _, NK, _), Nation(NK, _, RK, _), Region(RK, RN, _)

Q7(PK, SK, PN) :- Part(PK, PN, _, _, _), PartSupp(PK, **SK**, _, _, _)

Q8(CK, SK, CN) :- Customer(CK, CN, _, _, _), LineItem(OK, _, _, **SK**, _, _), Order(OK, CK, _, _, _, _)

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

Q10(OK, CK, OD) :- Order(OK, **CK**, _, _, **OD**, _)

Q11(CK, SK, PT) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(PK, _, _, PT, _)

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

Q13(CK, SK, PK, PN) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(PK, PN, _, _, _)

Q14(CK, SK, SN) :- Supplier(**SK**, SN, _, _, _), LineItem(OK, _, _, SK, _, _), Order(OK, **CK**, _, _, _, _)

Q15(CK, SK, CN) :- Customer(**CK**, CN, _, _, _), LineItem(OK, _, _, **SK**, _, _), Order(OK, CK, _, _, _, _)

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

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

Q18(CK, SK, PT, OK) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(PK, _, _, **PT**, _)

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

Q20(CK, SK, PK, PB, OD) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, OD, _), Part(PK, _, **PB**, _, _)

Q21(CK, SK, PB, OK) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(PK, _, **PB**, _, _)

Q22(CK, SK, PK, PN) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(**PK**, PN, _, _, _)

Q23(CK, SK, PK, OD) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, OD, _), Part(**PK**, _, _, _, _)

Q24(CK, SK, PK, OK) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(**PK**, _, _, _, _)

Q25(CK, SK, PK, PN, OD) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, **OD**, _), Part(**PK**, PN, _, _, _)

Q26(CK, SK, PK, OK, OD) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK** _, _, **OD**, _), Part(PK, _, _, _, _)

Q27(PK, SK, SN) :- PartSupp(**PK**, SK, _, _, _), Supplier(SK, SN, _, _, _)

Q28(SK, PK, SN) :- Supplier(**SK**, SN, _, _, _), PartSupp(**PK**, SK, _, _, _)

Q29(CK, SK, PK, CN) :- Customer(CK, CN, _, _, _), LineItem(OK, _, **PK**, **SK**, _, _), Order(OK, CK, _, _, _, _)

Q30(OK, PK, PN) :- Order(**OK**, _, _, _, _, _), LineItem(OK, _, PK, _, _, _), Part(PK, PN, _, _, _)

Q31(OK, PK, SK, PN, LI) :- Order(**OK**, _, _, _, _, _), LineItem(OK, LI, PK, SK, _, _), Part(PK, PN, _, _, _)

B The database schema description BNF

schema ::= DEFINE SCHEMA [schemaName] AS (DEFINE table)$^+$
table ::= TABLE tableName (fields)
fields ::= field (, field)*
field ::= fieldName fieldType [(constraint)]
fieldType ::= INT | FLOAT | STRING | DATE
constraint ::= PRIMARY KEY |
REFERENCES reference
reference ::= tableName (fieldName)
schemaName ::= identifier
tableName ::= identifier
fieldName ::= identifier

C The query language BNF

program ::= PROGRAM [programName] (DEFINE (query|node))$^+$

```
node ::= NODE origin ((link))+
link ::= LINK TO target
         LABEL label
         [COMPUTE queryName [INMAP map] [OUTMAP map]]
         ANCHOR anchor
map ::= variableName/variableName (, variableName/variableName )*
origin ::= skolemFunction
skolemFunction ::= skolFunctionName ([arguments])
arguments ::= variable (, variable )*
variable ::= synonym
target ::= skolemFunction | variable
query ::= QUERY queryName AS
            SELECT projection
            FROM domain
            [WHERE whereConditions]
            [GROUP BY expression [HAVING havingConditions]]
domain ::= tableDecl (, tableDecl)*
tableDecl ::= table variableName
projection ::= mapReference (, mapReference)*
mapReference ::= reference [AS synonym] | function [AS synonym]
reference ::= variableName.attributeName
expression ::= reference (, reference)*
function ::= functionName (reference)
functionName ::= MIN | MAX | AVG
whereConditions ::= whereCondition (AND whereCondition)*
havingConditions ::= havingCondition (AND havingCondition)*
whereCondition ::= whereTerm operator whereTerm
havingCondition ::= havingTerm operator havingTerm
whereTerm ::= reference | constant | parameter
havingTerm ::= whereTerm | function
operator ::= = | < | > | <= | >= | LIKE
programName ::= identifier
queryName ::= identifier
table ::= identifier
anchor ::= "identifier" | identifier
label ::= "identifier"
variableName ::= identifier
```

skolFunctionName ::= identifier
synonym ::= identifier
attributeName ::= identifier
parameter ::= &identifier

D The run-time schema specification BNF

runtimeSchema ::= RUNTIME SCHEMA [runtimeSchemaName] (DEFINE (view|function))$^+$
view ::= VIEW objectName AS sqlStatement
MAX AGE age
function ::= FUNCTION objectName AS sqlStatement
inputs
MAX SIZE constant
MIN HIT RATIO constant
MAX AGE constant
inputs ::= INPUT variableName (, variableName)*
sqlStatement ::= SELECT projection
FROM domain
[WHERE whereConditions]
[GROUP BY expression [HAVING havingConditions]]
domain ::= ObjectDecl (, ObjectDecl)*
ObjectDecl ::= objectName variableName
projection ::= mapReference (, mapReference)*
mapReference ::= reference [AS synonym] | aggFunction [AS synonym]
reference ::= variableName.attributeName
expression ::= reference (, reference)*
aggFunction ::= aggFunctionName(reference)
aggFunctionName ::= MIN | MAX | AVG
whereConditions ::= whereCondition (AND whereCondition)*
havingConditions ::= havingCondition (AND havingCondition)*
whereCondition ::= whereTerm operator whereTerm
havingCondition ::= havingTerm operator havingTerm
whereTerm ::= reference [(+)] | constant | parameter
havingTerm ::= reference | constant | parameter | aggFunction
operator ::= = | < | > | <= | >= | LIKE

runtimeSchemaName::= identifier
parameter ::= &identifier
variableName ::= identifier
objectName ::= identifier
attributeName ::= identifier
synonym ::= identifier

E The run-time policy language BNF

runtimePolicy ::= RUNTIMEPOLICY runtimePolicyName (DEFINE(query|node))$^+$
node ::= NODE origin (fAction)* ((link))$^+$
link ::= LINK.target
LABEL label
(qAction)
fAction ::= IF CONTEXT history
((UPDATE ([actionType]
function INMAP mapinput)$^+$)
qAction ::= [IF CONTEXT history]
[COMPUTE queryName [INMAP inmap] [OUTMAP outmap]]
ANCHOR anchor
[ELSE qAction]
history ::= historyNode (historyNode)*
historyNode ::= skolFunctionName ([historyFctargs])
outmap ::= variableName/variableName (, variableName/variableName)*
inmap ::= variableName/variableName (, variableName/variableName)*
mapinput ::= variableName/variableName (, variableName/variableName)*
origin ::= skolemFunction
skolemFunction ::= skolFunctionName ([skolFctargs])
historyFctargs ::= variableName (, variableName)*
skolFctargs ::= variable (, variable)*
variable ::= synonym
target ::= skolemFunction | variable
query ::= QUERY queryName AS sqlStatement
sqlStatement ::= SELECT projection
FROM domain

[WHERE whereConditions]
[GROUP BY expression [HAVING havingConditions]]
domain ::= ObjectDecl (, ObjectDecl)*
ObjectDecl ::= objectName variableName
projection ::= mapReference (, mapReference)*
mapReference ::= reference [AS synonym] | aggFunction [AS synonym]
reference ::= variableName.attributeName
expression ::= reference (, reference)*
aggFunction ::= functionName (reference)
functionName ::= MIN | MAX | AVG
whereConditions ::= whereCondition (AND whereCondition)*
havingConditions ::= havingCondition (AND havingCondition)*
whereCondition ::= whereTerm operator whereTerm
havingCondition ::= havingTerm operator havingTerm
whereTerm ::= reference | constant | parameter
havingTerm ::= whereTerm | aggFunction
operator ::= = | < | > | <= | >= | LIKE
actionType ::= - | | +
parameter ::= &identifier
synonym ::= identifier
attributeName ::= identifier
runtimePolicyName ::= identifier
variableName ::= identifier
queryName ::= identifier
objectName ::= identifier
skolFunctionName ::= identifier
label ::= "identifier"
function ::= identifier
anchor ::= "identifier" | identifier
variableName ::= identifier
queryName ::= identifier
skolFunctionName ::= identifier

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399