# Balancing Performance and Data Freshness in Web Database Servers

Alexandros Labrinidis
Department of Computer Science
University of Pittsburgh
labrinid@cs.pitt.edu

Nick Roussopoulos
Department of Computer Science
University of Maryland
nick@cs.umd.edu

## 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, and present experimental results.

## 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 [8, 3, 4, 5, 13] 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[15, 7, 6, 14]. However, unlike data warehouses, which are off-line during updates, most web servers maintain their back-end 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($\theta$), an adaptive algorithm for the Online View Selection problem. OVIS($\theta$) 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 re-used as necessary. Parameter $\theta$ 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($\theta$) 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 realtime stock information to subscribers. Updates to stock prices and other market derivatives are streamed



**Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003**

to the back-end 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 Section 3 we describe the proposed Online View Selection Algorithm and in Section 4 we discuss the results of our experiments. Section 5 summarizes related work. We conclude in Section 6.

## 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 (Figure 1). In this architecture, the

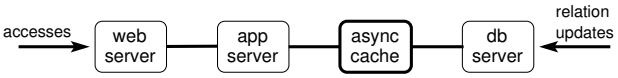


Figure 1: System Architecture

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 is 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 [1, 12].

### 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.
- **WebViews**, introduced in [11], are HTML or XML fragments. 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. 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.
- **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 which is generated by querying $R_i$. A web page $P_k$ is considered to be derived from WebView $W_j$ if $P_k$ contains $W_j$.

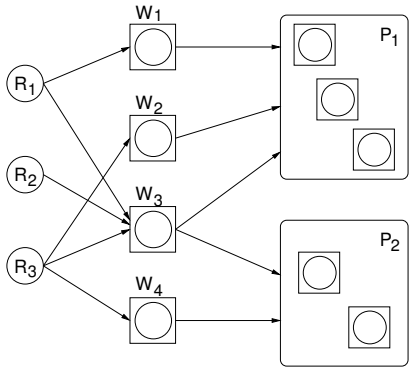


Figure 2: Web Page Derivation Graph

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 bigger 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 [3]. 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, which 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 are forwarded to the database server, whereas database updates do not affect them.

**Non-Materialized** (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 re-generated on a following 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, non-Materialized 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 a fresh non-materialized 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 non-materialized 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). 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. Dealing only with HTML WebViews means that the cost to generate any WebView from other WebViews will be negligible (simple concatenation of HTML fragments). Therefore, in this work we only consider materializing WebViews which are generated directly from relational data (stored in the database server). As was suggested in the literature [11, 18], response times for such WebViews 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.

### 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 ten-fold improvement in response time at the server (e.g., from 100 msec to 10msec) can stay undetected by end users who will receive their responses after a few seconds of network delay. However, a ten-fold performance improvement at the web server clearly improves **scalability**: the same web server configuration can serve ten times more users or handle sudden ten-fold 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 in the past. 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 off-line. 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 that are 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 Figure 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 (Figure 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 (Figure 3). Also, non-materialized WebViews $W_3$ and $W_4$ will be stale for the same interval $[t_4, t_5]$. On the 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.

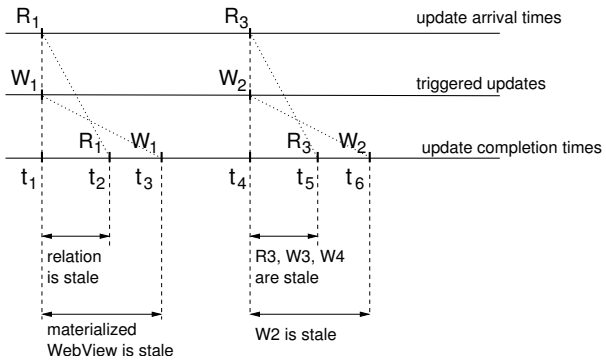


Figure 3: Staleness Example

We identify four types of data objects that can be stale: relations, non-materialized WebViews, materialized WebViews, and web pages.

- **Relations** are stale when an update for them has arrived, but not yet executed.
- **Non-materialized 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. In most occasions, a strict Boolean treatment of web page freshness like this will be inappropriate. For example, a personalized newspaper page with stock information 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 $a_{i,j} = 0$. By default, weight factors are set to $a_{i,j} = \frac{1}{n_j}$, where $n_j$ is the number of WebViews in page $P_j$ (which gives all WebViews equal importance within the same page). Weight factors can also be user-defined.

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. On 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 non-materialized 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 Figure 2), and the costs to access/update each relation/WebView.

Given the definition of QoD from Section 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 the accesses and updates are continuously streaming into the system, any algorithm that provides a solution to the view selection problem must decide at run-time and have the ability to adapt under changing workloads. Off-line 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).

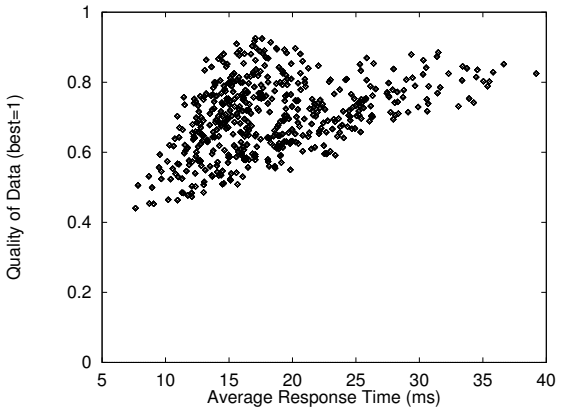


Figure 4: Perf/QoD of all Materialization Plans

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 Figure 4. The different materialization plans provide big variations in performance and Quality of Data. For example, plans in the bottom left corner of Figure 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 off-line 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($\theta$)*, an Online VIew Selection algorithm, where $\theta$ is a user-specified QoD threshold. OVIS($\theta$) strives to maintain the overall QoD above the user-specified threshold $\theta$ and also keep the average response time as low as possible. OVIS also monitors the access stream in order to prevent server backlog.

OVIS($\theta$) 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.

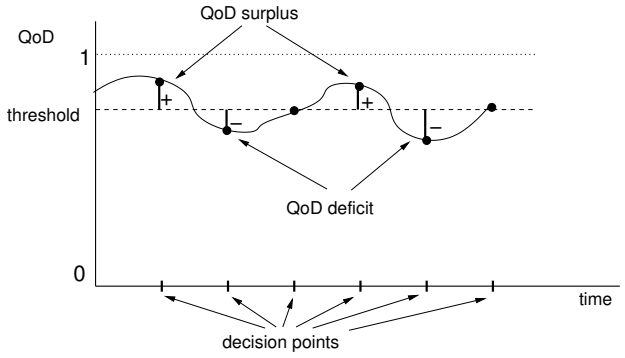


Figure 5: OVIS($\theta$) Algorithm

Figure 5 illustrates the main idea behind the OVIS($\theta$) 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 $\theta$, OVIS($\theta$) 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 $\theta$, the algorithm identifies a *QoD deficit*, for which it must compensate.

## 3.1 OVIS($\theta$) 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 next paragraphs.

### 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 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* [9] 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 [9]). As illustrated in Figure 5, the OVIS($\theta$) 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 minutes). Before each adaptation, we consolidate all statistics and generate estimates for the future. Using Equation 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.

### 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:

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

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 Equation 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 which do not depend on the materialization policy.

The hit ratio used in Equation 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 not depend on the Asynchronous Cache hit ratio (since all accesses are served from the Asynchronous Cache), but it will depend 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:

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

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 Equation 4.

Eq. 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 [13].

### Estimating the QoD

Similarly to performance, we use statistics to estimate the overall Quality of Data after adapting the materialization plan. Let us assume that $N_{fresh}(W_i \mid P_j)$ is the number of fresh accesses to WebView $W_i$ which originated from requests to page $P_j$. The overall QoD definition from Equation 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 Section 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:

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

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 a 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 Equation 8, the QoD contribution for each WebView $W_i$ will be:

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

where $n$ is the total number of web page requests.

### Estimating the differentials

At each adaptation step, OVIS($\theta$) 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 was not 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 $\Delta_{perf}$, the **performance differential** for WebView $W_i$, we use the cost formulas from Eq. 6 and Eq. 7. If $W_i$ is materialized, then we want to estimate how much the performance will change if $W_i$ stops being materialized:

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

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:

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

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 $\Delta_{QoD}$, 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:

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

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:

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

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($\theta$) Algorithm

The OVIS($\theta$) 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($\theta$) 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($\theta$) "invests" QoD surplus or tries to compensate for QoD deficit (Figure 5). In the following paragraphs we present the details of the OVIS($\theta$) algorithm.

### QoD Surplus

When the observed QoD $Q$ is higher than the user-specified threshold $\theta$, the algorithm will "invest" the surplus QoD ($= Q-\theta$) in order to decrease the average response time. This is achieved by materializing WebViews which were not 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, that picks WebViews based on their $\Delta_{perf}$ improvement provides a good solution, as we explain later.

### QoD Deficit

When the observed QoD $Q$ is less than the threshold $\theta$, the algorithm will have to compensate for the QoD deficit ($= \theta - Q$). In this case, OVIS($\theta$) 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 $\theta$. A greedy strategy, that picks WebViews based on their $\Delta_{QoD}$ benefit provides a good solution, as we explain later.

### Maximum Change Constraint

Allowing any number of WebViews to change materialization policy during a single adaptation step of the OVIS($\theta$) 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($\theta$) algorithm.

### Greedy Strategy

With the maximum limit in mind, the desired behavior for OVIS($\theta$) under *QoD surplus* can be summarized as follows: maximize improvement in performance and minimize decrease in QoD, while QoD $> \theta$, while changing the materialization policy of at most *MAX_CHANGE* WebViews. A knapsack-style greedy algorithm (i.e. $\Delta_{perf}$ per QoD unit) would be preferable if there was 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 $\Delta_{perf}$ is the best solution.

Similarly, the desired behavior for OVIS($\theta$) under *QoD deficit* can be summarized as follows: maximize improvement in QoD and minimize decrease in performance, while QoD $\leq \theta$, and while changing the materialization policy of at most *MAX_CHANGE* WebViews. With the maximum change constraint, a greedy algorithm selecting the top *MAX_CHANGE* WebViews with the highest $\Delta_{QoD}$ is the best solution.

### Server Lag Detection

From elementary queueing theory 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 what the server can handle, resulting in backlog, which we refer to as *server lag.*

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($\theta$) algorithm. We conclude that server lag is imminent, if: 1) the rate of increase for the average response time is too high (for example, a 100 msec increase in average response time over 1000 accesses), or, 2) 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.

### Pseudo-code

The OVIS($\theta$) algorithm is in Passive Mode most of the time, collecting statistics (Figure 5). Periodically, OVIS($\theta$) enters Active Mode in order to adapt the

**OVIS($\theta$) - 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
5.     and (qod_diff + Δ QoD(W_i)) > 0
6.        materialize W_i
7.        qod_diff + = Δ QoD(W_i)
8.        goto step 3
9.  else
10.       STOP
```

Figure 6: Pseudo-code for OVIS($\theta$) - QoD Surplus

**OVIS($\theta$) - 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
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
```

Figure 7: Pseudo-code for OVIS($\theta$) - QoD Deficit

materialization plan. Before deciding on a new materialization plan, the algorithm will check if there is server lag. If server lag is detected, OVIS($\theta$) 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($\theta$) skips an opportunity to adapt the materialization plan: for an initial *warm-up* period we forbid adaptation in order to collect enough statistics about the workload; 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 6 and 7 present the active mode of the OVIS($\theta$) algorithm under QoD Surplus and QoD Deficit conditions.

## 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 is fixed for the duration of the simulation. In adaptive mode, the materialization plan is modified at regular intervals using the OVIS($\theta$) algorithm (Figures 6 and 7). 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 from 10 to 20 WebViews. Access requests were distributed over web pages following a Zipf-like distribution [2] and the updates were distributed uniformly among relations. We also generated random Web Page Derivation Graphs (like the one in Figure 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 $\theta$ 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 seconds whereas the QoD threshold $\theta$ was set to 0.925.

Figure 8 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.

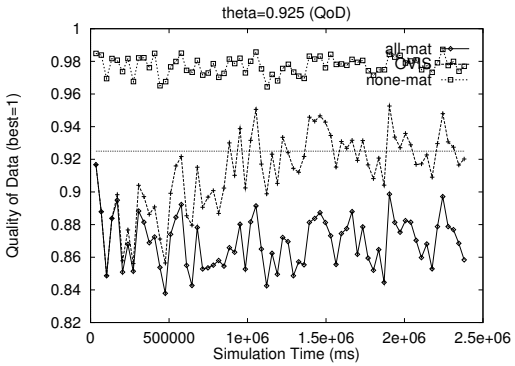


Figure 8: QoD over time for OVIS(0.925)

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 9 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 is the average response time for the OVIS algorithm, with $\theta = 0.925$. The high QoD with the no materialized 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 QoD. Clearly, the OVIS algorithm provides a good trade-off between these two extremes.

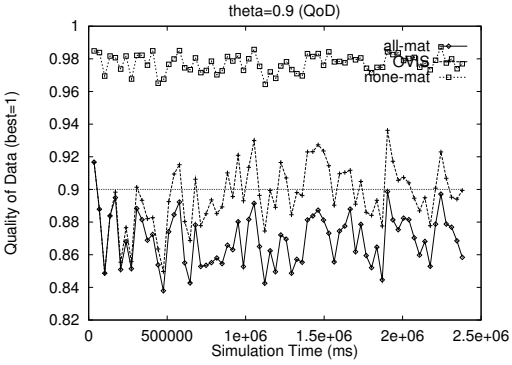


Figure 10: QoD for OVIS(0.90)

We changed the QoD threshold to 0.90 and ran the same experiment. We plot the QoD and the average response times in Figures 10 and 11. In this experiment, the QoD produced by the OVIS(0.90) algorithm is less than that of the OVIS(0.925) algorithm (from Figure 8). 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.

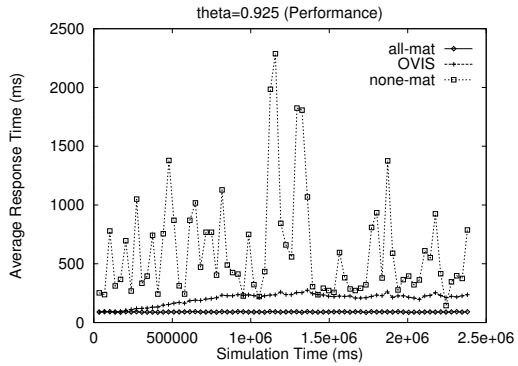


Figure 9: Performance for OVIS(0.925)

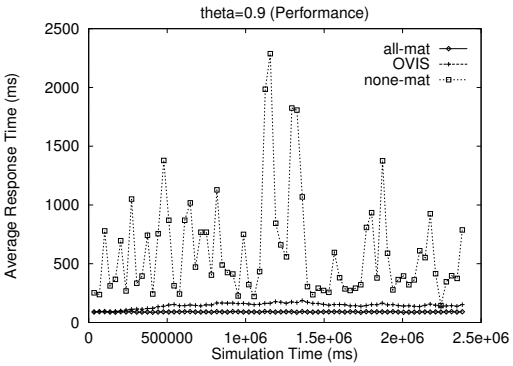


Figure 11: Performance for OVIS(0.90)

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 seconds. 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 12 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 that 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 Figure 13. The top line is when we do not materialize any WebView, the bottom one is when we material-

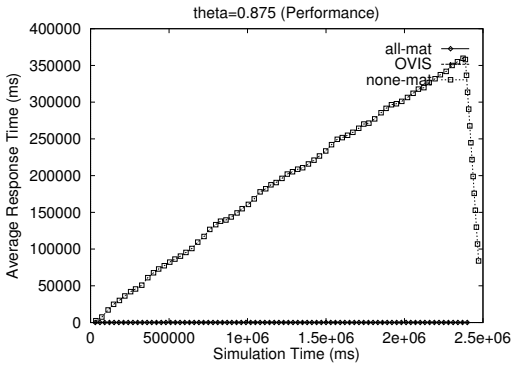


Figure 12: Performance for OVIS(0.875)

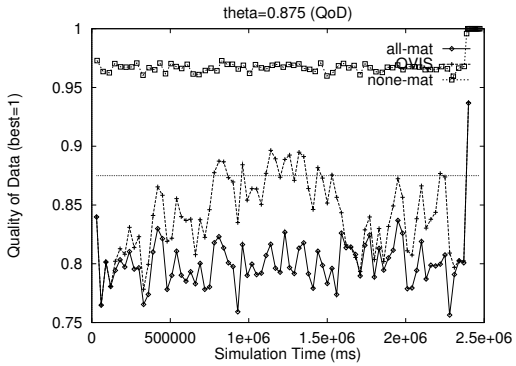


Figure 13: QoD over time for OVIS(0.875)

ize all WebViews and the middle line is when we use the OVIS algorithm to decide the materialization policy for each WebView. The OVIS algorithm tries to "climb" towards the QoD threshold 0.875, but, after a while (t=1563 sec), detects the server lag and "resets" to a fully-materialized policy, from which it starts to improve the overall QoD again. This behavior is more clear if we look at the average response times of just the fully materialized and the OVIS algorithm, in Figure 14. The response time under the OVIS algorithm increases slowly, then it 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 and thus reverting to a fully materialized policy.

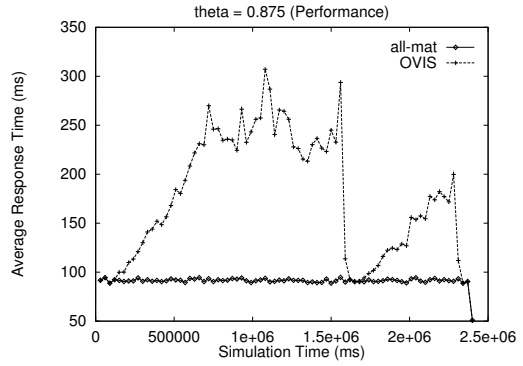


Figure 14: Performance for OVIS(0.875)

We ran the same experiment with different QoD thresholds, one higher (0.90) and one lower (0.85). In the experiment with the lower $\theta$, the QoD stabilizes around the threshold and we do not detect any server lag. On the other hand, in the experiment with the higher $\theta$, the OVIS algorithm detects server lag twice (while trying to reach the high QoD threshold) and resets to a fully materialized policy both times.

### 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-second interval, which was the duration of the experiment.

We plot the QoD and average response time in Figure 15 and Figure 16. In both plots, the top line is the case without any materialization (`none-mat`), the middle curve is when we ran OVIS with a threshold of 0.85 (`ovis(0.85)`), and the bottom line is the fully materialized case (`all-mat`). In Figure 16, 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.

## 5 Related Work

Our work stands between: 1) view selection and run-time buffer management for data warehouses, and 2) dynamic web caching.

View selection has been studied extensively in the context of data warehouses [15, 7, 6, 14]. However, in all of the current literature, the selection process is off-line, requiring complete knowledge of the access and update workloads in advance. This is an unrealistic assumption for web servers, which are always online.

Run-time buffer management for data warehouses is an online process [17, 16, 10]. However, updates in this environment are not online: the data warehouse is taken off-line during the maintenance window. In the context of the Web, we have to perform updates concurrently with user requests.

Dynamic web caching was introduced in [8]. Until recently, research has focused on providing an infrastructure to support caching of dynamically gener-

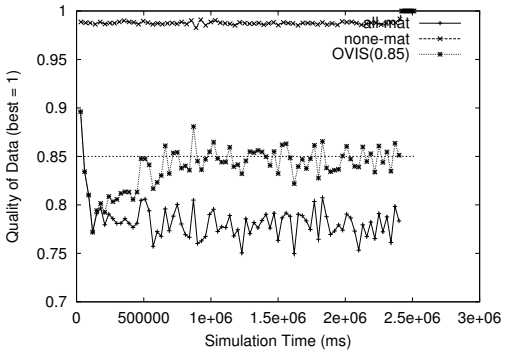


Figure 15: QoD - 25K updates

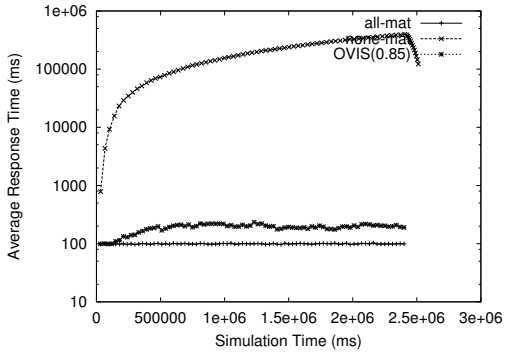


Figure 16: Performance - 25K updates

ated web pages [3, 18]. The decision of 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. [4, 5] presents a *Dynamic Content Accelerator* prototype that can cache fragments of dynamically generated web pages. [13] presents DBCache, which is an IBM DB2 database which can cache entire database tables transparently to the application server.

## 6 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($\theta$), 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 $\theta$ in OVIS is the level of data freshness that is considered acceptable for the current application. Through extensive experiments we showed that OVIS($\theta$) can: 1) provide the full spectrum of quality of data, and 2) detect and prevent server backlogs. We envision OVIS($\theta$) being used together with current dynamic content accelerators in order to build web-aware database servers that are self-manageable, robust, and scalable.

## References


[1] C. Bornovd et al. "DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures". In *Proc. of ACM SIGMOD*, 2003.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. "Web Caching and Zipf-like Distributions: Evidence and Implications". In *Proc. of INFOCOM*, 1999.

[3] J. Challenger et al. "A Publishing System for Efficiently Creating Dynamic Web Content". In *Proc. of INFOCOM*, 2000.

[4] A. Datta et al. "A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration". In *Proc. of VLDB*, 2001.

[5] A. Datta et al. "Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation". In *Proc. of ACM SIGMOD*, 2002.

[6] A. Gupta and I. S. Mumick, editors. *"Materialized Views: Techniques, Implementations, and Applications"*. MIT Press, June 1999.

[7] H. Gupta. "Selection of Views to Materialize in a Data Warehouse". In *Proc. of ICDT*, 1997.

[8] A. Iyengar and J. Challenger. "Improving Web Server Performance by Caching Dynamic Data". In *Proc. of USITS*, 1997.

[9] V. Jacobson. "Congestion Avoidance and Control". In *Proc. of ACM SIGCOMM*, 1988.

[10] Y. Kotidis and N. Roussopoulos. "DynaMat: A Dynamic View Management System for Data Warehouses". In *Proc. of ACM SIGMOD*, 1999.

[11] A. Labrinidis and N. Roussopoulos. "WebView Materialization". In *Proc. of ACM SIGMOD*, 2000.

[12] P. Larson, J. Goldstein, and J. Zhou. "Transparent Mid-Tier Database Caching in SQL Server". In *Proc. of ACM SIGMOD*, 2003.

[13] Q. Luo et al. "Middle-tier Database Caching for e-Business". In *Proc. of ACM SIGMOD*, 2002.

[14] H. Mistry et al. "Materialized View Selection and Maintenance Using Multi-Query Optimization". In *Proc. of ACM SIGMOD*, 2001.

[15] N. Roussopoulos. "View Indexing in Relational Databases". *ACM TODS*, 7(2):258–290, June 1982.

[16] P. Scheuermann, J. Shim, and R. Vingralek. "WATCHMAN : A Data Warehouse Intelligent Cache Manager". In *Proc. of VLDB Conference*, 1996.

[17] T. Sellis. "Intelligent caching and indexing techniques for relational database systems". *Information Systems*, 13(2), 1988.

[18] K. Yagoub et al. "Caching Strategies for Data-Intensive Web Sites". In *Proc. of VLDB Conf.*, 2000.