# Enabling Dynamic Content Caching for Database-Driven Web Sites

K. Selçuk Candan  Wen-Syan Li  Qiong Luo  Wang-Pin Hsiung  Divyakant Agrawal
C&C Research Laboratories, NEC USA, Inc., 110 Rio Robles, San Jose, CA 95134, USA
candan,wen,qiongluo,whsiung,agrawal@ccrl.sj.nec.com

## ABSTRACT

Web performance is a key differentiation among content providers. Snafus and slowdowns at major web sites demonstrate the difficulty that companies face trying to scale to a large amount of web traffic. One solution to this problem is to store web content at server-side and edge-caches for fast delivery to the end users. However, for many e-commerce sites, web pages are created dynamically based on the current state of business processes, represented in application servers and *databases*. Since application servers, databases, web servers, and caches are independent components, there is no efficient mechanism to make changes in the database content reflected to the cached web pages. As a result, most application servers have to mark dynamically generated web pages as non-cacheable. In this paper, we describe the architectural framework of the CachePortal system for enabling dynamic content caching for database-driven e-commerce sites. We describe techniques for intelligently invalidating dynamically generated web pages in the caches, thereby enabling *caching* of web pages generated based on database contents. We use some of the most popular components in the industry to illustrate the deployment and applicability of the proposed architecture.

## Keywords

Database driven web site, dynamic content caching, web acceleration, JDBC, application server, invalidation

## 1. INTRODUCTION

Web performance is a key differentiation among content providers; i.e., companies and e-commerce sites eager to reach, attract, and keep customers. Snafus and slowdowns with major web sites demonstrate the difficulty companies face trying to scale their services to large amounts of web traffic. Such slowdowns can be devastating for e-commerce sites. A recent Zona Research[27] study showed that with response times of less than 7 seconds, e-commerce sites find


*ACM SIGMOD 2001* May 21-24, Santa Barbara, California, USA

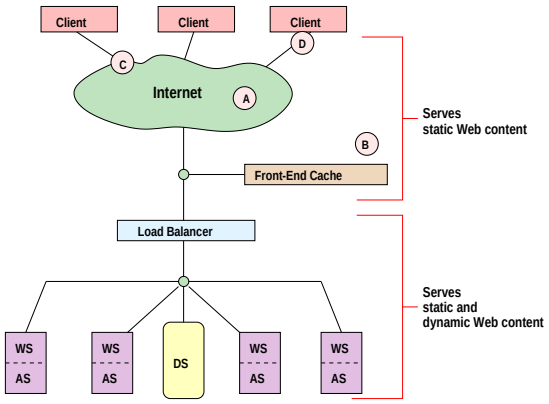




**Figure 1: A typical e-commerce site (WS: Web server; AS: Application server; DS:Database server)**

7% abandonment rates, whereas this rate increases sharply to 70% with response times greater than 12 seconds.

Figure 1 shows an overview of a typical web page delivery mechanism for web sites with backend systems, such as database management systems. In a standard configuration, there are a set of web/application servers that are load balanced using a traffic balancer, such as Cisco LocalDirector. In addition to the web servers, e-commerce sites utilize database management systems (DBMSs) to maintain business related data, such as prices, descriptions, and quantities of products. When a user accesses the web site, the request and its associated parameters, such as the product name and model number, are passed to an application server. The application server performs necessary computation to identify what kind of data it needs from the database and, then, sends appropriate queries to the database. After the database returns the query results to the application server, the application uses these to prepare a page and passes it to the web server, which then sends it to the user.

In contrast to a dynamically generated page, a static page, i.e., a page which has not been generated on demand, can be served to a user in a variety of ways. In particular, it can be placed in a proxy cache (Figure 1(A)), a web server front-end cache (as in reverse proxy caching, Figure 1(B)), an edge cache (i.e. a cache close to users and operated by content delivery services, Figure 1(C)), or a user side cache (i.e. user site proxy cache or browser cache, Figure 1(D)) for future use. With the static page in a cache, when a user requests this page, it can be delivered promptly from the cache without additional interaction with the web server. Note, however, that the application servers, databases, web

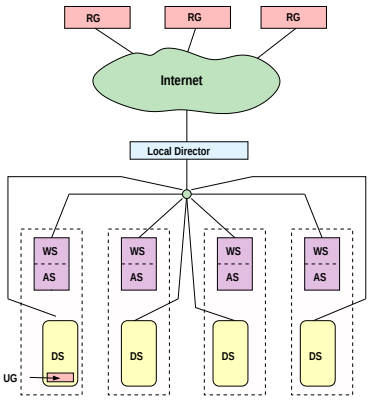



**Figure 2: Configuration I (replication); RGs are the clients (requests generators) and UG is the database where the updates are registered**

servers, and caches are independent components. Furthermore, there is no efficient mechanism to make database content changes to be reflected to the cached web pages. Since most e-commerce applications are sensitive to the freshness of the information provided to the clients, most application servers have to mark dynamically generated web pages as *non-cacheable* or make them expire immediately. Consequently, subsequent requests to dynamically generated web pages with the same content result in repeated computation in the backend systems (application and database servers) as well as the network roundtrip latency between the user and the e-commerce site.

There are various ways in which current systems are trying to tackle this problem. In some e-business applications, frequently accessed pages, such as catalog pages, are pregenerated and placed in the web server. However, when the data on the database changes, the changes are not immediately propagated to the web server. One way to increase the probability that the web pages are fresh is to periodically refresh the pages through the web server (for example, Oracle9i web cache provides a mechanism for time-based refreshing of the web pages in the cache) [19]. However, this results in a significant amount of unnecessary computation overhead at the web server, the application server, and the databases. Furthermore, even with such a periodic refresh rate, web pages in the cache can not be guaranteed to be up-to-date. Since caches designed to handle static content are not useful for database-driven web content, e-commerce sites have to use other mechanisms to achieve scalability. Below, we describe three (two traditional and one new) approaches to e-commerce site scalability:

## 1.1 Configuration I

Figure 2 shows the standard configuration, where there are a set of web/application servers that are load balanced using a traffic balancer, such as Cisco LocalDirector. Such a configuration enables a web site to partition its load among multiple web servers, therefore achieving higher scalability. Note, however, that since pages delivered by e-commerce sites are database dependent (i.e., puts computation burden on a database management system) replicating only the web servers is not enough for scaling up the entire architecture. Therefore, in this configuration, database servers are also replicated along with the web servers. This architecture has

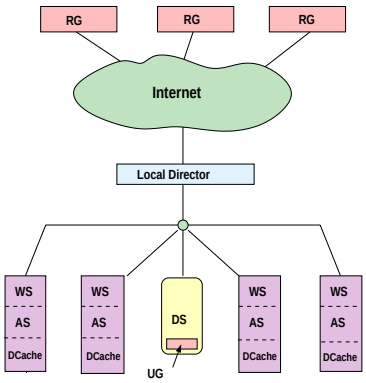



**Figure 3: Configuration II (middle-tier data caching)**

the advantage of being very simple. However, since it does not allow caching of dynamically generated content, it still requires redundant computation when clients have similar requests. Secondly, it is costly to keep multiple database synchronized in an update-intensive environment.

## 1.2 Configuration II

Figure 3 shows an alternative configuration that tries to address the shortcomings of the first configuration. As before, a set of web/application servers are placed behind a load balancing unit. In this configuration, however, there is only one DBMS serving all web servers. Each web server, on the other hand, has a middle tier database cache to prevent the load on the actual DBMS from growing too fast. Oracle 8i provides a middle-tier data cache [18], which serves this purpose. Since it uses middle-tier database caches (*DCaches*), this option reduces the redundant accesses to the DBMS; however, it can not reduce the redundancy arising from the web server and application server computations. Furthermore, although it does not incur database replication overheads, ensuring the currency of the caches requires a heavy database-cache synchronization overhead.

## 1.3 Configuration III

Finally, Figure 4 shows the configuration we are proposing in this paper: a dynamic-web-content cache sits in front of the load balancer to reduce the total number of web requests reaching the web server farm. In this configuration, there is only one DBMS. Hence, there is no data replication overhead. Also, since there is no middle-tier data cache, there is also no database/data-cache synchronization overhead. Redundancy is reduced at all (WS, AS, and DS) levels.

Network Appliance NetCache4.0 [17] supports an extended HTTP protocol, which enables demand-based *eject*ion of cached web pages. Similarly, recently, as part of its new application server, Oracle9i [19], Oracle announced a web cache that is capable of storing dynamically generated pages. In order to deal with changes, Oracle9i allows for time-based, application-based, or trigger-based invalidation of the cached pages. However, to our knowledge, it does not provide a mechanism through which updates in the underlying data can be used to identify which pages in the cache to be invalidated. The use of triggers for this purpose is likely to be inefficient and may introduce a large overhead on the underlying DBMSs, defeating the original purpose.

In this paper, we describe methods and systems for intel-

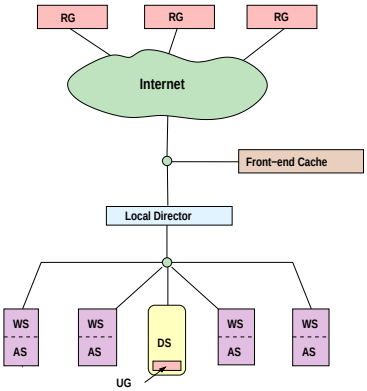



**Figure 4: Configuration III (web caching)**

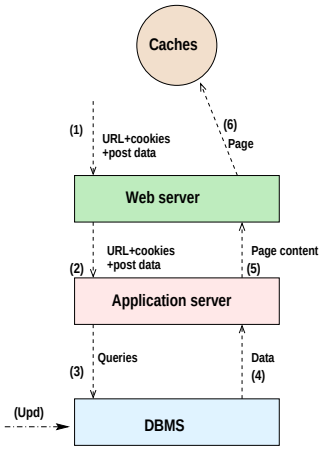



**Figure 5: Data flow in a database driven web site**

ligently invalidating dynamically generated web content in caches. We do not consider *data caching* at the database or application servers, although it could be used in conjunction with web page caches. An invalidator, which observes the updates that are occurring in the database identifies and invalidates cached web pages that are affected by these updates. Note that this configuration has an associated overhead: the amount of database polling queries generated to achieve a better-quality finer-granularity invalidation. The polling queries can either be directed to the original database or, in order to reduce the load on the DBMS, to a middle-tier data cache maintained by the invalidator.

We also show the applicability of our solution with the use of some of the most popular components in the industry (Oracle DBMS and BEA WebLogic web and application server). We compare the three configurations presented in this section with respect to the response time they provide to the end-users. The experiments and the presented results are for comparing the alternative architectures, and they are not intended to serve as benchmarks for the components.

# 2. DYNAMIC CONTENT CACHES

In this section, we provide an overview of the CachePortal *dynamic content cache management* architecture. Figure 5 shows the standard data flow processes in a database driven web site. Arrows **(1)**-**(6)**, in this figure, describe the steps that a typical dynamic content request takes:

- The process starts when the web server receives an HTTP request **(1)**. The request usually contains a URL string (with `GET` parameters), a set of `POST` parameters, and possibly a set of cookies.
- The web server uses the URL string to decide which application should serve the request. Then, the request is passed to an application server **(2)**.
- The application server accesses the DBMS through a set of queries **(3)** and processes the results **(4)** of these queries to create the page. It, then, passes this dynamically generated page to the web server **(5)**.
- Finally, the web server sends the dynamically generated page to the client browser or the proxy server which originated the request **(6)**. At this step, the page can be stored in various (front-end, proxy, or edge) caches for future use. Currently, however, these pages have to be tagged as non-cacheable.

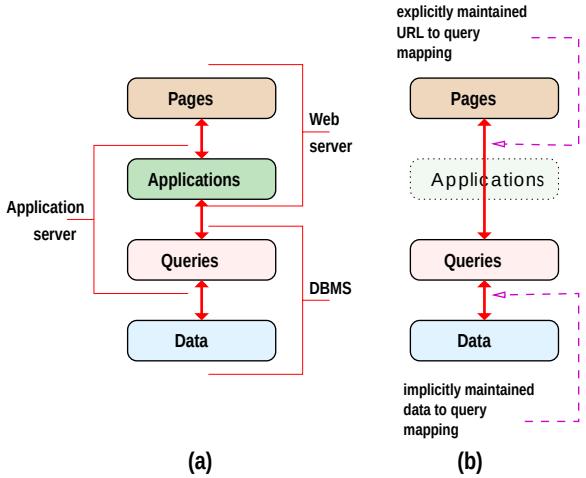



**Figure 6: (a) How different entities are related and which web site components are aware of them and (b) bi-level management of page to data mapping**

In addition to the web requests, a database driven web site also receives updates (through web or backend processes) to the underlying data in the databases. In the figure, the arrow **(Upd)** shows the updates to the database.

The major tasks of a dynamic content cache manager is to identify which updates to the underlying data affect which cached web content and to invalidate the affected cached content to prevent accesses to stale web pages by unsuspecting end-users. To accomplish this task, the dynamic content cache manager faces two major challenges.

## 2.1 Challenges

The first major challenge a dynamic content cache manager faces is to create a mapping among the cached web content and the underlying data elements. Figure 6(a) shows the dependencies between the four entities (pages, applications, queries, and data) involved in the creation of dynamic content. As shown in this figure, knowledge about these four entities are distributed on three different servers (web server, application server, and the database management server). There is no single entity which is aware of the page URL, the associated queries, and the underlying data used to answer those queries. Consequently, it is not straightforward to create a mapping between the data and the corresponding pages.

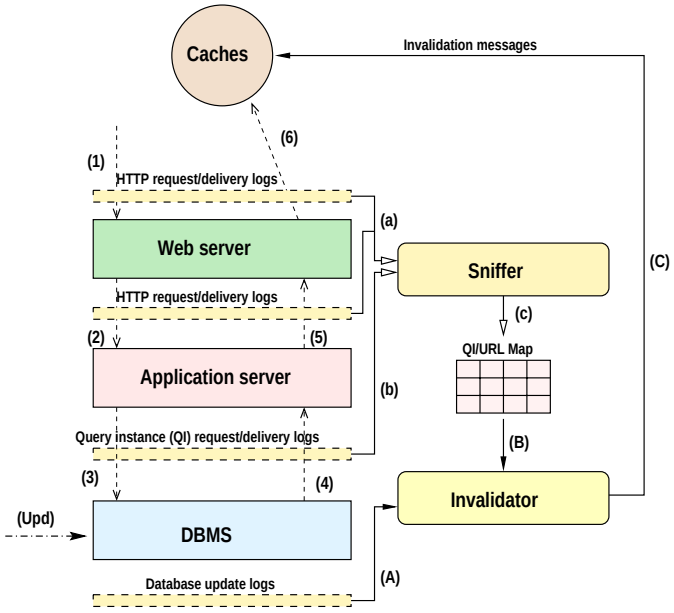



**Figure 7: The overall architecture**

The second major challenge is that timely web content delivery is a critical task for e-commerce sites and that any dynamic content cache manager must be very efficient (i.e., should not impose additional burden on the content delivery process), robust (i.e., should not increase the failure probability of the site), independent (i.e., should be outside of the web server, application server, and the DBMS to enable the use of products from different vendors), and non-invasive (i.e., should not require alteration of existing applications or special tailoring of new applications).

## 2.2 Proposed Solution

In this paper, we propose a solution which addresses these two challenges efficiently and effectively. Figure 6(b) illustrates the main ideas behind the proposed solution: instead of trying to find the mapping between all four entities in Figure 6(a), we divide the mapping problem into two: we find **(1)** the mapping between web pages and queries that are used for generating these pages and **(2)** the mapping between the queries and the data changes that affect these queries.

This bi-layered approach enables us to divide the problem into two components: *sniffing* or mapping the relationship between the web pages and the underlying queries and, once the database is updated, *invalidating* the web content dependent on queries that are affected by this update. Therefore, we propose an architecture (Figure 7) which consists of two independent components, a *sniffer*, which collects information about user requests and an *invalidator*, which discards cached pages that are affected by updates.

The invalidator sits on a separate machine which fetches the logs from the appropriate servers at regular intervals. Consequently, as shown in Figure 7, the architecture does not interrupt or alter the web request/database update processes. It also does not require a change in the servers or applications. Instead it relies on three logs (the HTTP request/delivery log, the query instance request/delivery log, and the database update logs) to extract the relevant information. Arrows **(a)-(c)** show the sniffer query instance/URL map generation process and arrows **(A)-(C)** show the cache content invalidation process. These two processes are complementary and are asynchronous.

## 2.3 Terminology

Before we describe the main components, we introduce relevant terminology we will use to describe the mapping and invalidation processes.

### *2.3.1 URL*

In this paper, we define URL (or page identifier) as a combination of three types of information contained within an HTTP request (using the Apache environment variable convention, the `HTTP_HOST` environment variable followed by the `QUERY_STRING` (`GET` parameters); the `HTTP_COOKIE` environment variable, and the HTTP message body which provides the post data (`POST` parameters). Note that, given an HTTP request, different `GET`, `POST`, or cookie parameters may have different effects on caching. Some parameters may need to be used as keys/indexes in the cache, whereas some other may not. In this paper, the term URL refers to the `HTTP_HOST` variables plus the parameters that has to be used as keys/indexes in the cache.

### *2.3.2 Query Type and Query Instances*

A query type is the definition of a database query. It is a valid SQL statement which may or may not contain variables. We denote a query type as $Q(V_1, \ldots, V_n)$, where each $V_i$ is a variable that has to be instantiated by the application server before the query is passed to the DBMS. Example:

```
SELECT * FROM R WHERE R.A > $V1 and R.B < 200;
```

where `$V1` is a query parameter. A query instance, is a bound query type with an associated request timestamp. We denote a bound query type as $Q^t(a_1, \ldots, a_n)$, where $t$ is the time at which application server passed the request to the DBMS and each $a_i$ is a value instantiated for variable $V_i$. Queries that are passed by the application server to the DBMS are bound queries. Therefore, multiple query instances can have the same query type.

## 2.4 Main Components of the Architecture

As shown in Figure 7, the main components of the proposed architecture are the sniffer and the invalidator. The sniffer creates the query instance to URL mapping (QI/URL map). Sniffer collects the query instances, but it does not interpret them. The URL information is gathered either before the web server using a module which listens to the incoming HTTP requests or before the application server using a module which uses the environment variables set by the application server. The QI/URL map contains **(1)** a unique ID for each row in the table representing the QI/URL map **(2)** the text of the SQL query to be processed by the invalidator, and **(3)** the URL information, including the values of the associated get, post, and cookie variables.

The invalidator, on the other hand, listens to the updates in the database and using the QI/URL map, identifies pages to be invalidated and notifies the relevant caches about the *staleness* of the cached page. For this purpose, it interprets the query instances in the QI/URL map.

As we mentioned earlier, neither the sniffer nor the invalidator should be a bottleneck in the system. Sniffer has to run as fast as the web server and this is not a problem be-

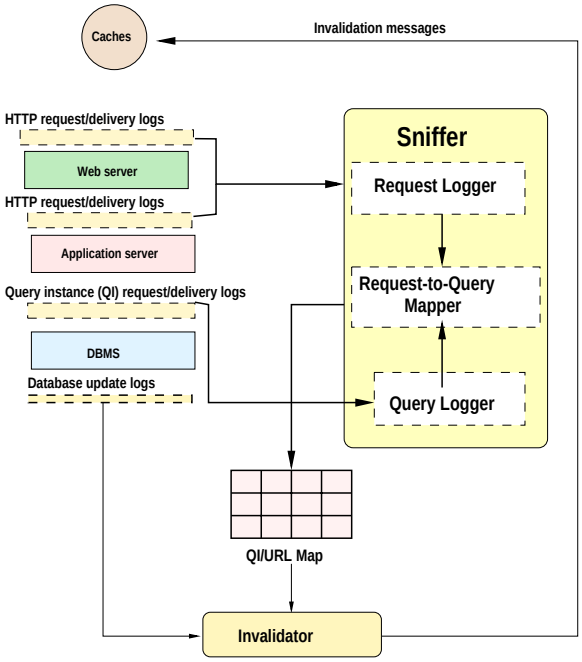



**Figure 8: Details of the CachePortal sniffer**

cause the web server has a lot more to do to serve a request than the sniffer. In the proposed architecture, the invalidator is also not a bottleneck, because it operates outside of the database and it has a control over how much time to spend for the invalidation process. Note, however, that if the invalidator needs to send extra queries to the database to gather relevant data for invalidation, then it *may* increase the load on the DBMS. If the web cache is used effectively, however, the reduction on the DBMS load due to the requests served from the web cache can account for this extra load. We will discuss the details of the invalidation process in Section 4 and the effects of the invalidation process on the performance of the system in Section 5. In the next section, we discuss the implementation details of the *sniffer* module.

# 3. THE SNIFFER MODULE

The sniffer module (Figure 8) consists of three loosely coupled parts: a *request logger*, which logs the HTTP requests, a *query logger* which creates a record of the database queries that the application server generates, and a *request-to-query mapper*, which generates a mapping between the HTTP requests and the corresponding database queries.

In this section, we use an e-commerce web site deployment which uses a BEA WebLogic web+application server and an Oracle DBMS to describe the implementation of the individual components of the sniffer (Figure 9(a)).

## 3.1 Request Logger

Request logger captures web requests that the e-commerce site receives. Note that, usually, web servers have an option to log all the requests they receive. One possibility is to use this logging option. However, different web servers have different logging capabilities. Furthermore, the logs created by web servers may not provide information (retrieval time and the parameters) with the necessary granularity. Clearly, an alternative option would be to modify the applications themselves to create these logs whenever they are involved.

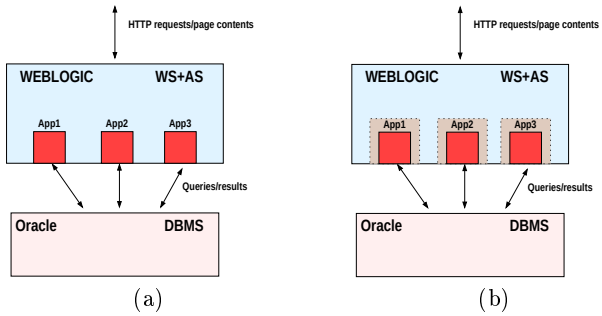



**Figure 9: (a) A site which uses BEA WebLogic and Oracle and (b) request sniffer architecture for BEA WebLogic web server (the dotted squares around the applications denote the request loggers)**

However, this would require modification of the applications, which would be invasive. Instead, we implement the request logger to work as a wrapper around the application servlets that are executed by BEA WebLogic (Figure 9(b)). This way we can log all requests that go through BEA WebLogic servlets. The logger extracts and stores **(1)** a unique ID for each request, **(2)** a request string which contains the page name and the get parameters, **(3)** a cookie string, **(4)** a post string which contains the post parameters, and **(5)** two (*receive* and *delivery*) time stamps.

In addition, the servlet wrapper has to translate `no-cache` cache directives into

```
Cache-Control:  private, owner="cacheportal"
```

directive so that CachePortal compliant caches can cache the resulting page. This process may require feedback from the invalidator to identify if the servlet uses a query which is marked non-cacheable by the invalidator. An alternative approach, which minimizes the interaction between the invalidator and the sniffer, would be to embed this logic within the wrapper code itself.

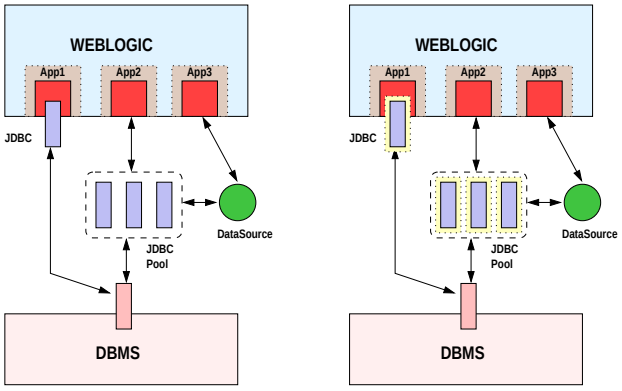
A temporal-sensitivity value describes how sensitive the servlet is (in milliseconds) to the updates in the underlying data. The pages generated by a servlet which is more sensitive than what CachePortal can accommodate are marked as non-cacheable. More specifically, the sniffer keeps the following information for each BEA WebLogic servlet: **(1)** a unique ID, **(2)** the original servlet string, **(3)** the cookie, get, and post parameters that are to be used as keys in the invalidation process, **(4)** associated statistics that will be collected to self-tune the invalidation process. **(5)** the temporal sensitivity of the servlet to the changes in the underlying data, and **(6)** the sensitivity of the pages generated by this servlet to the errors in the underlying data. Note that some of these parameters are used in the sniffing process and some are used as statistics that can be used in fine tuning the invalidation process.

## 3.2 Query Logger

Query logger captures the queries that are sent to the database by the application server. In order to achieve a non-invasive solution, we implement the query logger as a JDBC wrapper which provides a common interface to all JDBC drivers used by the application server. We are explaining the design of the query logger using BEA WebLogic:

**Servlets.** Servlets can access database in three different ways (Figure 10(a)).

- *Explicit JDBC drivers:* BEA provides two-tier JDBC




(a) (b)

**Figure 10: Sniffer architecture for BEA WebLogic web server (the dotted squares around the applications denote the request loggers and dotted rectangles around the JDBC drivers denote the query loggers)**

drivers, BEA WebLogic jDrivers, for various DBMSs. A servlet can explicitly use these drivers.

- *JDBC pools provided by the server:* BEA defines a connection pool as a named group of identical JDBC connections to a database that are created when the connection pool is registered. A servlet can access the database drivers through this pool.
- *DataSources provided by the server:* The recommended way to access JDBC connection pools is to bind a JDBC resource into the BEA WebLogic Server JNDI tree as a resource factory. In this case, instead of referring to the pool explicitly, servlets refer to the DataSource.

**Applets.** Applets can access database services provided by BEA WebLogic through RMI (standard-based) or T3 (proprietary) protocols. In both cases, applets can use a data source or an explicit pool.

**Enterprise Java Beans (EJBs).** An entity EJB can save its state (i.e., bean-managed persistence), or it can ask the container to save its non-transient instance variables automatically (i.e., container-managed persistence). In container-managed persistence, EJBs use deployment files that describe how BEA WebLogic container manages the persistence. This description include EJB connection pools.

The multitude of the ways the database can be accessed makes it essential to choose a single solution that can work with each option. Therefore, the query logger works as a wrapper around the JDBC drivers (Figure 10(b)). This way it is possible to log all queries that go through JDBC drivers, independent of how they are generated.

The JDBC wrapper is implemented as a driver class that provides interface to the actual JDBC drivers. The actual driver class name as well as the JDBC driver that has to be used by the wrapper are passed to the wrapper as an input in the database URL. The logger records the query string and the two timestamps, query receive time and result delivery. The JDBC wrapper is deployed in different ways depending on the access methods used by the servlet, applet, or EJB.

## 3.3 Request-to-Query Mapper Specification

Finally, the request-to-query mapper reads the query and request log files and creates a QI/URL map: during every request interval (i.e., between receive and delivery times of a requested page) in the the request log, it finds all queries that have been processed during the corresponding interval and writes this information to the QI/URL map. This map will be used by the invalidator module to identify which cached content has to be invalidated. In the next section, we describe details of the invalidator module.

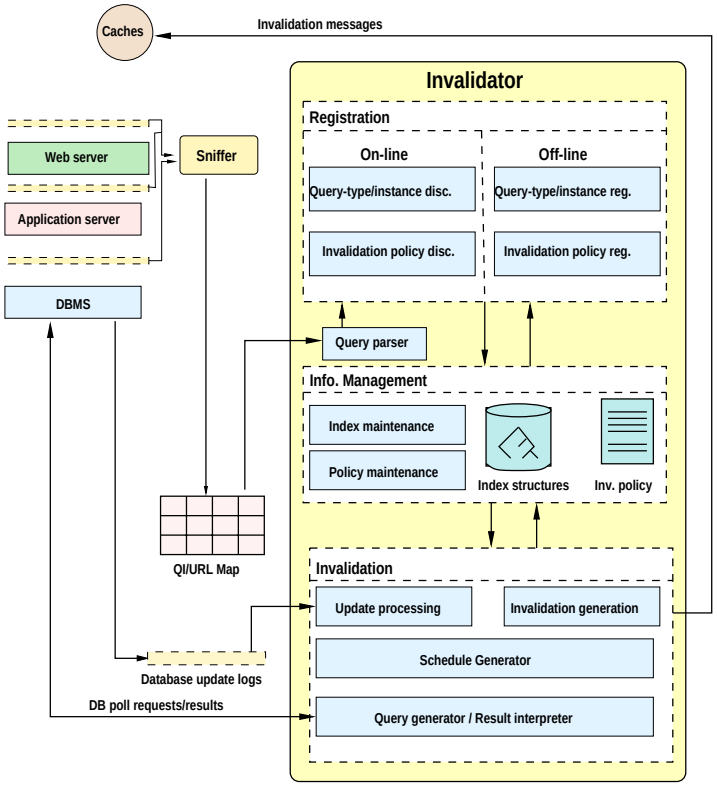



**Figure 11: Details of the Cacheportal invalidator**

# 4. INVALIDATOR MODULE

One way to achieve invalidation is to embed, into the database, update sensitive triggers which generate invalidation messages. The effectiveness of this approach, however, depends on the trigger management capabilities (such as tuple versus table level trigger activation and join-based trigger conditions) of the underlying database. It also puts heavy trigger management burden on the database. In addition, since the invalidation process depends on the requests that are cached, the database management system must also store a table of these pages. Finally, since the trigger management would be handled by the database management system, the invalidator would not have control over the invalidation process to guarantee timely invalidation.

Another way to overcome the shortcomings of the trigger-based approach is to use materialized views whenever they are available. In this approach, one would define a materialized view for each query type and then use triggers on these materialized views. Although this approach could increase the expressive power of the triggers, it would not solve the efficiency problems. Instead, it would increase the load on the DBMS by imposing view management costs.

In this section, we describe a third approach which achieves invalidation without the drawbacks of the above two alterna-

tives. Before describing this approach and the corresponding architecture of the invalidator module, we first provide an overview of the invalidation process using an example.

EXAMPLE 4.1. *Let us assume that we have an e-commerce application which uses a database that contains two tables,* `Car(maker, model, price)` *and* `Mileage(model, EPA)`. *Let us also assume that the following query,* $Query1$*, has been issued to produce a web page, say* $URL1$*:*

```
select Car.maker, Car.model, Car.price, Mileage.EPA
from Car, Mileage
where Car.maker = "Toyota" and
      Car.model = Mileage.model;
```

*If we observe that a new tuple* $(Mitsubishi, Eclipse, 20,000)$ *is inserted into the table* `Car`*, we can check whether the new tuple which is inserted into the relation* `Car` *does not satisfy the condition without any additional information. But if we observe that* $(Toyota, avalon, 25,000)$ *is inserted, then we can not check* whether or not the result is impacted *until we check the rest of the condition, which includes the table* `Mileage`*. That is, we need to check whether or not that the condition* `Car.model = Mileage.model` *can be satisfied. To check this condition, we can issue the following polling query,* $PollQuery$*:*

```
select Mileage.model, Mileage.EPA
from Mileage
where "Avalon" = Mileage.model;
```

*If there is a non-empty result for* $PollQuery$*, we know that the insert operation has had an impact on the query result of* $Query1$ *and consequently* $URL1$ *must be invalidated.* □

In general, we can find the results of a *polling query* either by issuing it to the database or by using external indexes kept within the invalidator, that can be quickly accessed:

- The size of the join index: If the index is reasonably small, it is preferable to build an index.
- The query frequency: If the query frequency is high it is preferable to build an index.
- The update frequency: The data in the index is part of the database but also replicated in the invalidator. Thus, if the index update cost is low, then it is preferable to build an index.

Invalidator consists of three tightly coupled components: a *registration* module, an *information management* module, and an *invalidation* module (Figure 11). The registration module creates the invalidation policies based on the queries that it receives. The information management module keeps track of the invalidation policies, and the *invalidation* module performs the actual invalidation task. In the following sections, we describe these modules in greater detail.

## 4.1 Registration Module

The registration module is responsible for **(1)** the creation of invalidation policies, **(2)** collection of relevant query statistics, and **(3)** passing of the relevant information to the information management module for the creation of auxiliary index and data structures. In other words, it is responsible for answering the question, *"what to invalidate"*. In its *off-line* mode, the administrator registers the query types (QTs) that the invalidator must listen to. Also, hard coded invalidation policies are registered at this mode. In its *on-line* mode, the registration module scans the QI/URL map and registers the new QTs and QIs. In this mode, it also creates query statistics and updates invalidation policies.

### 4.1.1 Query Type Registration

Domain experts may declare which types of queries are used by the applications. This module registers this information into the appropriate data structures. Each entry contains a query string, which includes **(1)** a query type name, **(2)** a unique ID, **(3)** a query string where $1, $2, ... denote the query parameters that CachePortal has to look for, and **(4)** associated statistics that will be collected to self-tune the invalidation process. These statistics include average and maximum invalidation times, average query and update frequencies, and the average invalidation ratio; i.e., the ratio of query instances invalidated by each update.

### 4.1.2 Query Type Discovery

The invalidator is also constantly listening to the QI/URL map to see if there are any query instances that it can not associate with the query types it knows of. If such query instances are encountered, the invalidator interprets them to identify their query types.

Since the number of query types and instances to be maintained can be large, instead of treating each query instance individually, the invalidator finds the related instances and process them as a group. Similarly, if multiple query types have parts that are similar/related to each other, these parts are treated in a coordinated manner. The new query types are written into an internal data structure.

### 4.1.3 Invalidation Policy Registration

Domain experts may establish some guidelines governing the invalidation process. These guidelines are registered into the invalidator's internal data structures. Each entry contains a policy rule which is either query-based or request-based.

### 4.1.4 Invalidation Policy Discovery

The invalidator constantly establishes/updates the performance guidelines governing the invalidation process. The decision can be either based on query types or the applications (such as servlets) that generate the pages. For example, a page which contains

- a query type that requires too much processing overhead may not be cached,
- a query type that invalidates more than a certain percentage of all query instances may not be cached,
- a query type/instance which is updated very often may not be cached.

Once the the query instances are processed to identify the possible query types, each query is assigned a cost, a priority, and a deadline. This information is then passed to the information management module.

## 4.2 Invalidation Module

The invalidation module is responsible for answering the question, *"when to invalidate"*. Unlike the registration module which listens to the queries from the QI/URL map, it **(1)** listens to the database update logs, and **(2)** it passes this information to the information management unit for the creation of auxiliary data structures and to the registration unit for the revision of invalidation policies. More importantly, it uses the invalidation policies and auxiliary data structures to **(3)** schedule the queries that will be sent to the DBMS (database polling) and to **(4)** send the appropriate invalidation messages to the caches.

This module consists of four subcomponents (Figures 11). The first component processes the update notifications received from the database. The second component creates invalidation schedules; i.e., identifies deadlines on invalidation messages. The third component uses this deadline to gather relevant data to minimize the unnecessary invalidation messages. Finally, the fourth component creates the actual invalidation messages.

#### 4.2.1 Update Processing

Since the number of updates to be processed can be large, instead of treating each update individually, the invalidator finds the related updates and process them as a group. For this task, at each synchronization point, it

1. pulls the update logs from the database and
2. for each relation, $R$, it creates two tables, $\Delta^+R$ which represents insertions to the relation and $\Delta^-R$ which represents the deletions.

#### 4.2.2 Schedule Generation

The invalidator uses the internal auxiliary structures to invalidate cached pages. However, in some cases, the internally maintained information may not be enough for this purpose. In these cases, this module identifies which information polling requests have to be generated and when these requests should be passed to the DBMS. Note that there is a tradeoff between the amount of polling required and the quality of the invalidation process. In general, it is possible to send detailed polling queries, hence spending more time, to identify which web pages in the cache **are not affected** by a given update. Therefore, it is possible to use this tradeoff between the amount of polling and the invalidation quality to schedule polling queries within the real-time constraints of an e-commerce site. This process requires parsing of the query type and identifying how various subcomponents of the query affect each other. Note that, especially when various queries share subqueries, this can be used in reducing the number and cost of polling queries.

#### 4.2.3 Query Generator/Result Interpreter

The polling information requests have to be converted into a form (i.e., SQL queries) understandable to the DBMSs and the results have to be converted into a form that the invalidator can use. This module handles these tasks.

#### 4.2.4 Invalidation Generator

Finally, the invalidation generator identifies which URLs are to be invalidated. Then, it creates an HTTP message which contains the invalidation requests and sends it to the appropriate caches.

Once the URL to be invalidated is identified, an invalidation cache-control message is generated. For this purpose, an extended cache control header, `Cache-Control: eject`, such as that provided by Network Appliance NetCache4.0 is used. This is simply an HTTP header that is sent as part of a "normal" client request.

### 4.3 Information Management Module

The information management module creates auxiliary data structures and invalidation policies that the invalidation module uses for decision making purposes. It maintains four major types of information:

- *Polling queries:* Polling queries and dependencies are maintained for optimizing the invalidation process.
- *Polling query results:* Some polling queries may be maintained for future use. Therefore, given a polling query instance that has to be maintained, the information management module
  1. creates the appropriate data structures for maintaining the results or the base data, depending on the instructions provided by the registration module.
  2. it initiates the daemon process that will watch the update logs and the polling queries issued by the invalidation module to update the stored data as necessary.
- *Invalidation policies:* Invalidation policies are maintained as query-based and request based formulas.
- *Statistics:* For every query type and web request (i.e., servlet) the information management module maintains statistics mentioned earlier.

## 5. EXPERIMENTS

In order to study the performance of the proposed dynamic content caching approach relative to the other alternative architectures, we set up various e-commerce web site deployment configurations. Figures 2, 3, and 4 presented in the Introduction show the three configurations we considered. In our set-up, we used Cisco LocalDirector to achieve load balancing. The web server farm behind the load distributor consists of four 200 MHz PCs with 768 MB main memory, running RedHat Linux 6.2 and Apache web servers. In addition to the web servers, we used Oracle8i database management servers.

We set up a hybrid (experiment+simulation) environment which allows us to experiment with real system components, while we simulate others (such as query generators) to evaluate the performance based on the most relevant parameters. This hybrid approach enables us to have absolute control of some experiment parameters (such as the request rate) through simulations and observe others (such as the response time provided by a web server until a given load) under real deployment conditions.

### 5.1 Experiment Parameters

In order to observe the performance of the proposed dynamic web content caching system with respect to other possible configurations, in our experiments we varied different system parameters and recorded the average response time that end-users observe. Table 1 provides an overview of the parameters involved in the evaluation of a database driven web site. These parameters are also visually depicted in Figure 12.

#### 5.1.1 System Parameters

Clearly, the two major parameters a database driven web site has to consider are the number of web requests ($num\_req$) it receives every second and the number of tuples that are in its database ($num\_tuples$). In addition, a database driven web site has to consider the number ($query\_per\_request$) of database queries generated by each web request. Each cache (web cache or database cache) used in such a system has an associated average hit ratio ($hit\_ratio$) which provides scalability with respect to increasing number of requests. This hit ratio is usually a function of the cache size ($cache\_size$). Web sites also benefit from resource replication ($rep\_rate$) for achieving better scalability. Database

**Table 1: System parameters**

| Parameter | Meaning |
|---|---|
| *num_req* | number of HTTP requests per second |
| *num_tuples* | number of tuples in the database |
| *num_query_types* | number of different query types the application generates |
| *cache_size* | number of web pages in the cache |
| *hit_ratio* | ratio of requests served from cache (function of cache size) |
| *query_per_request* | average number of DB queries each HTTP request generates |
| *update_rate* | average number of tuples updated per second |
| *poll_rate* | average number of polling queries generated per each query type |
| *poll_cost* | the cost of polling queries as a function of the cache size and update rate |
| *inval_rate* | the ratio of pages in the web cache that are invalidated (function of the number of polling queries) |
| *rep_rate* | database/cache/server replication ratio |
| *dist_synch_cost* | the cost of synchronizing distributed databases |
| *data_cache_synch_cost* | the cost of synchronizing the database with middle-tier data caches |

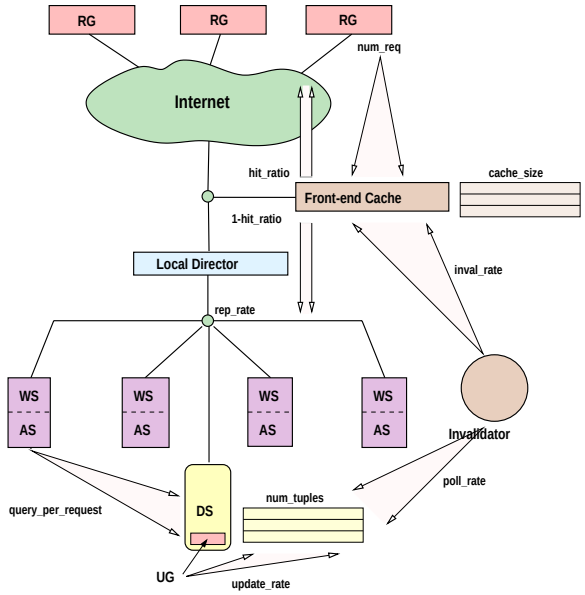



**Figure 12: System parameters (visual depiction)**

replication and data caching have associated synchronization costs (*dist_synch_cost* and *data_cache_synch_cost*).

As described in previous sections, the invalidator may need to generate polling queries to the database. The number (*poll_rate*) of polling queries generated is a function of the number of different query types (*num_query_types*) created by the underlying application. The cost of each polling query, on the other hand, depends on the number of cached pages (*cache_size*) and the rate with which the database is being updated (*update_rate*). Note that since the invalidator has to function in real-time, in reality, the amount of polling queries generated per second is limited. The effect of this is that the invalidation rate (*inval_rate*) increases as the cache size and the update rate increases. Over-invalidation, in turn, causes the hit ratio to decrease.

### 5.1.2 Evaluation Parameters

We use average response time observed by the end users to compare different configurations. Other parameters that we observed include the response time of each module, such as the DBMS, in the architecture. This enables us to observe how the bottleneck moves as the parameter value change.

## 5.2 Experiment Setup

In e-commerce deployments, depending on the characteristics of the underlying applications, these parameters can assume very different values. Therefore, in order to observe the affects of various parameter settings, we have created an artificial and flexible web architecture that

- lets us change the system parameters freely, and
- is deployed on a real hardware and software so that the results reflect real-life delay constraints.

Below, we describe different components of this architecture.

### 5.2.1 Application

We created a simple database driven e-commerce application. The database contains two tables: one small (500 tuples) and one large (2500 tuples) table. The two tables share a join attribute, which has 10 uniformly distributed values. We have intentionally kept the database size not very large to see if the web cache would be beneficial even when query processing cost is not overwhelmingly large. The application has three dynamically generated pages. A light-page request issues a select query on the small table, a medium-page request issues a select query on the large table, and a heavy-page request issues a select-join query on both tables. In all cases, the selectivity is 0.1.

### 5.2.2 Request Generators

The request generator module sends web requests to the web site and records the average response times. In our experiments, we have generated 30 requests per second: 10 light-, 10 medium-, and 10 heavy-page requests. In a static-page delivery system, such a load can easily be handled by a web server. Therefore, this load enables us to pinpoint to the bottlenecks in the application and database servers.

### 5.2.3 Update Generator

The update generator generates random updates to the database over the network.It enables us to observe the effect of varying update rates on the database query processing and synchronization costs. We have experimented with no-updates, 5 insertions and 5 deletions per table every second, and 12 insertions and 12 deletions per table every second.

### 5.2.4 Web Cache

The web cache reflects the hit ratio provided in the *hit_ratio* parameter; i.e., it answers to some of the requests immediately and passes the remaining web requests to the actual

web server. As described earlier, *hit_ratio* is a function of the *cache_size* and invalidation rate (*inval_rate*). In our experiments, we used a constant hit ratio of 70%.

The invalidator periodically sends polling queries. We assume that the invalidator maintains a data cache to which it poses the polling queries. Therefore, we have simulated the polling cost by issuing one query, which fetches the list of all recent updates, to the database every second.

#### 5.2.5 Data Caches

The middle-tier database caches reduce the number of database queries that the application generates. We simulated the behavior of the data cache by blocking some of the database queries generated at the application server and passing the rest to the actual database. In a sense, we assume that the cost of processing queries at the middle-tier cache is negligible with respect to processing them at the actual database. In our experiments, we used a constant hit ratio of 70%; i.e., only 30% of the database queries are directed to the database. We have simulated the cache synchronization cost by issuing one query, which fetches the list of updates, per cache to the database every second.

### 5.3 Experiment Results

In this subsection, we describe and evaluate the results of the experiments we ran to compare the three alternative configurations presented in the Introduction (Figures 2, 3, and 4). Tables 2 and 3 show the response times obtained in two sets of experiments: the first table shows results obtained under the assumption that data access costs at the middle-tier caches (Configuration II) are negligible and the second shows results obtained under the assumption that data access costs are not negligible.

Each table presents response times observed by end users under the three configurations (shown vertically) and three update loads (shown horizontally). Results for each configuration is further divided into three: **(1)** response times observed when the database has to be accessed (*miss*), **(2)** response times observed when the content (data or web page) is served from the cache (*hit*), and **(3)** the expected response times (*exp.*). The three update loads shown horizontally are as follows: no-updates, 5 insertions and 5 deletions from each table every second, and 12 insertions and 12 deletions from each table every second.

#### 5.3.1 Negligible Middle-Tier Cache Access Overhead in Configuration II

Table 2 shows that the first option (`Conf. I`), that is relying only on a load balancing solution with database replication, is not enough for an e-commerce site: under the given request load (30 requests per second or ~7 requests per database replica per second), even when there are no updates to the underlying database, the average response time observed by the end users are as high as 37 seconds. One third (12 sec.) of this is spent while accessing the database and the rest (25 sec.) is spent at the application and web servers. The significant amount of time spent at the application and web servers is due to resource starvation caused by processes holding to essential system resources, such as memory and network connection, while waiting for query results from the database. This shows that it is especially important to reduce the load on the database in an e-commerce web site, as a slight increase in database access times can have a large impact on the overall response time observed by the end users.

Second major observation from Table 2 is that, although the difference in average response times for configurations `II` and `III` is small (471ms vs. 450ms) when the update load is light, this difference get significantly higher as the rate of updates increases. In fact, when the rate of updates reaches around 50 updates per second, configuration `III` requires 20% less than the time required by configuration `II` (1147ms vs. 916ms). The reason for this increase is twofold:

First of all, the updates and requests are sharing the same network resources. Therefore, in configuration `II` a higher update rate increases the response time even in the case of hits. In configuration `III`, however, since the web cache is outside of the network which contains the database, the response time for cache hits are not affected by the updates to the underlying database.

Secondly, the database access penalty in the case of cache misses is higher in configuration `II` than in configuration `III`. This is because of the fact that, the application server uses the shared network to access the database; and, in configuration `II` the load on the shared network is higher than it is in configuration `III`. This leads to a consistently lower database access time in the proposed web cache configuration.

Note that in the above experiments, we assume that accesses to the middle-tier data cache in configuration `II` are instantaneous (i.e., the data is in the memory and query processing is negligible fast). We also assume that at every synchronization interval of 1sec, only the updates are sent to the caches instead of all table contents. In reality, the synchronization cost and data-cache access costs in configuration `II` is likely to be higher.

#### 5.3.2 Non-Negligible Middle-Tier Cache Access Overhead in Configuration II

In this set of experiments, we have implemented the middle-tier cache using a local database management system which keeps the materialized views of all queries. Therefore, each data cache access at Configuration II requires establishment of a connection to a database which serves as a local cache. On the other hand, the query processing cost at the cache is assumed to be negligible. As a result, although as in the previous set of experiments, data access from a middle-tier cache does not carry a query processing cost, now there is a cost (in terms of time and system resources) associated with making the connection with the cache.

The results in Table 3 shows that, even if the query processing cost at the cache is negligible, the consequences of many requests competing for the same cache resources can be very drastic (58 sec average response time even when there are no updates). This value is even significantly higher than the remote database access time. There are two reasons for this, First of all, **(1)** since when caches are effectively used (70% in our experiments) most queries are served from the cache, the race condition for database resources is especially significant in middle-tier data caches. Secondly, **(2)** the middle-tier data cache itself is sharing (and competing for) resources, such as CPU, with the web server and application server. Therefore, if any of these components suffer resource shortages due to an increased requirements in others, this increases the overall response time for the entire site. Therefore, it is better to have caches as lightweight

**Table 2: Experiment results with 70% cache hit rate (negligible middle-tier cache access overhead in Conf. II). Configuration I: One database per web server; Configuration II: One database for the system + middle-tier data caches at the application servers; Configuration III: Dynamic web-page caches**

| Average response times (ms) with 30 requests per second: 10 light-, 10 medium-, and 10 heavy-DB load per request | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UpdateRate $\langle ins_1, del_1, ins_2, del_2 \rangle$ | | Conf. I | | | | Conf. II | | | | Conf. III | | | |
| | | Miss | | Hit | Exp. | Miss | | Hit | Exp. | Miss | | Hit | Exp. |
| | | DB | Resp. | Resp. | Resp. | DB | Resp. | Resp. | Resp. | DB | Resp. | Resp. | Resp. |
| No Updates | (ms) | 15390 | 40775 | N/A | 40775 | 826 | 1291 | 119 | 471 | 773 | 1233 | 114 | 450 |
| $\langle 5, 5, 5, 5 \rangle$ | (ms) | 15480 | 41638 | N/A | 41638 | 1219 | 1903 | 145 | 672 | 1154 | 1602 | 73 | 532 |
| $\langle 12, 12, 12, 12 \rangle$ | (ms) | 16557 | 45443 | N/A | 45443 | 2556 | 3406 | 179 | 1147 | 2225 | 2944 | 47 | 916 |

**Table 3: Experiment results with 70% cache hit rate (non-negligible middle-tier cache access overhead in Conf. II). The numbers in small font are the same as in Table 2**

| Average response times (ms) with 30 requests per second: 10 light-, 10 medium-, and 10 heavy-DB load per request | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UpdateRate $\langle ins_1, del_1, ins_2, del_2 \rangle$ | | Conf. I | | | | Conf. II | | | | Conf. III | | | |
| | | Miss | | Hit | Exp. | Miss | | Hit | Exp. | Miss | | Hit | Exp. |
| | | DB | Resp. | Resp. | Resp. | DB | Resp. | Resp. | Resp. | DB | Resp. | Resp. | Resp. |
| No Updates | (ms) | 15390 | 40775 | N/A | 40775 | 11661 | 40116 | 58001 | 52632 | 773 | 1233 | 114 | 450 |
| $\langle 5, 5, 5, 5 \rangle$ | (ms) | 15480 | 41638 | N/A | 41638 | 13190 | 41206 | 51973 | 48845 | 1154 | 1602 | 73 | 532 |
| $\langle 12, 12, 12, 12 \rangle$ | (ms) | 16557 | 45443 | N/A | 45443 | 13247 | 40019 | 52696 | 48953 | 2225 | 2944 | 47 | 916 |

(in terms of access time as well as resource requirements) as possible. Furthermore, it is better to have potentially competing components on physically different machines.

To summarize, our initial experiments indicate that the proposed configuration (Conf. III) performs the best among all the alternatives while requiring the least amount of resources.

## 6. RELATED WORK

Various content delivery networks (CDNs) are currently in operation. These include Adero [1], Akamai [2], Digital Island [7], MirrorImage [16] and others. Although each one of these services are using relatively different technologies, they all aim to utilize a set of web-based network elements (or servers) to achieve efficient delivery of web content. Currently, all of these CDNs are mainly focussed on the delivery of static web content. Johnson *et al.* [11] provide a comparison of two popular CDNs (Akamai and Digital Island) and conclude that the performance of CDNs is more or less the same. It also suggests that the goal of a CDN should be to choose a *reasonably* good server, while avoiding *unreasonably* bad ones. Paul and Fei [20] provide concrete evidence to show that a distributed architecture of coordinated caches perform consistently better in terms of hit ratio, response time, freshness, and load balancing. Another study which shows that caching and content distribution indeed provides better performance is presented in [13]. Other related works include [10, 9], where authors propose a diffusion-based caching protocol that achieves load-balancing, [12] which uses meta-information in the cache-hierarchy to improve the hit ratio, [23] which evaluates the performance of traditional cache hierarchies and provides design principles for scalable cache systems, and [5] which notes that static client-to-server assignment may not perform well compared to dynamic server assignment.

Note that none of the works described above addresses the needs, such as dynamic content delivery, of e-commerce systems. The need for accounting for users' quality perception in designing web servers for e-commerce systems has been highlighted in [3]. The list of important QoS parameters listed in this study and others [27] includes the delay tolerance of users. In our work, we also consider this parameter, and discuss its impacts to the design of content delivery networks and caching systems.

SPREAD [22] is a system for automated content distribution. It proposes an architecture which uses a hybrid of *client validation*, *server invalidation*, and *replication* to maintain consistency across servers. Note that [22] focuses on static content and describes techniques to synchronize static content, which gets updated occasionally, across web servers. Therefore, in a sense, the invalidation and validation messages travel horizontally across web servers. Other works which study the effects of *invalidation* on caching performance are [24, 4, 8]. Consequently, there are several cache consistency protocol proposals which heavily rely on *invalidation* [22, 26, 15]. In our work, however, we concentrate on the updates of data in data sources, which are by design not visible to the web servers. Therefore, we introduce a *vertical* invalidation concept, where invalidation messages travel from database servers to web servers.

Recently, there has been some efforts aimed at preventing the database from becoming a bottleneck. One earlier solution was to cache business data outside of the DBMS to reduce the database access load. Oracle [18], for example, developed middle-tier data caching products along these lines. More recently, however, caching of the dynamically generated pages at the web servers has been shown to be more efficient than caching of the data itself [14]. Consequently, commercial DBMS and application server suppliers, such as Oracle, announced web caches [19] along with their more traditional middle-tier data caches [18].

Invalidating the content of these web caches, however, proved to be a significant challenge. Oracle web cache [19] as well as Dynamai [21], for example, addressed this challenge by providing time-based or event-based invalidation of the cache contents. The invalidation events can be generated by user supplied triggers or specially crafted application scripts. Oracle web cache, however, does not provide a framework for systematically generating invalidation messages in the presence of data updates. Challenger *et al.* proposed a solution,

based on explicitly maintained dependencies between data and cached objects, that addresses the update problem [6]. Similarly, Weave web site management system [25] uses a declarative language to describe the web site and benefits from this declarative specification to customize the appropriate materialization policy. Since, in most deployments, databases, application servers, and caches are independent components, however, maintaining such explicit dependencies is not always feasible. Furthermore, this solution puts additional responsibility, of expressing data dependencies between underlying data and cached objects, to the application program. In contrast, in our approach, the system does not maintain an explicit mapping between data and cached objects; furthermore, the application does not need to be specially tailored for web caching. This enables quick and flexible deployment of web caches as described in this paper.

## 7. CONCLUSION

In this paper, we presented an architectural framework for enabling dynamic content caching in database-driven e-commerce sites. More specifically, we described techniques for intelligently invalidating dynamically generated web content in the caches, thereby enabling *caching* of dynamically generated web content. The proposed solution achieves this by using a bi-layered approach which divides the problem into two components: *sniffing* or mapping the relationship between the web pages and the underlying queries and, once the database is updated, *invalidating* the web content dependent on queries that are affected by this update. Consequently, the proposed architecture consists of two independent components, a *sniffer*, which collects information about user requests and an *invalidator*, which removes cached pages that are affected by updates to the underlying data. We used some of the most popular components in the industry (Oracle DBMS and BEA WebLogic web and application server) to illustrate the deployment and applicability of the proposed architecture. We also presented a set of experiment results that verify the expected performance gains from the proposed architecture.

## Acknowledgements

The information presented in the paper about existing commercial products are obtained through public information provided over the web and it reflects our current knowledge about their publicly announced state-of-the-art. The experiments and the presented results are for comparing alternative web site configurations, and they are not intended and should not be viewed as benchmarks for the components.

## 8. REFERENCES

[1] Adero Inc. http://www.adero.com/.

[2] Akamai Technology. Information available at `http://www.akamai.com/html/sv/code.html`.

[3] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *WWW9*, pages 1–16, The Netherlands, 2000.

[4] P. Cao and C. Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4), 1998.

[5] R. Carter and M. Crovella. On the network impact of dynamic server selection. *Computer Networks*, 31(23-24):2529–2558, 1999.

[6] J. Challenger, A. Iyengar, and P. Dantzig. Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of the IEEE INFOCOM'99*, Mar. 1999.

[7] Digital Island, Ltd. Information available at `http://www.digitalisland.com/`.

[8] J. Gwertzman and M. Seltzer. World-wide web cache consistency. In *Proceedings of 1996 USENIX Technical Conference*, pages 141–151, San Diego, CA, Jan. 1996.

[9] A. Heddaya and S. Mirdad. Diffusion-based caching: Webwave. In *NLANR Web Caching Workshop*, pages 9–10, 1997.

[10] A. Heddaya and S. Mirdad. Webwave: Globally load balanced fully distributed caching of hot published documents. In *ICDCS*, 1997.

[11] K. Johnson, J. Carr, M. Day, and M. Kaashoek. The measured performance of content distribution networks. In $5^{th}$ *Int. Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[12] M. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. In *IEEE Workshop on Internet Applications*, pages 62–71, 1999.

[13] B. Krishnamurthy and C. Wills. Analyzing factors that influence end-to-end web performance. In *International World Wide Web Conference, WWW9*, pages 17–32, Amsterdam, The Netherlands, 2000.

[14] A. Labrinidis and N. Roussopoulos. WebView Materialization. In *ACM SIGMOD*, 2000.

[15] D. Li and P. Cao. Wcip: Web cache invalidation protocol. In $5^{th}$ *Int. Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[16] Mirror Image Internet, Inc. instaDelivery Internet services. http:// www.mirrorimage.com.

[17] Network Appliance Inc. http://www.netapp.com/products/netcache/.

[18] Oracle9i data cache. http://www.oracle.com/ip/deploy/ias/caching/index.html?database_caching.html.

[19] Oracle9i web cache. http://www.oracle.com/ip/deploy/ias/caching/index.html?web_caching.html.

[20] S. Paul and Z. Fei. Distributed caching with centralized control. In $5^{th}$ *Int. Web Caching and Content Delivery Workshop*, Portugal, May 2000.

[21] Persistent Software Systems Inc. http://www.dynamai.com/.

[22] P.Rodriguez and S.Sibal. Spread: Scaleable platform for reliable and efficient automated distribution. In *WWW9*, pages 33–49, The Netherlands, 2000.

[23] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distribued caching on the internet. In *ICDCS*, 1999.

[24] D. Wessels. Intelligent caching for world-wide web objects. In *INET-95*, 1995.

[25] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *VLDB 2000, Cairo, Egypt*, pages 188–199, 2000.

[26] H. Yu, L. Breslau, and S. Shenker. A scalable web cache consistency architecture. In *Proceedings of the ACM SIGCOMM'99*, Boston, MA, Sept. 1999.

[27] Zona Research. http://www.zonaresearch.com/.