The VLDB Journal (2004) 13: 240–255 / Digital Object Identifier (DOI) 10.1007/s00778-004-0131-7

Exploring the tradeoff between performance and data freshness in database-driven Web servers

Alexandros Labrinidis[1], **Nick Roussopoulos**[2]

[1] Department of Computer Science, University of Pittsburgh, Pittburgh, PA, 15260, USA (e-mail: labrinid@cs.pitt.edu)
[2] Department of Computer Science, University of Maryland, College Park, MD, 20742, USA (e-mail: nick@cs.umd.edu)

Edited by S. Abiteboul. Received: January 17, 2004 / Accepted: March 23, 2004
Published online: August 19, 2004 –

Abstract. Personalization, advertising, and the sheer volume of online data generate a staggering amount of dynamic Web content. In addition to Web caching, view materialization has been shown to accelerate the generation of dynamic Web content. View materialization is an attractive solution as it decouples the serving of access requests from the handling of updates. In the context of the Web, selecting which views to materialize must be decided online and needs to consider both performance and data freshness, which we refer to as the online view selection problem. In this paper, we define data freshness metrics, provide an adaptive algorithm for the online view selection problem that is based on user-specified data freshness requirements, and present experimental results. Furthermore, we examine alternative metrics for data freshness and extend our proposed algorithm to handle multiple users and alternative definitions of data freshness.

1 Introduction

The frustration of broken links from the early Web has been replaced today by the frustration of Web servers stalling or crashing under the heavy load of dynamic content. In addition to data-rich online Web services, even seemingly static Web pages are usually generated dynamically in order to include personalization and advertising features. However, dynamic content has significantly higher resource demands than static Web pages (at least one order of magnitude) and creates a huge scalability problem at Web servers.

Dynamic Web caching [11,5,7,8,17,1] has been proposed to solve this scalability issue. The biggest problem of employing caching techniques for dynamic Web content is the coupling of serving access requests and handling of updates, since an update that invalidates a cached object will result in the object being recomputed on the next access request. For example, imagine a cache that can store dynamically generated Web pages. During normal operation we get an 80% hit rate (which means that only 20% of the pages will need to be recomputed). If we get a small surge in the update stream, a big percentage of the cached pages could be invalidated, and the hit rate will drop significantly. A sudden drop in hit rate leads to a sudden increase in the average response time and possibly to server saturation. View materialization can solve this problem since it decouples the serving of access requests from the handling of the updates.

Selecting which views to materialize, the *view selection problem*, has been studied extensively in the context of data warehouses [24,10,9,21]. However, unlike data warehouses, which are offline during updates, most Web servers maintain their backend databases online and perform updates concurrently with user accesses. Therefore, in the context of the Web, selecting which views to materialize must be decided dynamically and needs to consider both performance and data freshness.

In this paper, we present OVIS(θ), an adaptive algorithm for the online view selection problem. OVIS(θ) acts as a knob in the system, determining at runtime which views should be materialized (cached and refreshed immediately on updates) and which ones should just be cached and reused as necessary. The parameter θ corresponds to the level of data freshness that is considered acceptable for the current application. In addition to maintaining high performance given the data freshness demands, OVIS(θ) also detects infeasible thresholds, when the freshness demands would create a backlog at the Web server.

Motivating example: Our motivating example is a database-driven Web server that provides real-time stock information to subscribers. Updates to stock prices and other market derivatives are streamed to the backend database and must be performed online. The Web server provides users with up-to-date information that includes current stock prices, moving average graphs, comparison charts between different stocks, and personalized stock portfolio summaries. In general, we are interested in data-intensive Web servers that provide mostly dynamically generated Web pages to users (with data drawn from a DBMS) and also face a significant online update workload.

Structure of paper: In the next section, we present our metrics for measuring system performance and data freshness. We also define the online view selection problem. In Sect. 3 we describe the proposed online view selection algorithm, and in Sect. 4 we discuss the results of our experiments. Section 5 examines alternative metrics for data freshness, whereas Sect. 6 extends the proposed online view selection algorithm to han-

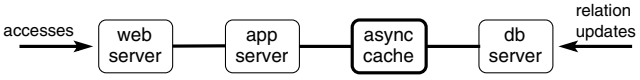

Fig. 1. System architecture

dle multiple users and alternative definitions of data freshness. Section 7 summarizes related work. We conclude in Sect. 8.

2 Problem definition

We extend the typical three-tier architecture of modern Web servers, by adding an *Asynchronous Cache* module, between the application server and the database server (Fig. 1).

In this architecture, the Web server module is responsible for serving user requests and the application server is responsible for Web workflow management. Instead of interfacing the application server directly to the database server, the Asynchronous Cache module acts as an intermediary. Unlike traditional caches in which data are simply invalidated on updates, data in the asynchronous cache can be *materialized* and immediately refreshed on updates. Recent products from IBM and Microsoft incorporate such an asynchronous middle-tier cache [2,15,16].

2.1 Web page derivation graph

There are three types of data objects in the system: relations, WebViews, and Web pages.

- **Relations** are stored in the database server and are the primary "storage" for structured data. They are affected by the incoming update stream. Relation updates are executed in order of arrival.
- **WebViews**, introduced in [18], are HTML or XML fragments. In other words, WebViews are simply parts of a Web page. WebViews are usually generated by "wrapping" database query results (i.e., database views) with HTML formatting commands or XML semantic tags. We allow WebViews to be formed from *any type* of database queries. In fact, the only assumptions that we must make for WebViews are that we:
  1. Have their current definition and corresponding query (which means that we must have the values of any parameters if the WebViews are generated by a parameterized query),
  2. Are able to determine when they become stale (in the most conservative case, this corresponds to whenever the source relations are updated), and
  3. Can uniquely address/name them (since we plan to cache them and must be able to perform lookups).

  We prefer the term WebView over the term "HTML fragment", which was introduced earlier, in order to stress that these HTML fragments are derived from a database. In fact, we will use the terms "view" and "WebView" interchangeably for the rest of the paper. In the general case, WebViews may be defined from other WebViews, but in this paper we focus on HTML WebViews that are directly derived from querying the database (as we will see later).
- **Web pages** are composed of one or more WebViews. Web pages are what the user is served with in response to his/her access requests.

A WebView W_j is derived from relation R_i if W_j includes data generated by querying R_i. A web page P_k is considered to be derived from WebView W_j if P_k contains W_j.

The associations between these data objects are depicted using a *Web page derivation graph*, which is a directed acyclic graph. The nodes of the graph correspond to relations, WebViews, or Web pages. An edge from node a to node b exists only if node b is derived directly from node a. A node can have multiple "parents"; therefore the in-degree of a node can be greater than one. Relations are the roots of the graph, with zero in-degree, and Web pages are the leafs of the graph, with zero out-degree. Figure 2 has an example of a Web page derivation graph. We assume a database with three relations (R_1, R_2, R_3), four WebViews (W_1, W_2, W_3, W_4), and two Web pages (P_1, P_2).

Figure 2 is a very small example of an actual Web page derivation graph. In practice, we usually have thousands of Web pages in a Web site, with dozens of HTML/XML fragments on each page [5]. However, we also expect to have a significant amount of WebView "sharing" among these Web pages. Imagine, for example, a personalized newspaper site. Each user selects the type of news to be included (e.g., local, national, economy), specifies a city for the weather forecast, and gives a list of stock symbols along with the purchase price and quantities for calculating his/her portfolio value. Although the combination of the above elements is most probably unique, there is clearly a finite number of cities/stock symbols that will be shared among thousands of users (in addition to the standard navigation/presentation fragments).

2.2 The Asynchronous Cache

All requests that require dynamically generated content are intercepted by the *Asynchronous Cache module* (ASC for short). ASC maintains WebViews using one of the following three policies.

Virtual WebViews are always executed on demand and never cached. Intercepted queries against virtual WebViews

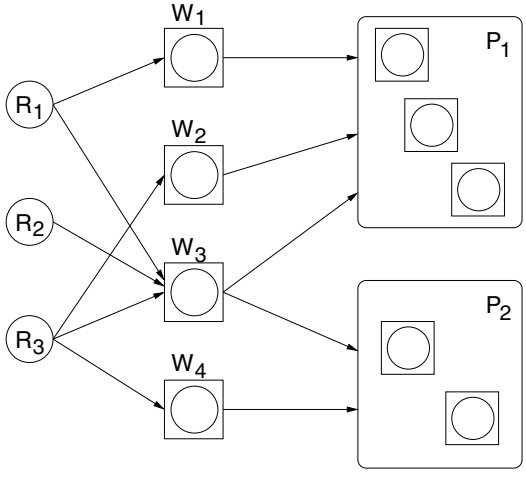

Fig. 2. Web page derivation graph

are forwarded to the database server, whereas database updates do not affect them.

Nonmaterialized (cached) WebViews are cached in ASC, in anticipation of future requests. While they are fresh, they are served very efficiently from the cache. When an update affects a WebView, the cached WebView is invalidated and needs to be regenerated on a subsequent request. This is similar to traditional caching with invalidation rather than a Time-to-Live (TTL) consistency protocol. Assuming that invalidating "dirty" WebViews in ASC is not a costly operation, nonmaterialized WebViews is always a better policy than virtual since, without any loss in data freshness, one obtains significant improvement in response time (for the times when a fresh version of the WebView is in the ASC). Serving from ASC results in two orders of magnitude improvement in response time compared to querying the DBMS.

Materialized WebViews are cached and continuously maintained in the presence of updates. Accesses to them are *always served from the ASC*. The response time is similar to that of a fresh nonmaterialized WebView. We assume that the response time remains almost constant since a materialized WebView is served from ASC even when it is not fresh. However, there is a limit as to how many WebViews should be materialized. Materializing too many WebViews increases the overhead of refreshing them all in the background and can have a negative effect on both server performance and WebView freshness.

The big difference between materialized and nonmaterialized WebViews is the **decoupling** of serving access requests from updating WebViews. With materialization, updates are not in the critical path of serving user requests. Without materialization, updates must be taken care of while serving user requests (i.e., by refreshing a stale WebView before responding). This decoupling helps materialized WebViews avoid increases in response times when there is an increase in update volume (and thus an increase in the percentage of invalidated cached WebViews). In addition to providing data storage, the asynchronous cache module is responsible for automatically selecting which WebViews to materialize.

In this work we consider HTML WebViews only, since we expect them to have the most impact (being more widely deployed than XML WebViews). Dealing only with HTML WebViews means that the cost to generate any WebView from other WebViews will be negligible (simple concatenation of HTML fragments) compared to the cost of generating WebViews from relational data. This would be in contrast, for example, to XML WebViews, which may have complicated transformations and derivations from other (parent) XML WebViews. Such an environment becomes very similar to that of the traditional view materialization problem, where one must consider the view derivation dependencies [24,10,9,20,29,21].

Since the cost to generate WebViews from other WebViews is negligible, in this work we only consider materializing WebViews that are generated directly from relational data (stored in the database server) and do not consider materializing WebViews derived from other WebViews. As was suggested in the literature [18,30], response times for WebViews generated from relational data can be reduced dramatically if they are materialized. Thus, they are the only ones that could offset the overhead of materialization (keeping them up to date in the background). Finally, we assume that WebViews are refreshed by recomputation. This is the general case, which assumes that there is no method for incrementally refreshing the materialized WebViews.

2.3 Measuring performance

We define the performance of data-intensive Web servers by observing the incoming access request stream for a time interval T and measuring the average response times for each user request.

Definition 1. Performance *is measured as the average response time for user requests.*

Specifically, we measure the time between the arrival of the request at the Web server and the departure of the response. We measure response times at the Web server since all our techniques aim at improving the performance of the Web server.

Improving Web server performance might actually not be visible to the end user. Even a tenfold improvement in response time at the server (e.g., from 100 ms to 10 ms) can stay undetected by end users who will receive their responses after a few seconds of network delay. However, a tenfold performance improvement at the Web server clearly improves **scalability**: the same Web server configuration can serve ten times more users or handle sudden tenfold surges in traffic without the cost of additional hardware.

2.4 Measuring quality of data

The "goodness" of the results generated by data-intensive Web servers has been neglected. However, with Web servers being used for increasingly important applications (e.g., stock market information), it is crucial to measure and improve the *quality of data* served to the users. One common characteristic across data-intensive Web servers is their *online nature*: updates to source data are applied concurrently with user accesses since Web servers are always available and never offline. Therefore, the freshness of data served is the most important measure of quality of data.

Definition 2. Quality of data (QoD) *for data-intensive Web servers is the average freshness of the served Web pages.*

When an update to a relation is received, the relation and all data objects derived from it become *stale*. Database objects remain stale until an updated version of them is ready to be served to the user.

We illustrate this with an example. Let us assume the Web page derivation graph of Fig. 2 and that only WebViews W_1 and W_2 are materialized. If an update on relation R_1 arrives at time t_1, then relation R_1 will be stale until time $t_2 \geq t_1$, the time when the update on R_1 is completed (Fig. 3). Although we do not cache relations, relation R_1 will be considered stale because of the unapplied update during $[t_1, t_2]$. On the other hand, materialized WebView W_1 will be stale from time t_1 until time $t_3 \geq t_2$, when its refresh is completed. If an update on relation R_3 arrives at a later time, t_4, then relation R_3 will be stale for the $[t_4, t_5]$ time interval, until t_5, when the update on R_3 is completed (Fig. 3). Also, nonmaterialized WebViews W_3 and W_4 will be stale for the same interval $[t_4, t_5]$. On the

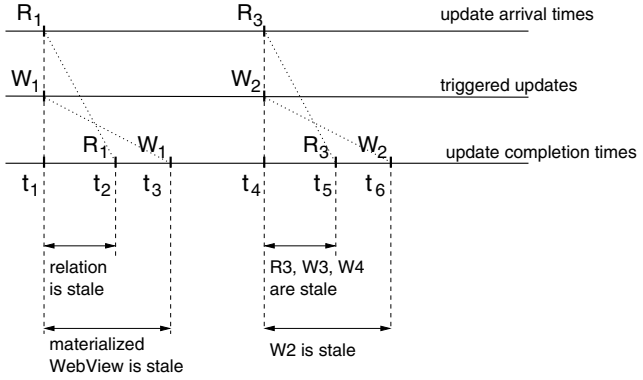

Fig. 3. Staleness example

other hand, materialized WebView W_2 will be stale from time t_4 until time $t_6 \geq t_5$, when its refresh is completed.

We identify four types of data objects that can be stale: relations, nonmaterialized WebViews, materialized WebViews, and Web pages.

- **Relations** are stale when an update for them has arrived but not yet executed.
- **Nonmaterialized WebViews** are stale when an update for a parent relation has arrived but not yet executed.
- **Materialized WebViews** are stale if the WebViews have not been refreshed yet (after an update to a parent relation).
- **Web pages** are stale if a parent WebView is stale.

In order to measure freshness, we observe the access request stream and the update stream for a certain time interval T. We view the access stream during interval T as a sequence of n access requests:

$$\ldots,\ A_x,\ A_{x+1},\ A_{x+2},\ \ldots,\ A_{x+n-1},\ \ldots\ .$$

Access requests A_x are encoded as pairs (P_j, t_x), where t_x is the arrival time of the request for Web page P_j. Each Web page P_j consists of multiple HTML fragments (WebViews).

We define the freshness function for a WebView W_i at time t_k as follows:

$$f(W_i, t_k) = \begin{cases} 1, \text{ if } W_i \text{ is fresh at time } t_k \\ 0, \text{ if } W_i \text{ is stale at time } t_k. \end{cases} \quad (1)$$

A WebView W_i is *stale* if W_i is materialized and has been invalidated, or if W_i is not materialized and there exists a pending update for a parent relation of W_i. A WebView W_i is *fresh* otherwise.

In order to quantify the freshness of individual access requests, we recognize that Web pages are based on multiple WebViews. A simple way to determine freshness is by requiring that all WebViews of a Web page be fresh in order for the Web page to be fresh. Under this scheme, even if one WebView is stale, the entire Web page will be marked as stale. On most occasions, a strict Boolean treatment of Web page freshness like this will be inappropriate. For example, a personalized newspaper page with stock and weather information should not be considered completely stale if all the stock prices are up to date but the temperature reading is a few minutes stale.

Since a strict Boolean treatment of Web page freshness is impractical, we adopt a *proportional definition*. Web page freshness is a rational number between 0 and 1, with 0 being completely stale and 1 being completely fresh. To calculate $f(A_k)$, the freshness value of Web page P_j returned by access request $A_k = (P_j, t_k)$ at time t_k, we take the weighted sum of the freshness values of the WebViews that compose the Web page:

$$f(A_k) = f(P_j, t_k) = \sum_{i=1}^{n_j} a_{i,j} \times f(W_i, t_k)\,, \quad (2)$$

where n_j is the number of WebViews in page P_j and $a_{i,j}$ is a weight factor.

Weight factors $a_{i,j}$ are defined for each (WebView, Web page) combination and are used to quantify the *importance* of different WebViews within the same Web page. Weight factors for the same Web page must sum up to 1, or $\sum_{i=1}^{n_j} a_{i,j} = 1$, for each Web page P_j. When a WebView W_i is not part of Web page P_j, then the corresponding weight factor is zero, or $a_{i,j} = 0$. The user does not have to specify weight factors. By default, these are set to $a_{i,j} = \dfrac{1}{n_j}$, where n_j is the number of WebViews in page P_j (which gives all WebViews equal importance within the same page). However, weight factors can also be user-defined to reflect greater importance of a WebView (fragment) within a page compared to the other WebViews in the same page. Such definitions are always page-dependent.

The overall quality of data for the stream of n access requests will then be

$$QoD = \frac{1}{n} \times \sum_{k=x}^{x+n-1} f(A_k). \quad (3)$$

2.5 Online view selection problem

The choice of WebViews to materialize will have a big impact on performance and data freshness. At the one extreme, materializing all WebViews will give high performance but can have low quality of data (i.e., views will be served very fast but can be stale). On the other hand, keeping all views nonmaterialized will give high quality of data but low performance (i.e., views will be as fresh as possible, but the response time will be high).

We define the *online view selection problem* as follows: in the presence of continuous access and update streams, dynamically select which WebViews to materialize so that overall system performance is maximized, while the freshness of the served data (QoD) is maintained at an acceptable level. In addition to the incoming access/update streams, we assume that we are given a Web page derivation graph (like the one in Fig. 2) and the costs to access/update each relation/WebView.

Given the definition of QoD from Sect. 2.4, an acceptable level of freshness will be a threshold $\theta \in [0, 1]$. For example, a threshold value of 0.9 will mean that roughly 90% of the accesses must be served with fresh data (or that all Web pages served are composed of about 90% fresh WebViews).

The view selection problem is characterized *online* for two reasons. First, since updates are performed *online*, concurrently with accesses, we must consider the freshness of the served data (QoD) in addition to performance. Second, since

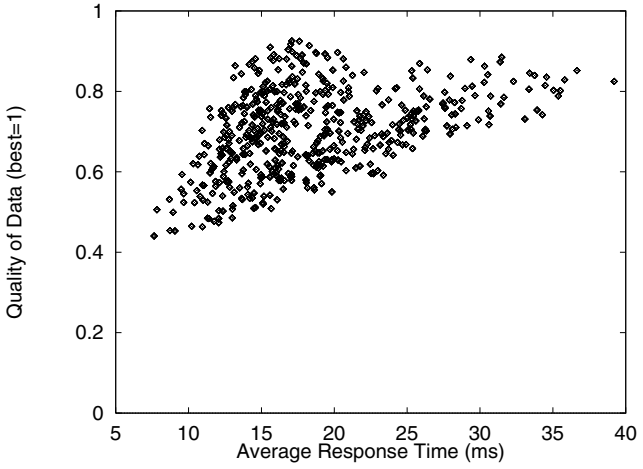

Fig. 4. Performance/QoD of all materialization plans

the accesses and updates are continuously streaming into the system, any algorithm that provides a solution to the view selection problem must decide at runtime and have the ability to adapt under changing workloads. Offline view selection cannot match the wide variations of Web workloads.

We will use the term *materialization plan* to refer to any solution to the online view selection problem. We do not consider the virtual policy for WebViews since caching will always give as fresh data as the virtual policy and will reuse results, giving better performance. In this paper we assume that the Asynchronous Cache module has infinite size and thus there is no need for a cache replacement algorithm (which would distort the comparison).

To visualize the solution space for the online view selection problem, we enumerate all possible materialization plans for a small workload and compute the performance and QoD in Fig. 4. The different materialization plans provide big variations in performance and QoD. For example, plans in the bottom left corner of Fig. 4 correspond to materializing most WebViews (with very low average response time and low QoD), whereas plans in the top of the plot correspond to not materializing most WebViews (with very high QoD and high average response times).

3 The OVIS algorithm

Traditional view selection algorithms work offline and assume knowledge of the entire access and update stream. Such algorithms will not work in an online environment since the selection algorithm must decide the materialization plan in real time. Furthermore, updates in an online environment occur concurrently with accesses, which makes the freshness of the served data an important issue. Finally, the unpredictable nature of Web workloads mandates that the online view selection algorithm be *adaptive* in order to evolve under changing Web access and update patterns.

In this section we describe *OVIS(θ)*, an Online VIew Selection algorithm, where θ is a user-specified QoD threshold. OVIS(θ) strives to maintain the overall QoD above the user-specified threshold θ and also keep the average response time as low as possible. OVIS also monitors the access stream in order to prevent server backlog.

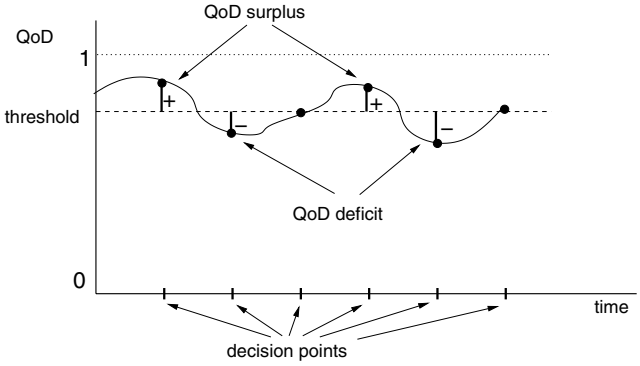

Fig. 5. OVIS(θ) algorithm

OVIS(θ) is inherently adaptive. The algorithm operates in two modes: passive and active. While in *passive mode*, the algorithm collects statistics on the current access stream and receives feedback for the observed QoD. Periodically the algorithm goes into *active mode*, where it will decide if the current materialization plan must change and how.

Figure 5 illustrates the main idea behind the OVIS(θ) algorithm. By constantly monitoring the QoD for the served data the algorithm distinguishes between two cases when it must change the materialization plan. When the observed QoD is higher than the threshold θ, OVIS(θ) identifies a *QoD surplus*, which chooses to "invest" in order to improve the average response time. On the other hand, when the observed QoD is less than the threshold θ, the algorithm identifies a *QoD deficit*, for which it must compensate.

3.1 OVIS(θ) statistics

We want to be able to estimate the change in average response time and overall QoD after adapting the materialization plan. We also want to accurately observe the QoD for the served data in order to determine whether we have a QoD surplus or a QoD deficit. For that purpose we maintain statistics for each WebView and use them to estimate future behavior. Specifically, we estimate:

- The access frequency for each WebView,
- The performance contribution for each WebView in case it will be materialized and in case it will not be materialized,
- The overall data freshness (QoD) contribution of each WebView in case it will be materialized and in case it will not be materialized, and,
- The amount of change in performance and QoD (differentials) if we change the materialization policy for a WebView.

We explain these statistics, along with the estimation methods, in the following paragraphs.

3.1.1 Estimating the access frequency

The most important statistic in our system is the number of accesses each WebView gets. We must consider popularity because the materialization decision for popular WebViews will have a great impact on both the average response time and the overall QoD. We maintain the total number of accesses for a WebView W_i, which we write as $N_{acc}(W_i)$. The $N_{acc}(W_i)$ counter is incremented whenever there is an access request for a Web page that contains W_i.

We use the *recursive prediction error method* [12] to estimate the number of accesses a WebView will have in the near future. According to this method, we use the measurement for the current period, m, and the previous estimate, a, to generate a new estimate a' using the following formula:

$$a' = (1-g)a + gm\,, \tag{4}$$

where g is a gain factor, $0 < g < 1$. Gain was set to 0.25 for all of our experiments (as was suggested by [12]). As illustrated in Fig. 5, the OVIS(θ) algorithm is executed *periodically* in order to adapt the materialization plan. Periods can be defined either by the number of Web page requests received (e.g., adapt every 1000 page requests) or by time intervals (e.g., adapt every 2 min). Before each adaptation we consolidate all statistics and generate estimates for the future. Using Eq. 4, we have

$$N'_{acc} = (1-g)N_{acc} + gN^m_{acc}, \tag{5}$$

where N'_{acc} is the new estimate for the number of accesses, N_{acc} is the old estimate for the number of accesses, and N^m_{acc} is the number of accesses measured for the current interval.

3.1.2 Estimating performance

We estimate the overall cost for implementing a materialization policy and use it to quantitatively compare the changes in performance. High cost will correspond to high average response times and thus low performance.

If a WebView W_i is not materialized, then the overall cost will depend on the *Asynchronous Cache hit ratio*, or how many times we have a cache miss versus a cache hit. Cache misses mandate recomputation of W_i, whereas cache hits will lower the overall cost. We use "$W_i \not\rightsquigarrow$ mat" to denote that W_i will not be materialized. If H_r is the estimate of the hit ratio for WebView W_i, we have:

$$\begin{aligned} cost(W_i \not\rightsquigarrow \text{mat}) = &\underbrace{H_r \times N_{acc} \times A_{hit}}_{\text{cache hits}} \\ &+ \underbrace{(1-H_r) \times N_{acc} \times A_{miss}}_{\text{cache misses}}, \end{aligned} \tag{6}$$

where N_{acc} is the estimate for the number of accesses for W_i, A_{hit} is the access cost for a cache hit on W_i, and A_{miss} is the access cost for a cache miss on W_i. All estimates are computed using Eq. 4. For readability, we do not use the W_i subscripts whenever they can be easily inferred.

The hit ratio, H_r, depends on the materialization policy. If a WebView is materialized, we expect a high hit ratio, because WebViews are refreshed immediately after an update. On the other hand, if a WebView is not materialized, we expect a lower hit ratio (even if eventually the user receives fresh results after cache misses). For that purpose, we maintain separate statistics depending on whether the WebView was materialized or not. When we are trying to estimate the overall cost for a WebView that *will not be* materialized, we use the statistics from when the WebView *was not* materialized. When we are trying to estimate the overall cost for a WebView that *will be* materialized, we use the statistics from when the WebView *was* materialized. The only exception to this is the estimation for the number of accesses and the number of updates that do not depend on the materialization policy.

The hit ratio used in Eq. 6 is based on statistics from when WebView W_i was not materialized. If such statistics are not available (because W_i was always materialized in the past), then we use an optimistic estimate for the hit ratio, $H_r = 100\%$.

If a WebView W_i is materialized, the overall cost will depend, not on the Asynchronous Cache hit ratio (since all accesses are served from the Asynchronous Cache), but on the update rate. Updates lead to immediate refreshes and thus impose a computational "burden" on the system. We use $W_i \rightsquigarrow$ mat to denote that W_i will be materialized. The overall cost in this case will be

$$\begin{aligned} cost(W_i \rightsquigarrow \text{mat}) = &\underbrace{N_{acc} \times A_{hit}}_{\text{accesses}} \\ &+ \underbrace{R_r \times N_{upd} \times U_{mat}}_{\text{refreshes}}, \end{aligned} \tag{7}$$

where R_r is an estimate of what percentage of source updates leads to WebView refreshes for W_i, N_{upd} is the estimate of the number of source updates that affect W_i, and U_{mat} is the cost to refresh WebView W_i. The refresh ratio, R_r, is not always 100% because sometimes refreshes are "batched" together (e.g., when there is an update surge). Finally, all estimates are computed using Eq. 4.

Equation 7 assumes that the cost of refreshing the materialized WebViews in the asynchronous cache will impact the response time of serving access requests. This is true when all three software components (Web server, Asynchronous Cache, DBMS) reside in the same machine, which is a typical configuration for data-intensive Web servers today [17].

3.1.3 Estimating the QoD

Similarly to performance, we use statistics to estimate the overall QoD after adapting the materialization plan. Let us assume that $N_{fresh}(W_i \mid P_j)$ is the number of fresh accesses to WebView W_i that originated from requests to page P_j. The overall QoD definition from Eq. 3 can be rewritten as follows:

$$QoD = \frac{1}{n} \times \sum_i \sum_j \left[a_{i,j} \times N_{fresh}(W_i \mid P_j) \right]$$

for all WebViews W_i and all Web pages P_j, where n is the total number of page access requests and $a_{i,j}$ are the weight factors defined in Sect. 2.4. Weights $a_{i,j}$ sum up to 1.0 for all WebViews in the same Web page.

Instead of separate $N_{fresh}(W_i \mid P_j)$ counters for all (WebView, page) combinations, we maintain only one *weighted* counter, $N_{fresh\text{-}a}(W_i)$, for each WebView W_i. We increment

$N_{fresh\text{-}a}(W_i)$ by the weight value $a_{i,j}$ for each fresh access to W_i originating from a request to page P_j. We have that $N_{fresh\text{-}a}(W_i) = \sum_j \left[a_{i,j} \times N_{fresh}(W_i \mid P_j)\right]$ for all Web pages P_j. Therefore, the QoD definition can be simplified as

$$QoD = \frac{1}{n} \times \sum_i N_{fresh\text{-}a}(W_i). \quad (8)$$

To estimate the contribution of an individual WebView W_i to the overall QoD, we maintain a *freshness ratio*, F_r, defined as $\frac{N_{fresh\text{-}a}}{N_{acc\text{-}a}}$, where $N_{acc\text{-}a}$ is the N_{acc} counter computed similarly to $N_{fresh\text{-}a}$ for each WebView W_i. The difference is that $N_{acc\text{-}a}$ is incremented by $a_{i,j}$ on every access, not just the accesses that produced fresh results, which is the case for $N_{fresh\text{-}a}$. The freshness ratio depends on the materialization policy; therefore we need to maintain separate statistics for when the WebView was materialized and for when it was not materialized, similarly to the hit ratio estimation in the previous section. Given the freshness ratio, F_r, and Eq. 8, the QoD contribution for each WebView W_i will be

$$QoD(W_i) = F_r \times \frac{N_{acc\text{-}a}}{n}, \quad (9)$$

where n is the total number of Web page requests.

3.1.4 Estimating the performance/QoD differentials

At each adaptation step, OVIS(θ) must decide if changing the materialization policy for a particular WebView is warranted or not. In other words, it must determine whether it should stop materializing a materialized WebView or whether it should begin materializing a WebView that had not been previously materialized.

After estimating the performance and QoD for all WebViews using the formulas from the previous paragraphs, we compute the performance and QoD differentials for switching materialization policies. For example, if a WebView W_i is currently materialized, we compute the difference in performance and QoD if W_i were to stop being materialized.

To estimate Δ_{perf}, the **performance differential** for WebView W_i, we use the cost formulas from Eqs. 6 and 7. If W_i is materialized, then we want to estimate how much the performance will change if W_i stops being materialized:

$$\Delta_{perf} = cost(W_i \not\rightsquigarrow \text{mat}) - cost(W_i \rightsquigarrow \text{mat}). \quad (10)$$

Similarly, if W_i is not currently materialized, then we want to estimate how much the performance will change if W_i starts being materialized:

$$\Delta_{perf} = cost(W_i \rightsquigarrow \text{mat}) - cost(W_i \not\rightsquigarrow \text{mat}). \quad (11)$$

A positive performance differential means that the average response time will increase, whereas a negative performance differential means that the average response time will decrease (which is an improvement).

To estimate Δ_{QoD}, the **QoD differential** for WebView W_i, we use the QoD formulas from Eq. 9. If W_i is materialized, then we want to estimate how much the QoD will change if W_i stops being materialized:

$$\Delta_{QoD} = QoD(W_i \not\rightsquigarrow \text{mat}) - QoD(W_i \rightsquigarrow \text{mat}). \quad (12)$$

Similarly, if W_i is not currently materialized, then we want to estimate how much the QoD will change if W_i starts being materialized:

$$\Delta_{QoD} = QoD(W_i \rightsquigarrow \text{mat}) - QoD(W_i \not\rightsquigarrow \text{mat}). \quad (13)$$

A positive QoD differential means that the QoD will increase (which is an improvement), whereas a negative QoD differential means that the QoD will decrease.

3.2 OVIS(θ) algorithm

The OVIS(θ) algorithm constantly monitors the QoD of the served data and periodically adjusts the materialization plan (i.e., which WebViews are materialized and which ones are not materialized). By maintaining the statistics presented in the previous subsection, OVIS(θ) has a very good estimate of how big an effect on the overall performance and QoD the changes in the materialization plan will have. As we outlined at the beginning of this section, OVIS(θ) "invests" QoD surplus or tries to compensate for a QoD deficit (Fig. 5). In the following paragraphs we present the details of the OVIS(θ) algorithm for the case of QoD surplus and QoD deficit. We also explain why we need to impose a constraint on the maximum amount of change to the materialization plan in a single adaptation step, and we describe how to detect server lag. Finally, we provide the pseudocode for the OVIS(θ) algorithm.

3.2.1 QoD surplus

When the observed QoD Q is higher than the user-specified threshold θ, the algorithm will "invest" the surplus QoD $(= Q - \theta)$ in order to decrease the average response time. This is achieved by materializing WebViews that were previously materialized. For the algorithm to take the most profitable decision, we just need to maximize the total performance benefit, $\sum \Delta_{perf}$, for the WebViews that become materialized, while the estimated QoD "losses", $\sum \Delta_{QoD}$, remain less than $Q - \theta$. A greedy strategy, which picks WebViews based on their Δ_{perf} improvement, provides a good solution, as we explain later.

3.2.2 QoD deficit

When the observed QoD Q is less than the threshold θ, the algorithm will have to compensate for the QoD deficit $(= \theta - Q)$. In this case, OVIS(θ) will stop materializing WebViews, thus increasing QoD, at the expense of increasing the average response time. For the algorithm to take the most profitable decision, we just need to maximize the total QoD benefit, $\sum \Delta_{QoD}$, for the WebViews that stop being materialized, while the estimated overall QoD does not increase above the threshold θ. A greedy strategy, which picks WebViews based on their Δ_{QoD} benefit, provides a good solution, as we explain later.

3.2.3 Maximum change constraint

Allowing any number of WebViews to change materialization policy during a single adaptation step of the OVIS(θ) algorithm can have detrimental effects. Since we do not have prior

knowledge of the future, any estimate of future performance and QoD after a materialization policy change is just an estimate and can be wrong. Therefore it is preferable to take smaller adaptation "steps", which should result in a more stable algorithm. For this reason, we impose a limit on the number of WebViews that can change materialization policy during a single adaptation step. We specify this limit as a percentage over the total number of WebViews in the system and denote it as *MAX_CHANGE*. For example, if *MAX_CHANGE* = 5%, and we have 1000 WebViews in our system, then at most 50 of them can change materialization policy at a single adaptation step of the OVIS(θ) algorithm.

3.2.4 Greedy strategy

With the maximum limit in mind, the desired behavior for OVIS(θ) under *QoD surplus* can be summarized as follows:

- Maximize the improvement in performance, and
- Minimize the decrease in QoD

while

- Changing the materialization policy of at most *MAX_CHANGE* WebViews, and
- QoD $> \theta$.

A knapsack-style greedy algorithm (i.e., pick the WebViews with the highest Δ_{perf} per QoD unit) would be preferable if there were no limit to the number of WebViews. However, with the maximum change constraint, a greedy algorithm selecting the top *MAX_CHANGE* WebViews with the highest Δ_{perf} is the best solution.

Let us see why. In the general case, we assume that, because of the *MAX_CHANGE* constraint, we will not be able to reach our goal of QoD $= \theta$ in a single step. However, we would like to change the materialization policy so that we get as close as possible to that goal. For this reason, we want to maximize the overall improvement in a single step. For example, if we have 1000 WebViews, and *MAX_CHANGE* is 10, then we need to identify the ten WebViews that would give us the highest overall improvement in a single step. Clearly, a greedy strategy that selects the ten WebViews with the highest Δ_{perf} is the best solution.

Similarly, the desired behavior for OVIS(θ) under *QoD deficit* can be summarized as follows:

- Maximize the improvement in QoD, and
- Minimize the decrease in performance,

while

- Changing the materialization policy of at most *MAX_CHANGE* WebViews, and
- QoD $\leq \theta$.

With the maximum change constraint, a greedy algorithm selecting the top *MAX_CHANGE* WebViews with the highest Δ_{QoD} is the best solution.

3.2.5 Server lag detection

From elementary queueing theory [13] we know that system performance worsens dramatically as we approach 100% utilization. In practice, there can be cases where the incoming access and update workload generate more load than the server can handle, resulting in backlog, which we refer to as *server lag*. Figure 6 has an example of server lag, which is visible in the response times of the access requests.

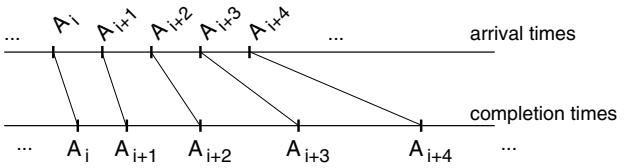

Fig. 6. Server lag example

It is crucial to detect server lag in an online system. For users, server lag means near-infinite response times; this holds for both current (i.e., those still waiting a response) and future users of the system. For system administrators, failure to identify server lag can lead to long, ever-increasing backlogs, which will eventually crash the server.

We detect server lag by monitoring the average response time and the QoD of the served results. Specifically, we compute the *rate of change* between consecutive calls to the OVIS(θ) algorithm. We conclude that server lag is imminent, if:

- The rate of increase for the average response time is too high (for example, a 100-ms increase in average response time over 1000 accesses), or,
- The rate of decrease for the average QoD is too high (for example, a 0.1 drop in QoD over 1000 accesses).

A sudden increase in the average response time is a textbook case for server lag. A sudden decrease in the average QoD indicates that our system, with the current configuration of materialization policies, has surpassed its capacity to handle updates in a timely manner as a result of server lag.

Server lag is used to detect *infeasible QoD thresholds*. For example, a QoD threshold very close to 1 will most likely lead to a server meltdown and should be detected, since no WebView could be materialized, and thus the system will be vulnerable to overloads.

3.2.6 Pseudocode

The OVIS(θ) algorithm is in passive mode most of the time, collecting statistics (Fig. 5). Periodically, OVIS(θ) enters active mode in order to adapt the materialization plan. Before deciding on a new materialization plan, the algorithm will check if there is server lag. If server lag is detected, OVIS(θ) makes all WebViews materialized. This action corresponds to pressing a "panic" button.

Making all WebViews materialized will have the best performance and thus should help alleviate server backlog before it is too late. Materialization essentially "protects" accesses from overload by removing the handling of the updates from the critical path. Assuming "well-behaved" update processes, a surge in updates will lead to reduced QoD without impact on performance.

There are two cases when OVIS(θ) skips an opportunity to adapt the materialization plan:

1. For an initial *warmup* period we forbid adaptation in order to collect enough statistics about the workload;

OVIS(θ) - QoD Surplus

```
0.  qod_diff = QoD − θ > 0
1.  ignore all materialized WebViews
2.  ignore all WebViews with Δ_perf ≥ 0
3.  find W_i with min Δ_perf
4.  if MAX_CHANGE not reached and
5.  (qod_diff + Δ_QoD(W_i)) > 0 then
6.      materialize W_i
7.      qod_diff + = Δ_QoD(W_i)
8.      goto step 3
9.  else
10.     STOP
```

Fig. 7. Pseudocode for OVIS(θ) – QoD surplus

OVIS(θ) - QoD Deficit

```
0.  qod_diff = θ − QoD > 0
1.  ignore all WebViews not materialized
2.  ignore all WebViews with Δ_QoD ≤ 0
3.  find W_i with max Δ_QoD
4.  if MAX_CHANGE not reached then
5.      stop materializing W_i
6.      qod_diff − = Δ_QoD(W_i)
7.      if qod_diff > 0
8.          goto step 3
9.      else
10.         STOP
11. else
12.     STOP
```

Fig. 8. Pseudocode for OVIS(θ) – QoD deficit

2. After detecting server lag, we impose a short mandatory *cool-down* period, during which we do not allow any plan adaptations, in order to let the system reach a stable state again.

Figures 7 and 8 present the active mode of the OVIS(θ) algorithm under surplus and deficit conditions.

3.2.7 Implementing OVIS on a real system

Although we have not yet implemented OVIS as part of a commercial Web server, we believe that doing so would not be very difficult. There are two main issues that need to be addressed: (1) invalidating cached WebViews and (2) collecting statistics about QoD and response times at the server.

In order to recognize which of the WebViews cached inside the Asynchronous Cache have been invalidated, we propose to rely on the existing replication facilities of modern database management systems [16]. In order to collect the required statistics, we propose to instrument the Web server appropriately. We have done so successfully for our WebView Materialization study [18] by instrumenting the page request module of the Apache Web Server to collect response time information.

4 Experiments

In order to study the online view selection problem, we built `osim`, a data-intensive Web server simulator in C++. The database schema, the costs for updating relations, the costs for accessing/refreshing views, the incoming access stream, the incoming update stream, and the level of multitasking are all inputs to the simulator. The simulator processes the incoming access and update streams and generates the stream of responses to the access requests, along with timing information. Among other statistics, the simulator maintains the QoD metric for the served data.

`osim` runs in two modes: *static mode* and *adaptive mode*. In static mode, the materialization plan is prespecified and fixed for the duration of the simulation. In adaptive mode, the materialization plan is modified at regular intervals using the OVIS(θ) algorithm (Figs. 7 and 8). We report the average response time and the observed QoD for each experiment.

We used synthetic workloads in all experiments. The database contained 200 relations, 500 WebViews, and 300 Web pages. Each relation was used to create 3–7 WebViews, whereas each Web page consisted of from 10 to 20 WebViews. Access requests were distributed over Web pages following a Zipf-like distribution [3] and the updates were distributed uniformly among relations. We also generated random Web page derivation graphs (like the one in Fig. 2). Although updates were distributed uniformly among relations, this did not correspond to uniform distribution of updates to WebViews because of the random view derivation hierarchy. Interarrival rates for the access and the update stream approximated a negative exponential distribution. The cost to update a relation was 150 ms, the cost to access a WebView from the Asynchronous Cache was 10 ms and the cost to generate/refresh a WebView was 150 ms in all experiments.

4.1 Providing the full spectrum of QoD

In this set of experiments we vary the QoD threshold θ in order to produce the full spectrum of choices between the (low QoD, high performance) case of full materialization and the (high QoD, low performance) case of no materialization. The workload had 35,000 accesses and 32,000 updates. The duration of the experiment was 2400 s, whereas the QoD threshold θ was set to 0.925.

Figure 9 shows the QoD over time. The top line is the QoD for the no-materialization case (i.e., only caching), and the bottom line is the QoD for the fully materialized case. Both policies correspond to static materialization plans. The middle line is the QoD over time for the OVIS algorithm (our adaptive policy), and the straight line is the QoD threshold, 0.925. Initially, all WebViews under OVIS start as being materialized. However, in this experiment, the QoD for OVIS quickly "climbs" to the threshold levels and stays around the threshold for the duration of the experiment.

Figure 10 shows the fluctuation of average response time. The top line is the average response time for the no-materialization case, and the bottom line is the average response time for the fully materialized case. The middle line

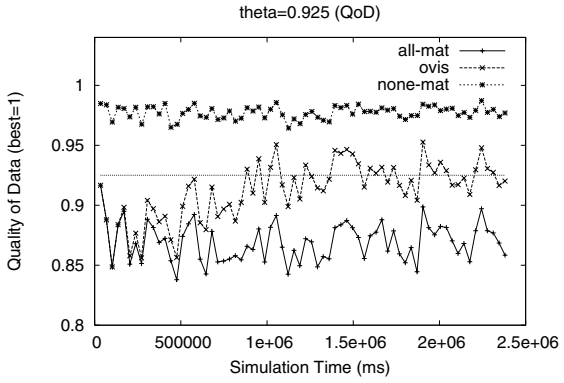

Fig. 9. QoD over time for OVIS(0.925)

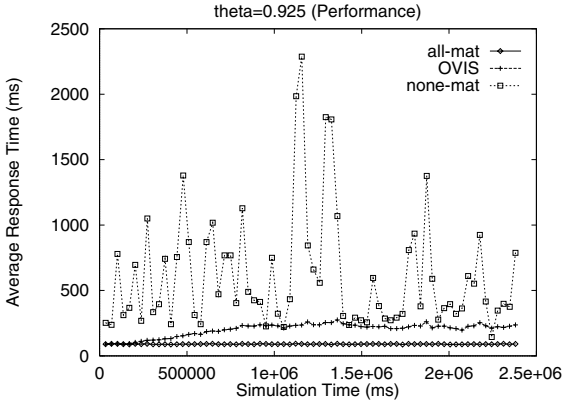

Fig. 10. Performance for OVIS(0.925)

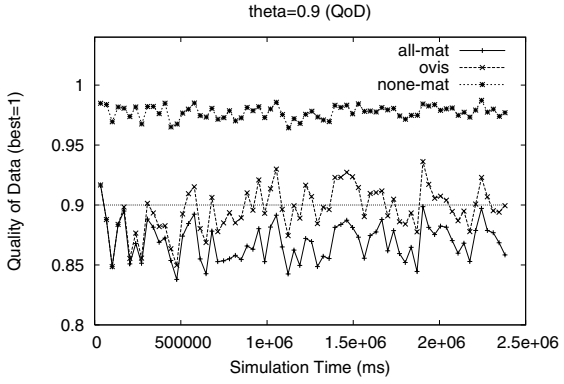

Fig. 11. QoD for OVIS(0.90)

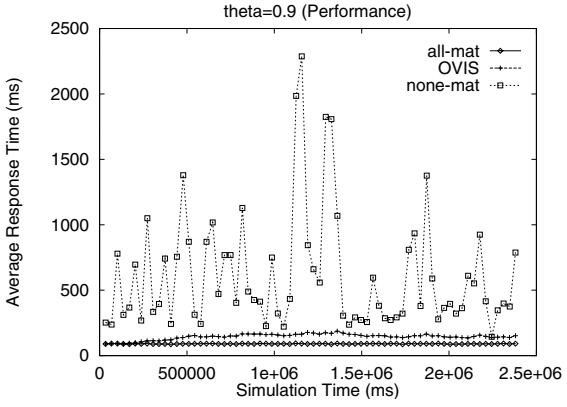

Fig. 12. Performance for OVIS(0.90)

is the average response time for the OVIS algorithm, with $\theta = 0.925$. The high QoD with the no-materialization case is "penalized" by widely fluctuating response times (up to ten times worse than OVIS). On the other hand, the near-constant response times for the fully materialized case correspond to relatively poor QoDClearly, the OVIS algorithm provides a good tradeoff between these two extremes.

We changed the QoD threshold to 0.90 and ran the same experiment. We plot the QoD and the average response times in Figs. 11 and 12. In this experiment, the QoD produced by the OVIS(0.90) algorithm is less than that of the OVIS(0.925) algorithm (from Fig. 9). In other words, OVIS seems to track the specified QoD threshold.

Finally, we ran the same experiment with a threshold value of 0.85 for OVIS. Since the QoD for the fully materialized policy is very close to 0.85, the behavior of OVIS mirrored that of the fully materialized case. This was true for both the data freshness and the average response time.

4.2 Detecting infeasible QoD thresholds

In this set of experiments we wanted to see the behavior of the OVIS algorithm under server lag conditions. The workload had 40,000 accesses and 35,000 updates. The duration of the experiment was 2400 s. Under this workload, without materialization, the server exhibits significant lag, the average response times increase monotonically, and the server essentially crashes under the heavy load.

Figure 13 has the average response time for the three policies: no materialization (`cached`), full materialization (`mat`), and that produced by the adaptive OVIS algorithm (`ovis`). The response times for the fully materialized and OVIS are very close together, at the bottom of the graph. The average response time for the no-materialization policy increases constantly because the server has been saturated. At the end of the experiment, the average response times without materialization are three orders of magnitude worse than the OVIS or fully materialized policies. Clearly, this is a situation we want to avoid, regardless of how good the QoD is under the no-materialization policy.

We plot the QoD for the three different policies in Fig. 14. The top line represents when we do not materialize any WebView, the bottom line when we materialize all WebViews, and the middle line when we use the OVIS algorithm to decide the materialization policy for each WebView. The OVIS algorithm tries to "climb" toward the QoD threshold 0.875 but after a while ($t = 1563$ s) detects the server lag and "resets" to a fully materialized policy, from which it starts to improve the overall QoD again. This behavior is clearer if we look at the average response times of just the fully materialized and the OVIS algorithm, in Fig. 15. The response time under the OVIS algorithm increases slowly, then stabilizes (when the QoD is also stabilized around the QoD threshold), but at some point a sudden increase in response time leads to server lag detection, thus reverting to a fully materialized policy.

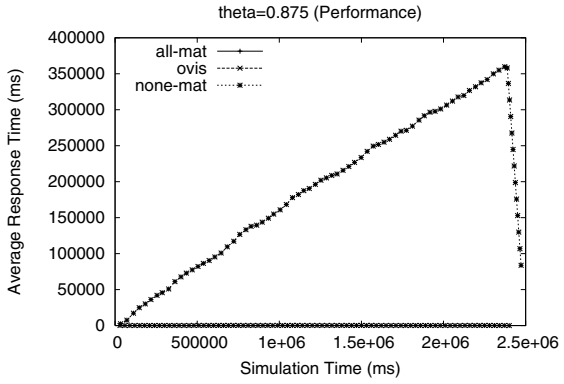

Fig. 13. Performance for OVIS(0.875)

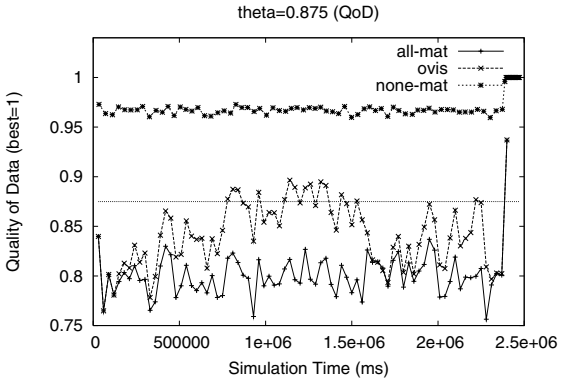

Fig. 14. QoD over time for OVIS(0.875)

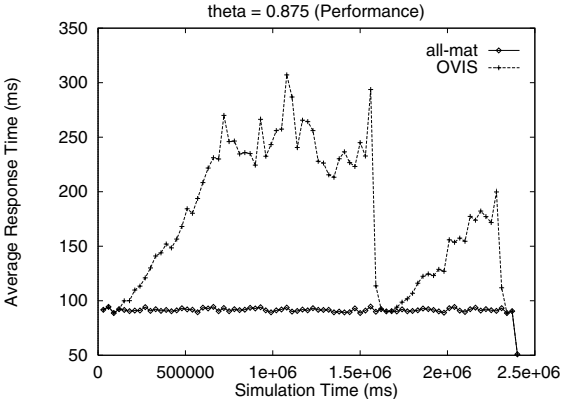

Fig. 15. Performance for OVIS(0.875)

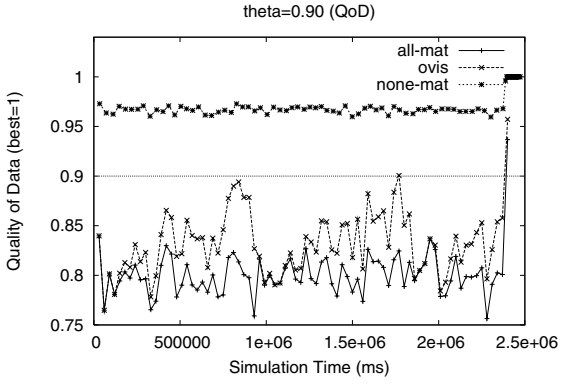

Fig. 17. QoD for OVIS(0.90)

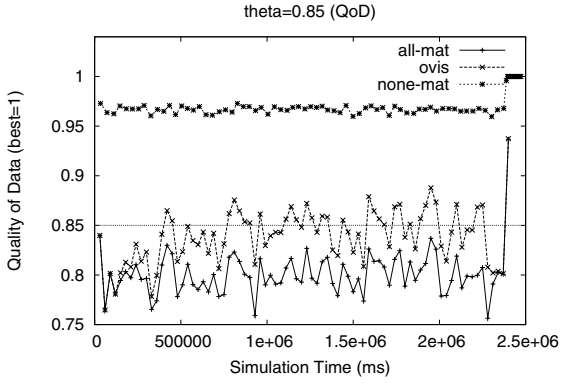

Fig. 16. QoD for OVIS(0.85)

We ran the same experiment with different QoD thresholds, one higher (0.90) and one lower (0.85). In the experiment with the lower θ, the QoD stabilizes around the threshold and we do not detect any server lag (Fig. 16). On the other hand, in the experiment with the higher θ (Fig. 17), the OVIS algorithm detects server lag twice (while trying to reach the high QoD threshold) and resets to a fully materialized policy, at $t = 932\,\mathrm{s}$ and at $t = 1834\,\mathrm{s}$.

4.3 Scaling the number of WebViews

In this set of experiments we evaluated the behavior of the OVIS algorithm with a higher number of WebViews. The workload had 800 relations, 2000 WebViews, and 1500 Web pages. Each Web page consisted of 10 to 20 WebViews, whereas each relation was used to generate 5 to 15 WebViews. A total of 40,000 Web page accesses and 25,000 relation updates occurred in a 2400-s interval.

We plot the QoD and average response time in Figs. 18 and 19. In both plots, the top line respresents the case without any materialization (`none-mat`), the middle curve when we ran OVIS with a threshold of 0.85 (`ovis(0.85)`), and the bottom line the fully materialized case (`all-mat`). In Fig. 19, the y-axis (average response time) is in logarithmic scale in order to distinguish between the OVIS and all-mat curves. Clearly, even at a higher number of WebViews, OVIS continues to provide a hybrid solution between the fully materialized and no-materialization cases. Also, OVIS avoids the server overload that is exhibited by the materialize-nothing approach (top curve). In fact, OVIS is able to track the user-specified QoD threshold of 0.85 very well.

We ran another set of experiments, with the same setup, but with a heavier update workload: 30,000 relation updates instead of 25,000. We plot the QoD and average response time for this set of experiments in Figs. 20 and 21. In both plots, the top line is the case without any materialization (`none-mat`), the middle curve when we ran OVIS with a threshold of 0.85

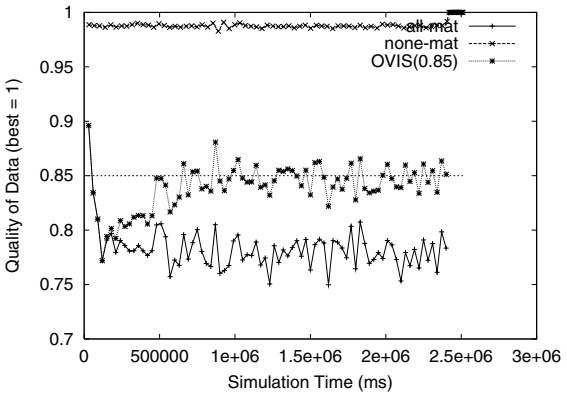

Fig. 18. QoD – 25K updates

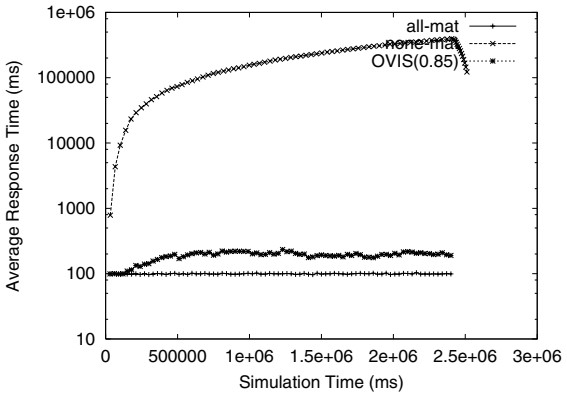

Fig. 19. Performance – 25K updates

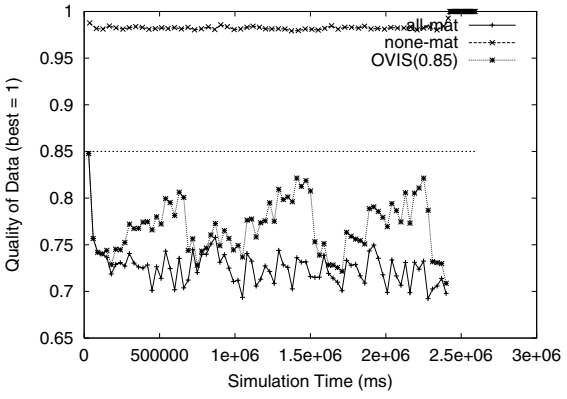

Fig. 20. QoD – 30K updates

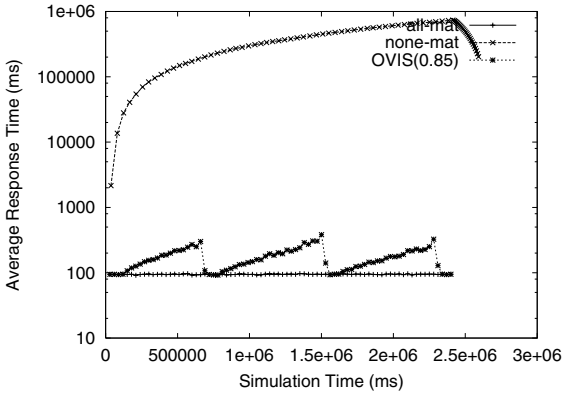

Fig. 21. Performance – 30K updates

(`ovis(0.85)`), and the bottom line the fully materialized case (`all-mat`). The y-axis (average response time) is in logarithmic scale for Fig. 21 in order to distinguish between the OVIS and all-mat curves. In this set of experiments, the user-specified QoD threshold of 0.85 is *infeasible*: although OVIS attempts to reach the threshold, the system detects server lag and reverts to the fully materialized policy, thus avoiding meltdown. The behavior is evident by the zig-zag on the QoD (Fig. 20) and the average response time (Fig. 21). Nevertheless, OVIS(0.85) still provides a hybrid solution, with higher QoD than the fully materialized case, and slightly worse average response time. Without materialization we have server overload, resulting in average response times that are three to four orders of magnitude worse than those of the OVIS algorithm.

5 Alternative QoD metrics

In this section we present an overview of QoD metrics that can be used to measure the freshness of dynamic Web pages. We distinguish three dimensions of such metrics: how individual fragment freshness is computed, how it is aggregated at the page level, and how it is aggregated over multiple accesses. We present the different alternatives for each dimension in the following paragraphs.

5.1 Fragment freshness

Measuring the freshness of an individual WebView (or HTML fragment) is the most fundamental component of any QoD metric. We classify such fragment freshness metrics (FFMs) into two categories:

- **Boolean**, when a true/false answer to the "is this fragment stale?" question is used. In Boolean FFMs, the assigned value is typically 0 if the fragment is stale and 1 if it is fresh. Note that there are no intermediate values.
  We used a Boolean FFM in the presentation of the OVIS(θ) algorithm, although the algorithm will work (with trivial modifications) for any type of FFM, since OVIS(θ) is based on QoD differentials and not on the actual values for QoD.
- **Numeric**, when a numeric value is used to further specify how fresh a certain fragment is. In numeric FFMs, the assigned value can either be bounded (e.g., when the FFM is in the [0, 1] range) or completely unbounded. We further explore the different types of numeric FFMs in the following paragraphs. We adopt the terminology from [22] in our presentation.

5.1.1 Time-based

Time-based numeric FFMs use the time elapsed from the previous update to quantify how stale a certain data item (or fragment in our case) is. Such metrics are especially useful in

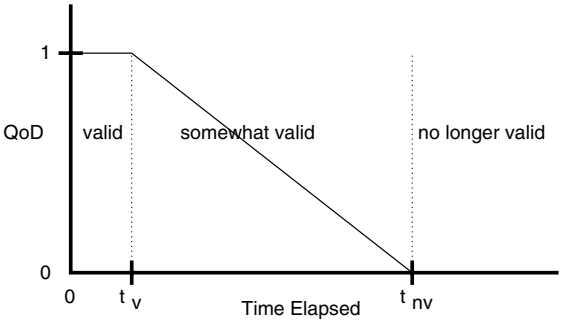

Fig. 22. Mapping of time-based FFM to [0,1] space

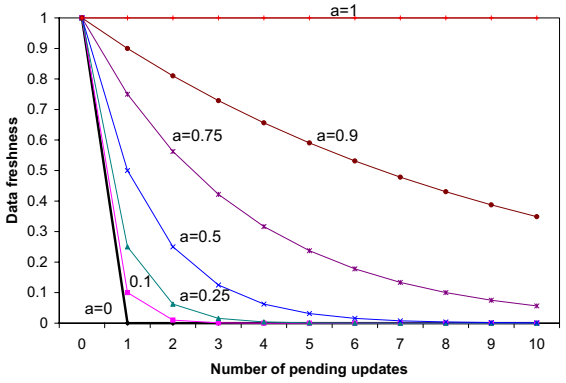

Fig. 23. Freshness decay under various a

distributed environments where the exact time of the next update is not known, but instead time-to-live (TTL) information is used as an approximation [6].

One problem with this approach is that the resulting values are unbounded and also their interpretation would vary according to applications and data domains. For example, a stock quote that is 30 min old would be practically useless, whereas temperature information that is 30 min old is still considered a reasonable approximation of the current temperature.

We propose to alleviate this problem by introducing a *mapping* of the time since last update to a 0-1 range, which can be defined on a per-application basis. This mapping, which is defined similarly to the QoS curves from Aurora [4], is as follows. There is an initial period after the last update, t_v, for which the value is considered valid and the data item is fresh (and thus the FFM has a value of 1). After this period, the "freshness" of the data item declines according to a function (which can be linear or any other monotonically decreasing function). At time t_{nv} the freshness of the data item drops to 0. After that time, the freshness remains 0. Figure 22 illustrates the concept.

The major advantage of this proposal is that it maps the unbounded time-based metric to an intuitive [0,1]-metric, which can be customized for each specific application domain (by modifying the values of t_v and t_{nv}). The only disadvantage is the small overhead in computing the metric.

5.1.2 Lag-based

Lag-based numeric FFMs use the number of unapplied updates to quantify how stale a certain fragment is. Such metrics are especially useful when we have exact information on the upcoming updates (i.e., when invalidations are propagated from the origin servers). Like the time-based case, the currently used lag-based FFMs are unbounded and as such are not very intuitive.

In the past, we proposed measuring the "freshness" of an item as a decreasing function of the number x of updates missed, and specifically:

$$\textit{freshness}(x) = f(x) = a^x \quad x = 0, 1, \ldots . \tag{14}$$

We define a as the *freshness decay rate*, which has a value between 1 and 0. The value $a = 1$ corresponds to the "do not care" case: the user considers the object perfectly fresh, no matter how many updates it is lagging. The value $a = 0$ corresponds to an extremely demanding user who considers as useless everything unless it is perfectly up-to-date (we assume that $0^0 = 1$). Notice that each object i has its own freshness decay rate a_i (for example, one object is a stock price that has to be very fresh, while another is a computer manual, which is still useful even if it is slightly outdated).

Figure 23 illustrates the freshness value under various a values. It is up to the user to set the appropriate value for a and thus determine the behavior of the freshness decay.

The proposed FFM definition has three important properties:

- Being a [0,1] metric, it is much more intuitive to use rather than the unbounded divergence/lag metrics that simply count the number of pending updates [31,22].
- a is a knob that can "record" the varying characteristics of the data items with regards to freshness. It can also be used to record user preferences.
- This model subsumes the typically used binary model (0 if something is stale, 1 if something is fresh), which has been used in many approaches in the past (e.g., [19]). The binary model can be seen as a special case (with $a = 0$).

5.1.3 Divergence-based

Divergence-based numeric FFMs compare the current version with the most up-to-date version and quantify the difference in values (i.e., the divergence). Such metrics are especially useful when data items are simple values (e.g., a stock price) but do not work as well on entire HTML fragments because it is difficult to accurately quantify the difference between two arbitrary HTML/XML fragments.

In the case of simple values, one can either use the absolute value difference as the divergence metric [27] or normalize the difference (by dividing with the current value) to compute a relative percentage. Obviously, the relative percentage approach cannot be used in cases where the value domain includes 0. Finally, it should be noted that in both the absolute and relative cases the resulting FFM will be unbounded, as even the relative percentage can be more than 100% for large differences.

5.2 Aggregating at the page level

Given a fragment freshness metric, FFM, the question arises of how to combine it for fragments that are part of the same Web page. There are two main approaches:

- The **minimum** value of the FFM of the component fragments is used when a "guarantee" is needed on the freshness of all fragments on the page. This is especially useful when used in combination with numeric FFMs. For Boolean metrics, it will lead to an all-or-nothing behavior, which is usually undesirable.
- The **average** value of the FFM of the component fragments is used when a strict guarantee is not needed but instead we want to get an indication of the freshness of the entire page. This can be used in combination with Boolean FFMs to indicate what percentage of fragments should be fresh. This is the approach taken by the OVIS(θ) algorithm.
  In the general case, the average can be *weighted* to indicate the different levels of importance of the fragments composing the Web page.

In a system where nesting of fragments is supported, not all fragments should be considered when computing the freshness at the page level. Assume that we have an arbitrary hierarchy of fragments, expressed as a directed acyclic graph. The nodes with zero out-degree are the Web pages. Clearly, in such a case only the freshness of the leaf fragments should be considered when computing the freshness at the page level. If we have a leaf fragment A, then all other fragments that are derived from it will "inherit" the freshness value of A. This is also true in the case of a fragment composed of multiple lower-level fragments. Therefore, when nesting of fragments is supported, only the innermost fragments must be considered when computing the freshness at the page level.

5.3 Aggregating over multiple access requests

Given an FFM and a way to aggregate it at the page level, the question arises as to how to combine it for multiple Web page accesses. There are two main approaches:

- Using the **minimum** value of page freshness will provide a guarantee on the freshness of the data returned back to the user. This is especially useful when combined with the minimum approach of aggregating FFM at the page level.
- Using the **average** value of page freshness will provide, not a guarantee, but instead what is the freshness for the general case and thus allow fluctuations. This is the approach taken by the OVIS(θ) algorithm.

Both approaches can be combined with either approach for aggregating freshness at the page level, with different semantics for each combination.

6 Extensions to the OVIS algorithm

In this section we outline how the OVIS(θ) algorithm can be extended to handle alternate QoD metrics and QoD requirements from multiple users.

6.1 Supporting alternate QoD metrics

For the presentation of the OVIS(θ) algorithm, we used a Boolean fragment freshness metric, FFM, with aggregation using averages both at the page level and among multiple access requests. Since the basic idea behind our algorithm is based on performance/QoD differentials, as such, the algorithm is not tied to any particular FFM. Any Boolean or numeric FFM could be used, as long as the differentials of Eqs. 12 and 13 can be easily computed.

The current implementation of OVIS(θ) assumes that FFMs are aggregated using averages. If we defined QoD to provide strict guarantees, i.e., using minimum instead of average for aggregating at the page level and among multiple access requests, then the OVIS(θ) algorithm must change as follows. Assume that the required QoD threshold is θ_1. At every adaptation point we must go through all WebViews and compare each one's QoD against θ_1. For those below θ_1, we must act in the same way as in the QoD deficit case of Fig. 8. For those that are above θ_1, we must act in the same way as in the QoD surplus case of Fig. 7. In other words, we can only "invest" our QoD surplus for those WebViews that are strictly above the threshold, and we must also compensate for any WebViews that are below the threshold. The *MAX_CHANGE* constraint will still be applicable, so we must first deal with the QoD deficit cases.

6.2 Handling multiple user requirements

In the current version of the OVIS(θ) algorithm, only a single QoD threshold θ was used. We propose to extend OVIS to handle multiple users as follows. Assume that we have n users, with different QoD threshold requirements $\theta_1, \theta_2, \ldots, \theta_n$. Only two changes are needed to the OVIS algorithm to support multiple users:

- Pick the stringiest of these QoD requirements and use that as the systemwide QoD threshold, i.e., $\theta = \max \theta_i$. This approach is similar to that used for a hierarchy of caches in [25].
- If server lag is detected, then this means that the stringiest QoD threshold is *infeasible*. In this case, after reverting to the all-materialized state, we must reset the systemwide QoD threshold to be the second stringiest QoD threshold among all users. If another server lag is detected, we repeat the process, further reducing the QoD threshold until a feasible threshold is reached.

7 Related work

Our work stands between (1) view selection and runtime buffer management for data warehouses and (2) dynamic Web caching.

View selection has been studied extensively in the context of data warehouses [24,10,9,20,29,21]. However, in all of the current literature, the selection process is offline, requiring complete knowledge of the access and update workloads in advance. This is an unrealistic assumption for Web servers, which are always online.

Research in runtime buffer management for data warehouses is closer to our work because the proposed algorithms are online (i.e., they do not require knowledge of the entire access and update stream).

Sellis [26] deals with caching the results of frequent or expensive queries in secondary storage. A cost model is presented and ranking-based replacement policy introduced. Cached results are updated *on demand*, i.e., the next time they are requested in a query.

WATCHMAN, a data warehouse cache manager, is presented in [28]. Cache admission and cache replacement are integrated using a "profit metric" that considers the access rate, size, and execution cost of the cached results. This work, however, does not deal with warehouse updates.

DynaMat, a data warehouse cache manager that unifies the view selection and view maintenance problems under a single framework, is presented in [14]. Materialized views are stored in a *view pool* and are refreshed during predetermined update windows. Replacement decisions are made either when the pool becomes full (space-bound case) or because there might not be enough time within the update window to refresh some of the materialized results (time-bound case). Updates in DynaMat are *offline*, since queries are not answered during the update window.

Finally, a scheduling framework that takes into account freshness when it decides to replicate OLAP data or route transactions in a database cluster is presented in [23].

Performing updates concurrently with user queries is the main difference between existing work on materialized view selection and our work. The current state of the art in materialized view selection aims to improve overall performance (QoS). However, in the Web server environment, updates are online and thus we need to consider both QoS and quality of data (QoD).

Dynamic Web caching was introduced in [11]. Until recently, research has focused on providing an infrastructure to support caching of dynamically generated Web pages [5,30]. The decision about which pages to cache, when to cache them, and when to invalidate or refresh them is left to the application program or the Web site designer. There is recent work on cache management for dynamic Web content. A *Dynamic Dynamic Content Accelerator* prototype that can cache fragments of dynamically generated Web pages is presented in [7,8]. DBCache, which is an IBM DB2 database capable of caching entire database tables transparently to the application server is presented in [17,1].

The biggest problem of employing caching techniques for dynamic Web content is having the handling of updates in the critical path of serving access requests, which can have detrimental effects on server performance. In contrast, OVIS(θ) will effectively mix caching with view materialization to strike the best balance between performance and data freshness, while avoiding server backlogs.

8 Conclusions

Traditional caching techniques, if used in isolation to accelerate dynamic Web content, face the possibility of server backlogs because the handing of updates is in the critical path of serving access requests. In this paper, we have introduced the online view selection problem: dynamically select which views to materialize in order to maximize performance while keeping data freshness at acceptable levels. We presented OVIS(θ), an adaptive algorithm which combines view materialization with caching and effectively allows for the decoupling of serving of access requests and handling of updates. Parameter θ in OVIS is the level of data freshness that is considered acceptable for the current application. Through extensive experiments we showed that OVIS(θ) can (1) provide the full spectrum of quality of data and (2) detect and prevent server backlogs. We envision OVIS(θ) being used together with current dynamic content accelerators in order to build Web-aware database servers that are self-manageable, robust, and scalable.

Acknowledgements. The authors would like to thank the anonymous reviewers whose insightful comments helped improve the paper and Christos Faloutsos for his invaluable help. This material is based upon work supported by the U.S. Army Research Laboratory and the U.S. Army Research Office under contract/grant number DAAD19-01-1-0494, by NASA under award No NCC8235, and by a startup grant from the School of Arts and Sciences at the University of Pittsburgh.

References

1. Altinel M, Bornhovd C, Krishnamurthy S, Mohan C, Pirahesh H, Reinwald B (2003) Cache tables: paving the way for an adaptive database cache. In: Proceedings of the 29th conference on very large data bases (VLDB), Berlin, Germany, pp 718–729
2. Bornovd C, Altinel M, Krishnamurthy S, Mohan C, Pirahesh H, Reinwald B (2003) DBCache: Middle-tier database caching for highly scalable e-business architectures. In: Proceedings of ACM SIGMOD, San Diego, p 662
3. Breslau L, Cao P, Fan L, Phillips G, Shenker S (1999) Web caching and Zipf-like distributions: evidence and implications. In: Proceedings of INFOCOM 1999, New York, pp 126–134
4. Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Seidman G, Stonebraker M, Tatbul N, Zdonik S (2002) Monitoring streams: a new class of data management applications. In: Proceedings of the 28th international conference on very large data bases (VLDB), Hong Kong, pp 215–226
5. Challenger J, Iyengar A, Witting K, Ferstat C, Reed P (2000) A publishing system for efficiently creating dynamic Web content. In: Proceedings of INFOCOM 2000, Tel Aviv, Israel, pp 844–853
6. Cho J, Garcia-Molina H (2003) Effective page refresh policies for Web crawlers. ACM Trans Database Sys 28(4):390–426
7. Datta A, Dutta K, Thomas HM, VanderMeer DE, Ramamritham K, Fishman D (2001) A comparative study of alternative middle tier caching solutions to support dynamic Web content acceleration. In: Proceedings of the 27th international conference on very large data bases (VLDB), Rome, pp 667–670
8. Datta A, Dutta K, Thomas HM, VanderMeer DE, Suresha, Ramamritham K (2002) Proxy-based acceleration of dynamically generated content on the World Wide Web: an approach and implementation. In: Proceedings of ACM SIGMOD, Madison, WI, pp 97–108

9. Gupta A, Mumick IS (1999) Materialized views: techniques, implementations, and applications, MIT Press, Cambridge, MA
10. Gupta H (1997) Selection of views to materialize in a data warehouse. In: Proceedings of ICDT, Delphi, Greece, pp 98–112
11. Iyengar A, Challenger J (1997) Improving Web server performance by caching dynamic data. In: Proceedings of the USENIX symposium on Internet technologies and systems, Monterey, CA
12. Jacobson V (1988) Congestion avoidance and control. In: Proceedings of ACM SIGCOMM, Stanford, CA, pp 314–329
13. Jain R (1991) The art of computer systems performance analysis. Wiley, New York
14. Kotidis Y, Roussopoulos N (1999) DynaMat: A dynamic view management system for data warehouses. In: Proceedings of ACM SIGMOD, Philadelphia, pp 371–382
15. Larson P, Goldstein J, Zhou J (2003) Transparent mid-tier database caching in SQL Server. In: Proceedings of ACM SIGMOD, San Diego, p 661
16. Larson P, Goldstein J, Zhou J (2004) Transparent mid-tier database caching in SQL Server. In: Proceedings of the 20th international conference on data engineering, Boston, pp 177–189
17. Luo Q, Krishnamurthy S, Mohan C, Pirahesh H, Woo H, Lindsay BG, Naughton JF (2002) Middle-tier database caching for e-business. In: Proceedings of ACM SIGMOD, Madison, WI, p 662
18. Labrinidis A, Roussopoulos N (2000) WebView materialization. In: Proceedings of ACM SIGMOD, Dallas, TX, pp 367–378
19. Labrinidis A, Roussopoulos N (2001) Update propagation strategies for improving the quality of data on the Web. In: Proceedings of the 27th international conference on very large data bases (VLDB), Rome, pp 391–400
20. Ligoudistianos S, Sellis TK, Theodoratos D, Vassiliou Y (1999) Heuristic algorithms for designing a data warehouse with SPJ views. In: Proceedings of the 1st international conference on data warehousing and knowledge discovery (DaWaK), Florence, Italy, pp 96–105
21. Mistry H, Roy P, Sudarshan S, Ramamritham K (2001) Materialized view selection and maintenance using multi-query optimization. In: Proceedings of ACM SIGMOD, Santa Barbara, CA, pp 307–318
22. Olston C, Widom J (2002) Best-effort cache synchronization with source cooperation. In: Proceedings of ACM SIGMOD, Madison, WI, pp 73–84
23. Rohm U, Bohm K, Schek H-J, Schuldt H (2002) FAS – A freshness-sensitive coordination middleware for a cluster of OLAP components. In: Proceedings of the 28th international conference on very large data bases (VLDB), Hong Kong
24. Roussopoulos N (1982) View indexing in relational databases. ACM Trans Database Sys 7(2):258–290
25. Shah S, Dharmarajan S, Ramamritham K (2003) An efficient and resilient approach to filtering and disseminating streaming data. In: Proceedings of the 29th international conference on very large data bases (VLDB), Berlin, Germany, pp 57–68
26. Sellis T (1988) Intelligent caching and indexing techniques for relational database systems, Inf Sys 13(2):175–185
27. Shah S, Ramamritham K, Shenoy PJ (2002) Maintaining coherency of dynamic data in cooperating repositories. In: Proceedings of the 28th international conference on very large data bases (VLDB), Hong Kong, pp 526–537
28. Scheuermann P, Shim J, Vingralek R (1996) WATCHMAN: A data warehouse intelligent cache manager. In: Proceedings of the international conference on very large data bases, Bombay, India, pp 51–62
29. Theodoratos D, Ligoudistianos S, Sellis TK (2001) View selection for designing the global data warehouse. Data Knowl Eng 39(3):219–240
30. Yagoub K, Florescu D, Issarny V, Valduriez P (2000) Caching strategies for data-intensive Web sites. In: Proceedings of the 26th international conference on very large data bases (VLDB), Cairo, Egypt, pp 188–199
31. Yu H, Vahdat A (2000) Design and evaluation of a continuous consistency model for replicated services. In: Proceedings of the 4th symposium on operating systems design and implementation, pp 305–318