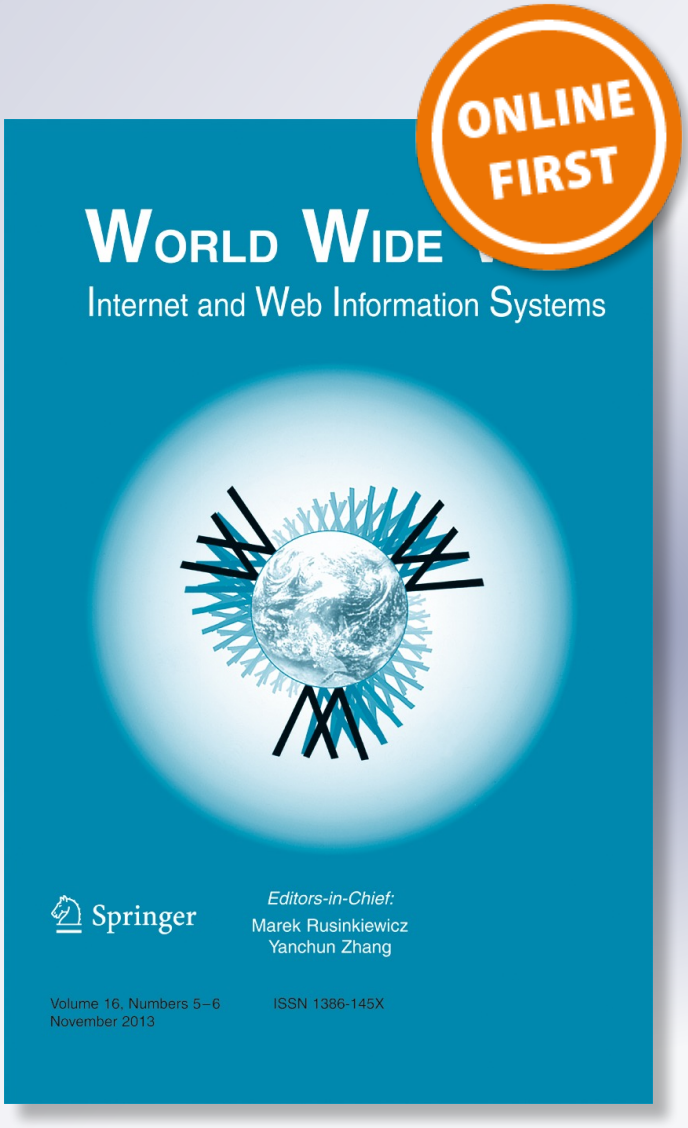
Performance vs. freshness in web database applications

Stavros Papastavrou, Panos K. Chrysanthis & George Samaras

World Wide Web
Internet and Web Information Systems

ISSN 1386-145X

World Wide Web
DOI 10.1007/s11280-013-0262-0



Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science +Business Media New York. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Author's personal copy

World Wide Web
DOI 10.1007/s11280-013-0262-0

Performance vs. freshness in web database applications

Stavros Papastavrou · Panos K. Chrysanthis · George Samaras

Received: 2 June 2013 / Revised: 17 September 2013 / Accepted: 1 October 2013
© Springer Science+Business Media New York 2013

Abstract In this work, we present a novel approach for the efficient materialization of dynamic web pages in e-commerce applications such as an online retail store with millions of items, hundreds of HTTP requests per second and tens of dynamic web page types. In such applications, user satisfaction, as measured in terms of response time (QoS) and content freshness (QoD), determines their success especially under heavy workload. The novelty of our materialization approach over existing ones is that, it considers the data dependencies between content fragments of a dynamic web page. We introduce two new semantic-based data freshness metrics that capture the content dependencies and propose two materialization algorithms that balance QoS and QoD. In our evaluation, we use a real-world experimental system that resembles an online bookstore and show that our approach outperforms existing QoS-QoD balancing approaches in terms of server-side response time (throughput), data freshness and scalability.

Keywords Caching · Dynamic web content · QoD · QoS · Data freshness

This work was part of the first author's doctoral dissertation at the University of Cyprus and was partially supported by the USA National Science Foundation Award OIA-1028162 and co-funded by the EU Project CONET (INFSO-ICT-224053) and the project FireWatch (#0609-BIE/09), sponsored by the Cyprus Research Promotion Foundation.

S. Papastavrou (✉) · G. Samaras
Department of Computer Science, University of Cyprus, Nicosia, Cyprus
e-mail: stavros@schoolfortheblind.net

G. Samaras
e-mail: cssamara@ucy.ac.cy

P. K. Chrysanthis
Department of Computer Science, University of Pittsburgh, Pittsburgh, USA
e-mail: panos@cs.pitt.edu

Published online: 24 November 2013

Author's personal copy

1 Introduction

Today as well as in the future, no business can succeed without an Internet presence. In fact, in the near future the boundaries between "conventional" and "electronic" commerce (e-commerce) will become increasingly blurred as businesses move their sales and services onto the web, with no barriers of time or distance. Our work focuses on e-commerce applications, such as an online retail store or an e-bookstore, built on top of web databases (webdb). Specifically, these applications are implemented by dynamic web pages that are generated on-demand by executing resource-hungry template scripts that access local or remote databases to produce HTML content.

Reportedly, billions of dollars are lost every year due to excessive delays in e-commerce web pages that force users to abandon their session [14]. This typical happens in a flash-crowd situations, where a web database server suddenly experiences a huge spike in requests/traffic and cannot keep up with it. It could also happen when the expected traffic in the web site is underestimated and before it is fixed by adding more resources.

The study in [18] presents a comprehensive and comparative listing of early approaches for enhancing *quality-of-service* QoS (user-perceived latency) under heavy workload. These approaches offer better QoS at the expense of uncontrolled *quality-of-data* QoD (freshness of data served). Improving on [18], the approaches in [3, 5, 7, 10, 21, 23] attempt to balance QoS and QoD by re-using from the cache as much as necessary stale content in order to spare computational resources and boost QoS. However, an open challenge has been the quantification of data freshness (QoD) of content and how this can be traded with QoS. In other words, the problem that we address is which pages or parts of pages (also known as *content fragments*) are 'less important' at a given time for 'that particular user' so that they can be re-used from cache under heavy or bursty server workload.

In an earlier version of this paper [19], we posed that current QoS-QoD balancing approaches fail to meet the requirements of modern e-commerce, web database applications under those server workload circumstances for the following two reasons:

- *Link Dependencies.* There is no consideration for the navigation needs of a user: If a content fragment is reused from cache, then it may be missing a needed valid HTML link for further user navigation at that given point in time. For example, a link on the upper right part of a web page of an e-bookstore may be recommending to the user to add the current book in the shopping cart, however, that link may be invalid since its containing fragment was reused from cache.
- *Set-View Dependencies.* There is no consideration for content fragments that must be synchronized (i.e., present consistent information) at the same time. For example, a part of the web page is showing book search results while another part is showing irrelevant suggested book listings from a previous search.

We addressed these shortcomings by proposing two new metrics for QoD and two algorithms that optimize these metrics. In this paper, we elaborate on the realization of our algorithms and present a thorough evaluation of the effectiveness of their key components in regulating QoS when balancing QoS-QoD. In addition, in order to better illustrate our proposed solution, we discuss how content fragments are

Author's personal copy

designed and organized into template files and analyze their dependency in the context of an e-bookstore.

Contributions The three key contributions of this paper are:

1. We enhance the notion of QoD with the inclusion of the above content dependencies (i.e., dependencies of web page content fragments). To encapsulate link dependencies, we introduce the metric of QoLF that considers the freshness of links in the content served and, thus, measures the ability of the user to navigate to the next page. To encapsulate set-view dependencies, we introduce the metric of QoSF that measures the degree of synchronization between content parts served.
2. We present two novel content materialization algorithms, namely, QLS and QLSV, that balance performance (QoS) with data freshness (QoD) in terms of the proposed QoLF and QoSF metrics.
3. We experimentally show that our algorithms outperform existing QoS-QoD ones in terms of throughput (i.e., better server-side response time), increased data freshness and scalability by sustaining more user sessions. Our performance evaluation is carried out using a real-world bookstore web database application, which is the canonical example of the majority of e-commerce web applications and online stores.

Roadmap In the next section, we first present the underlying assumptions of our work and then introduce our motivating application, providing a detailed analysis of its content fragments and their dependencies. In Section 3, we discuss existing content materialization approaches and their shortcomings. In Section 4, we present our approach for QoS-QoD balancing for materialization and in Section 5 our materialization algorithms. In Section 6, we discuss our performance evaluation and conclude in Section 7.

2 Background

In this section, we first present our assumed system model and discuss the generation of dynamic web pages from content fragments. Then we illustrate the use of content fragments and their dependencies in an e-bookstore application which we refer to throughout the rest of the paper.

2.1 Basic assumptions

System model We assumed a user-driven, personalized e-commerce web database applications with infrequent database updates and a typical client/proxy/web server/application server/application database(s) architecture as shown in Figure 1. All the components may have a data cache, however, we focus on the cache of the application server (middle-tier) which is the module responsible for content materialization as well as regulating QoS by varying the quantity of cached content served [8, 9, 11]. Moreover, we do not assume a common shared cache across all user sessions. We distinguish between individual user sessions with the use of cookies in user browsers.

Author's personal copy

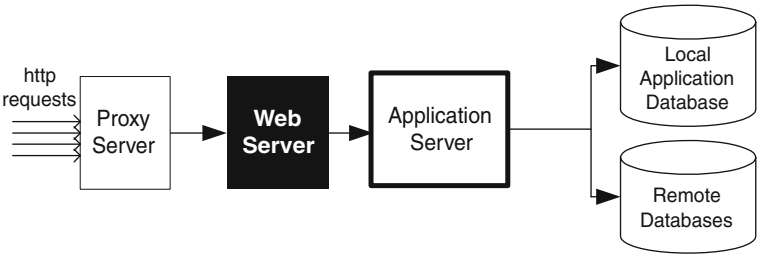


Figure 1 The typical layered web architecture supporting web db applications

The performance of the system is primarily determined by computation resources and not storage resources—even though a cache could be quite large, the computation latency may not be able to fully utilize it. Thus, the performance of the system is measured in terms of server throughput and the average server-side response time. The server-side response time is defined to be the lapsed time from the moment that a user request arrives at the web server until the response is transmitted to the user. This includes any computation and communication latency to the database back-end. A well provisioned server is expected to provide for an average server-side response time specified by a *threshold*.

The performance of the system is not measured in terms of the client-perceived latency because this depends on network delays and other client related aspects such as a slow client computer, which are beyond the purview and control of the server.

Templates of content fragments The web server is the public entry point of the application and directly serves request for static content (e.g., style sheets, images). Requests for a dynamic web page are routed to an application server that executes the corresponding *t*emplate file to generate the content. For modularity and performance, template files, such as ASP or PHP files, consist of script blocks that relate to different parts of a web page or content fragments. Content fragments as also referred to as "modules". A typical template file contains a simple HTML layout using either HTML Tables or Dividers (DIV) in which the content fragments are graphically aligned. A content fragment, for example one that presents the search results for books, (as we explore below in our motivation application), contains at least one SQL statement on the application database. The results of the statement are appropriately wrapped with HTML and style sheets for presentation.

The content and purpose of each fragment as well as their positioning in their corresponding templates is determined by the functional and presentation requirements of the application. Fragments in a template can be added, removed or repositioned within a template dynamically and at runtime. For example, the Cold Fusion server-side scripting language (Cold Fusion[1]) allows a custom HTML tag to be assigned to fragment, which in turn is included in a template file. This method allows for "cleaner" and modular development since fragments can easily be imported or moved inside template files.

[1] http://www.adobe.com/products/coldfusion-family.html

Author's personal copy

Content fragmentation using template files is a convenient and popular approach for developing, administrating and maintaining a dynamic webdb application with the use of a content management system (e.g., Plone CMS[2] and Drupal[3]).

When processing a template request, the containing fragments can be either retrieved from cached or freshly materialized possibly by accessing the back-end database. When all the fragments in the template became available, they are assembled together according to the template file's HTML layout and transmitted to the user through the web server.

The typical architecture described above can also facilitate webdb applications based on Asynchronous XML (or AJAX) [20] with the addition of the appropriate client-side (typically JavaScript) scripts. AJAX applications differ from traditional web-based approaches since the underlying assumption is that an application can "live and run" inside a single instance of a dynamic web page. The most typical example is that of Google's Gmail [26], where a basic template remains loaded on the browser and certain areas change dynamically to load search and folder results as well as compose a new message. The big advantage of AJAX applications is increased performance, since only the necessary data is loaded on every user request. The major drawback, however, is that AJAX applications are stateless, in the sense that the notion of "back" and "forward" for the browser does not exist. In other words, a user can loose the whole setup and state of the application if the "forward" or "back" buttons are pressed. Although in this work we do not consider this type of stateless applications, the underlying principles of our materialization approach can be used to support selective loading of data in AJAX.

2.2 Motivating application

Let us consider as a motivating example an online bookstore application, modeled after the popular Amazon webdb application,[4] which typically has about 20 different templates of which the four most popular are a *search page*, a *view book page*, a *shop box page* (shopping cart), and a *used books page*. These four pages account for more than 95 % of user accesses. Other less popular web pages include editorial reviews pages, feedback pages, user profile pages, as well as check-out specific pages. Figure 2 illustrates a breakdown (in fragments) of the three most popular bookstore templates. The arrows denote that a user can use a link or an HTML Form from within the source fragment to navigate to another (or the same) template. Some fragments have link dependencies to more than one templates.

Template search.dyn (S) Fragment $F1_{topresults}$ displays the top book search results and $F2_{allresults}$ displays all the results in a page-numbered way. $F3_{suggested1}$ and $F4_{suggested2}$ show suggested related sponsored results. $F5_{searchform}$ contains the HTML search Form for submitting book queries plus a few suggested terms for repeating search. $F6_{resubmit1}$ and $F7_{resubmit2}$ support search resubmission according to book tags and product categories, respectively. Finally, a bottom fragment $F8_{history}$ carries a

[2] www.plone.org

[3] www.drupal.org

[4] www.amazon.com

Author's personal copy

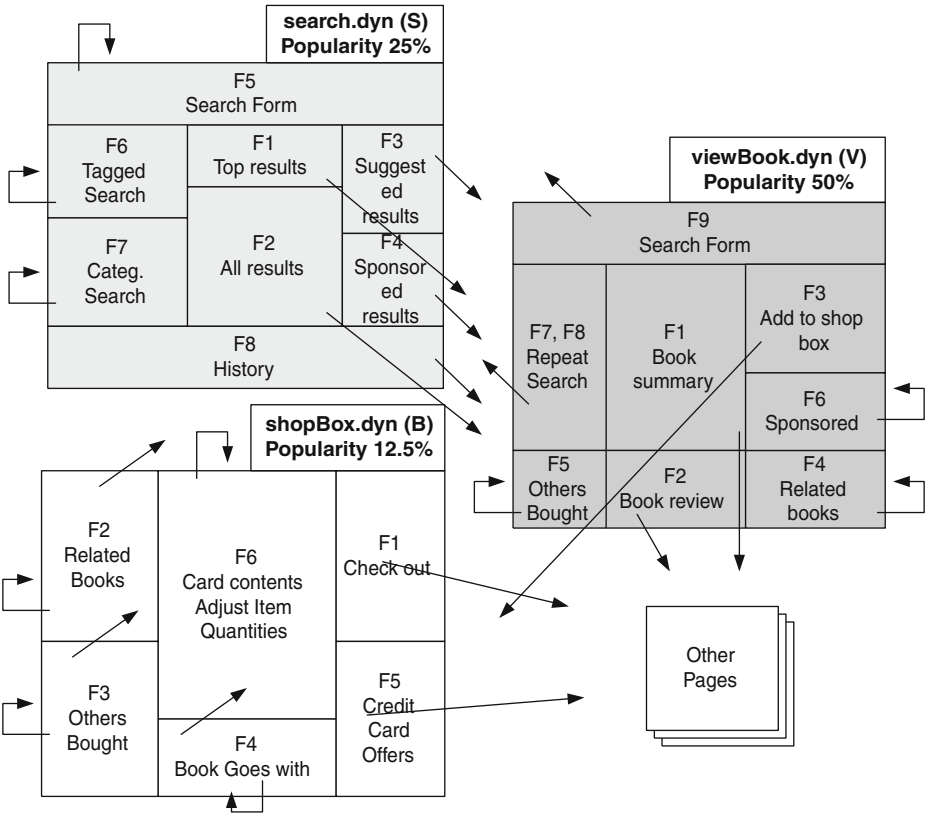


Figure 2 The fragments of the three most popular pages of the bookstore application with their linking dependencies to other templates

history of the recently viewed books. Fragments $F1_{topresults}$, $F2_{allresults}$, $F3_{suggested1}$, $F4_{suggested2}$ and $F8_{history}$ of `search.dyn`, that contain book listings, link the users to `viewBook.dyn`, the most popular page of our store which generates and displays information on a specific book. The rest ($F5_{searchform}$, $F6_{resubmit1}$ and $F7_{resubmit2}$) link again back to the `search.dyn` page for repeating a search.

Template viewBook.dyn (V) The `viewBook.dyn` page synthesizes all information related to a book. The main fragment $F1_{bookinfo}$ displays its cover picture and summary info, while $F2_{bookreviews}$ shows reviews and ratings. $F3_{addtoshopbox}$ contains a panel for adding the book into a shopping box through various financial/shipping ways. Fragments $F4_{related1}$, $F5_{related2}$ and $F6_{related3}$ show listings for other related books (what other people bought, sponsored, used etc.) linking again to `bookView.dyn`. The fragments $F7_{repeatsearch1}$ and $F8_{repeatsearch2}$ link back to `search.dyn` for search resubmission according to book tags and product categories. Finally, fragment

Author's personal copy

$F9_{searchform}$ contains the search form plus a few related suggested terms for repeating search.

Template shopBox.dyn (B) Fragment $F3_{addtoshopbox}$ of the previous template links to the shopping box of our application, implemented by the `shopBox.dyn` template. This template is called when a book or product is added to the shopping cart from the `viewBook.dyn` template. $F1_{checkout}$ of this template links to a less popular checkout page with various parameters that relate to shipping and payment. Fragments $F2_{related1}$, $F3_{related2}$ and $F4_{related3}$ show related book listings and link to `viewBook.dyn`. They also link to `shopBox.dyn` for adding the book directly into the shopping box without viewing it. $F5_{creditoffers}$ links to special credit card offerings. Finally, $F6_{adjustquantities}$ displays the shopping cart contents with options for increasing/decreasing quantities that link to the same page `shopBox.dyn`.

Fragment materialization in our model is analogous to virtual WebViews [6]. However, WebViews fragments are oblivious to their contents and usage. In our context, the fragments are assumed to contain HTML form and URL links with dynamic parameters that provide the user with the means of navigating between dynamic web pages. Links point statically to a target template and have appended dynamic parameters according to the application semantics, i.e., the link "/doBook.php?bookid=2345 & action=changeQuantity& value=−1" instructs the target template to perform specific tasks. We assume that a fragment reused from cache always contains outdated links since their parameters would refer to a previous user application-specific state and, therefore, would be invalid. Hence, according to our system model, fragments that are reused from cache do not have any freshness weight (importance). On the other hand, a fragment that is materialized upon a user request is considered fresh within is containing web page.

Definition 1 (Freshness of a content fragment) A dynamic web content fragment is considered fresh if it has been materialized on a user request, according to the user-submitted parameters.

Definition 2 (Freshness of a dynamic Web page) A dynamic web page is considered fresh if all the fragments in its corresponding template are served fresh.

3 Current approach and its shortcomings

Existing dynamic web page materialization methods balance QoS and QoD by varying the number of fresh fragments per template request according to the individual importance of their containing fragments [3, 7, 10, 21]. The "less important" fragments are the first to be reused from cache when server workload increases. The importance factor or weight of a fragment is template-specific and measures only the fragment's contribution to the overall freshness of its containing template. The sum of the weights of all fragments inside a template sums up to 1, which is the maximum value of freshness when all the fragments of a requested template are materialized.

Author's personal copy

A fragment F contributes to the freshness of a template T, if it is materialized when T is requested.

Definition 3 (weightIF(F,T)) Let $weightIF(F, T)$ be the freshness importance factor of an individual fragment F in template T. If $F_1, F_2, \ldots, F_n$ are all the member fragments of a template T, then $weightIF(F_i, T) \in (0, 1)$ and

$$\sum_{i}^{n} weightIF(F_i, T) = 1$$

Definition 4 (countIF(F,T)) $countIF(F, T)$ of a fragment F in template T is

$$countIF(F, T) = \begin{cases} 1 & \text{if } F \text{ is materialized in } T \\ 0 & \text{if } F \text{ is reused from cache in } T. \end{cases}$$

Basically, existing materialization methods focus on the importance of individual fragments and for this reason, we refer to their underlying approach as the *QoIF (Quality of Individual Fragments)* approach and their adopted QoD metric as the QoIF metric.

Definition 5 (QoIF(T)) Let $F_1, F_2, \ldots, F_n$ be all the member fragments of template T. $QoIF(T)$ is the freshness of template T whose value is

$$QoIF(T) = \sum_{i}^{n} weightIF(F_i, T) \times countIF(F_i, T)$$

The problem with the traditional QoIF approach is that, it considers the templates and their fragments as independent by ignoring content dependencies within and across templates. More specifically these problems are as follows.

3.1 No provision for link dependencies

Definition 6 (Link dependency) A fragment F_{source} is link-dependent on a template T_{dest}, if there is at least one URL link or HTML Form inside fragment F_{source} that links to T_{dest}.

For example, fragment $F3_{\text{addtoshopbox}}$ in template `viewBook.dyn` has a link dependency to template `shopBox.dyn`.

As discussed above, the links between templates are dynamic, in the sense that their parameters are not hardcoded. If $F3_{\text{addtoshopbox}}$ is reused from cache, because of its relative low QoIF importance weight, then it may not contain the correct HTML links to navigate to `shopBox.dyn` and add the requested book in the cart. Thus, even though the QoIF(`viewBook.dyn`) might be high, the absence of a freshly materialized $F3_{\text{addtoshopbox}}$ (an unsatisfied dependency—also called "broken link") stalls the user session which can only resume as soon as a fresh $F3_{\text{addtoshopbox}}$ is materialized and delivered to the user.

Author's personal copy

3.2 No provision for set-view dependencies

Definition 7 (Set-view dependency) A fragment F_i in template T_x is set-view dependent on fragment F_j in the same template T_x if both fragments must present consistent-to-each-other information.

For example, in template `shopBox.dyn`, fragment $F4_{related3}$ is set-view dependent on fragment $F6_{adjustquantities}$. If $F4_{related3}$ is reused from cache, then it may show inconsistent results with a fresh fragment $F6_{adjustquantities}$, since it may suggest the purchase of a book that already exists in the cart of the user. In general, a set-view dependency relates to the necessity of having two content fragments synchronized in order to simultaneously present consistent information on the user's browser. Clearly, the QoIF approach which handles fragments independently cannot synchronize the materialization of interdependent fragments.

4 Our approach to QoS-QoD balancing

Our approach for balancing performance (QoS) with data freshness (QoD) takes into account link and set-view content dependencies when materializing a dynamic page, thus reducing broken links and unsynchronized content. In a nutshell, our goal is to select the right set of fragments to materialize per page request, given the current server workload constrains. Under light workload, all fragments are materialized and all content dependencies are met. Under heavier workload, the right set of cached fragments is reused so that the most important-to-the-user link and set-view dependencies are met at that particular point in time. Our approach, illustrated in Figure 3, is broken down into the following three sub-goals (building blocks, or modules):

- *Ensuring QoS.* Constantly calculating the maximum possible number of fragments per template request that need to be materialized in order to keep the average response time below a predefined QoS threshold (in *ms*),
- *Speculation.* Employing user access patterns to 'guess' the next template that a user will request,

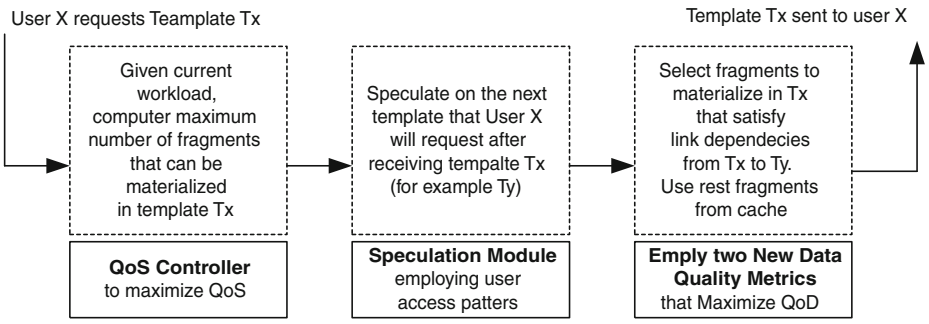


Figure 3 Overview of our proposed approach

Author's personal copy

– *Maximizing QoD*. Selecting the appropriate mixture of fragments per template request that satisfy link and set-view dependencies to the highest possible degree, given the 'guessed' next template that the user will request, in order to reduce broken links and unsynchronized content.

In the rest of this section, we elaborate on how we achieve these three sub-goals. In the next section, we bring together these three sub-goals and show how they are used by our proposed materialization algorithms.

4.1 Ensuring QoS and the QoS controller

4.1.1 Overview

The QoS Controller is responsible to ensure the specified QoS. It is based on the idea of regulating QoS by increasing or decreasing the number of fragments that are materialized per template request so that performance goals are met. Those fragments that are not materialized are reused from cache. This procedure is essentially symmetric to the QoD-centric OVIS algorithm found in [7], according to which effort is first made to keep data quality within acceptable margins. The reason for adopting such a straightforward QoS-centric approach over a QoD-centric one is because, there is no guarantee that by reducing the overall QoD would yield improved performance. Specifically, by enforcing a lower QoD per template does not imply that fewer fragments are materialized, since fragments do not have the same data quality weight. Instead, by explicitly reducing the number of materialized fragments per template, we secure performance gains since computational resources are immediately spared.

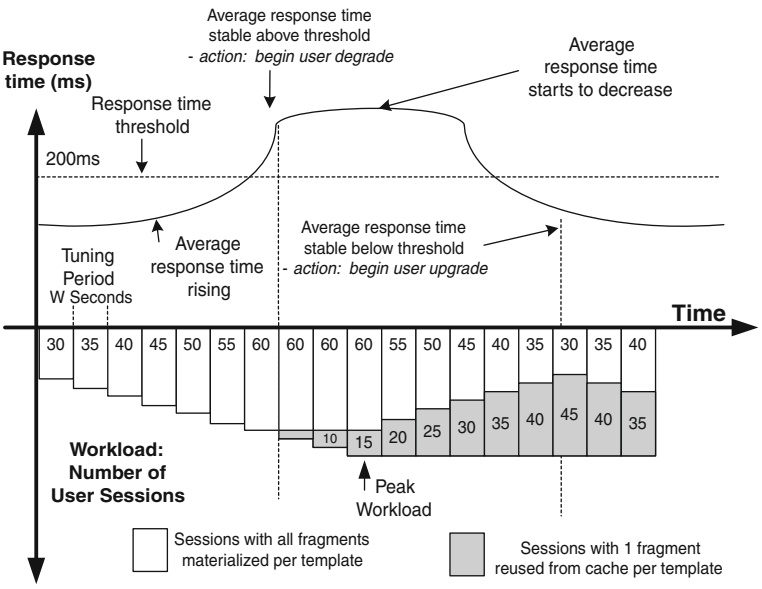


Figure 4 How QoS controller regulates QoS

Author's personal copy

Figure 4 illustrates the methodology of controlling QoS with a simplified example. Two essential factors for the QoS Controller are the maximum tolerable response time (*QoS-threshold*) (set at 200 ms) and the average response time of currently active user sessions (curved line at about 175 ms at t = 0). The former defines a threshold for the latter which, when violated/crossed, triggers the QoS Controller to initiate a corrective action as follows: At run time, the average response time of active user sessions is computed every *tuning period of W sec* (e.g., 5 s). If that is found steadily higher than the maximum tolerable QoS-threshold (at about 60 users), then initially one fragment per template is reused from cache for a small percentage of users.

As more users log on to the system (peak workload at 75 users) and performance goals are yet not met, more users must be affected. In the example, performance goals are met only when 45 users are have one fragment reused from cache per template request. From that point on, the procedure is reversed by decreasing the number of users that are affected. We refer to the action of applying a decrease on the number of fresh fragments for a user session as "degrading the user" or "dropping one fragment from a user". We refer to "upgrading a user" for the opposite action.

In the case where all users are degraded and still the average response time is steadily above the QoS threshold, then an additional second fragment is reused from cache initially for a small percentage of users and so on. Note that, all users must be degraded before issuing an additional drop to any user.

The QoS-threshold depends on the application requirements and can be modified accordingly, at any time, by the system administrator. A typical value would be in the order of hundreds of milliseconds [22]. As we stated earlier in Section 2.1 where we presented our system model, the use the threshold does not guarantee that all users receive their requested pages after that threshold has elapsed due to unaccounted network and client-side delays. What is essentially achieved is the regulation of the total time a user request is allowed to live in the system from the time it is received until it has finished processing and dispatched to the user. This regulation is then translated into enhanced server throughout as we discuss in Section 6 where we present our performance results of our evaluation.

The QoS tuning period (W) and the percentage of users (Userdiff) to degrade per period depend on the arrival rates which do not follow any known distribution [2, 25]. Both parameters define the "aggressiveness" of the QoS Controller. We study their effectiveness using linear and bursty arrival rates as part of our evaluation in Section 6.

4.1.2 The details

Since the number of fragments in each template and their execution time are not equal, we introduce an abstraction layer that applies a common denominator on how many fragments are dropped per template in each tuning period. For this reason, the QoS Controller maintains a *Global QoS Level Index*. Its value reveals the magnitude (or coefficient) of dropped fragments per user necessary for the system to maintain QoS and is independent from the number of fragments reused from cache per template. In addition, it maintains a *Local QoS Level Index* for every user session, in order to keep track the number of times a user has been degraded.

Figure 5 displays the pseudocode of the QoS Controller. At initialization, the Global QoS Level is set to 0 (line 1), it implies that the current trend is to materialize

Author's personal copy


```
Procedure QoS Controller()
 1  Global_QoS_level = 0;    // no workload at system startup
 2  while (True)
 3
 4   wait( W seconds );   // do QoS Threshold check every Tuning Period
 5
 6   if ( average response time   is SteadilyAbove  QoS Threshold )
 7   |  // this directive is used by the QLS algorithm
 8   |  QoS_Directive  =  " must favor QoS "
 9   |  if ( all active users have local QoS_level = Global_QoS_level )
10   |  |   Global_QoS_level --
11   |
12   else if ( average response time   is SteadilyBelow   QoS Threshold )
13   |   QoS_Directive  =   " must favor content quality  "
14   |  if ( all active users  have local QoS_level = Global_QoS_level )
15   |  |   Global_QoS_level ++
16   end  while
```

Figure 5 The QoS controlling loop

all fragments per template request for all user sessions due to light workload. Every tuning period of W seconds (line 3), the algorithm checks the average response time (line 6). If that is found above the specified QoS threshold, then the QoS Controller issues a *QoS directive* to the materialization algorithms (e.g., our new QLS algorithm) for a need to increase performance (line 8). Finally, the QoS Controller decides weather to decrease or not the Global QoS Level Index (lines 9–10). As mentioned earlier, this happens only if all users all active users have been degraded equally. On the other hand, if the average response time is found below the specified QoS threshold, then the QoS Controller issues a *QoS directive* for more fragment materialization (line 13) and increases the Global QoS Level Index (lines 14–15) accordingly.

In the next section, we examine in detail how the QoS directive and the Global and Local QoS Level Indexes are used by our materialization algorithms.

4.2 Speculation and usage plans

The second sub-goal of our approach is a speculation mechanism on the next template that a user will request. This speculation is required so that link dependencies are considered when selecting which fragments to materialize per template request. Since user speculation is not the main focus of this work, we only briefly discuss a simple speculation scheme based on data mining findings and pattern matching, which we use to implement our speculation module. A detailed description of the speculation module can be found in [17].

4.2.1 Recurrent user behavior

According to [25], the popularity of dynamic pages (and of templates) obeys a zipf-like distribution similar to static documents and media files. In other words, fewer templates account for more requests in a structured, almost predictable manner: the most popular template is accessed roughly at a rate of 50 %, the second most popular at a rate of 25 % and so on. It has also been shown that for webdb applications, a small

Author's personal copy

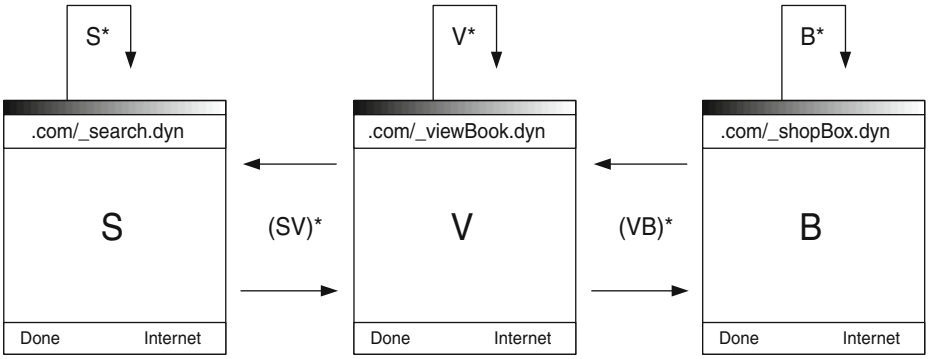


Figure 6 Five usage plans of the bookstore application: three uni-usage plans S*, V*, B*, and two bi-usage plans (SV)* and (VB)*

set of templates (approximately four) account for almost 95 % of the requests [1], where this set of templates is stable over time [16]. Similarly, [4, 15] refer to "mostly working" user sessions, in which users exhibit a very strong temporal locality in their request patterns on a small set of documents.

In our motivating application, the template access pattern of a typical user session is

$$S \rightarrow S \rightarrow V \rightarrow V \rightarrow B \rightarrow V \rightarrow B \rightarrow \ldots$$

The session begins with the user performing a couple of book searches using the `search.dyn` template (S). Subsequently, the user picks a book for viewing from the results by navigating to `viewBook.dyn` (V). Following that, the user picks a second book for viewing from a related book listing inside V. The book is added to her shopping box by navigating to `shopBox.dyn` (B), and so on.

4.2.2 Encoding the recurrent user behavior

To encode recurrent access patterns, we develop a scheme called *Usage Plans* whose two components are: (a) *Uni-Usage Plans* (uni-UP) that capture looping requests for the same template, and (b) *bi-Usage Plans* (bi-UP) that capture looping requests between the same two templates. Figure 6 presents five Usage Plans of the bookstore application with the three most popular templates of the application.

Note that two Usage Plans do not overlap and do not share the same template transition. In other words, *every transition between two templates is a member of only one Usage Plan*. This restriction is very important because it allows us to define a *session to consist solely of a sequence of Usage Plans*. Let us demonstrate this with an example of a user session in our motivating application, shown in Figure 7: The user performs first a search for a book three times in a row using S, views a couple of books using V and adds the last viewed book in the shopping basket using B. Then from within B, the user picks a suggested book to view using V, and then adds it to the shopping basket using B. The sequence of templates is shown along with the projected Usage Plans that emerge.

Author's personal copy

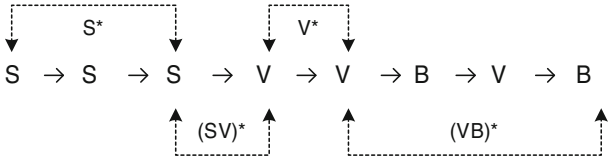


Figure 7 A session illustrated as a sequence of usage plans. Note that each usage plan is immediately followed by another one

4.2.3 Speculation on user behavior

Our profile-based speculation works in three distinct steps:

Step 1 A user session is constantly monitored and encoded in terms of Usage Plan switching using a FSM with three states: (a) "currently on a uni-usage plan", (b) "currently on a bi-usage plan" and (c) "partial bi-usage plan". The first state implies that the user has requested the same template again for at least one time (for example, S→S). The second state implies that the user has at least looped once between the same two templates (for example, S→V→S). The third state implies that the user has completed only one transition of a bi-Usage Plan.

Step 2 A speculation on the next template to be requested by the user is issued on every user request by considering prior user patterns on Usage Plan switching. These patterns are captured in a user-wise pattern-mapping table in which each row is a quintuple that represents a possible change in user behavior. Each quintuple encodes (a) the previous state of the user, (b) its current state, (c) the template that the user is currently viewing, (d) the template that the user is currently requesting by clicking on a URL link or through an HTML Form, and (e) the speculated Usage Plan that relates to the first four entries of the quintuple. An example of a quintuple is ("currently on a uni-usage plan", "partial bi-usage plan", S, V, V*) which means that the user has requested template `search.dyn` for at least two times, is currently requesting template `viewBook.dyn` and the user's speculated behavior is to request again template `viewBook.dyn`. The list of patterns can be set a-priori, based on the semantics of the application. However, these may be refined/adjusted dynamically using web mining techniques (e.g., [24]).

Step 3 For feedback purposes, the speculated template is compared to the actual user behavior. If there is a conflict, the fifth entry on the corresponding quintuple of the pattern-mapping table is corrected.

4.3 Maximizing QoD and the new QoLF and QoSF data freshness metrics

The third sub-goal of our approach is the selection of the right set of fragments to materialize per template request so that link and set-view dependencies are met. The pre-requisites are: (a) the maximum number of fragments per template (computed by the QoS Controller) and (b) the speculation on the next template in a user's access pattern (computed by the Speculation Module), as presented above.

As opposed to existing methods, we consider the importance of a fragment to be a multi-faceted metric which relates (i) to the ability of a user to request the next template and (ii) to the relation of that fragment to other fragments inside its

Author's personal copy

template. Thus, to select which fragment to materialize per template request, we make use of two novel data quality metrics, namely, QoLF and QoSF, These two new metrics substitute the traditional QoIF metric of existing methods.

4.3.1 QoLF—Quality of link fragments

The metric of QoLF quantifies the existence of freshly materialized fragments inside a template T_s with link dependencies toward a target template T_d. It captures this by means of *QoLF importance weight* of a fragment toward a link-dependent template.

Definition 8 (weightLF(F_i, T_s, T_d)) Let $weightLF(F_i, T_s, T_d)$ be the QoLF importance factor of fragment F_i in template T_s toward template T_d. For all F_i in T_s with a link dependency to template T_d, $weightLF(F_i, T_s, T_d) \in (0, 1)$ and

$$\sum_{i}^{n} weightLF(F_i, T_s, T_d) = 1$$

In other words, weightLF(F_i, T_s, T_d) measures the navigation/linking importance of fragment F_i in template T_s toward template T_d. In this way, the importance of F_i is dynamic since it depends on a target template T_d.

For example, consider a user viewing a book through template `viewBook.dyn`. If the expected behavior of the user is to add the book in the shopping cart by linking to `shobBox.dyn`, then the importance of fragment $F3_{addtoshopbox}$ in template `viewBook.dyn` which links the user to `shopBox.dyn` is relatively higher to other fragments that link the user to other templates. Alternatively, if the expected behavior of the user, after viewing the book in `viewBook.dyn`, is to repeat search using suggested book listings, the importance of fragments $F7_{repeatsearch1}$ and $F8_{repeatsearch2}$ that perform tagged book search by linking to template `search.dyn` is relatively higher to the importance of fragment $F3_{addtoshopbox}$.

In any case, if all fragments inside template T_s with link dependencies to T_d are materialized when T_s is requested by a user, then the QoLF for template T_s toward T_d has the maximum value of 1.

Definition 9 (QoLF(T_s,T_d)) For all fragments F_i in template T_s with link dependency to T_d, then

$$QoLF(T_s, T_d) = \sum_{i}^{n} weightLF(F_i, T_s, T_d) \times countIF(F_i, T_s)$$

As in the case of QoIF, if a particular linking fragment from T_s toward T_d is not materialized, then the QoLF value is reduced according to the QoLF importance weight of that fragment toward T_d.

4.3.2 QoSF—Quality of set-view fragments

The metric of QoSF quantifies the overall set-wise consistency of set-view dependent fragments inside a template. This is achieved by means of *QoSF importance weight* which measures the importance of having synchronized materialization of two fragments of a template.

Author's personal copy

Definition 10 (weightSF(F_i, F_j, T)) Let $weightSF(F_i, F_j, T)$ be the QoSF importance weight between fragments F_i and F_j in template T. For all F_i and F_j which are set-view dependent T, $weightSF(F_i, F_j, T) \in (0, 1)$ and

$$\sum_{i,j}^{n} weightSF(F_i, F_j, T) = 1$$

In other words, weightSF(F_i, F_j, T) measures the importance of having fragments F_i and F_j in template T synchronized/present consistent content. The QoSF importance between two fragment is set according to the application semantics, as discussed in Section 3, using a decimal value between 0 and 1. In addition, all the QoSF importance weights in a particular template must sum up to 1.

Given that QoSF considers pairs of fragments, only synchronized pairs contribute to and counted toward the freshness of their template.

Definition 11 (countSF(F_i, F_j, T)) $countSF(F_i, F_j, T)$ of a pair of fragments F_i and F_j in template T is

$$countSF(F_i, F_j, T) = \begin{cases} 1 & \text{if } F_i \text{ and } F_j \text{ are synchronized} \\ 0 & \text{otherwise.} \end{cases}$$

Fragments F_i and F_j are synchronized if when their template is requested by a particular user, either both are materialized on this request, or both are used from cache but were previously generated on the same request. When all set-view dependent fragments of a template T are synchronized then T is fully set-view consistent and its QoSF has the maximum value of 1.

Definition 12 (QoSF(T)) For all fragment pairs F_i and F_j in template T, then

$$QoSF(T) = \sum_{i,j}^{n} weightSF(F_i, F_j, T) \times countSF(F_i, F_j, T)$$

In other words, if all pairs of set-view dependent fragments in a template are synchronized when T is requested by a user, then the template T is fully set-view consistent. If one pair of set-view dependent fragments is not synchronized, then the overall set-view consistency of their template is reduced according to the QoSF importance of that particular set-view dependency.

5 Materialization algorithms

In the previous section, we presented the three building blocks of our content materialization approach. We explained how QoS is regulated by a QoS Controller by using a Global QoS Level Index. We then introduced the notion of Usage Plans for user access speculation and explained how those are employed by a very simple speculation module that 'guesses' the next template that a user will request. Finally, we introduced the metrics of QoLF and QoSF for measuring data freshness, given the link and set-view dependencies of content fragments. In this section, we bring

Author's personal copy

together these three building blocks and demonstrate our content materialization algorithms that balance QoS with QoD.

5.1 The MP selection table

We organize together the Global QoS Level index, the Usage Plans and the new data quality metrics into one structure called the *MP Selection Table*. In brief, this table summarizes all combinations of fresh/cached fragments, for a specific template, into groups according to a QoS Level Index and calculates their respective QoLF and QoSF values. The combinations are called *Materialization Plans (MP)*. The intuition behind this grouping is that, the MPs in each group require approximately the same execution time.

Lets consider for example template `search.dyn` (S) and assume that it consists of only 4 fragments for ease of presentation. The corresponding MP Selection Table for template S is shown in Figure 8. At QoS Level Index equal to 0, there is only 1 MP, the '1111' and implies that all fragments are materialized. At QoS Level Index equal to −1, there are 4 possible MPs. The MP '1101' implies that all fragments are materialized except the third one which is retrieved from cache. For each MP, a QoLF value for each template to which template S links to is computed according to Definition 9 in Section 4.3.1. In our example, template S links to its self and template

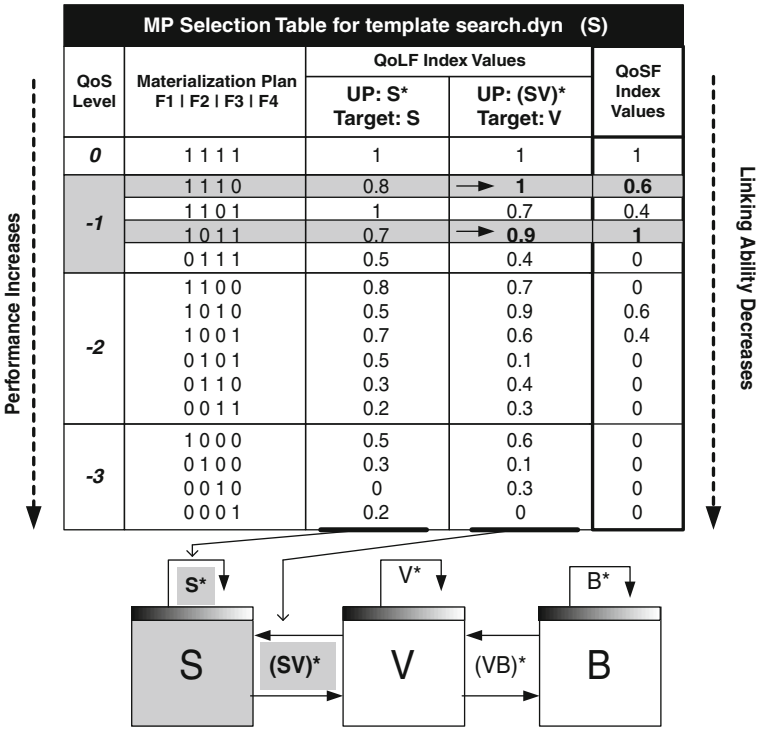


Figure 8 The MP selection structure for template `search.dyn`

Author's personal copy

viewBook.php (V) when the user is on usage plan S* and (SV)* respectively. In the right-most column of the MP Selection Table, the QoSF for each MP is computed according to Definition 12 in Section 4.3.2.

5.2 QLS: The QoL-sensitive algorithm

The first algorithm we propose, namely *QLS*, balances QoS with QoD in terms of the QoLF metric. It is sensitive to link dependencies meaning that its goal is to facilitate seamless user navigation even at high workloads where more fragments per template are served from cache (and contain outdated links). Recall that seamless user navigation relates to the ability of a user to request the next template in its request sequence by using a valid link. An instance of the QLS algorithm is instantiated for every new user session.

Figure 9 shows the pseudo-code of QLS. Triggered on every user request for a template, QLS's main loop secures first QoS by computing the approximate quantity of fragments to materialize in the requested template (lines 4–16). To do that, first it reads the QoS directive from the QoS Controller. If the directive is, for example, to favor QoS due to high workload, then it degrades the user by reducing its Local QoS Level Index. Note that, the number of users that are degraded per tuning period W is limited to a percentage (UserDiff) of active user sessions.

QLS Materialization Algorithm (for user X)

```
1   while (the user X requests templates)                          First,
2     user has requested template Tx                               secures
3                                                                  QoS
4     get the  server workload ( QoS_Directive), from the  QoS Module
5     degrade or upgrade user X, if necessary
6       if (QoS_Directive = "must favor QoS")      then
7           if ( % of users degraded  in current Check Period < UserDiff )      then
8               Local_QoS_Level --
9       else
10      if (QoS_Directive = "must favor QoS")      then
11          if ( % of users upgraded in current Check Period < UserDiff )       then
12              Local_QoS_Level ++
13
14    compute apprx. quantity of fragments in Tx, using Local_QoS_Level
15        get candidate MPs from MP Selection Structure of Tx,
16        with QoS level = Local_QoS_Level
17
18    get the next speculated template    of user X,                 Then,
19        from the   Speculation Profiling Module                    maximizes
20    compute the  right set of fragments   in Tx to materialize     Data Quality
21        get MP from candidate MPs, that maximizes QoLF             in terms of
22                                                                   QoLF
23    materialize the fragments of template Tx with 1s
24    retrieve from cache the fragments in Tx with 0s
25    assemble the all fragments in template Tx and send to user
26  end while
```

Figure 9 The QLS algorithm

Author's personal copy

Given the user's Local QoS Level Index, QLS then secures data quality (lines 18–24) First, it reads the speculated template that the user will request from the Speculation Module. Then, it isolates the group of MPs from the MP Selection table with QoS Level equal to the user's QoS Level Index. In the example highlighted in Figure 8, the the user has requested template `search.dyn` and its Local QoS Level Index is set to −1. There are now four possible combinations of cached/fresh fragments to select from (column 2 from the table). QLS will then pick the MP with the highest possible value of QoLF toward the speculated next template in the user's access pattern. If the speculated template is `viewBook.dyn` (V), then the selected MP is the '1110' whose QoLF value is 1. Finally, the fragments with the corresponding 1s in the requested template are materialized while those with the 0s are retrieved from cache. All fragments, fresh and cached, are then assembled in one file according to the template and send to the user.

5.3 The QLSV algorithm

Our proposed second algorithm, called *QLSV*, is a variation of QLS and balances QoS with QoD in terms of both the QoLF and QoSF metrics. Similar to QLS, its goal is to enable seamless navigation for users at high workloads with the additional goal of strengthening data synchronization in the template as much as possible. Specifically, we added to QLS the extra goal of selecting the materialization plan with the highest possible index for QoSF with respect to a relax factor on QoLF (line 21 of the QLS algorithm).

We demonstrate this with an example as highlighted in Figure 8. If the user's Local QoS Level Index is −1 and the speculated Usage Plan is (SV)*, then only the MP '1110' with QoLF equal to 1 and QoSF equal to 0.6 is considered. However, a relax factor of 10 % implies that the algorithm **additionally** considers the MP '1011', which has QoLF equal to 0.9 and QoSF equal to 1. The effect of the relax factor is that reduces the linking ability the user in order to improve data synchronization inside the template. In the highlighted example, a 10 % relax factor on QoLF yields a 40 % gain on the QoSF of the selected MP. Finally, a relax factor of 0 % implies that the algorithm considers only the MPs with the highest possible QoLF index (as the original QLS algorithm does).

6 Performance evaluation

In this section, we present the performance evaluation of our proposed materialization algorithms. In our evaluation, we first assert the effectiveness of QoS Controller and then compare the performance of our materialization algorithms with the current QoIF approach in terms of data freshness, throughput (i.e., server-side response time) and scalability (i.e., maximum user sessions support).

The evaluation is performed on an experimental platform that emulates a real-world web database application similar to the bookstore application discussed in Section 2.1. Every experiment was run at least 5 times to secure statistically significant results.

Author's personal copy

6.1 Experimental platform: e-Bookstore

6.1.1 Webdb system

Our main server machine (a dual CPU, 2GB RAM, RAID 0) hosted our Java-based web server structured according to the multi-threaded system model. On the same machine, we deployed an application server according to our proposed architecture in Section 4. The application database runs on a separate machine (also a dual CPU, 2GB RAM, RAID 0) on the same local network and it is implemented on Microsoft SQLServer 2005. The database holds the data for a bookstore with more than a hundred thousand books, in addition to data for book availability, authors, shopping baskets, orders etc.

We prepared a mixture of templates, each containing eight to ten fragments. The fragments and their content dependencies are setup according to the bookstore application. Every fragment contains script code that manipulates the results of one read-only query on the application database. In addition, one fragment of the `shopBox.dyn` template executes one update on the application database for placing (or removing) a book in a user's shopping box.

6.1.2 Workload

On a separate machine, we developed and deployed a multi-threaded User Generator engine capable of emulating a large number of user browsers. We chose to create our own user generator engine in order to have greater control over our experiments in terms of user statistical traces and fragment handling. Specifically, our browser emulators can issue a special HTTP GET request for receiving a fresh version of a fragment when a user selects a link embedded in a fragment served from the cache which is broken.

Our synthetic workload follows basic principles according to the transactional web e-Commerce benchmark (TPC-W) [13], In particular: (a) the popularity of documents follows a zipf-like distribution, (b) a small set of documents (around four) account for at least 95 % of total user requests, (c) this set is stable over time, (d) consecutive user requests occur about every ten seconds [12].

6.2 Evaluating the QoS controller

In our first group of experiments, we evaluate the QoS Controller in order to identify its effectiveness in regulating QoS and explore the impact of each tunable parameter.

Experiment 1 In our first experiment, we test the QoS Controller's sensitivity to the number of user's degraded (UserDiff) per turning period W using a linear rate for the arrival of new user sessions. We set QoS-threshold at 200 ms and the tuning period W at 10 s. We start the experiment with active 25 user sessions and new users arrive at a rate of one per five seconds. We stop the experiment when the number of concurrent users reaches 250. We perform three runs of the experiment with UserDiff values equal to 20 %, 60 % and 100 % respectively.

The results (Figure 10a) show that the average response time is maintained below the QoS-threshold for all three runs, however, the average response time is more aggressively pushed lower when a higher UserDiff value is used. This is

Author's personal copy

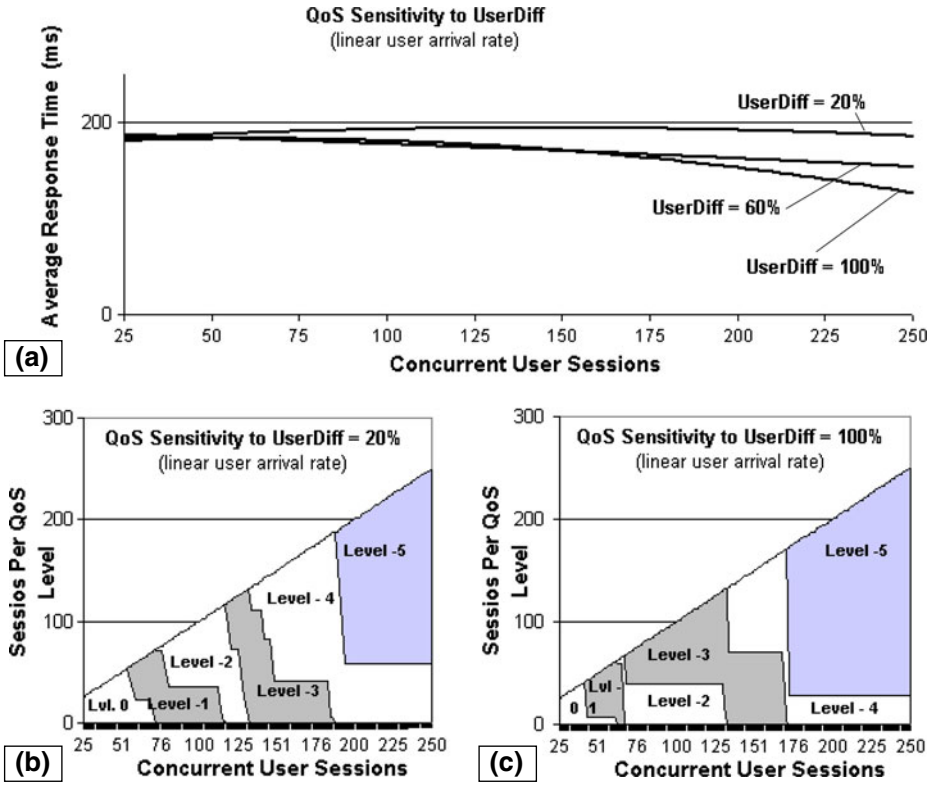


Figure 10 Evaluation of the QoS controller—sensitivity to UserDiff

attributed to the rate user sessions are degraded to lower QoS levels as the number of concurrent users increases. A lower UserDiff value results in more gradual degrades (Figure 10b) as opposed to a higher UserDiff value that results in more rapid degrades (Figure 10c).

Experiment 2 In our second experiment, we test the QoS Controller's sensitivity to the length of the tuning period W, again using a linear rate for the arrival of new user sessions. This time, we fix UserDiff at a moderate 60 % and the routine of the experiment is the same as previous. We run the experiment for three settings using a tuning period of 5, 10 and 15 s, respectively. To our surprise, the results (Figure 11a) show that there is no notable difference by using different tuning periods. The charts in Figure 11b and c reveal that this similarity is due to the symmetry in degrading user sessions. A smaller tuning period W has shorter and more frequent degrades, as opposed to a larger period that has longer but less frequent degrades.

Experiment 3 Our next experiment evaluates the sensitivity of the QoS Controller to UserDiff under a bursty arrival episode of new user sessions. We run the experiment for three settings with UserDiff values equal to 20, 60 and 100 % while having the tuning period fixed at a modest ten seconds. We start the experiment with 100 loaded user sessions in Local QoS Level equal to −3. At this workload,

Author's personal copy

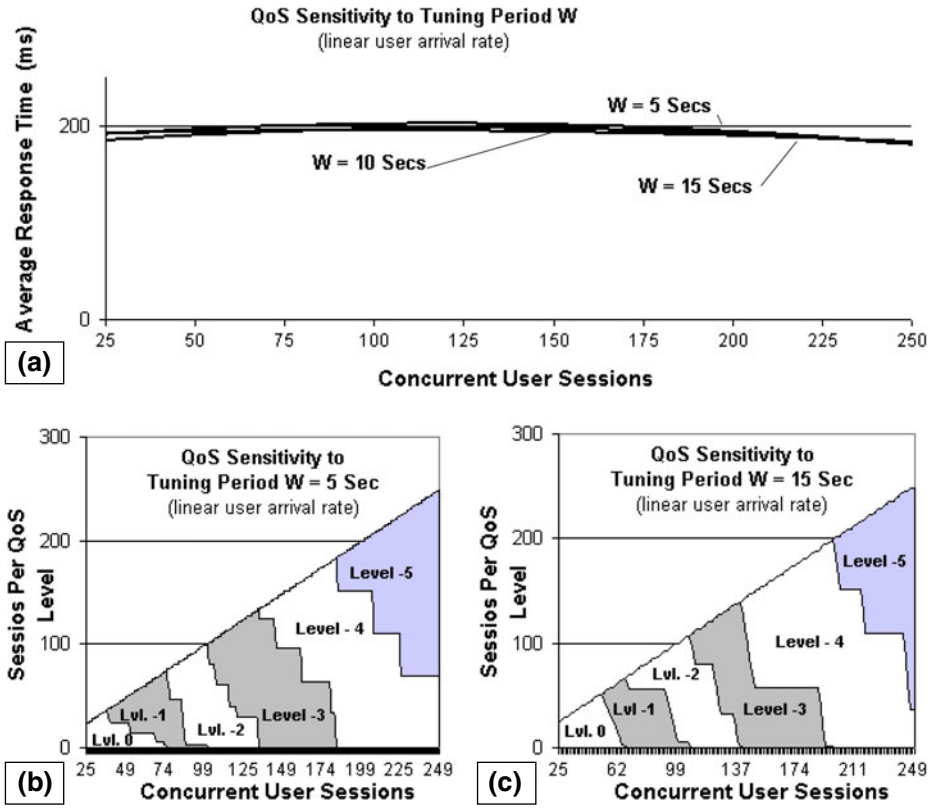


Figure 11 Evaluation of the QoS controller—sensitivity to tuning period W

the average response time is below the QoS-threshold. We continue by adding 50 new user sessions in the next ten seconds and stop the experiment when the average response time is stabilized below the QoS-threshold.

The results of this experiment (Figure 12a), show that the system is more responsive to bursty user arrivals when the UserDiff value is higher. On the other hand, a system with a lower UserDiff value not only requires considerably more time to stabilize the average response time but also suffers from increased response times. This is attributed to the slow pace of degrading users to lower QoS levels, as already discussed in our previous experiment of using linear arrival of users.

Experiment 4 We run a similar experiment as Experiment 3 in which we test the system's sensitivity to the length of the tuning period W with values of 5, 10 and 15 s. The UserDiff is fixed at a modest 60 %. The experiment produced some very interesting results (Figure 12b). A more lengthy tuning period leads to a slower initial response to the bursty arrival of users. However, it stabilizes the average response time faster due to the fact that more time is given to the system to degrade users between successive tunings. On the other hand, a short tuning period identifies the bursty arrival of users earlier and starts to respond quicker. However, it leads to

Author's personal copy

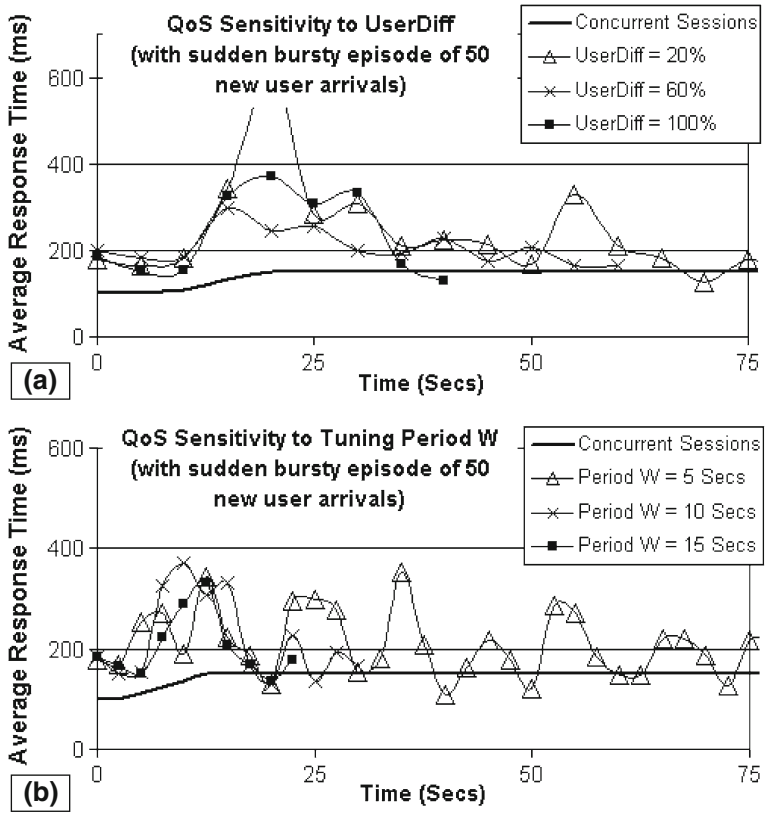


Figure 12 Evaluation of the QoS controller—bursty arrival of users

lengthy periodic increases and decreases of the average response time until that is stabilized below the QoS-threshold. This is because a shorter tuning period degrades only a fraction of user sessions between successive tunings and, as a result, response times are only momentarily pushed below the QoS-threshold.

6.3 Evaluating the QLS algorithm

For evaluating our QLS algorithm against the QoIF approach, we use a linear rate of user arrivals because we are only interested in examining how data quality is affected by the number of concurrent user sessions. For this reason, we use a low UserDiff value of 20 % that has proven to maintain the average response time below the QoS-threshold without being aggressive. In addition, we use a tuning period W of ten seconds which, in our previous experiments, was found to be effective without being aggressive.

Our first set of experiments compares QLS to the QoIF approach on the percentage of pages served with broken links. The results of the experiments (Figure 13, dotted lines) show that the percentage of pages with broken links is proportional

Author's personal copy

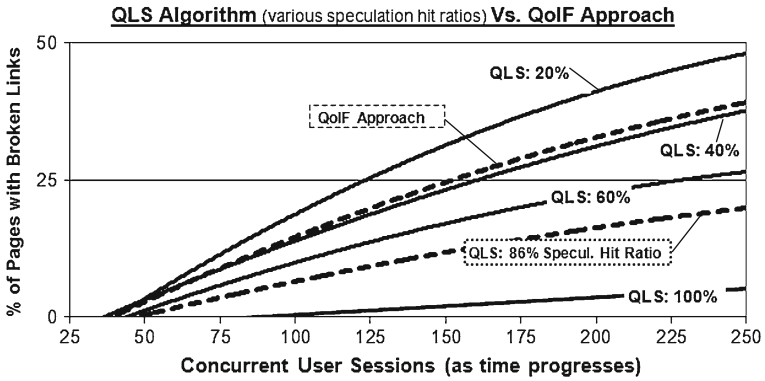


Figure 13 Evaluation of the QLS Vs traditional approach on broken links

to the workload. This is because increased workload implies that more users are degraded toward lower QoS levels, and therefore more fragments are served from cache having outdated links. Moreover, the results clearly state that QLS generates approximately 50 % less pages with broken links than the QoIF approach, even at high workload. This is because QLS selects the fragments for materialization with link dependencies on the next speculated template of the user.

Our analysis has shown that the Speculation Profiling Module used by QLS has a hit ratio of 86 % in speculation correctly the next template that the user will request. Figure 13 (solid lines) also plots the performance of QLS by setting the speculation hit ratio manually. The results suggest that our QLS is still better than the QoIF approach even at a *p*oor speculation ratio of 40 %, without the use of any sophisticated web mining technique.

6.4 Evaluating the QLSV algorithm

In this set of experiments, we compare our QLSV algorithm to the QoIF approach on the percentage of broken links and then on the percentage of unsatisfied set-view dependencies. That is pairs of set-view dependent fragments that are served to the user unsynchronized. For these experiments, we run QLSV with relax factors for QoLF equal to 0, 10, 20 and 30 %. Recall that, the relax factor reduces the maximum possible QoLF of the materialization plans in order for the algorithm to select the plan with the maximum possible QoSF value.

The results, plotted in Figure 14 show that QLSV serves less unsynchronized set-view dependent fragments than the QoIF Approach that has no related provision whatsoever. The gains are greater by using a higher QoLF relax factor of 30 %. However, the results come at a cost for the QoLF. Figure 15 plots the percentage of broken links for the four runs of QLSV. The obvious reductions on the previous gains of QLS are attributed to the reduced QoLF imposed by the QoLF relax factor.

6.5 Throughput and scalability

As mentioned earlier, we express performance in terms of throughput (server-side response time) and scalability in terms of number of concurrent user sessions. Our

Author's personal copy

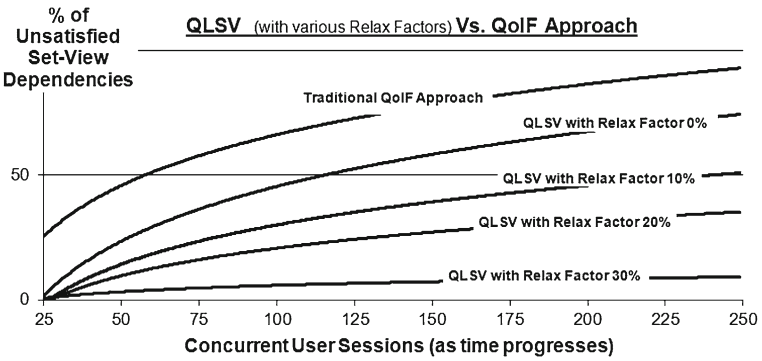


Figure 14 QLSV vs traditional approach on unsynchronized content

last experiment measures the maximum throughput and concurrent users that can be sustained by the QoIF approach, the QLS algorithm and its QLSV variation with relax factors of 0, 10, 20 and 30 %. In other words, this experiment measures the "industrial potential" of our algorithms. This experiment differs from the previous since it provides support for handling broken links in cached fragments on the user's browser. To implement this, we alter the normal request sequence of a user when a template with a cached fragment containing a needed link is received. When this occurs, the user issues an extra special HTTP GET request to the server *in order to receive afresh only the missing fragment* containing the valid link. Subsequently, the user resumes its template request sequence.

The results of this experiment, illustrated in Figure 16 show that both QLS and QLSV outperform the QoIF approach. QLS in particular achieves higher throughput by sustaining about 25 % more concurrent users than the QoIF approach. This is attributed to 50 % less extra load on the server to handle the special HTTP GET request issued by users for missing fragments. Subsequently, the gains are reduced for QLSV since a higher relax factor on QoLF generates more broken links than QLS.

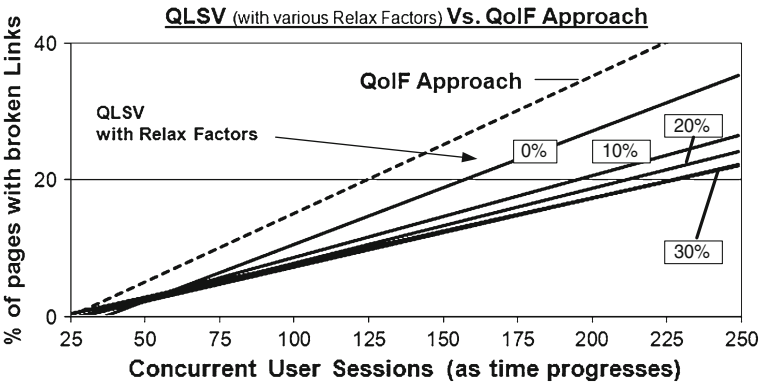


Figure 15 Evaluation of the QLSV vs traditional approach on broken links

Author's personal copy

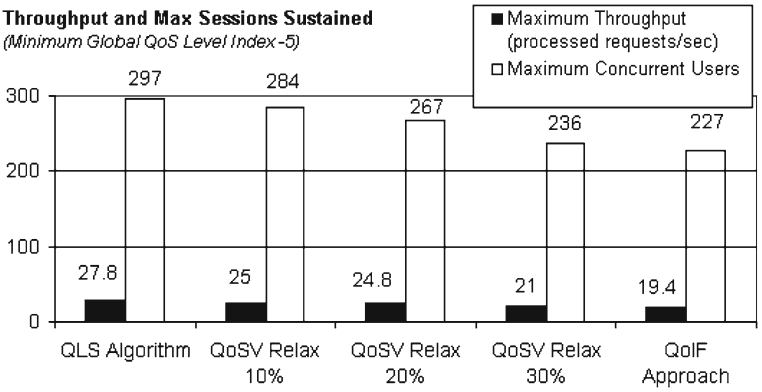


Figure 16 Throughput and maximum user sessions of QoIF, QLS and QLSV

7 Conclusion and future work

In this paper, we considered the problem of meeting user QoS and QoD expectations in dynamic web database applications under heavy load and identified the shortcomings of current dynamic web page materialization approaches, which blindly trade QoD for QoS. These shortcomings are basically failings to consider data dependencies between content fragments in a dynamic Web page. In particular, we identified as two such key dependencies, the link dependencies that ensure correct navigation and the set-view dependencies which ensure data consistency among the fragments of a page.

To mitigate these shortcomings, we introduced two new QoD metrics, namely QoLF and QoSF and proposed two novel materialization algorithms of content fragments, namely QLS and QLSV, that optimize these new metrics. The performance advantages of our two materialization algorithms, including their scalability, were experimentally demonstrated by using a real web database application of an online bookstore. Our proposed algorithms achieve their performance by considering content dependencies and user access patterns when selecting which fragments to materialize and which to reuse from the cache when generating a web page.

Both QLS and QLSV are able to meet user and application QoS requirements while incurring less impact on QoD compared to previous QoS-QoD balancing methods. As opposed to QLSV, the QLS algorithm is more suitable in situations characterized by more frequent user clicks—more impatient users—where response time matters the most. The QLSV algorithm is more suitable for users that prefer to spend more time exploring a single instance of a dynamic Web page.

Although the online bookstore is the canonical example of the majority of e-commerce web applications and online stores, our next step is to evaluate our approach in the context of other web database applications with larger web sites and larger databases such as social networks, technical forums and newsgroups.

Currently, our proposed solution does not allow for the weights of the fragments in our QoD metrics to be dynamic themselves. These weights capture the importance of the dependencies of the individual fragments and are predefined for all users. We are planning to develop a more dynamic fragment weighting mechanism that customizes

Author's personal copy

QoD metrics based on user access patters as a supplement to our user speculation module. Such a mechanism would further enhance performance by achieving a more personalized QoS-QoD balance.

References

1. Arlitt, M.: Characterizing web user sessions. SIGMETRICS Perform. Eval. Rev. **28**(2), 50–63 (2000)
2. Arlitt, M.F., Williamson, C.L.: Internet web servers: workload characterization and performance implications. IEEE/ACM Trans. Networking **5**(5), 631–645 (1997)
3. Bright, L., Raschid, L.: Using latency-recency profiles for data delivery on the web. In: VLDB, pp. 550–561 (2002)
4. Cunha, C., Bestavros, A., Crovella, M.: Characteristics of www client-based traces. Boston University, Tech. Rep. TR-95-010 (1995)
5. Guirguis, S., Sharaf, M.A., Chrysanthis, P.K., Labrinidis, A., Pruhs, K.: Adaptive scheduling of web transactions. In: ICDE, pp. 357–368 (2009)
6. Labrinidis, A., Roussopoulos, N.: Webview materialization. SIGMOD Rec. **29**(2), 367–378 (2000)
7. Labrinidis, A., Roussopoulos, N.: Exploring the tradeoff between performance and data freshness in database-driven web servers. VLDB J. **13**(3), 240–255 (2004)
8. Labrinidis, A., Luo, Q., Xu, J., Xue, W.: Caching and materialization for web databases. Foundations and Trends in Databases **2**(3), 169–266 (2009)
9. Larson, P.-Å., Goldstein, J., Zhou, J.: Mtcache: transparent mid-tier database caching in sql server. In: ICDE, pp. 177–189 (2004)
10. Li, W.-S., Po, O., Hsiung, W.-P., Candan, K.S., Agrawal, D.: Engineering and hosting adaptive freshness-sensitive web applications on data centers. In: WWW, pp. 587–598 (2003)
11. Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B.G., Naughton, J.F.: Middle-tier database caching for e-business. In: SIGMOD, pp. 600–611 (2002)
12. Mah, B.A.: An empirical model of http network traffic. In: INFOCOM, p. 592 (1997)
13. Menascé, D.A.: Testing e-commerce site scalability with tpc-w. In: CMG Conference, pp. 457–466 (2001)
14. Olshefski, D.P., Nieh, J., Nahum, E.: Ksniffer: determining the remote client perceived response time from live packet streams. In: OSDI, pp. 333–346 (2004)
15. Oke, A., Bunt, R.B.: Hierarchical workload characterization for a busy web server. In: TOOLS, pp. 309–328 (2002)
16. Padmanabhan, V.N., Qiu, L.: The content and access dynamics of a busy web site: findings and implications. SIGCOMM Comput. Commun. Rev. **30**(4), 111–123 (2000)
17. Papastavrou, S.: Semantics-based Metrics and Algorithms for Dynamic Content in Web Database Applications. Ph.D. dissertation, University of Cyprus (2009)
18. Papastavrou, S., Samaras, G., Evripidou, P., Chrysanthis, P.K.: A decade of dynamic web content: a structured survey on past and present practices and future trends. IEEE Communications Surveys & Tutorials **8**(2), 52–60 (2006)
19. Papastavrou, S., Chrysanthis, P.K., Samaras, G.: Exploring content dependencies to better balance performance and freshness in web database applications. In: WISE, pp. 512–525 (2012)
20. Paulson, L.D.: Building rich web applications with ajax. IEEE Comput. **38**(10), 14–17 (2005)
21. Qu, H., Labrinidis, A.: Preference-aware query and update scheduling in web-databases. In: ICDE, pp. 1–10 (2007)
22. Schmitt, B., Oberlnder, S.: Access evaluation of digital libraries: characteristics and performance of web opacs. In: Second Int. Workshop on New Developments in Digital Libraries (2002)
23. Schroeder, B., Harchol-Balter, M.: Web servers under overload: how scheduling can help. ACM Trans. Internet Technol. **6**(1), 20–52 (2006). doi:10.1145/1125274.1125276
24. Srivastava, J., Cooley, R., Deshpande, M., Tan, P.-N.: Web usage mining: discovery and applications of usage patterns from web data. SIGKDD Explor. Newsl. **1**(2), 12–23 (2000)
25. Wang, Q., Makaroff, D., Edwards, H.K., Thompson, R.: Workload characterization for an e-commerce web site. In: CASCON, pp. 313–327 (2003)
26. Wilson, T.: Review of gmail. Inf. Res. **10**(1) (2004). http://www.informationr.net/ir/reviews/sofrev17/sofrev17.html