# Exploring Content Dependencies to Better Balance Performance and Freshness in Web Database Applications

Stavros Papastavrou[1], Panos K. Chrysanthis[2,*], George Samaras[1,**]

[1] Dept. of Computer Science, University of Cyprus, Nicosia, Cyprus
stavros@schoolfortheblind.net,
cssamara@ucy.ac.cy
[2] Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA
panos@cs.pitt.edu

**Abstract.** In this paper, we present a novel approach for materializing dynamic web pages by exploiting content dependencies and user access patterns. We introduce two new semantic-based data freshness metrics and show that our approach out-performs traditional balancing QoS-QoD approaches in terms of server throughput, increased data freshness and scalability. In our evaluation we use a real-world experimental system that resembles an online bookstore web database application.

**Keywords:** Caching, Dynamic Web Content, QoD, QoS, Data Freshness.

## 1 Introduction

Our work focuses on e-commerce web applications such as an online bookstore. Those 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 [1]. The study in [2] presents a comprehensive and comparative listing of early approaches for enhancing QoS (user-perceived latency) under heavy workload in the blind expense of QoD (freshness of data served). Improving on [2], the approaches in [3, 4, 5, 6, 7, 8] 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, 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.

* Partially Supported by the USA National Science Foundation Award OIA-1028162.
** Co-funded by the EU Project CONET (INFSO-ICT-224053) & The Project FireWatch (#0609-BIE/09), Sponsored by the Cyprus Research Promotion Foundation.

In this paper, we pose that current QoS-QoD balancing approaches fail to meet the requirements of modern Web database applications 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 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.

**Contributions.** 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, 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. We present two content materialization algorithms that balance QoS with data freshness in terms of the proposed QoLF and QoSF metrics. Our experimental findings show that our algorithms outperform traditional QoS-QoD approaches 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.** Next, we present the underlying assumptions of our work and existing content materialization approaches. In Section 3, we present our approach for QoS-QoD balancing for materialization and in Section 4, our materialization algorithms. In Section 5, we discuss our performance evaluation and conclude in Section 6.

## 2 System Model and Related Work

### 2.1 Basic Assumptions

The system model for user-driven, personalized e-commerce web database applications with infrequent database updates is based on the typical client / proxy / web server / application server / application database(s) architecture. All the components may have a 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 [9, 10, 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.

The web server is the public entry point of the application and immediately serves request for static content (style sheets, images). Requests for a dynamic web page are routed to an application server that executes the corresponding template file. Template files include script blocks that relate to web page content fragments that are either materialized from scratch or reused from the cache. The materialization of a fragment includes queries on the application database(s) and formatting/wrapping of their results with HTML. Finally, all the fragments, cached or freshly materialized, are assembled together according to the template file and transmitted to the user through the web server.

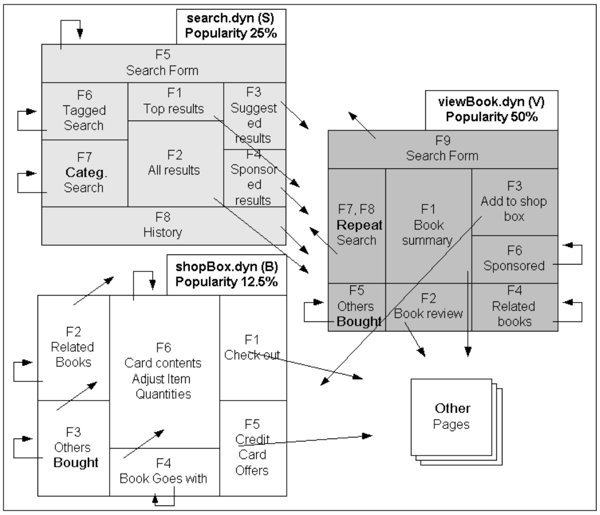


**Fig. 1.** Breakdown of the bookstore application

Let us take, for example, an online bookstore application with 20 different templates for dynamic web pages of which 4 account for the 95% of user accesses (Fig. 1). The most popular page is a search page (template: search.php) that provides search results, category listings and suggested books in various fragments. The second most popular page is the book viewing page (template: viewBook.php) that presents all information about a selected book in two fragments, related books listing in other 5 fragments plus 1 fragment for adding the book into the shopping card with different options ($F_{add}$). Typically, the user can navigate from search.php to viewBook.php by picking up a book link. The third book is the shopping card page (template: shopBox.php) that confirms the addition of a book into the shopping card and provides additional suggested listings for direct addition into the shopping card or for viewing. Typically, the user navigates from viewBook.php to shopBox.php by clicking on a link from within the $F_{add}$ fragment).

Fragment materialization in our model is analogous to virtual WebViews [12]. 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.*

## 2.2 Current Approach (QoIF Approach) and Shortcomings

Current approaches balance QoS and QoD by varying the number of fresh fragments per template request according to the *individual* importance of their containing fragments [5, 6, 7, 8]. 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. 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 $F_1, F_2, ..., F_n$ are all the member fragments of a template T, then weightIF(F,T) ∈(0,1) and*

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

**Definition 4.** countIF(F,T) *The countIF(F,T) of a fragment F in template T is*

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

Given that the current approach focuses on the importance of individual fragments, we refer to their adopted QoD metric as QoIF (Quality of Individual Fragments) and to the current approach as the *QoIF approach.*

**Definition 5.** (QoIF) *Let F1, F2, ..., Fn 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 the problems are:

*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 link inside fragment $F_{source}$ that links to $T_{dest}$.*

The links between templates are dynamic, in the sense that their parameters are not hardcoded. If a fragment that includes a needed link for navigation is reused from cache, because of its relative low QoIF importance weight, then it does not contain valid links for the user to navigate. This unsatisfied dependency (also called 'broken link') stalls the user session until a fresh version of the fragment is received.

*2) No Provision for Set-View Dependencies.*

**Definition 7.** (Set-View Dependency) *A fragment $F_i$ in template T is set-view dependent on fragment $F_j$ in the same template T if both fragments must present consistent (synchronized) information.*

The QoIF approach, which handles fragments independently, fails to synchronize the materialization of set-view dependent fragments since the importance factor employed is fragment-wise and does not force two fragments to be materialized or reused from cache at the same user request for their template.

## 3 Our Approach for QoS-QoD Balancing

Our approach for balancing QoS with data freshness 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 at met at that particular point in time. Our approach is broken down into the following three sub-goals (or components):

– **Ensure QoS.** Constantly calculate the maximum possible quantity of fragments per template request that must be materialized in order to keep the average response time below a predefined QoS threshold (measured in ms),
– **Speculation.** Employ user access patterns to 'guess' the next template that a user will request,
– **Ensure QoD.** Indicate the appropriate mixture of fragments per template request that satisfy link and set-view dependencies to the highest possible degree in order to reduce broken links and unsynchronized content.

Since our main focus in this paper is content dependencies, we discuss only in brief how we regulate QoS and the methodology of request speculation. The detailed descriptions on QoS regulation and user speculation can be found in [20].

## 3.1 Ensuring QoS – The QoS Controller

Two essential parameters for ensuring QoS are the maximum tolerable response time (*QoS-threshold*) and the average response time of currently active user sessions (*QoS-average*). The former defines a threshold for the latter which, when violated/crossed, triggers the QoS Controller to take a corrective action. At run time, the QoS-average of active user sessions is computed every tuning period of W seconds. If it is found steadily higher than QoS-threshold, the QoS Controller attempts to lower it by issuing a controlled decrease on the suggested maximum number of fragments that are materialized per template until the next tuning period. This decrease is progressively applied to a percentage of active user sessions per tuning period. In addition, all active user sessions must be affected at least once before the QoS Controller issues any additional decreases as necessary. As soon as the average response time is stabilized below the threshold, the decrease is suspended. In this case, the procedure can be reversed by issuing an increase on the maximum number of fragments per template for materialization. We refer to the action of applying a decrease to a user session as "degrading the user" or "dropping the user". We refer to "upgrade" for the opposite action.

In order to implement this QoS policy, we use two QoS level indexes. The first is called *Global QoS Level* and indicates the suggested number of fragments to be reused from cache per template request. The second is called *User QoS Level* and indicates the actual number of fragments per template to be reused from cache for a particular user. Initially, at light workload, the Global QoS Level is set to 0. The User QoS Level is also set to 0 for all currently active users. Every W seconds, the QoS-average is checked. If it is found to be steadily below the QoS-threshold, then the Global QoS Level is decreased to -1. If workload continues to increase, then a requirement for any extra decrease to the Global QoS Level to -1 is that, all current users have been degraded to -1. In Section 4, we examine how the materialization algorithm regulates the User QoS Level and the directive flags.

## 3.2 Speculation – The Usage Plans

The second sub-goal of our approach is the speculation on the next template that a user will request. Since user speculation is not the main focus of this paper, we only briefly discuss here a simple speculation scheme based on data mining findings, which we use to implement the speculation module of our materialization algorithms.

According to [13], 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 web database applications, a small set of templates (approximately four) account for almost 95% of the requests [14], where this set of templates is stable over time [15]. Similarly, [16], [17] 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 order to encode recurrent access patterns, we introduce the notion of *Usage Plans* (UP) that encapsulate looping user behavior. Figure 2 presents five Usage Plans of the bookstore application of the three most popular templates of the application. Note how two Usage Plans do not share the same template transition. In other words, every transition between two templates is a member of only one UP. This restriction is very important because it allows us to define a session to consist solely of a sequence of non-overlapping UP. For example, a user initially performs a search for a book three times in a row using template 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. This sequence of template requests is shown in Figure 3 along with the projected Usage Plans that emerge (S*, (SV)* etc.).

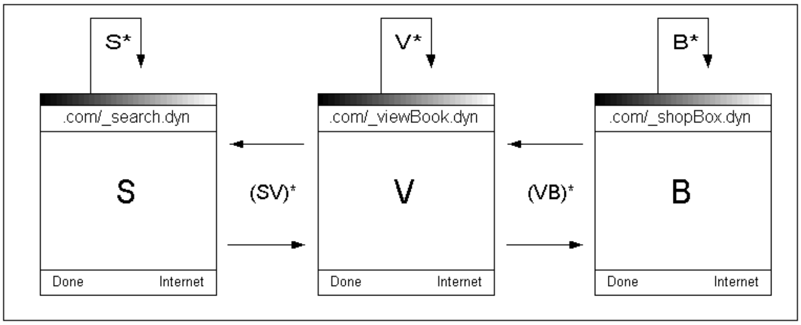


**Fig. 2.** Five Usage Plans of the Bookstore Application: Three uni-usage plans S*, V*, B*, and two bi-usage plans (SV)* and (VB)*

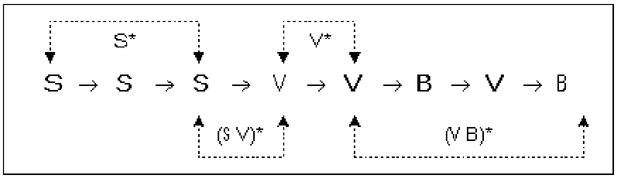


**Fig. 3.** A Session Illustrated as a Sequence of Usage Plans. Note that each usage plan is immediately followed by another one.

Having established that a user session consists of non-overlapping UPs, we present in brief our simple speculation methodology. We distinguish the UPs to uni-UPs and bi-UPs. The former involve only one-template looping, such as S* and V*. The latter involve two templates such as (SV)* or (VB)*. For each user, we use a FSM Module with three states: (a) "the user is on a uni-UP", (b) "the user is on a bi-UP, and (c) "the user is a moving from a uni-UP to a bi-UP". As the user links between templates, the state on the FSM is changed accordingly. To "speculate" on the next template that the user will request, we use pattern matching that encodes the typical behavior of a user using the FSM and the user's pervious behavior as inputs.

### 3.3 Ensure QoD – The New QoLF and QoSF Data Freshness Metrics

To better weight the importance of content fragments, we introduce two new semantics-based metrics as follows:

**Quality of Link Fragments** (QoLF): This metric quantifies the existence of freshly materialized fragments inside a template $T_s$ with link dependencies toward a target template $T_d$. QoLF applies importance weights on fragments toward link-dependent templates.

**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 $T_d$. 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*

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

If a 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 Td.

**Quality of Set-view Fragments** (QoSF): The metric of QoSF quantifies the overall set-wise consistency of set-view dependent fragments inside a template. Similarly, we use an importance weight that measures the importance of materializing two set-wise dependent fragments in a template.

**Definition 10.** *(weightSF(Fi,Fj,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 in T, weightSF($F_i$, $F_j$ , T) $\in$ (0, 1) and*

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

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

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

Fragments $F_i$ and $F_j$ are synchronized in template T if both are materialized or reused from cache. 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*

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

## 4 Materialization Algorithms

In the previous section, we examined how QoS is regulated by the increase or decrease of the Global QoS Level index and introduced the notion of Usage Plans and the new metrics of QoLF and QoSF for measuring data freshness, given the link and set-view dependencies of content fragments. In this section, we explain how we organize QoS Level index, Usage Plans and the new data quality metrics into one convenient structure called MP Selection Table and show how it is used by our materialization algorithms.

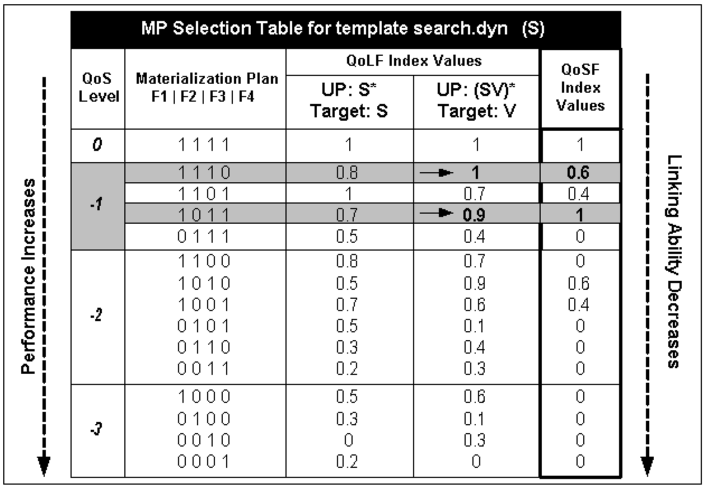
### 4.1 Putting It All Together: The MP Selection Table

The MP Selection Table is a structure that summarizes all combinations of fresh/cached fragments, for a specific template, into groups according to a QoS Level Index. Those combinations are called *Materialization Plans* (MP). For example, at level -1, the table lists 4 possible MP of 3 fresh and 1 cached fragments. Figure 4 shows the MP Selection Table for template search.php (S) (for ease of presentation we show only 4 fragments). The MP '1111' of a template with four fragments implies that all fragments are materialized. 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 is computed (see Definition 9). In our example, template S links to its self and template viewBook.php (V). In the right-most column (Figure 4), the QoSF for each MP is computed (see Definition 12).

**The QLS Algorithm.** We first present the QLS algorithm that considers only link dependencies. One instance of the materialization algorithm (shown in Figure 5) is attached to every user session. On each user request for template $T_x$, the algorithm first secures QoS for the user by increasing or decreasing the User QoS Level, if necessary (lines 4 to 5). Using the user's new QoS level, the algorithm isolates the group of candidate MPs from the MP Selection Table (line 6).

To ensure QoD, the algorithm uses the FSM Module to speculate on the next template that the user will link to from template $T_x$ (line 8-9). Then, the algorithm selects the MP from the group of candidate MPs with the user's QoS Level that maximizes QoLF toward the speculated template. Finally, the fragments that correspond to '1' in the selected MP are materialized from scratch while the fragments with '0' are reused form cache.

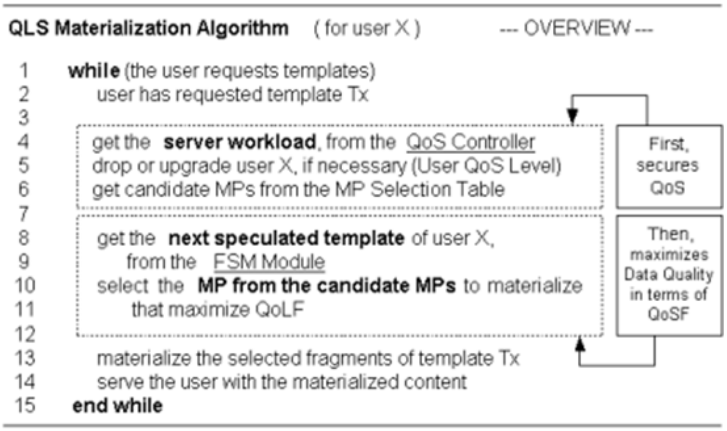


**MP Selection Table for template search.dyn (S)**

| QoS Level | Materialization Plan F1 \| F2 \| F3 \| F4 | QoLF Index Values | | QoSF Index Values |
|---|---|---|---|---|
| | | UP: S* Target: S | UP: (SV)* Target: V | |
| *0* | 1 1 1 1 | 1 | 1 | 1 |
| *-1* | 1 1 1 0 | 0.8 | ⟶ **1** | **0.6** |
| | 1 1 0 1 | 1 | 0.7 | 0.4 |
| | 1 0 1 1 | 0.7 | ⟶ **0.9** | **1** |
| | 0 1 1 1 | 0.5 | 0.4 | 0 |
| *-2* | 1 1 0 0 | 0.8 | 0.7 | 0 |
| | 1 0 1 0 | 0.5 | 0.9 | 0.6 |
| | 1 0 0 1 | 0.7 | 0.6 | 0.4 |
| | 0 1 0 1 | 0.5 | 0.1 | 0 |
| | 0 1 1 0 | 0.3 | 0.4 | 0 |
| | 0 0 1 1 | 0.2 | 0.3 | 0 |
| *-3* | 1 0 0 0 | 0.5 | 0.6 | 0 |
| | 0 1 0 0 | 0.3 | 0.1 | 0 |
| | 0 0 1 0 | 0 | 0.3 | 0 |
| | 0 0 0 1 | 0.2 | 0 | 0 |

Performance Increases
Linking Ability Decreases

**Fig. 4.** The MP Selection Table for template search.php



```
QLS Materialization Algorithm   ( for user X )          --- OVERVIEW ---

1   while (the user requests templates)
2       user has requested template Tx
3
4       get the  server workload, from the  QoS Controller
5       drop or upgrade user X, if necessary (User QoS Level)
6       get candidate MPs from the MP Selection Table
7
8       get the  next speculated template  of user X,
9           from the  FSM Module
10      select the  MP from the candidate MPs  to materialize
11          that maximize QoLF
12
13      materialize the selected fragments of template Tx
14      serve the user with the materialized content
15  end while
```

**Fig. 5.** The QLS Materialization Algorithm

For example, if a user with QoS level equals to -1 has requested template search.php (S), and the speculation FSM Module returns that the user will subsequently request template viewBook.php (V), the algorithm will examine the candidate MPs with QoS Level -1 index and select the MP '1110' which has the highest QoLF value.

**The QLSV Algorithm.** The QLSV variation of the QLS algorithm considers additionally the QoSV value of the candidate MPs toward increasing the synchronization of set-view dependent fragments. An additional Relax Factor is used to indicate the tolerance on the loss of the QoLF of the selected MP. In our example above, with a Relax Factor of 0%, the algorithm would have select the MP '1110' with a relatively low QoLF value of 0.6 while with a Relax Factor of only just 10%, the MP '1011' with much higher QoSV is selected. In other words, the QLSV variation increases the synchronization of content in a dynamic web page at the expense of linking dependencies.

## 5 Evaluation

**Setup.** The evaluation is performed on an experimental platform that emulates a real-world bookstore web database application. 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 IV. 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 SQL Server 2008. 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.php template executes one update on the application database for placing (or removing) a book in a user's shopping box.

For the client 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 only receiving a fresh version of a fragment that was served from cache. Our synthetic workload follows basic principles according to the transactional web e-Commerce benchmark (TPC-W) [18], 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 [19].

**Evaluation of the QLS Algorithm**. Our first set of experiments compares QLS to the current QoIF approach on the percentage of pages served with broken links. The results of the experiment (Figure 6a, dotted lines) show that this percentage is proportional to the workload. This is because increased workload implies that more users are dropped toward lower QoS levels, and therefore more fragments are served from cache with 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 Module used by QLS has a hit ratio of 86% in speculating correctly the next template that the user will request. However, Figure 6a (solid lines) plots the performance of QLS by setting the speculation hit ratio manually. The results suggest that our QLS outperforms QoIF even at such a low speculation hit ratio of 40%.

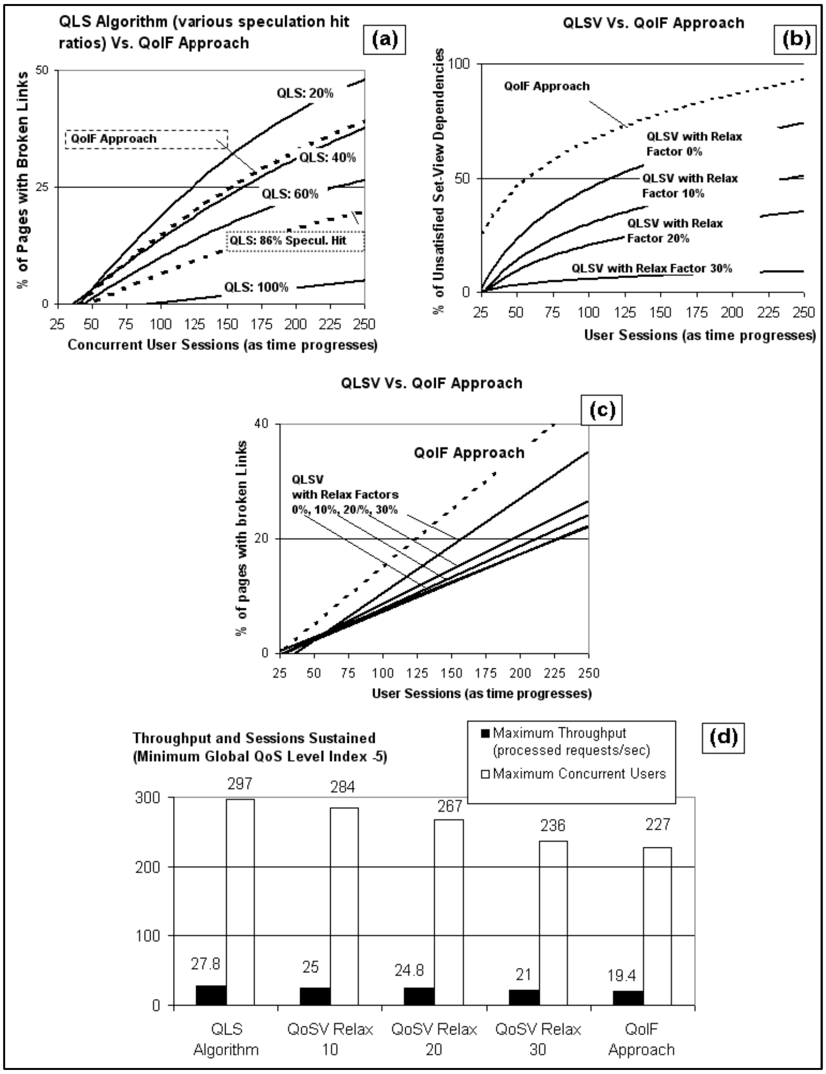


**Fig. 6.** The Performance Results

**Evaluation of the QLSV Variation.** In this set of experiments, we compare our QLSV algorithm to the QoIF approach. First, we compare the two approaches on the percentage of unsatisfied set-view dependencies. That is pairs of set-view dependent fragments that are served to the user unsynchronized. Then, we compare them on the percentage of broken links. 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 materialization plans in order for the algorithm to select the plan with the maximum possible QoSF value. The results (Figure 6b) 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 6c 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.

**Throuput and Maximum Sessions Sustained.** Our last experiment measures the maximum throughput and concurrent users that can be sustained by the QoIF approach, QLS algorithm and its QLSV variation using QoLF 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. 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 special request to the server in order to receive fresh only the missing fragment that contains valid links. Subsequently, the user resumes its template request sequence. The results of this experiment (Figure 6d) 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 at 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 generates more broken links than QLS.

## 6 Conclusion

In this paper, we considered the problem of meeting user QoS expectations in dynamic web database applications under heavy load and we identified the shortcomings of current approaches, which trade QoD for QoS. To mitigate these shortcomings, we proposed two new materialization algorithms, namely QLS and QLSV, for dynamic web pages that can meet user QoS requirements while incurring less impact on the 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. 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. 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. 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 technical forums and newsgroups.

## References

1. Olshefski, D.P., Nieh, J., Nahum, E.: Ksniffer: Determining theremote client perceived response time from live packet streams. In: OSDI 2004, pp. 333–346 (2004)
2. Papastavrou, S., Samaras, G., Evripidou, P., Chrysanthis, P.K.: Adecade of dynamicweb content: A structured survey on past and present practices and future trends. IEEE CS & T 8(2), 52–60 (2006)
3. Schroeder, B., Harchol-Balter, M.: Web servers under overload: Howscheduling can help. ACM Trans. Inter. Tech. 6(1), 20–52 (2006)
4. Guirguis, S., Sharaf, M.A., Chrysanthis, P.K., Labrinidis, A., Pruhs, K.: Adaptive scheduling of web transactions. In: ICDE, pp. 357–368 (2009)
5. Bright, L., Raschid, L.: Using latency-recency profiles for datadelivery on the web. In: VLDB, pp. 550–561 (2002)
6. Labrinidis, A., Roussopoulos, N.: Exploring the tradeoff between performance and data freshness in database-driven web servers. VLDB J. 13(3), 240–255 (2004)
7. Li, W.S., Po, O., Hsiung, W.P., Candan, K.S., Agrawal, D.: Engineeringand hosting adaptive freshness-sensitive web applications on data centers. In: WWW, pp. 587–598 (2003)
8. Qu, H., Labrinidis, A.: Preference-aware query and update scheduling in web databases. In: ICDE, pp. 1–10 (2007)
9. Larson, P.-A., Goldstein, J., Zhou, J.: Mtcache: Transparent mid-tier database caching in sql server. In: ICDE, pp. 177–189 (2004)
10. Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B.G., Naughton, J.F.: Middle-tier database caching for ebusiness. In: SIGMOD 2002, pp. 600–611 (2002)
11. Labrinidis, A., Luo, Q., Xu, J., Xue, W.: Caching andMaterialization in Web Databases. Foundations and Trends in Databases 3(2), 169–266 (2009)
12. Labrinidis, A., Roussopoulos, N.: Webview materialization. SIGMOD Rec. 29(2), 367–378 (2000)
13. Wang, Q., Makaroff, D., Edwards, H.K., Thompson, R.: Workloadcharacterization for an e-commerce web site. In: CASCON, pp. 313–327 (2003)
14. Arlitt, M.: Characterizing web user sessions. SIGMETRICS Perform. Eval. Rev. 28(2), 50–63 (2000)
15. 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)
16. Cunha, C., Bestavros, A., Crovella, M.: Characteristics of www client-based traces. Boston University, Tech. Rep. TR-95-010 (1995)
17. Oke, A., Bunt, R.B.: Hierarchical workload characterization for abusy web server. In: OOLS, pp. 309–328 (2002)
18. Menasce, D.A.: Testing e-commerce site scalability with tpc-w. In: CMG Conference, pp. 457–466 (2001)
19. Mah, B.A.: An empirical model of http network traffic. In: INFOCOM, p. 592 (1997)
20. Papastavrou, S.: Semantics-based metrics and algorithms for dynamic content in web database applications. PhD dissertation, LC: TK5105.5.P37, CSD, University of Cyprus (2009)