Caching Strategies for Data-Intensive Web Sites

Khaled Yagoub, Daniela Florescu, Valérie Issarny, Patrick Valduriez

INRIA-Rocquencourt

Domaine de Voluceau, 78153 Le Chesnay Cédex, France

{lastname.firstname}@inria.fr

Abstract.

Data-intensive Web sites serve large volumes of pages whose content is dynamically extracted from a database. Such Web sites have very high software development and maintenance costs and in general offer poor response times due to the heavy interaction with the database system. This paper introduces the Weave management system developed at INRIA, which alleviates the above shortcomings of data-intensive Web sites. Weave relies on the declarative specification of Web sites and offers a number of tools for the easy implementation, deployment and monitoring of the specified site. Weave features a customizable cache system that implements the optimal data materialization strategy according to the Web site's specifics: it can cache database data, XML fragments and HTML files. To explore Weave's performance we have built a Web site based on the TPC/D benchmark database using the WeaveBench test platform. We conducted a number of experiments with various data materialization strategies supported by our system. Results clearly show that in the general case, a mix of different caching policies is required to achieve optimal performance.

1 Introduction

Confronted to the rapid growth of the Internet and the need to quickly deploy effective solutions, many researchers have devoted their energy to improving Web performance by reducing client latency and bandwidth consumption and increasing servers scalability and availability. Proposed solutions include predictive prefetching, caching of Web objects, and the architecting of network and Web servers. Analyses show that existing solutions are beneficial but not yet satisfactory. Proxy caches are currently the most effective mechanism to improve Web performance, and yet traces clearly show that these caches only manage to attain a maximal hit rate of about 50% [27]. This limitation is mainly due to the dynamic nature of many HTML documents, which prevents them to be cached at the proxy level. Dynamic documents are typically generated using CGI scripts or they include the result of a query to a database. The flourishing of database-centric e-commerce applications is making the current state of affairs even worse, rapidly increasing the percentage of dynamic Web documents.

As more and more Web sites put their data content under the control of dedicated database management systems (DBMSs) to ensure data's persistence, availability and consistency, improving the access performance to database generated documents becomes a key issue in the improvement of the overall Web performance. This paper addresses the design, implementation and performance of data-intensive Web sites: Web sites that provide access to a large number of pages whose content is extracted from relational databases. Current data-intensive Web sites contain large amounts of ad-hoc code which is application and platform specific, leading to an unbearable system complexity. Complexity translates in very high development and maintenance costs and reduced optimization opportunities. We believe that it is necessary to provide development tools and construction methods based on site's high-level specification. Declarative Web site specifications can dramatically reduce development and maintenance costs while, by making the system's overall structure explicit, they help to automate the detection and deployment of performance improvement solutions. In this paper we describe Weave: a data-intensive Web site management system developed at INRIA[1] that is based on the above principle. Weave supports a high level methodology for the easy design, implementation, profiling and optimization of Web sites built on relational databases.

The performance problem of data-intensive Web sites lies in addressing the latency reduction of pages produced by the site. A system for which every single page would have to be constructed from scratch from the site's underlying DBMS, would surely have very limited performance and would definitively not scale very much. We believe that adequate materialization strategies are mandatory to attain reasonable and scalable performance out of data-intensive Web sites. Improving performance of data-

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

[1] http://caravel.inria.fr/Eprototype_WEAVE.html.

intensive Web sites through data materialization has so far been addressed in two ways: (i) materializing the results of frequently asked SQL queries [13], (ii) selectively materializing pages on the Web server [22]. Although both strategies perform well under certain circumstances, both suffer from lack of generality. The bottlenecks of data-intensive Web sites have large degrees of variation. They typically depend on the hardware and software environment, on the database statistics, and on the Web site's structure and access patterns [13]. We claim that there is no universal evaluation strategy which is optimal for all Web sites and which is independent of their particular parameters. Provisioning data-intensive Web sites requires a management system that is able to implement all the materialization strategies, and which selects the optimal strategy on a case-by-case basis according to the characteristics of the given Web site.

Weave offers a high-level language for Web site specification, and supports an innovative site architecture that is based on a 3-tier customizable cache system. The cache system can cache database data, XML fragments and HTML files, supporting data materialization at various levels. Data materialization is further tailored according to the database characteristics, Web site size, data freshness and response time constraints, and user access patterns. Before going into the particular solution of Weave, Section 2 discusses general issues related to the specification and implementation of data-intensive Web sites. Section 3 describes the particular solution adopted in the Weave system, describing the system's key elements for the easy construction of Web sites running optimal data materialization strategies. Section 4 discusses the experiments we have run so as to assess the benefits of Weave, analyzing the performance of a Weave Web site derived from the TPC/D benchmark database[2]. Section 5 then compares our solution with related work. Finally, Section 6 concludes.

2 Building Declarative Data-Intensive Web Sites

The general architecture of declarative data-intensive Web site management systems can be summarized by the following five fundamental principles:

1. The data content of the Web site is extracted from a (not necessarily dedicated) DBMS. The database can be either a primary database or the result of a data integration process, virtual or materialized.
2. The specification of the Web site is distinguished from its implementation. By specification of a Web site, we mean the description of the site's HTML pages, which must be separated from the code to be executed for solving page requests.
3. The specification of Web site's structure and content is separated from that of the graphical presentation of its pages. The former relates to the set of pages in the Web site, the data content attached to each page and the hyperlinks emanating from each page, while the latter concerns the page's layout.
4. The structure and the data content of the Web site are described in terms of a logical model. Many different models have been considered, most of them being (naturally) based on the notion of graph.

[2]TPCD Benchmark. http://www.tpc.org.

5. The mapping between the raw data and the logical model of the Web site is described via a declarative view definition language.

Concerning the separation between the Web site's content from the graphical presentation, note that this approach is already globally accepted (at least in theory) since the XML standard aims at isolating the data content of a Web site from the graphical presentation, usually described as independent XSLT programs. Hence, in the rest of the paper, we consider XML as the natural candidate language to describe Web site's structure and content and ignore the other possible Web site models. In this context, a Web site specification is done in two steps. First, the specification of the mapping between the raw data (relational in our case) and the Web site's logical model (XML in our case) has to be given. This is equivalent to defining an *XML view* over the relational data[12, 6]. Second, the definition of the Web site graphical presentation is given in terms of a set of XSLT programs.

2.1 Evaluation Strategies for Data-Intensive Web Sites

An important question that we address in this paper is whether it exists an optimal strategy for the evaluation of HTML pages. A multitude of strategies may be applied, ranging from purely dynamic to fully static evaluation.

Under a purely dynamic evaluation strategy, the Web server triggers the generation of an XML fragment corresponding to the requested page upon each request. In order to produce the result, the appropriate parameterized SQL queries have to be executed on the database, according to the site specification, and the result has to be packaged in XML format. The XML fragment is then sent to the HTML generator, which applies the appropriate XSLT program and generates the final HTML file. Under such an evaluation strategy, the total waiting time for a complete Web page can be decomposed as: (1) the *network communication time*, (2) the *HTTP connection time*, (3) the *Web application startup time*, (4) the *DBMS connection time*, (5) the *SQL execution time*, (6) the *XML generation time* and (7) the *HTML generation time*. In the case of dynamic evaluation of HTML pages from large databases, the entire process can be unacceptably slow.

Various solutions have been proposed to reduce the waiting time for a page in this context. Some solutions focus on reducing or eliminating one of the above 7 costs. For example, expensive CGI calls can be replaced by efficient APIs, or servlets. Furthermore, most products avoid systematic database connection through a pool of connections. However, to the best of our knowledge, no previous work studied the ratio between the various components of the response time and analyzed the real bottlenecks of such a system. It is clear that understanding these ratios is a mandatory step prior to considering any work on performance improvement and that the ratios will in general vary depending on the particular Web site. Hence, the local solutions presented above only slightly reduce the performance problem, but do not tackle completely the issues raised in the dynamic evaluation of Web sites.

A more general solution, used by most existing products [22] and prototypes, relies on materializing HTML pages. The materialization can be done either on the fly or off-line, before any user starts browsing. Despite good response times, the static (off-line) evaluation strategy has several major drawbacks. First, it incurs significant space overhead, which can be even greater than duplicating the entire database since the same data item may appear in multiple pages. Furthermore, the same HTML template is replicated for various pages. Second, propagating updates from the database to the Web site is a serious problem once the site has been materialized. Third, the materialization granularity (i.e. a page) is not always appropriate: different fragments in a page can have different update frequencies, and materializing at the page level imposes the recomputation of the entire page, even if some parts of the page did not change. Finally, the static approach cannot always be applied since it cannot accommodate forms (i.e. the page content depends on the values of some parameters which are only known at runtime).

The solution proposed in [13] relies on the observation that, in the case of pure dynamic evaluation, the parameterized queries issued from the Web server and executed on the DBMS server share much of their computation, leading to redundant work. The proposed solution is thus to cache in the DBMS, the results of parameterized SQL computation, under the form of relational tables called cache functions, and reuse the results for subsequent requests. This approach has several advantages. First, if the SQL execution time is the dominant cost and if this cost is high, there are significant performance improvements. Second, since the cached data and the raw data are under the control of the same DBMS, simpler update propagation mechanisms can be deployed. Third, it is possible to control the granularity of the cached data, ranging from entire parameterized SQL queries to simpler sub-queries or combinations of subqueries using outerjoins [13]. However, this solution may alter the Web site's performance under certain conditions. Since the cached data is under the control of the DBMS, every cache action (e.g. search, insert, delete) accesses the DBMS via expensive SQL statements: using a cache with a low hit ratio incurs a large penalty. The SQL execution time is not always the most prominent cost in evaluating a page and may even be negligible. In this case, the overhead of caching tuples in the DBMS may outweigh the performance improvement. Finally, another drawback of this solution is a possible overload the DBMS server.

An alternative to the above solutions is to cache intermediate XML representations of data. Compared to caching HTML files, XML caching has the advantage of storing less data. Moreover, XML representations allow for carefully controlling the granularity of cached data, ranging from complete pages to page fragments. For instance, we can cache the name of a product, which is somehow stable, but not its price which varies a lot over time. However, caching XML data instead of HTML files does not exhibit a clear advantage in terms of space saving when the ratio between the size of the XML representation and that of the corresponding HTML file is close to 1. Under such conditions, caching XML data would not bring any benefit in terms of space saving, and would additionally incur runtime overhead for converting XML data into HTML on the fly. Compared to DBMS caching [13], caching XML has the advantage of eliminating (in the case of a hit) the costs of database connection, SQL execution, and of generating XML data. This technique also allows to reduce the load generated on the DBMS by the Web server. On the other hand, update propagation from the DBMS to the cached data is made more difficult (e.g. [21]).

From the above, we can conclude that there is no universally good solution for improving the performance of a data-intensive Web site. Each of the above materialization techniques (i.e. HTML, XML or DB) will be effective under certain circumstances and disastrous under others. It is thus mandatory for data-intensive Web sites to flexibly support the materialization at all levels: HTML, XML, and DB data. Notice that better response times may be offered to clients by pipelining the operations required to produce an HTML page from a query. Unfortunately, despite some encouraging attempts [20, 26], streaming components are not yet available for XML and HTML generation. In addition, even when such components will be available, this will not solve the overall performance problem of data-intensive Web sites, which requires accommodating the high database and server load. Caching remains here a key solution to address this scalability issue.

2.2 Materialization Strategies

An agreed upon solution for reducing the cost involved in the pure dynamic evaluation of database-generated Web pages seems to be data materialization, reusing intermediate results of various computations to answer subsequent queries. In order to deploy such a solution, the following issues must be addressed.

1. What kind of data should be materialized? As we showed before, data go through various levels of abstraction between the data producer (i.e. relational tables) and the data consumer (i.e. HTML files). Hence, there is a choice of materializing either the result of relational queries (as tables on a DBMS), either XML fragments or directly HTML files.

2. When must materialization be performed? Possible answers to this question are: (i) data items are materialized *proactively*, before users start interacting with the Web site, (ii) data items computed *upon request* are cached, and reused for subsequent requests, (iii) data items are *prefetched* according to their probability of being accessed.

3. Where should the materialized intermediate results be placed for effective performance improvement? Data items can be stored within part or all of the following nodes: *database server*, *Web server*, *proxy*, and *Web client* (in the case of XML fragments and HTML files). Caching a particular data item among eligible nodes then depends both on behavioral information (e.g. access pattern) and on the processing capability of the given node (e.g. Web clients will not support XML generation in general).

4. How are updates from the database propagated to the materialized data? Updates can be propagated in either: (i) a *push* fashion, i.e. an update to the database *immediately* triggers the deletion or the recalculation of the materialized data invalidated by the update, or (ii) a *pull* fashion, i.e. the materialized data items are periodically checked for freshness, and the appropriate action is performed upon inconsistency. There are several comments to make about push *vs.* pull strategies. First, only the push strategy can guarantee up-to-date data delivery. In the case of a pull update propagation strategy, the Web site may eventually deliver outdated data, and this may be unacceptable for certain Web sites, while it can be clearly acceptable for others. On the other hand, a push strategy can only be deployed at the expense of using a costly trigger mechanism. Moreover, the applicability of the push strategy is drastically limited if the data is materialized outside the Web server itself, due to the knowledge that is required (e.g. which data items are materialized and where) [21].

5. Which particular data items must be materialized and which ones must be computed upon request? Intuitively, the data items (i.e. tables, XML fragments or HTML files) satisfying the following three criteria are good candidates for materialization: (i) they are expensive to compute, (ii) they do not require frequent recomputation for update propagation, and (iii) they are requested with high frequency. For the other data items, it is not obvious that the gain obtained from materialization outweighs the overhead.

The right answer to the above questions depends on the database itself, the Web site size, freshness and response time constraints, the Web site usage patterns, and, last but not least, on the particular hardware and software environment. The problem that we are addressing here is how to build a management system that can support all the above evaluation strategies and therefore being useful in most situations.

3 The Weave Web Site Management System

Weave has been designed to support the evaluation strategies described in the previous section. The Web site specification is given in Weave as: (i) a *WeaveL* program that describes the site's structure and content, and (ii) a set of XSLT templates that describe the site's graphical presentation. The WeaveL program describes abstractly the site's pages, the data attached to each type of page, and the links between pages. In the absence of any additional information, the specified Web site is interpreted by Weave as being purely dynamic, and is executed as such. For the cases where complex materialization strategies are desired, Weave offers an extension of WeaveL, called *WeaveRPL*, which allows the specification of complex runtime policies. A runtime policy prescribes which data have to be materialized and under which form, and how the updates are to be propagated from the database to the materialized data, etc. The Weave system includes all the necessary components to deploy and interpret at runtime such complex policies.

In this section we first detail Weave's declarative Web site specification. Subsequently, we describe the Weave site architecture, the associated language for runtime policy customization, and the current implementation of Weave.

3.1 Declarative Web Site Specification using WeaveL

A WeaveL program consists of a set of *site class* specifications. A site class models a collection of homogeneous pages in a Web site (e.g., collections corresponding to pages of customers or suppliers). Each page in the site can then be seen as an instance of a particular class, and identified and distinguished from the other instances by a set of items from the underlying database (e.g. a page of a particular customer can be identified by the key of the customer in the database). Hence, a page request handled by the Web server must specify the class of the page that is requested and zero or more parameter values, which uniquely identify the content of the requested page.

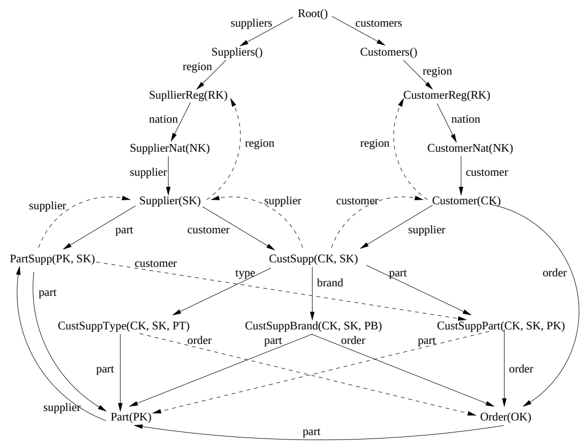

Figure 1: The TPC/D site schema

In WeaveL, the specification of each Web site class includes: (i) the declaration of the parameters identifying an instance of the class, (ii) the SQL query whose result gives all possible instances for the above parameters (describing how to produce all the instances of that class), (iii) the specification of the data contained in an instance of the respective class (i.e. the parameterized query that retrieves this data from the database), (iv) the specification of the hyperlinks emanating from an instance of the respective class (i.e. the database queries that have to be evaluated in order to build the correct links between the pages) and (v) the specification of the forms embedded in the page. Finally, in addition to the WeaveL program which gives the site's XML view definition, a complete Web site specification consists of a set of XSLT programs, all instances of a given class sharing the same XSLT.

Example 3.1 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 desired Web site is organized according to the hyperstructure represented in Figure 1. There is a root page with two links to suppliers (node labeled Suppliers()) and customers (node labeled Customers()). Both suppliers and customers are grouped by geographical region (e.g., CustomerReg(RK)), and within each region by nationality (e.g., CustomerNat(NK)). Suppliers and customers have further links to detailed information about the orders, as depicted in Figure 1. For illustration, we give below the WeaveL specification of the CustomerNat class. A Web page instance of this class depends on a single parameter (i.e. the key in the database of the given nation) and it contains (i) the name of the particular nation and (ii) hyperlinks to all the pages corresponding to customers in that nation. The mapping between the attributes and the hyperlinks contained in an instance of this class and data in the database is described using (parameterized) SQL queries as follows.

```
define class CustomerNat ($NK)
{instances using Q0 }
{
 data  nation_name using Q1 ;
 link customer to Customer($CK) using Q2 ;
}
define query Q0 as select nationkey as $NK from Nation;
define query Q1 as select name as nation_name
                    from Nation where nationkey=$NK;
define query Q2 as select custkey as $CK, name as anchor
                    from Customer where nationkey=$NK;
```

Given a page request (e.g. a particular binding for the parameter $NK of the class CustomerNat), it is necessary to first produce the XML fragment corresponding to the respective Web page. In Weave, the XML Generator has the task of evaluating the parameterized queries from the WeaveL specification and producing the corresponding XML data. An important feature of the Weave XML Generator is that it can be invoked with the request of generating complete XML pages or only some fragments of them. In the latter case the resulting fragment keeps track of the missing pieces, which are computed on the fly when needed. The XML data produced in Weave adhere to a unique DTD/schema, which is independent of the database schema and the structure of the Web site. For example, the complete XML fragment describing the content of the page identified by the class CustomerNat and $NK=6 is the following:

```
<XML_fragment id=" CustomerNat_6 ">
    <class> CustomerNat </class>
    <parameter> 6 </parameter>

    <data_fragment name=" nation_name ">
        <data_value> France </data_value>
    <data_fragment>

    <link_fragment name=" customer ">
        <link_item>
            <XML_fragment id=" Customer_402 ">
                <class> Customer </class>
                <parameter> 402 </parameter>
            </XML_fragment>
            <anchor> Customer#000000402 </anchor>
        </link_item>
        ....
    </link_fragment>
</XML_fragment>
```

3.2 Site Architecture

Declarative Web site specification only worries about what data will populate the Web site, and how the site is structured, but does not specify how the site is implemented. This separation of concerns is important because it gives the freedom of choosing the most adequate evaluation strategy. In this subsection, we detail the Weave site architecture enabling customized data materialization (or *runtime policy*); the next subsection details the WeaveRPL language for the specification of customization. The Weave site architecture is based on the following main components (see Figure 2):

- The *scheduler* has the task of interpreting the runtime policy, and coordinating the behavior of the other components. It receives HTTP requests and redirects them to the *cache manager* components out of which data relating to a given page may be retrieved.
- The individual *cache managers* enforce the runtime policy by handling data requests as forwarded by the scheduler. They further undertake appropriate actions regarding environmental constraints (e.g. management of data replacement and of data consistency). Environmental constraints are handled through the signal of associated events (e.g. time set for a timer has elapsed, maximum size set for a data container is reached).
- The individual *caches* (or *repositories*), which actually store the data and with which the cache managers interact. Notice that there may be various caches related to a given manager, e.g., for scalability purpose.
- The *XML generator* is in charge of issuing queries to the DBMS (including the DB cache) and producing XML fragments.
- The *HTML generator* generates HTML pages from XML fragments and XSLT programs.
- The *statistics manager* is in charge of storing and summarizing the data describing the Web site's runtime behavior. These data include: statistics about the Web site's traces and access patterns, statistics about the response times of the various Weave components (e.g. XML generator, XML cache, HTML cache), and statistics describing Weave caches usage (e.g. hit ratios, maximum size).

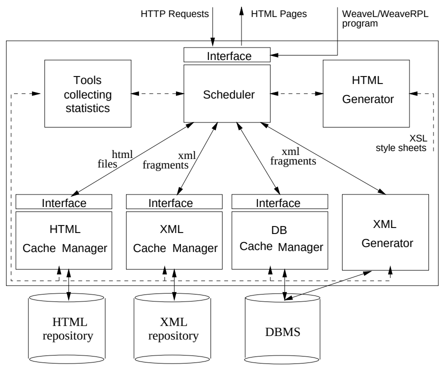

Figure 2: The architecture of the Weave system

Example 3.2 As an illustration of the interaction patterns among Weave site components, consider a request for a page whose associated data are asked to be cached as both XML and HTML. The page is first searched for within the HTML cache. If present, the page is simply returned to the scheduler, which returns it as a result of the HTTP request. If the page is absent from the HTML cache, the corresponding XML fragment is sought within the XML cache. If the fragment is present and corresponds to the entire page, it is returned to the scheduler, which computes the corresponding HTML page to forward it to both the HTML cache manager and the initiator of the HTTP request. Finally, if either the data is absent from the XML cache or the data retrieved from the XML cache corresponds to a subset of the page, the missing data is requested to the XML generator. The XML generator further requests the data to the DB cache manager, the data being ultimately retrieved from the underlying database. The retrieved data then serve computing the corresponding XML fragment, which is processed in order to update the

XML and HTML caches and to service the HTTP request. The above scenario illustrates the interaction among the components of a Weave site when all three data repositories are involved. It is then quite easy to infer alternative interaction patterns when only a subset of the caches is involved.

So far, we have introduced Weave sites with respect to the execution of a given runtime policy. The behavior of Weave sites is dynamically customizable with respect to the enforced runtime policy by taking as input WeaveRPL policy specifications. When the runtime policy is to be modified (or initialized), the new policy is first delivered to the scheduler, which distributes it among the Weave-specific components according to their functionality (see Figure 2).

3.3 Customizing runtime policies

Runtime policies specified using WeaveRPL give the behavior of the three data caches with respect to environmental constraints and data requests, which further implicitly describe the global algorithm that has to be followed in order to solve HTTP requests. This behavioral specification is similar for the three types of caches, and decomposes in three parts:

(1) The specification of the data items that are subject to materialization in each particular data cache. The set of items that are subject to materialization in each cache are logically grouped in a number of *containers*. A container is the unit of storage management and allows grouping together objects (i.e. tuples, XML fragments or HTML files) that have similar access and evolution patterns. Containers enable fine tuning of data materialization and provide a convenient basis for the physical distribution of caches. The definition of a container describes in a declarative fashion the items that are eligible for materialization in that container. In the case of XML or HTML, the items are identified by the name of a site class together with caching conditions upon its instances (e.g. parameter value, instance size, access frequency or computing time). In the case of database tuples, items are identified by the name of a table and a set of key values.

(2) The global constraints over all the data items to be cached in a given container. These constraints relate to the items' size (size), age (age) and access frequency (frequency).

(3) A set of Even-Condition-Action (ECA) rules, which dictates the way the cache manager responds to various events (e.g. request for data, size overflow, data aging, etc). Events that are currently supported relate to initialization (onInit), cache overflow (onFull parameterized with the container size), aging (onTimer parameterized with the timer value). Conditions that may be expressed are as for global constraints. Regarding supported actions, these include: prefetching specified data items within the cache (compute), removing specified items from the cache (remove), applying a given function to the cache content, e.g., for data replacement (apply), and reinitializing the cache content (reinit).

The above specification of containers drives the behavior of the scheduler when servicing HTTP requests. Given an object request, the global scheduler knows from which caches the object can possibly be retrieved, and how it must route the request in order to solve it completely.

Example 3.3 The TPC/D site definition is now complemented with the specification of the associated runtime policy. The specification associated with the HTML cache is given below.

```
Cache HTML:
  {  /* Container definition */
       define container contHTML as
          select Part,
             PartSupp(PK) where (PK < 100 and SK < 100),
             CustSupp(PK, SK) where (size < 2KB and
                                          frequency > 0.3);
       /* Global constraints */
          frequency > 0.2 and size < 10 KB;
       /* ECA rules */
          onInit compute PartSupp,
                         Part(PK) where PK < 100;
          onTimer(5mn)
                  remove Part(PK) where (PK > 100 and
                                          age > 30 mn);
          onFull(200MB) remove all where size > 2KB;
  }
```

The HTML cache contains a single container, which stores instances of the site classes that are listed after the select clause. Stored instances must have a size less than 10KB and an access frequency greater than 0.2. Further constraints are set over instances of the PartSup and CustSupp classes: instances of the former must have values for PK and SK that are both less than 100, instances of the latter must have a size that is less than 2KB and an access frequency that is greater than 0.3. Upon initialization (handling of the onInit event), the HTML cache is fed with the instances of the Part and PartSupp classes, which meet the aforementioned global constraints over cached instances. The content of the cache is refreshed every 5mn using the onTimer event; stored instances of the Part class whose value for PK is greater than 100 and whose age is greater than 30mn are removed. When the cache is full (space consumption greater than 200MB), all the stored instances whose size is greater then 2KB are removed (see handling of the onFull event). Specification for the XML cache follows.

```
Cache XML:
  {  /* Container definition */
       define container contXML as
          select Supplier(SK):fragments{name, customer}
          where SK < 100;
       /* Global constraints */
              frequency > 0.2 and size < 100 KB;
       /* ECA rules */
             onInit compute all;
             onTimer(5mn) reinit;
             onFull(50MB) apply LRU;
  }
```

The XML cache also contains a single container, which stores fragments of instances of the Supplier class whose value for SK is less than 100. The stored fragments correspond to parts of the HTML page that give the supplier's name and list of customers. Upon initialization, the cache is fed with all the instances that met the specified conditions until the cache is full. The cache is refreshed every 5mn by removing all the stored instances and recomputing them (reinit action). When the cache is full, a traditional LRU algorithm is applied for the replacement of the instances.

Similarly to the way an XML (resp. HTML) cache holds sets of XML fragments (resp. HTML files) that are logically defined by predicates and physically grouped into containers, a DB cache holds sets of tuples that are logically defined by predicates and physically grouped into tables. The tuples can either belong to some *materialized views*, or be part of *function tables*, which maintain the results of parameterized queries (with their respective inputs) [13]. The DB cache specification contains the definition of the views and/or the cache functions as illustrated below, each of them corresponding in our terminology to a single container. Both the materialized views and the cache functions are tables holding tuples. The difference resides in when the content of the tables is computed: statically (onInit) for the views and upon request (onRequest) for the cache functions. Moreover, views are complete (i.e. they contain *all* the tuples satisfying the given predicate), while cache functions can contain only a subset of the tuples satisfying the given predicate. Hence, both materialized views and cache functions can be specified using the same formalism (i.e. same container definitions and same ECA rules) as for the case of XML and HTML caches. More details about the DB cache can be found in [13].

```
Cache DB:
 { /* Container definition */
   define container CACHE_FUNCTION as
       select o.o_custkey, l.l_supkey, p.p_partkey, p.p_name,
         p.p_type, o.o_orderdate, o.o_orderkey, p.p_brand
       from LINEITEM l, ORDER o, PART p
       where o.o_orderkey = l.l_orderkey and
                p.p_partkey = l.l_partkey
   input o_custkey, l_supkey;
     /* ECA rules */
   onRequest(CustomerSupplier($CK,$SK))
       compute all
           where o_custkey=$CK and l_supkey=$SK;
   onTimer(30mn) remove all;
 }
 { /* Container definition */
   define container VIEW as
       select o.o_custkey, l.l_supkey, l.l_partkey, o.o_orderdate,
             o.o_orderkey, l.l_linenumber
       from ORDER o, LINEITEM l
       where o.o_orderkey = l.l_orderkey
   input custkey, supkey;
     /* ECA rules */
   onInit compute all;
   onTimer(24h) reInit;
 }
```

To conclude this section we note that our final goal is to offer a system, which would analyze the declarative specification and produce automatically the "optimal" runtime policy. At this point, our system still requires a Web site administrator to generate by hand the desired runtime policies. However, in order to help the administrator choosing the best materialization strategy, Weave offers a powerful testing and tracing component. For example, this component analyses and summarizes the Web site execution statistics, and highlights the potential performance problems, hence simplifying considerably the Web site administrator's task.

3.4 Prototype implementation

The Weave site architecture is highly modular, allowing the use of off-the-shelf components for most of the architecture elements but the Weave-specific ones, which are the scheduler and the cache managers. All the Weave site components, presented in Figure 2, are wrapped in Java (JDBC is used for interfacing with the DBMS), using simple but powerful interfaces. Implementing Weave then consists of providing implementations corresponding to the component interfaces, together with the WeaveRPL compiler. In the current Weave implementation, except for the Weave-specific components, we have either reused existing components when available (i.e. IBM LotusXSL for the HTML generator, and IBM XML4J as part of the XML generator, which we enriched for supporting the generation of XML fragments relating to part of a page), or implemented in a quite trivial way some of the others (i.e., we use the underlying file system for the HTML and XML caches).

Particular attention has been devoted to the design and implementation of the three cache managers. Weave contains a single cache manager interface and a single cache manager implementation, which proved to simplify a lot Weave implementation. The three data repositories holding the materialized data share the same interface; changing the real data repository requires only to change the implementation of this interface. This feature allows us, for example, to experiment with several XML repositories (persistent DOM, Excelon and files).

Notice that the specification of XML views over relational databases and their efficient implementation, is a distinct area of active research [26, 12, 6]. Although applicable to the Web domain, the proposed solutions are not specifically designed for it. In particular, they do not allow manipulating entire XML elements and/or fragments. However, Weave is designed such that the XML Generator can easily be replaced in the future when powerful and efficient such components become available. Finally, notice that the current implementation of Weave supports only a subset of all the materialization strategies that were discussed in Section 2. The main limitation concerns data update propagation, currently offered only through a pull model.

3.5 Weave on the Web

Up to this point, we have been concentrating on the use of Weave for building sites at the Web server level. In addition, we have presented an example of Weave site incarnations with exactly one instance of each type of cache (e.g. DB, XML and HTML), all three caches being used and further located on a single site. Unsurprisingly, Weave has been designed to enable the deployment of sites with different configurations, in term of cache components and their location over the various Web nodes (i.e. Web server, proxy, client). We do support the distribution and replication of the Weave site's components over these nodes, and the implementation of sites comprising only a subset of the three caches. Among the benefits of Weave in this context, Weave site components are valuable for coping with thin Web clients (e.g. wireless PDAs), which are foreseen as future prominent Web actors. Dedicated proxy caches may exploit the XML fragments for the convenient customization of Web pages. Due to the lack of space, we do not discuss any further the usage of Weave in the overall Web environment.

4 Experimentation

To measure the performance of various data materialization strategies, we built a test platform called WeaveBench and performed various experiments. In the following, we present the test platform and its configuration, the experiments and the performance results.

4.1 The WeaveBench test platform

We built our own test platform WeaveBench. WeaveBench generates a load for testing Weave on a Web server by running one or more Web client processes on one or more client computers. Each client process sends requests as fast as it can receive data back from the server. Thus, it generates a load much heavier than that of a single interactive user. A single process manages the testing done by the client processes. It starts the benchmark runs, each one by a different client process, and combines the performance results into a single summary report.

WeaveBench considers a Web site derived from the TPC/D database factor 1 with a database size of 1.2 GB and 15 million pages. The Web site contains in average 5 SPJ queries per page, and no aggregate queries have been used. Each client submits page requests based on a trace file that describes the pages of the TPC/D database. Thus, we must generate a trace file that captures a realistic workload of page accesses. In the experiments, each client sends requests to the Web server according to a trace file. All the trace files for a given run are generated based on the same probability distribution, as follows. First, a workset of N (N=10.000 in the current implementation) distinct pages have been chosen from the entire Web site. In so doing, the pages closer to the root of the Web site are considered with a (slightly) higher probability. Second, C (C being the number of clients) sequences of length M (M=1000 in the current experiments) of pages are chosen from the workset, according to a Zipf distribution.

The database is stored in Oracle v8 on a dedicated Ultra Sparc I machine (143MHz and 384MB of RAM), running SunOS Release 5.5. The cache global manager, the three caches and the Web server, Apache v3.3.3, are all on the same machine, a 300MHz Pentium with 520MB of RAM, running Linux Release 6.1. We use the Apache JServ servlet engine to run the cache global manager, which we implemented as a servlet. Two other machines are used to hold WeaveBench clients. Each machine, a 200MHz Pentium with 96MB of RAM, ran about 1-50 clients in increments of 10. All the machines in the test platform are interconnected by a 10Mbps network that can be isolated from other networks.

4.2 Experiments

We have run the following experiments: dynamic evaluation, static[3] evaluation, DB caching only, XML caching only, HTML caching only, and mixed caching. Notice that the dynamic evaluation is the worst case scenario where client requests always yield access to the database, generation of the XML fragment and generation of the HTML page. On the other hand, the static evaluation is the "ideal" scenario where all the HTML files have been statically generated and put in the HTML cache. This scenario is unrealistic because of the high costs for generating (after each change) and storing the files. For DB caching, XML caching, and HTML caching, the goal is to measure the respective benefit of caching data within the database, the XML cache, and the HTML cache. Finally, mixed caching is the realistic scenario where all three caches are used, each one for a different type of page according to the particular parameters. The goal is to assess the benefit of caching at various levels (database, XML fragment, HTML files).

The results are presented as a two-dimensional table where each row corresponds to a page class (Customer, Supplier, etc.) and gives the percentage of the various execution times (wsconnect, queryexec, etc.) out of the *total Web server response time* for the instances of that class. The table is sorted by decreasing order of response time. We highlighted in bold faces the components that cause performance problems, i.e. the components with execution times that are for more than 30% of the overall processing. All results are shown for 1 client generating 500 page requests and in the absence of data updates. Experiments with up to 50 clients showed large degradations of the HTML generation time. This is mainly due to the fact that existing XSLT processors consume large amounts of memory and do not scale properly. These results suggest that Web site architectures based on XML and XSLT can be used in real applications only when better XSLT processors will be available. However, there is no doubt that this will be the case in the future since efficient XSLT processing is receiving significant attention.

4.3 Performance results

Dynamic pages. Table 1 shows the results when there is no caching. The average response times are between 15 s[4] and 300 ms. For page classes Customer, Supplier, CustSup, CustSuppPart and CustSuppBrand, the query execution time dominates because the queries involve joins with large tables. For page classes CustomerNat and SupplierNat, the XML and HTML generation times dominate because the queries are simple but produce a large amount of data. For the other page classes, the response time drops to 500 ms and below, and is divided between the Web server connection and the HTML generation. These page classes need not be optimized and, thus, they are not shown in the experiments results.

Precomputed pages. Table 2 shows the results when the HTML files have been statically generated. Thus, the only relevant execution times are those for connecting to

[3]In this experiment, all the Web pages have been precomputed. However, they are still served through the Weave application, and not by the Web server directly from the file system.

[4]The large response time can be explained by the hardware configuration used for the experiments.

page class	wsconnect(%)	queryexec(%)	xmlgen(%)	htmlgen(%)	resptime(ms)	size(KB)
Customernat	1.52	0.81	**44.36**	**53.32**	15202.34	660
CustSupp	1.84	**97.38**	0.04	0.74	12530.95	2
CustSuppBrand	1.85	**96.73**	0.05	1.37	12459.98	2
CustSuppType	1.91	**96.57**	0.05	1.47	12063.53	2
Supplier	2.30	**82.96**	6.36	8.38	10044.03	100
CustSuppPart	2.52	**96.51**	0.06	0.92	9160.41	2
PartSupp	8.23	**88.40**	0.28	3.09	2802.67	1
Suppliernat	22.30	3.63	25.78	**48.29**	1034.73	60
Customer	30.26	**43.80**	6.73	19.21	762.52	15

Table 1: The results of the dynamic evaluation

page class	wsconnect(%)	transftime(%)	resptime(ms)
Customernat	0.35	**99.65**	3027.30
Supplier	3.30	**96.70**	323.60
Suppliernat	4.87	**95.13**	219.04
Customer	**31.12**	**68.88**	34.30
CustSuppBrand	**70.22**	29.78	15.20
CustSuppType	**70.65**	29.35	15.11
CustSuppPart	**73.37**	26.63	14.55
CustSupp	**77.46**	22.54	13.78
PartSupp	**78.21**	21.79	13.65

Table 2: The results of the static evaluation

the Web server and transferring the HTML files to the client (transftime). Response times are optimal and are between 3 s and 13 ms for the most expensive pages. For page classes of larger HTML size (CustomerNat, Supplier, SupplierNat and Customer), the time to transfer the HTML files dominates. For the others (2 KB and below), the Web server connection time dominates and the response time gets quite small.

DB caching. Table 3.**a** shows the results with DB caching only. We have used three cache containers: one for pages of class Customer only, one for pages of class Supplier and PartSupp and one for pages of class CustSupp, CustSuppPart, CustSuppType and CustSuppBrand. The selection of these containers is close to optimal and was found after several trials. No cache was used for the other page classes because the number of pages or the locality of reference are low. Thus, the results are not shown for those because they are the same as in Table 1. For pages of class CustSupp, CustSuppPart, CustSuppType and CustSuppBrand, the improvement in response time is quite significant (factor 11 or more) The response time for pages of class Supplier has been improved by 2.4. However, that for pages of class PartSupp has been worsened by a factor 1.5 because the hit ratio does not compensate for the cost of inserting tuples in the container.

XML caching. Table 3.**b** shows the results with XML caching only. In our configuration, the XML cache takes memory away from the Web server. XML caching essentially improves on query execution time and XML generation time. For page classes with a good hit ratio (CustSuppType, CustSupp, CustSuppBrand, CustSuppPart, Supplier and PartSupp), the response time can be improved by a factor between 3 and 7. For the other page classes, there is either small improvement or slight degradation due to competing memory access by the Web server. Note that our implementation of an XML cache could be improved by using a persistent XML store. Compared to DB caching, the improvement is not as good mainly because of the relatively high memory consumption.

HTML caching. Table 3.**c** shows the results with HTML caching only. HTML caching essentially improves on HTML generation time, in addition to the improvements of XML caching. The performance improvement is slightly higher than that of XML caching because the HTML cache consumes less memory (all files are stored on disk). But the improvement is still not as good as that of DB caching.

Mixed caching. Table 3.**d** shows the results with a mixed strategy where we combine DB caching, XML caching and HTML caching. For each cache, the decision of what to cache and how was taken according to the observations made in separate experiments with either DB, XML or HTML caching only. We use the DB cache for pages of class CustSupp, CustSuppPart, CustSuppType and CustSuppBrand. However, we use the XML cache for Supplier and Customer pages. We cache only the fragments corresponding to the customers of a given Supplier and suppliers of a given *Customer.* We assume that these fragments are not frequently updated and worth being cached. The other fragments (i.e. parts of a Supplier and orders of a Customer) are supposed to be frequently updated and so should be built on demand. Finally, the other page classes, except pages of class PartSupp which are computed on the fly without any caching, are statically generated and put in the HTML cache. As a result, the performances are much better than with each cache alone. The highest improvements in response time are with the DB cache (factor 11 or more). The improvement of precomputed pages (e.g., CustomerNat, SupplierNat, etc.) and of XML caching for pages of classes Supplier and Customer is also significant. The results for the uncached pages remain the same as in Table 1. To summarize the results, Figure 3 gives the response times of the different caching strategies for our TPC-D Web site and clearly shows that caching at different levels is a good strategy.

5 Related Work

There is a number of efforts towards easing the construction of data-intensive Web sites through declarative specification [3, 11, 2, 9, 14, 7], and towards optimizing the response times offered by such sites [13, 22]. However, to the best of our knowledge, there is little work addressing both issues in conjunction, which we consider as mandatory

page class	wsconnect(%)	queryexec(%)	xmlgen(%)	htmlgen(%)	resptime(ms)
a. Results of DB caching					
PartSupp	5.42	**93.50**	0.18	0.89	4244.72
Supplier	5.64	**58.12**	16.30	19.93	4080.14
CustSupp	26.78	**69.08**	0.53	3.62	859.64
CustSuppPart	28.70	**65.86**	0.67	4.78	802.09
CustSuppBrand	30.22	**62.20**	0.71	6.88	761.79
CustSuppType	**34.74**	**56.94**	0.78	7.54	662.60
Customer	**38.26**	**35.57**	8.63	17.54	601.61
b. Results of XML caching					
Customernat	3.45	0.36	22.18	**74.00**	7337.22
CustSuppType	8.02	**89.27**	0.07	2.64	3157.17
CustSupp	8.37	**90.57**	0.07	0.01	3025.36
CustSuppBrand	8.79	**89.58**	0.07	1.55	2880.49
CustSuppPart	10.01	**88.39**	0.10	1.50	2528.84
Supplier	13.7	21.65	18.09	**46.53**	1843.35
Suppliernat	28.44	1.47	16.67	**53.42**	890.21
Customer	**60.27**	9.46	5.82	24.45	420.05
PartSupp	**68.50**	20.83	0.91	9.76	369.61
c. Results of HTML caching					
Customernat	3.33	0.38	23.40	**72.89**	7517.07
CustSuppType	8.16	**91.02**	0.07	0.75	3071.73
CustSupp	8.62	**90.80**	0.06	0.51	2905.92
CustSuppBrand	9.20	**89.92**	0.07	0.80	2723.02
CustSuppPart	10.96	**88.10**	0.10	0.84	2286.69
Supplier	13.46	24.24	18.51	**43.79**	1862.03
Suppliernat	29.01	1.53	16.70	**52.76**	863.83
PartSupp	**46.33**	**49.66**	0.61	3.39	540.91
Customer	**62.51**	10.66	6.10	20.73	400.92
d. Results of mixed caching					
Customernat	8.07	0.00	0.00	**91.93**	3010.50
PartSupp	8.84	**89.55**	0.29	1.32	2747.78
Supplier	11.63	25.95	18.19	**44.24**	2090.05
CustSupp	29.69	**66.01**	0.57	3.73	818.44
CustSuppPart	30.77	**63.83**	0.62	4.78	789.89
CustSuppBrand	32.02	**60.83**	0.69	6.46	758.88
CustSuppType	**39.82**	**50.23**	0.87	9.08	610.32
Customer	**52.83**	6.71	7.57	**32.89**	460.02
Suppliernat	**97.51**	0.00	0.00	2.49	249.10

Table 3: Results of the dynamic materialization strategies

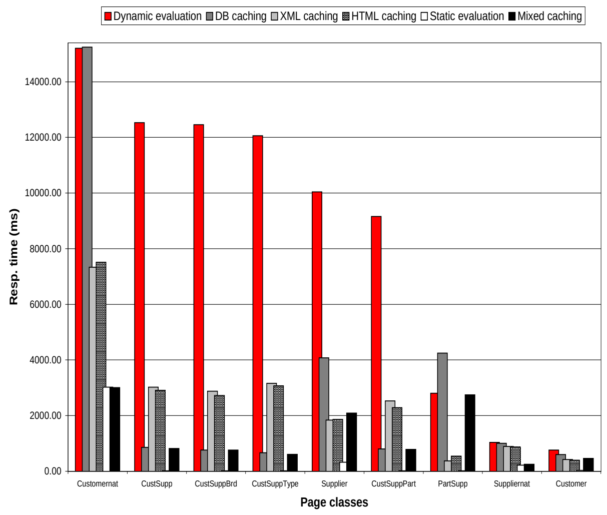

Figure 3: Comparison of different caching strategies

for the effective deployment of data-intensive Web sites. In this context, the work proposed in [13] exploits declarative Web site specifications for data caching and lookahead computation within the database system according to the users' access patterns and the update frequency of the data items. This improves the performance of handling database queries, allows for efficient update propagation, and enables caching of data that are shared among various pages. However, this solution addresses only DB caching, which has been shown to be valuable only in certain circumstances.

Related work originates from research results in the Web area, which have been investigating ways to overcome the performance penalty caused by uncacheable dynamic objects. Solutions have been proposed at the level of proxy and server caches. Weave shares common ideas with some of the solutions proposed for proxy caches. In a way similar to combining XML fragments with XSLT programs, the proposal of [10] argues for dividing dynamic objects into a static part (or template) and a dynamic part. Retrieval of the former may then benefit from caching while retrieval of the latter leads to Web server access. In this way, the performance cost of accessing a dynamic object is reduced to the minimum. A more general solution is the introduction of cache applets [5] which customize the caching policy for each document. Cache applets enable Web servers to attach computation with Web objects, which may be conveniently exploited by proxy caches (e.g. the proxy cache may request the server only for the document part that is actually dynamic). Close to this work is the proposal of [4], which introduces the CacheL language for customizing cache management policies according to the specifics of accessed documents and/or services. Regarding proxy cache management, Weave offers features close to the above solutions. Weave additionally deals with the improvement of the Web server latency, which is not addressed by the above work. This aspect is mandatory since, in the case of query processing, the load offered to the Web server remains unchanged and hence still requires adequate management at the server level for effective performance improvement.

Caching of dynamic objects at the level of Web servers has been investigated in [21], which introduces an algorithm for efficient update propagation to caches. It is a first step towards improving Web performance when handling database queries. However, it is aimed at a specific service, i.e., the 1998 Olympic Game Web site. In particular, the targeted service allowed for storing in memory all the dynamic pages without overflow, yielding a cache hit rate close to 100% without applying a replacement algorithm. This assumption cannot be made for all the Web servers interfacing with databases, especially data-intensive Web sites. Furthermore, this solution is a very specific component of the entire Web site, which may easily be integrated within Weave sites given their architectural modularity. On the other hand, our solution is general, providing methods and tools for the easy design, implementation, deployment, and maintenance of efficient data-intensive Web sites.

Close to the Weave system are products that aim at easing the development of Web-based applications. In particular, we find Microsoft Active Server Pages [19], and Sun JavaServer Pages [18] that are component-based architectures offering a number of base components to build Web servers delivering dynamic pages. However, it is up to the developer to tailor the site's implementation by providing the needed additional components. The IBM Websphere [15] and ColdFusion [16] are Web application servers that ease the development of Web sites through the provision of powerful base components for HTML caching, database access, and scheduling. However, customization of the Web site's runtime policy is less flexible than in Weave and is disseminated in the implementation of the various components composing the site. This alters the ease of specification and maintenance of the caching policies; it also diminishes the chances that such caching policies will be generated automatically in the future.

Dynamai [17] is a configurable cache allowing caching dynamic documents. Dynamai sits in front of a site's Web application and intercepts HTTP requests. If a request comes in for content not yet in the cache or that Dynamai cannot cache, Dynamai passes the request to the Web server, caching the response in the first case and ignoring it in the second case. Dynamai allows the site administrator to identify cache-able dynamic content, declare the events that may result in changes to the content, and, notify the cache when these events occur. Weave is close to Dynamai from the standpoint of offering support for the effective caching of dynamic objects. However, Weave is a superset of this product in that it is not only a caching tool, but also provides a framework for the specification of data-intensive Web sites. We believe that a global knowledge of the data supply chain (i.e. from the data producer all the way to the data consumer) is needed in order to derive the appropriate caching policy. By ignoring the source of the data and the way the Web site is generated only limited solutions can be found.

Finally, three other systems explore issues that nicely complement our work. In [7] the authors propose a high level Web site specification model, more powerful then the one we currently support. However, they did not address the performance problem and the supported Web evaluation strategies are the simple ones: static and dynamic. In [22] the authors address the optimization problem. For each particular page, the disadvantages obtained from materialization are compared with the performance improvements, thus deriving an "optimal" materialization strategy. However, due to the large number of (parametrized) Web pages the proposed optimization method cannot be easily adapted to our context. The space of possible materialization policies examined in [22] is also significantly more restricted then the one we would like to explore. Finally, in [1] the authors propose an interesting high level specification methodology for e-commerce applications based on a rule-based language. These systems complement our own; we believe that a complete solution for *fast* and *high quality* Web application deployment can only be obtained by an harmonious combination of these technologies.

6 Conclusions and Future Work

In a data-intensive Web site, returning a Web page may require costly interaction with the database system (connection and querying) to dynamically extract its content. The database interaction cost adds up to the non-negligible base cost of Web page delivery, thereby increasing much the response time. Although useful, techniques for proxy and Web server caches do not help reducing the Web latency in this case since they work at the level of HTML pages. In this paper, we have addressed the performance problem of accessing dynamic Web pages in data-intensive Web sites. This work has been done in the context of the Weave Web site management system developed at INRIA.

Our approach relies on the declarative specification of the Web site through a logical model, and the customization of the site's data materialization strategy based on high-level specification. The logical model of the Web site is based on the XML graph data model. A site schema then represents an XML view definition over a relational database. Thus, we can manage the data of the Web site at three levels: database, XML fragments, HTML files. In this context, we have proposed a customizable cache system architecture and its implementation. Our solution enables data caching at the various levels of data elaboration within the Web site: database data, XML fragments or HTML files. In addition, Weave comes along with the WeaveRPL language for specifying both the Web site's content and customized data materialization within the site. Given a site graph, cache management may be specified at a fine grain by attaching caching actions to each site class. Furthermore, the Web site being specified abstractly, the site's developer is relieved from dealing with low-level implementation details, which further eases the site's maintenance and evolution. Our solution has been illustrated using a Web site derived from the TPC/D benchmark database. Based on the result of our experiments, we have assessed the performance of various caching strategies: dynamic pages (worst case), precomputed pages (best case), DB caching, XML caching, HTML caching, and mixed caching (combining DB, XML and HTML caching). The results clearly show that a mixed strategy is generally optimal.

This paper has presented the building blocks of Weave, which we have further implemented so as to assess our approach. Work still needs to be done for further assessment and improvement of Weave. As a short term objective, we are currently working on the enhancement of the current Weave prototype with respect to optimizing the performance of the components composing a Weave site. As longer term research objectives, we plan to investigate four directions. First, we would like to eliminate a strong limitation of our current system: the fact that our declarative specifications can only model Web sites that *read* data from the database, but not Web sites that *update* a database. It is particularly important (especially for e-commerce Web sites) to be able to model in a declarative fashion data transfers in both directions, from the database to the the Web site and vice versa. Supporting the TPC-W benchmark is our next goal and this functionality will be required. The second interesting direction is to consolidate the replication and distribution of the Weave components on the proxies and/or on the clients. Finally, it is important to be able to generate the run-time policies automatically, based on the information extracted from the execution statistics, from the database statistics and from the Web site constraints in terms of data freshness and response time.

Our ultimate, and ambitious, goal is to obtain a self adaptive Web site management system which dynamically changes its own run-time policies in response to the behavior of a running system.

Acknowledgements: The authors would like to acknowledge Cezar Cristian Andrei for his participation to the implementation and evaluation of the Weave prototype.

References

[1] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active views for electronic commerce. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[2] G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, database and Webs. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 1998.

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

[4] J. F. Barnes and R. Pandey. Providing dynamic and customizable caching policies. In *Proc. of the USENIX Second Symp. on Internet Technologies and Systems*, 1999.

[5] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proc. of IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 1998.

[6] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *Proc. of the Int. Workshop on Web and Databases (WebDB)*, 2000.

[7] S. Ceri, Piero, Fraternali, and A. Bongio. Web modeling language (WebML): a modeling language for designing Web sites. In *Proc. of the Int. World Wide Web Conf.*, 2000.

[8] B. Chidlovskii and U. M. Borghoff. Semantic caching of Web queries. *VLDB Journal*, 9(1):2–17, 2000.

[9] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 1998.

[10] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proc. of USITS'97 – USENIX Symp. on Internetworking Technologies and Systems*, 1997.

[11] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a Website management system. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 1998.

[12] M. Fernandez, W.-C. Tan, and D. Suciu. Silkroute : Trading between relations and XML. In *Proc. of the Int. World Wide Web Conf.*, 2000.

[13] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run time management of data intensive Web sites. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[14] P. Fraternali. Tools and approches for developing data-intensive Web applications: a survey. *ACM Computing Surveys*, 1999.

[15] http://www-4.ibm.com/software/webservers/appserv/.

[16] http://www1.allaire.com/Products/coldfusion/.

[17] http://www.dynamai.com/home.html.

[18] http://www.java.sun.com/products/jsp/index.html.

[19] http://www.microsoft.com/.

[20] Z. G. Ives, A. Y. Levy, and D. S. Weld. Efficient evaluation of regular path expressions over streaming XML data. Technical Report UW-CSE-2000-05-02, University of Washington, 2000.

[21] A. Iyengar, J. Challenger, and P. Dantzig. A scalable system for consistently caching dynamic Web data. In *Proc. of IEEE INFOCOM*, 1999.

[22] A. Labrinidis and N. Roussopoulos. WebView materialization. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2000.

[23] Q. Luo, J. F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active query caching for database Web servers. In *Proc. of the Int. Workshop on Web and Databases (WebDB)*, 2000.

[24] T. Nguyen and V.Srinivasan. Accessing relational databases from the World Wide Web. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 1996.

[25] L. Quan, L. Chen, and E. A. Rudensteiner. Argos: Efficient refresh in an XQL-Based Web caching system. In *Proc. of the Int. Workshop on Web and Databases (WebDB)*, 2000.

[26] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficient generating XML documents from relational data. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2000.

[27] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative Web proxy cahing. In *Proc. of SOSP'99 – 17th ACM Symp. on Operating Systems Principles*, December 1999.

[28] K. Yagoub, D. Florescu, V. Issarny, and C. Andrei. Building and customizing data-intensive Web sites using Weave. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2000. (software demonstration).