Run-time Management of Data Intensive Web-sites

Daniela Florescu
Daniela.Florescu@inria.fr

Alon Levy
alon@cs.washington.edu

Dan Suciu
suciu@research.att.com

Khaled Yagoub
khaled.yagoub@inria.fr

1 Introduction

The World Wide Web (WWW) has been proven to be an excellent medium for businesses to disseminate information. As a result, the ability to populate web sites with content derived from large databases has become 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. In parallel, a new paradigm for building and maintaining web sites based on declarative specifications, has arisen in the research community [6, 2, 4, 1, 15]. Two main features underly this paradigm. First, a declarative specification is based on a *logical* model of the web site which captures the content and structure of the web site. Second, the logical model of the site is defined as a view, in some declarative language, over the data underlying the site.

As an example of this paradigm, 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. 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 [14]. One approach is to materialize the site completely. This approach has several obvious disadvantages. First, precomputation cannot be applied to sites with forms. Second, materializing the site would imply an important space overhead, often even greater than duplicating the entire database. 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: (a) some queries may be too expensive to evaluate at run-time, which is unacceptable in the interactive nature of web access, (b) evaluating queries at run-time may result in repeated computation. For example, an obvious repetition occurs when multiple browsers request the same page. A second interesting observation is that successive queries issued while browsing a site share much of their computation.

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: (1) current caching techniques do not cache dynamically generated pages and (2), even if caching techniques are extended, the granularity of an entire HTML page is too coarse-grained. Clearly, a deeper semantic analysis of the site definition is required in order to exploit the optimization opportunities raised by the second observation above. Since a first priority of web site managers today is to ensure reasonable response time, they end up hard-wiring 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.

One obvious solution to this 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 [18, 8, 9, 3], index selection, function caching [12, 11, 5], multiple query optimization [17] and reusing query invariants [13, 16]. 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 the structure of a Web site can lead to significant savings over and above the application of the known techniques mentioned above. Our final goal is to automatically compile a declarative specification into a *run-time policy*, which decides which actions to perform and which queries to evaluate depending on the browsing history. Run-time policies should be 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 that we present in this paper.

The paper is organized as follows. Section 2 describes declarative web site management systems and different run-time management techniques. Section 3 describes the optimization techniques that we propose in this paper and evaluates their impact.

2 Web sites specification

We begin by describing the general architecture of declarative web site management systems, as embodied in the STRUDEL-R system [7]. We note that the key architectural aspects of the STRUDEL-R are common to other systems for declarative web site management [1, 2, 15, 4].

STRUDEL-R is based on a *logical* representation of a web site, called a *site graph*, which is independent of its graphical presentation or of the underlying data management systems. The site graph models the pages in the web site, the links between them and the data associated with each page. Formally, 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. 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.

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 [6]. We refer to the collections of pages in a web site as *site collections*. Nodes in the site graph (and hence in the website) 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 $F(a_1, \ldots, a_n)$, where F is a site collection name and $a_1, \ldots, a_n$ are a list of data items.

A site graph in STRUDEL-R is defined intensionally, via a *site schema*, rather than extensionally, one node at a time. Formally, 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 the internal nodes $F_1(\bar{X}_1)$ and $F_2(\bar{X}_2)$ in the site schema is labeled by a conjunctive query q where $\bar{X}_1 \cup \bar{X}_2$ are distinguished variables of q. The query q denotes the condition

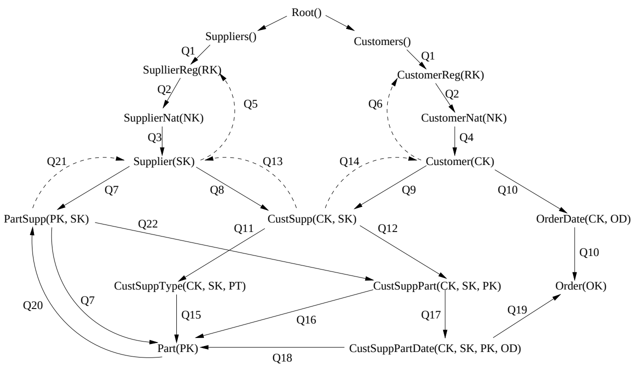

Figure 1: The site schema for the TPC/D example.

for the existence of the arc in the site graph between particular instances of $F_1(\bar{X}_1)$ and $F_2(\bar{X}_2)$. Similarly, a data arc in the site schema between $F_1(\bar{X}_1)$ and Y is labeled by a conjunctive query denoting the condition for the association of a particular data item to a web page.

Example 2.1: We use the following example throughout the paper and in our experiments. Suppose we want to produce a browsable version of the data contained in the TPC/D benchmark. The database contains information about products, customers and client orders. 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. The definitions of the queries in the site schema are given in the figure 5.

Applying the site schema to a particular instance of the database results in a site graph. We identify two extreme approaches to computing the web site from a site schema: a static and a dynamic approach. In the static approach, the entire site graph is computed ahead of time, by evaluating each query in the site schema. For each tuple in the result of such a query, a new arc is created in the site graph. Nodes are created as a side effect of arc creation; a node $F(X)$ in the site schema will generate a single node $F(\bar{a})$ for each binding $\bar{a}$ of the variables X. The site graph can be then 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 and all the outgoing HTML links. This translates in the execution at run-time of a set of parametrized queries, i.e. the set of queries labeling all the outgoings arcs from the node $F(\bar{X})$ in the site schema, with the extra selection $(\bar{X} = \bar{a})$. In figure 5, the attribute names in boldface are the bound variables in the dynamic evaluation.

Both of the aforementioned approaches have significant disadvantages. The static approach cannot be applied to sites with forms, and incurs significant space overhead of duplicating the database. 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.

3 Optimization techniques

Our final goal is to develop 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. In this paper we study several particular techniques for optimizing the dynamic evaluation of web sites 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. Our experiments were performed on a web site derived from the TPC/D benchmark (total size 1.84GB), whose site schema is shown in Figure 1. 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 random browsing sequences.

3.1 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, consider the node CustSuppPart in Figure 1. To construct the web page corresponding to the node CustSuppPart for given values ck, sk, pk, we have to compute:

Q16(CK,SK,PK,PN) :- Order(OK,CK,_,_,_,_),Part(PK,PN,_,_,_), LineItem(OK,_,PK,SK,_,_), CK=ck, SK=sk, PK=pk.

However, we observe that we could have reached this node from CustSupp(CK,SK) via the edge Q12, i.e., the constants ck,sk,pk are in the answer set of the query: Q12(CK,SK,PK,PN) :- Order(OK,CK,_,_,_,_), Part(PK,PN,_,_,_), LineItem(OK,_,PK,SK,_,_).

Hence, it is possible to expand then CustSuppPart(ck,sk,pk) node by computing the following simpler query:

Q16'(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 several reasons. First, the rewritten query is not safe for the static evaluation. Second, we may not use this query even during dynamic evaluation if the time between page requests exceeds the age limit of the site. 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 a query, it always reduces its running time. In our example, we obtain up to a 4 fold speedup in performance.

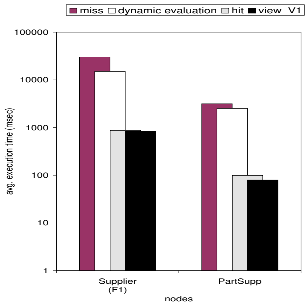

Figure 2: Benefits of view materialization and caching.

3.2 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 to materialize in order to optimize the behavior of the site. This problem of choosing a set of materialized views and the appropriate set of indices for a given query workload has received significant attention in recent literature [10, 18, 8, 9, 3].

Example 3.1: Let's assume we decide to materialize the following view with an index on the attribute SK:

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

The view V1 can be used in answering both Q8 and Q22. We measured considerable speedup rates: 32 for PartSupp nodes and 18 for Supplier nodes. However, the additional space needed for the view and the indexes is around 600M (27% of the size of the original database).

3.3 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 [5]. In this way we can store less data and still obtain significant speedups in certain cases. Furthermore, it is often cheaper to periodically invalidate data in a cache than to pay the cost of view maintenance.

Our caching optimization saves the results of parameterized queries in special database 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 $\bar{a}$.

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 3.2: Suppose that instead of the view V1 we want to maintain a semantically equivalent function F1, updated in the node Supplier(sk):
F1(SK,CK,PK,CN) :- Customer(CK,CN,_,_,_),
LineItem(OK,_,PK,SK,_,_), Order(OK,CK,_,_,_,_), input SK.

We can use F1 to rewrite queries Q8 and Q22 as follows:

$Q8_{F1}$(CK,SK,CN) :-F1 (SK,CK,_,CN), SK=sk
$Q22_{F1}$(CK,SK,PK,CN) :- F1(PK,SK,CK,CN),PK=pk, SK=sk

At run-time, when we compute $Q8_{F1}$, we first check to see if sk is already cached in 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.

Figure 2 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, (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 25 and 17 compared to case (4). Most interestingly, the overhead of case (3) compared to case (4) is relatively low (a slowdown of 2%) due to the extra cache check and update.

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 or a super query of a query appearing on an arc.

3.4 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 query:
Q17(CK,SK,PK,OD) :- Order(OK,CK,_,_,OD,_), Part(PK,_,_,_,_), LineItem(OK,_,PK,SK,_,_), CK=ck, SK=sk, PK=pk.

In the next node, CustSuppPartDate(CK,SK,PK,OD) we need to compute query Q19:

Q19(CK,SK,PK,OK,OD) :- Order(OK,CK,_,_,OD,_), Part(PK,_,_,_,_), LineItem(OK,_,PK,SK,_,_), CK=ck, SK=sk, PK=pk, OD=od.

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

By applying this technique to the nodes CustSupp, CustSuppType, 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,PT) :- Order(OK,CK,_,_,OD,_), LineItem(OK,_,PK,SK,_,_), Part(PK,PN,_,PT,_), Input CK, SK.

Optimistic lookahead performs in one node an 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_,_,OD,_), LineItem(OK,_,PK,SK,_,_), CK=ck.

In the next node, CustSupp(CK,SK) we need to compute query Q12:

Q12(CK,SK,PK,PN) :- Order(OK,CK,_,_,_), Part(PK,PN,_,_,_), LineItem(OK,_,PK,SK,_,_), CK=ck, SK=sk.

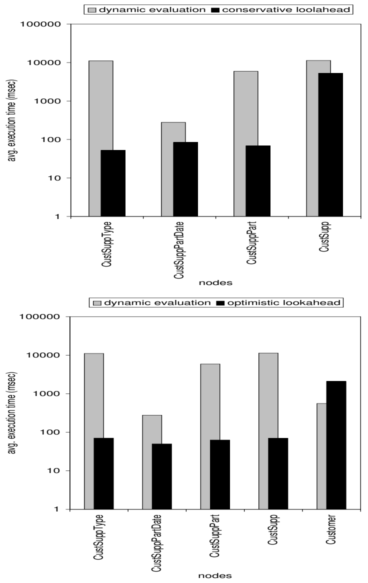

Figure 3: Benefits of lookahead computations.

Suppose we want to define a function for Q9 that also performs all the necessary computation for the query Q12. 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 Q12 not included in the intersection.

Specifically, we would define the function:

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

This function is updated in the node Customer but can also be used in the evaluation 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 the nodes CustSupp, CustSuppType, CustSuppPart and CustSuppPartDate.

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

Finally, figure 4 shows the performances of two complex run-time policies. In the upper figure we alloted enough space for the web site to be able to materialize a sizeable view (we allowed an additional 1GB to the original size of the database). In the lower figure we only alloted an additional 10MB. In the first case the run-time policy included the materialized join between Order and LineItem with 4 indexed columns. The view is used in almost all the nodes, and as a result, all the queries in the site ran in less than 8 seconds, and all but three

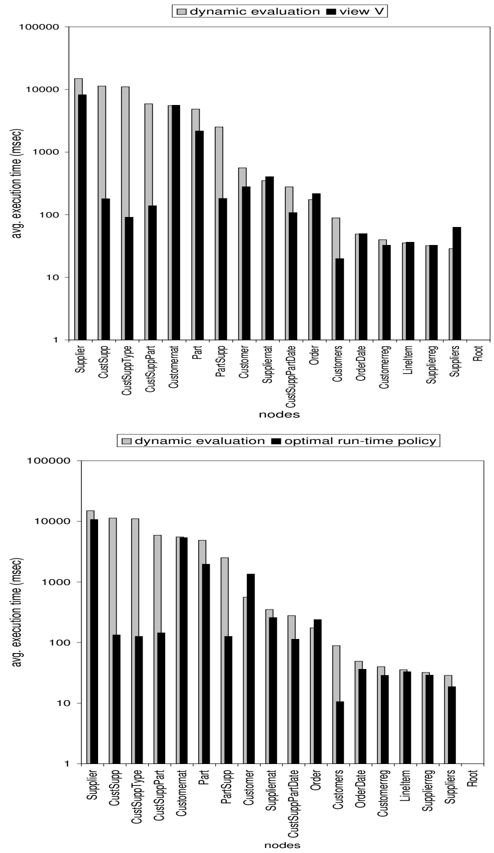

Figure 4: Results of two run-time policies.

in less than 400 milliseconds. In the second case the run-time policy includes a conservative lookahead in the node Supplier (which benefits the node PartSupp), and an optimistic lookahead computation in the node Customer which benefits the nodes CustSuppDate, CustSupp, CustSuppType and CustSuppPart. The running times of all the nodes are comparable in the two policies. This example highlights the savings obtained purely by exploiting the structure of the web site, with practically no memory overhead.

We note 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.

4 Conclusions

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.

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, PK, PN) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(PK, PN, _, _, _)
Q13(CK, SK, SN) :- Supplier(**SK**, SN, _, _, _), LineItem(OK, _, _, SK, _, _), Order(OK, **CK**, _, _, _, _)
Q14(CK, SK, CN) :- Customer(**CK**, CN, _, _, _), LineItem(OK, _, _, **SK**, _, _), Order(OK, CK, _, _, _, _)
Q15(CK, SK, PK, PN, PT) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(PK, PN, _, **PT**, _)
Q16(CK, SK, PK, PN) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, _, _), Part(**PK**, PN, _, _, _)
Q17(CK, SK, PK, OD) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, OD, _), Part(**PK**, _, _, _, _)
Q18(CK, SK, PK, PN, OD) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK**, _, _, **OD**, _), Part(**PK**, PN, _, _, _)
Q19(CK, SK, PK, OK, OD) :- LineItem(OK, _, PK, **SK**, _, _), Order(OK, **CK** _, _, **OD**, _), Part(PK, _, _, _, _)
Q20(PK, SK, SN) :- PartSupp(**PK**, SK, _, _, _), Supplier(SK, SN, _, _, _)
Q21(SK, PK, SN) :- Supplier(**SK**, SN, _, _, _), PartSupp(**PK**, SK, _, _, _)
Q22(CK, SK, PK, CN) :- Customer(CK, CN, _, _, _), LineItem(OK, _, **PK**, **SK**, _, _), Order(OK, CK, _, _, _, _)

Figure 5: Queries on the arcs of the site schema in example 2.1.

This paper described several techniques for optimizing the run-time behavior of web sites, and propose 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. In term of future work, the problem of developing compilation algorithms, which automatically translate a declarative specification into run-time policies, that are both efficient and produce high-quality run-time policies clearly deserves significant attention.

References

[1] G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, databases and webs. In *ICDE*, Orlando, Florida, 1998.

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

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.

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

[5] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

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

[7] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Run-time management of data intensive web-sites. INRIA Technical report. Available from http://www-rodin.inria.fr/ yagoub/pubs/paper.ps, 1999.

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

[9] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, 1999.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[11] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *SIGMOD*, pages 423–434, 1996.

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

[13] F. Llirbat, F. Fabret, and E. Simon. Eliminating costly redundant computations from sql trigger executions. In *SIGMOD*, pages 428–439, 1997.

[14] T. Nguyen and V. Srinivasan. Accessing relational databases from the world wide web. In *SIGMOD*, Montreal, Canada, 1996.

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

[16] J. Rao and K. A. Ross. Reusing invariants: A new strategy for correlated queries. In *SIGMOD*, pages 37–48, 1998.

[17] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

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