Building and Customizing Data-Intensive Web Sites using Weave

Khaled Yagoub, Daniela Florescu, Valérie Issarny, Cezar Andrei*
INRIA-Rocquencourt
Domaine de Voluceau, 78153 Le Chesnay Cédex, France
{firstname.lastname}@inria.fr

1 Overview

We call a *data-intensive* Web site a Web site that provides access to large numbers of pages whose content is dynamically extracted from a database. In this context, producing a Web page may require costly interaction with the database system for connection and querying. The database interaction cost adds up to the non-negligeable base cost of Web page delivery, thereby increasing much the client waiting time.

Various solutions have been proposed to improve Web performance by reducing the waiting time for a page. These solutions include predictive prefetching, caching of Web objects, and architecting the network and Web servers for scalability and availability [1]. Among these works, Cao et al. [3] introduce cache applets, which enable Web servers to attach a piece of Java code to each dynamic document. This code is run whenever a request for a cached document is received. A cache applet can either rewrite the cached document and return it, or request the cache to either fetch or regenerate the document. Close to this work, Barnes et al. [2] propose a domain-specific programming language, CacheL, for defining customizable caching policies. The language allows defining how objects are cached, replaced, and kept consistent. While these solutions present benefits, they remain insufficient to improve the Web latency of pages built from database content when the query execution cost dominates.

We demonstrate Weave[1], a data-intensive Web site management system developed at INRIA. The system addresses the performance problem of accessing dynamic Web pages in the case of sites whose content is derived from relational databases.

*The current address of this author is "Politehnica" University of Bucharest. The work has been done while Cezar was visiting INRIA.

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

In a previous research, we addressed the problem of runtime management of data-intensive Web sites [4]. The proposed solution was to cache in the database the results of parameterized queries, under the form of relational tables, and reuse the results for subsequent requests. This improves performance of handling database queries, allows for efficient update propagation, and enables caching of data that are shared among various pages. However, this solution can alter the Web site's performance in the case of a low hit ratio or when the query execution time is not the most prominent cost (every cache action access the database via expensive SQL statements).

In the current version of Weave, we generalize this work. Our goal is to reduce the response time for serving page requests through an appropriate *customizable* cache system. Our solution enables data caching at various levels of data elaboration within the site. The system allows to cache the results of *database queries*, intermediate *XML fragments* and *HTML files*. In addition, it provides a *declarative language* (*WeaveL*) for specifying Web site structure and content, and an extension of this language for specifying the customized cache management policy within the site.

2 System description

Figure 1 depicts the architecture of the Weave system. In the following, we briefly discuss the key features and the corresponding components of Weave.

Declarative Web site specification. Weave [4] adopts the XML graph data model to describe the structure and the content of the Web site independently of its graphical representation (different models can be seen in [1]). An instance of this data model corresponding to a particular Web site is an *XML site graph*, which is a directed labeled graph with two types of nodes: internal nodes corresponding to Web pages, and leaf nodes corresponding to data values attached to pages. Links between pages are modeled as arcs between the internal nodes representing them in the graph. An XML site graph is defined *intensionally*, via an *XML site schema*, rather than extensionally (i.e., one Web page/XML fragment at a time). Therefore, a site schema represents nothing else than an XML view definition over a database. In this view, the Web pages are classified into

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

homogeneous collections called *site classes*. Applying the site schema to a particular instance of the database results in a complete XML site graph. The graphical representation of the site is described using XSL style sheets.

Two components are fundamental in the Weave system: the XML Generator, which applies the definition of the site schema to the underlying data and produces (fragments of) the XML site graph, and the HTML Generator, which applies XSL style sheets to XML fragments, resulting in browsable HTML pages.

The WeaveL language. XML site schemas are described using the WeaveL language. A WeaveL program consists of a set of site class specifications. Each class specification contains the declaration of the parameters identifying an instance of the class, the SQL query whose result gives all possible instances for the above parameters (describing how to produce all instances of the class), the specification of the data contained in an instance, and the specification of the hyperlinks from an instance of the respective class.

Suppose we want to produce a browsable version of the data contained in the TPC/D benchmark [4]. A fragment of a program written in WeaveL is presented bellow. It describes Supplier pages, where each such page contains the name of the supplier, a set of links to the pages of his customers, and a form allowing a user to search information about the parts the supplier produces. The query given in the clause instances specifies how to obtain from the database all the possible values of the parameter $SK. This information is needed for the static evaluation of the Web site.

```
define class Supplier($SK)
{instances using Q0}
{
      data name using Q1;

      link customer to Customer($CK) using Q2);

      form parts
       input $partname text
       link to PartList($partname);

}
// where Q1, Q2,... are (parameterized) SQL queries defined as:

define query Q0 as
      select s.s_suppkey as $SK from Supplier s ;

define query Q1 as
      select s.s_name as name
      from Supplier s
      where s.s_suppkey = $SK ;
```

In order to support page components modeling, the link clause can be prefixed with the keyword embedded. In the example, if the clause **link** *customer* was prefixed by embedded, the system would compute all the customers pages related to the given supplier and embed them in the supplier page instead of creating links to these pages. Moreover, for performance purpose, a link clause can be prefixed with the keyword prefetch. In our example, if prefetched was used, a request for a Supplier page would trigger a self (artificial) request to Weave for all the customer pages reachable via the given supplier page.

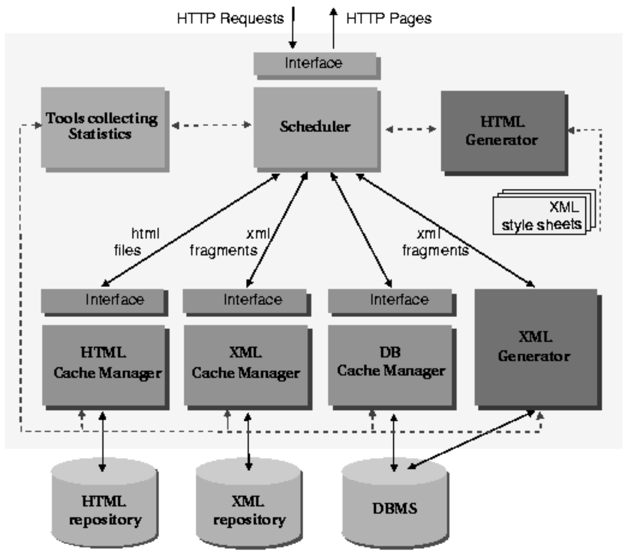


Figure 1: Architecture of the Weave system.

The XML Generator is responsible for the evaluation of the queries in the site schema and producing the corresponding XML data. Given a particular binding for the parameter $SK of the class Supplier, the XML generator produces the XML fragment corresponding to the Web page. For example given $SK=421, the following XML fragment is generated:

```
<XML_fragment id="Supplier_421">
  <class> Supplier </class>
  <parameter> 421 </parameter>

  <data_fragment name="name">
    <data_value> Supplier#000000421 </data_value>
  <data_fragment>

  <link_fragment name="customer">
    <link_item>
      <XML_fragment id="Customer_2">
         <class> Customer </class>
         <parameter> 2 </parameter>
      </XML_fragment>
      <anchor> Customer#000000002 </anchor>
    </link_item>
    ...
  </link_fragment>

  <form_fragment name="parts">
      <XML_fragment>
         <class> PartList </class>
         <input> $partname </input>
      </XML_fragment>
      <input type="text"> $partname </input>
  </form_fragment>

</XML_fragment>
```

Three-level caching architecture. For performance sake, Weave can cache data at three levels of abstraction: tables, XML, and HTML. As in [4], the DB cache manager controls caching and lookahead computation within the database system. It interfaces with the system, and offers additional capabilities such as the pooling of database connections.

Compared to the HTML cache, which caches HTML files on disk, the XML cache has the advantage of storing less data and allows for carefully controlling the granularity of the cached data, ranging from the entire page to fragments of the page. Moreover, it allows diminishing the load generated on the database by the Web server.

The architecture (Figure 1) includes a manager for each individual cache and a cache scheduler. The cache managers share the same interface and implement standard cache operations for data retrieval like addition and removal. These operations are triggered by events such as HTTP requests and data invalidations. The scheduler coordinates the execution of cache managers. It interacts with one or more of the individual caches according to the caching policy set for the given page.

The system architecture is modular and can easily be distributed. The replication of components on proxies and clients is also possible.

Customized cache management. Our final goal is to automatically compile a declarative specification of a Web site into a customizable *runtime policy*. A runtime policy controls the runtime behavior of the Web site so to make optimal usage of the caches according to the users' access patterns and the update frequency of the data. It specifies which data to prefetch or to cache (HTML pages, XML fragments, relational tables, or any combination of those), which particular items to prefetch or cache (e.g, which particular HTML pages or XML fragments), and which actions to perform under different events, such as page requests, data updates or environmental changes.

So far our system does not support such an automatic generation of runtime policies. However, to ease the task of the Web site administrator, we introduce *WeaveRPL*, a high level language for the abstract specification of the cache system's behavior (the specification is similar for each cache). The language is based on event-condition rules. It enables to explicitly specify the global policies implemented by an individual cache manager for setting overall features such as the maximum cache size, and the actions to be carried out upon a global event such as a cache overflow. Furthermore, it builds upon the declarative Web site specification, and allows the definition of per-site-class customized caching (basically, how to handle events related to data retrieval, addition, removal, and staleness).

In the following example, the HTML cache is configured as a single container, named HTML_CONT, which stores instances of classes Supplier, Customer, and Part. The container can be used to cache all Part instances and only instances of the Supplier and Customer classes that satisfy the following constraints: instances of Supplier must have a size less than 2KB, and instances of Customer must have an access frequency that is greater than 0.3. Upon initialization (handling of the onInit event), the HTML cache is fed with all the instances of class Part, and only with the instances of class Supplier, whose value of key SK is less than 100 and which meet the aforementioned constraint on size over cached instances. The content of the cache is refreshed every 30 minutes using the onTimer event, removing all the stored instances whose age is greater than 10 minutes. Whenever the cache is full (onFull event), a traditional LRU algorithm is applied for the replacement of the instances.

```
Cache HTML
{
 //Container definitions
   define container HTML_CONT as
   select Supplier where size <= 2KB,
      Customer where frequency >= 0.3,
      Part;
 //ECA rules
   onInit compute Part, Supplier(SK) where SK<100;
   onTimer(30mn) remove all where age>10mn;
   onFull(200M) applay LRU;
}
```

The XML cache specified in the example below also contains a single container (XML_CONT).

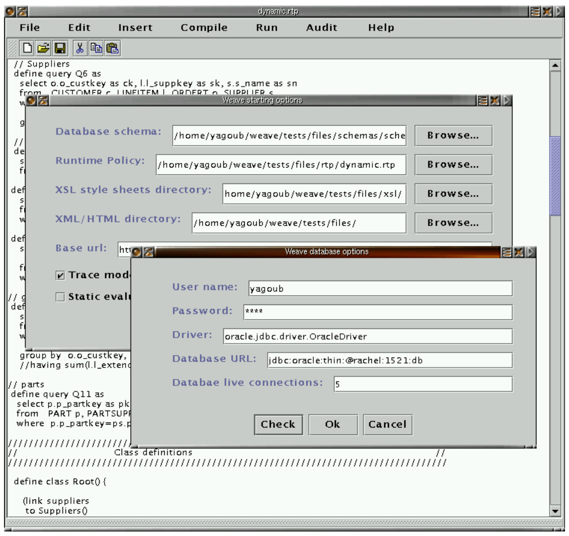
```
Cache XML
{
 //Container definitions
   define container XML_CONT as
      select Customer : fragments{name, supplier};
 //ECA rules
   onInit compute all;
   onTimer(120mn) reinit;
}
```

As opposed to its HTML counterpart, the XML cache container is used only to cache the fragments corresponding to the name of a given Customer and the set of links to his suppliers' pages. We assume that these fragments are not frequently updated and worth being cached. The other fragments, which are links to orders' pages of a Customer, are supposed to be frequently updated and therefore should be built on demand. The container is refreshed every 2 hours by removing and recomputing all the instances (reinit action).

Even in the absence of an automatic way to generate runtime policies, a high level language for specifying such policies does ease the production of data intensive Web sites, compared to existing approaches. By using our system, a Web site administrator is only requested to abstractly describe the desired cache management. Besides this language, Weave provides also an *API* for Web application programs to explicitly control the content of the caches.

3 Demonstration

The main focus of the demonstration is on Weave's ability of improving the performance of Web sites through a customized cache management. To this end, we compare different runtime policies with a Web site derived from the



```
// Suppliers
define query Q6 as
 select o.o_custkey as ck, l.l_suppkey as sk, s.s_name as sn
 from   CUSTOMER c, LINEITEM l, ORDERT o, SUPPLIER s
```

Database schema: /home/yagoub/weave/tests/files/schemas/sche

Runtime Policy: /home/yagoub/weave/tests/files/rtp/dynamic.rtp

XSL style sheets directory: home/yagoub/weave/tests/files/xsl/

XML/HTML directory: /home/yagoub/weave/tests/files/

Base url: htt

[✓] Trace mod

[ ] Static eval

User name: yagoub

Password: ****

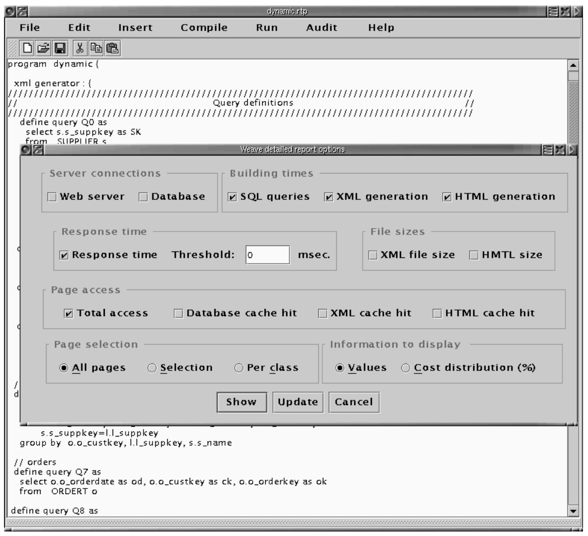
Driver: oracle.jdbc.driver.OracleDriver

Database URL: jdbc:oracle:thin:@rachel:1521:db

Databae live connections: 5

```
 group by  o.o_custkey,
 //having sum(l.l_extend

// parts
define query Q11 as
 select p.p_partkey as pk
 from   PART p, PARTSUPP
 where  p.p_partkey=ps.p

/////////////////////////////////////////////////////////////////////////////////
//                         Class definitions                                   //
/////////////////////////////////////////////////////////////////////////////////

 define class Root() {

   (link suppliers
    to Suppliers()
```

Figure 2: Weave configuration window.


dynamic.rtp
File Edit Insert Compile Run Audit Help
Weave detailled report options

```
program  dynamic {

 xml generator : {
/////////////////////////////////////////////////////////////////////////////////
//                              Query definitions                              //
/////////////////////////////////////////////////////////////////////////////////
  define query Q0 as
   select s.s_suppkey as SK
   from   SUPPLIER s
```

Server connections: [ ] Web server [ ] Database

Building times: [✓] SQL queries [✓] XML generation [✓] HTML generation

Response time: [✓] Response time Threshold: 0 msec.

File sizes: [ ] XML file size [ ] HMTL size

Page access: [✓] Total access [ ] Database cache hit [ ] XML cache hit [ ] HTML cache hit

Page selection: [● All pages] ○ Selection ○ Per class

Information to display: [● Values] ○ Cost distribution (%)

```
     s.s_suppkey=l.l_suppkey
 group by  o.o_custkey, l.l_suppkey, s.s_name

// orders
define query Q7 as
 select o.o_orderdate as od, o.o_custkey as ck, o.o_orderkey as ok
 from   ORDERT o

define query Q8 as
```

Figure 3: Execution reports main window.

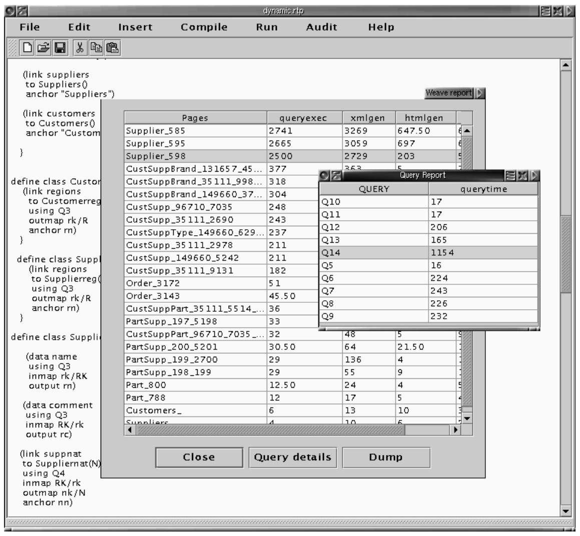
TPC/D benchmark database. The data is loaded into the Oracle8 DBMS.

Figure 2 shows the Weave user interface that allows a Web administrator to edit, modify and compile a WeaveL program or a runtime policy. The administrator can also set the execution parameters of Weave like the selected runtime policy and the maximum number of reusable database connections through this interface.

The demonstration starts by showing how to specify the Web site using the WeaveL language and a set of XSL style sheets. Then, we automatically run the Web site under two extreme conditions: when the entire site is precomputed before the pages are requested (purely static evaluation) and when each page is computed on the fly (purely dynamic evaluation). We show that it is immediate to derive from the declarative specification each of the above extreme approaches. In doing so, we run the Web site according to a particular trace (which can be given or generated based on a certain probability distribution).

For each execution (Figure 3), the system can be configured to generate reports containing information about the distribution of the various execution costs (database access, XML generation, HTML generation), the average total response time and the average XML and HTML file size. This information can be generated per page or per site class. Figure 4 shows an example of a Weave report obtained after browsing the TPC/D Web site. The report also displays the details of the execution time for all of the queries involved in building a particular page (see Figure 4).

Based on the results of the execution reports, the Web site administrator can tailor a specific caching strategy aimed at performance improvement. We show how complex caching strategies can be expressed in our formalism by simply using high level runtime policies. Given a particular runtime policy, the Web site can be rerun; the system will interpret the runtime policy and utilize the caches appropriately. Finally, by comparing the execution reports



```
  (link suppliers
   to Suppliers()
   anchor "Suppliers")

  (link customers
   to Customers()
   anchor "Custom

  }

define class Custor
  (link regions
   to Customerreg
   using Q3
   outmap rk/R
   anchor rn)
  }

 define class Suppl
   (link regions
    to Supplierreg(
    using Q3
    outmap rk/R
    anchor rn)
  }

define class Suppli

   (data name
    using Q3
    inmap rk/RK
    output rn)

  (data comment
   using Q3
   inmap RK/rk
   output rc)

 (link suppnat
  to Suppliernat(N)
  using Q4
  inmap RK/rk
  outmap nk/N
  anchor nn)
```

Pages	queryexec	xmlgen	htmlgen
Supplier_585	2741	3269	647.50
Supplier_595	2665	3059	697
Supplier_598	2500	2729	203
CustSuppBrand_131657_45...	377	363	5
CustSuppBrand_35111_998...	318		
CustSuppBrand_149660_37...	304		
CustSupp_96710_7035	248		
CustSupp_35111_2690	243		
CustSuppType_149660_629...	237		
CustSupp_35111_2978	211		
CustSupp_149660_5242	211		
CustSupp_35111_9131	182		
Order_3172	51		
Order_3143	45.50		
CustSuppPart_35111_5514_...	36		
PartSupp_197_5198	33		
CustSuppPart_96710_7035_...	32	48	5
PartSupp_200_5201	30.50	64	21.50
PartSupp_199_2700	29	136	4
PartSupp_198_199	29	55	9
Part_800	12.50	24	4
Part_788	12	17	5
Customers_	6	13	10
Suppliers	4	10	6

QUERY	querytime
Q10	17
Q11	17
Q12	206
Q13	165
Q14	1154
Q5	16
Q6	224
Q7	243
Q8	226
Q9	232

Figure 4: Query execution time report.

obtained from various runtime policies we show that a mixed strategy (caching data at different levels) is generally optimal, and therefore desirable.

References

[1] http://caravel.inria.fr/˜yagoub/webdbase.html.

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

[3] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents (objects) on the Web. In *Proc. of Middleware*, 1998.

[4] D. Florescu, V. Issarny, P. Valduriez, and K. Yagoub. Caching strategies for data-intensive Web sites. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2000.