CS-TR-4343

Online View Selection for the Web

Alexandros Labrinidis
Department of Computer Science & ISR
University of Maryland, College Park
`labrinid@cs.umd.edu`

Nick Roussopoulos*
Department of Computer Science & ISR
University of Maryland, College Park
`nick@cs.umd.edu`

March 9, 2002

Abstract

View materialization has been shown to ameliorate the scalability problem of data-intensive web servers. 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. In such environments, the selection of views to materialize must be performed online; both performance and data freshness should be considered. In this paper, we discuss the *Online View Selection problem*: select which views to materialize in order to maximize performance while maintaining freshness at acceptable levels. We define Quality of Service and Quality of Data metrics and present OVIS(θ), an adaptive algorithm for the Online View Selection problem. OVIS(θ) evolves the materialization decisions to match the constantly changing access/update patterns on the Web. The algorithm is also able to identify *infeasible* freshness levels, effectively avoiding saturation at the server. We performed extensive experiments under various workloads, which showed that our online algorithm comes close to the optimal off-line selection algorithm.

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 advertising features and personalization[BBC+98]. However, dynamic content has significantly higher resource demands than static web pages and creates a huge scalability problem at web servers.

View materialization has been proposed as the solution to the scalability problem for dynamic content [LR99, LR00, YFIV00]. With view materialization, dynamic query results are cached outside the DBMS and re-used for answering future requests. Updates on base data are performed immediately in the DBMS and trigger a refresh on the materialized views. Although refreshes are applied immediately, the method

*Also with the Institute for Advanced Computer Studies does not provide any hard guarantees for the freshness of the responses sent to the users. For example, if all views were materialized, the update workload could crash the server and create a backlog, resulting in stale responses. Fast query response is of paramount importance only if the data is fresh, otherwise it may be more harmful than slow or even no data service. This is especially true for data-intensive web servers being used for critical applications, where serving stale data can have catastrophic consequences.

The view selection problem aims at balancing the trade-off between performance improvement and maintenance overhead because of materialization. View selection has been studied extensively in relational databases and data warehouses [Rou82, Han87, Sel88, BM90, GM95, RCK$^+$95, SSV96, Gup97, CD97, KR99]. In data warehouses, View Selection is performed off-line during the down time of the warehouse. Web servers, on the other hand, must remain online all the time and thus, updates are applied in the back-end database while the web server continues to serve user requests. Therefore, in a web server environment, view selection needs a) a run-time method for deciding which views to materialize, and, b) a cost model that takes into account both system performance and data freshness guarantees observed under this view selection.

In this paper we introduce the *Online View Selection problem*: in the presence of continuous access and update streams, dynamically select which views to materialize, so that overall system performance is maximized, while a guarantee on the freshness of the served data is maintained. We define Quality of Service (QoS) metrics to measure system performance and Quality of Data (QoD) metrics to measure the freshness of the served data.

An algorithm that solves the Online View Selection problem must

- be *adaptive*: rapidly evolve the selection decisions based on changes in the access/update patterns,
- recognize *infeasible QoD guarantees*, when the user-specified QoD guarantee cannot be met under the current workload without additional resources,
- be *scalable*: handle large datasets and heavy access/update volumes.

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 to the back-end database and must be performed online. The web server is required to provide users with up-to-date information on specific stocks. This information 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. Several recent studies on the access workloads of data-intensive web servers [BCF$^+$99, PQ00, Mar01, LR01b] indicate that accesses are highly skewed, with a handful of pages corresponding to a big percentage of the overall access volume.

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 has a brief summary of related work. We conclude in Section 6.

2 Online View Selection Problem

We assume a system architecture like the one in Figure 1. The web server is responsible for serving user requests. Depending on the complexity of the web site, we may have an application server responsible for web workflow management. Instead of interfacing the application server directly to the database server, an *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. This allows for significantly faster response times for materialized data, but incurs overhead for refreshes. Finally, the *update scheduler* intercepts all incoming updates and is responsible for invalidating cached content in the asynchronous cache, propagating the relation updates to the database server, and triggering the refreshment of materialized data in the asynchronous cache.

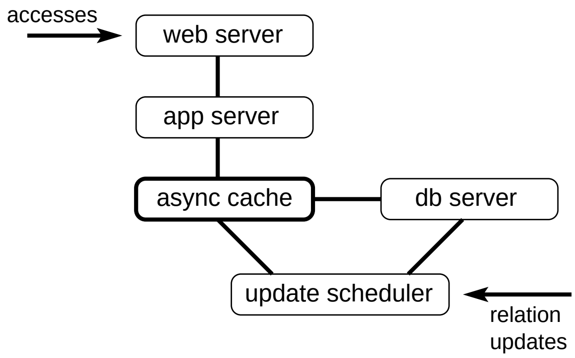

Figure 1: System Architecture

2.1 Web Page Derivation Graph

We distinguish 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. The incoming update stream affects relations only. Relation updates are executed in order of arrival.
- *WebViews* [LR99] are HTML or XML fragments. WebViews are usually generated by "wrapping" database query results (views) with HTML formatting commands or XML semantic tags (views for the Web). A WebView can be any type of HTML or XML fragment, even if it does not include data drawn from a database.
- *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.

We assume that we are given a directed acyclic graph, the *Web Page Derivation Graph*, which represents the derivation paths for all web pages. The nodes of the graph correspond to data objects in the system. An edge from node a to node b exists only if node b is derived directly from node a. A node b 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, whereas the web pages are the leafs of the graph, with zero out-degree.

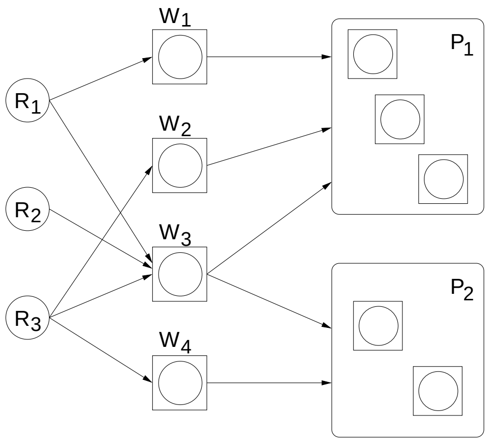

Figure 2: Web Page Derivation Graph

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 and P_2).

WebViews are derived by querying relational data and are generated by "webifying" relational views. We impose the requirement that WebViews must either be derived from relational data or from other WebViews. This means that for multi-source WebViews which are generated from other WebViews and also using relational data, we must first "wrap" the relational data with additional WebViews before using them to derive the multi-source WebViews.

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 [CIW$^+$00]. 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 (through database queries) are intercepted by the *Asynchronous Cache (ASC)*. ASC deals with objects that can be maintained under the following policies:

- **Virtual** WebViews are always executed on demand and are not cached. Intercepted queries against Virtual WebViews are simply forwarded to the database server. Database updates do not affect Virtual

WebViews. However, accessing them is 1-2 orders of magnitude slower than cached or materialized WebViews.

- **Non-Materialized** 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, it is invalidated and needs to be re-generated on a following request. This is similar to traditional caching with invalidation rather than Time-to-Live (expiration) time. Caching WebViews is always a better policy than Virtual, since, without any loss in data freshness, one obtains significant improvement in response time for all the times that a fresh version of the WebView is in the ASC.

- **Materialized** WebViews are materialized and continuously maintained under updates in the background. Accesses to them are always served from the ASC. The response time is similar to a fresh non-materialized WebView and remains almost constant since a Materialized WebView is served from ASC even if it is invalidated. However, there is a limit as to how many WebViews should be materialized. More materialized WebViews means an increase in the overhead of refreshing them and has 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 handling updates. 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 bringing the fresh version of 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. We plan to study XML WebViews in the future. Dealing only with HTML WebViews means that the cost to generate any WebView from other WebViews will be negligible (simple embedding or 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). Response times for such WebViews can be reduced by up to two orders of magnitude if materialized [LR00] and thus are the only ones that could offset the overhead of keeping them up to date in the background.

2.3 Measuring Quality of Service

In order to measure the performance of a data-intensive web server, we observe the incoming access request stream for a certain time interval T. We define *Quality of Service (QoS)* as the average response time for user requests, which corresponds to the average time required to service a web page access request. Therefore, improving the QoS is equivalent to reducing the average response time. Note that we record the request arrival and completion times *at the server* (and not at the client) in order to factor out the network latency from our measurements.

2.4 Measuring Quality of Data

We define *Quality of Data (QoD)* for data-intensive web servers as 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). 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.

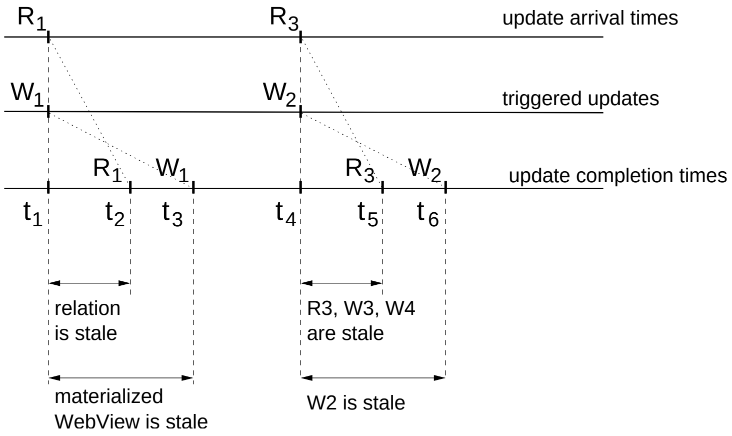

Figure 3: Staleness Example (W_1, W_2 assumed materialized)

We have four types of data objects that can be stale:

- *relations*, when an update for them has arrived, but not yet executed,
- *non-materialized WebViews*, when an update for a parent relation has arrived, but not yet executed,
- *materialized WebViews*, if the WebViews have not been refreshed yet (after an update to a parent relation),
- *web pages*, 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 an observation 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. Note that each web page P_j consists of multiple HTML fragments (WebViews). Similarly, we view the update stream during an observation interval T as a sequence of m update instructions:

$$\ldots,\ U_y,\ U_{y+1},\ U_{y+2},\ \ldots,\ U_{y+m-1},\ \ldots$$

Update instructions U_y are encoded as pairs (R_d, t_y), where t_y is the arrival time of the update instruction for relation R_d. Note that we are only interested in the portion of the update stream that arrived concurrently with the access requests under observation, or $t_x \leq t_y \leq t_{y+m-1} \leq t_{x+n-1}$.

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

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

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. This means that 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 can be a rational number between 0 and 1, with 0 being completely stale and 1 being completely fresh. To calculate freshness of web page P_j at time t_k, we take the weighted sum of the freshness values of the WebViews that compose the web page:

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

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 $\sum_{i=1}^{n_j} a_{i,j} \ = \ 1, \forall$ 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 weight factors can be user-specified or one can simply use the default value of $a_{i,j} \ = \ \frac{1}{n_j}$, where n_j is the number of WebViews in page P_j, thus giving all WebViews equal importance within the same page. For example, from Figure 2, the default weight factors will be: $\frac{1}{3}$ for P_1 WebViews (W_1, W_2, and W_3) and $\frac{1}{2}$ for P_2 WebViews (W_3 and W_4).

Let $f(A_k)$ be the freshness value of web page P_j returned by access request $A_k = (P_j, t_k)$. Then the overall Quality of Data is:

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

where n is the total number of access requests.

2.5 Online View Selection Problem

Clearly, the choice of WebViews to materialize will have a big impact on QoS and QoD. On the one extreme, materializing all WebViews will give high QoS, but can have low QoD (i.e. views will be served very fast, but can be stale). On the other hand, keeping all views non-materialized will give high QoD, but low QoS (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 (QoS) is maximized, while a guarantee on the freshness of the served data (QoD) is maintained. In addition to the incoming access/update streams, we assume that we are given a web page derivation graph, and the costs to access/update each relation/WebView.

Given the definition of QoD from Section 2.4, a freshness guarantee 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 necessarily an *online* process 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 the QoS. Second, since the accesses/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. Off-line view selection cannot match the wide variations of web workloads.

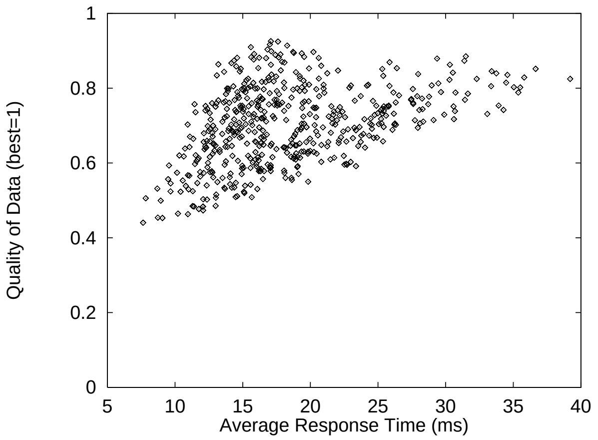

Figure 4: All Materialization Plans in QoD/QoS space

We will use the term *materialization plan* to denote possible solutions to the Online View Selection problem. A materialization plan simply lists which WebViews should be materialized, while the remaining WebViews should be just cached. 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 QoS.

To visualize the solution space for the Online View Selection Problem we enumerate all possible materialization plans for a small workload and compute the QoS and QoD (Figure 4). The different materialization plans can 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 right corner of the plot correspond to non-materializing most WebViews (with very high average response time and high QoD).

3 Online View Selection 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 realtime. 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)**, a benefit-based 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 keep the average response time as low as possible. OVIS also monitors the access stream in order to prevent server backlog.

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.

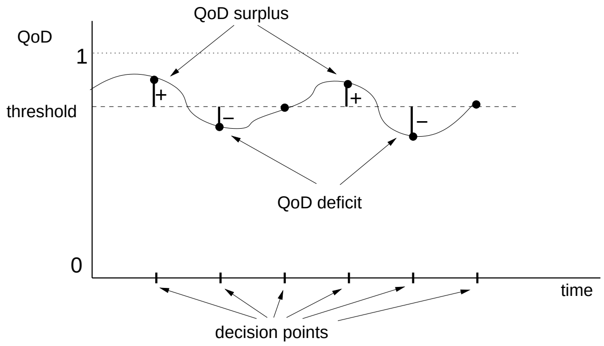

Figure 5: OVIS(θ) Algorithm

The main idea behind the OVIS(θ) algorithm is illustrated in Figure 5. By constantly monitoring the QoD for the served data, the algorithm distinguishes between two cases when it must decide on 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.

In the next paragraphs we explain the statistics that are maintained during the algorithm's passive mode, and present the pseudocode for the OVIS(θ) algorithm.

3.1 OVIS(θ) Statistics

The OVIS(θ) algorithm maintains statistics in order to accurately observe the QoD for the served data, and, to predict the effects on the QoD and average response time from changes in the materialization plan.

We have three counters for every WebView W_i. $N_{acc}(W_i)$, is the total number of accesses to W_i, $N_{fresh}(W_i)$ is the number of those accesses that were served fresh back to the user, and $N_{hits}(W_i)$ is the number of accesses that found a fresh version of WebView W_i in the cache. The difference between $N_{fresh}()$ and $N_{hits}()$ is that $N_{fresh}()$ counts freshness on the final, served results, whereas $N_{hits}()$ counts freshness based on the first attempt to read from the Asynchronous Cache (ASC).

	version in cache is fresh	version in cache is stale
WebView W_i is materialized	$N_{acc}(W_i)$++ $N_{fresh}(W_i)$++ $N_{hits}(W_i)$++	$N_{acc}(W_i)$++
WebView W_i is non-materialized	$N_{acc}(W_i)$++ $N_{fresh}(W_i)$++ $N_{hits}(W_i)$++	$N_{acc}(W_i)$++ $N_{fresh}(W_i)$++

Table 1: OVIS(θ) Counters

Table 1 illustrates the differences between the three counters. Accesses to web pages are translated to WebView accesses for all the WebViews that are part of a web page. We assume that WebViews are invalidated whenever updates to their parent relations are received. $N_{hits}(W_i)$ is incremented on an access to W_i only if the WebView is materialized and fresh, or if non-materialized, but the cached version is fresh. It will not be incremented if the cached copy of the WebView is stale, even if eventually the Asynchronous Cache will compute a fresh version of W_i and serve it to the user. The purpose of the $N_{hits}(W_i)$ counter is to help us predict the effects of changing the materialization plan (if a non-materialized WebView becomes materialized and vice-versa).

All counters are kept up to date for the duration of passive mode and are initialized every time the OVIS(θ) algorithm goes into active mode. The three aforementioned counters monitor the access and update workload for the immediate past and are used by the OVIS(θ) Algorithm to estimate the access and update patterns for the immediate future.

Calculating the QoS Benefit We use the term *materialize* a WebView to denote that a WebView gets materialized and the term *dematerialize* a WebView when we stop materializing it. Using a simple cost model we can estimate the QoS differentials of materializing or dematerializing a WebView W_i.

Let us assume that the cost to access a WebView W_i from the ASC is $A_{mat}(W_i)$ and the cost to compute a WebView W_i on demand is $A_{virt}(W_i)$. The benefit on QoS from materializing WebView W_i is

$$\begin{aligned}
\Delta_{cost}(\text{mat}) &= \mathit{cost}(W_i \text{ is not mat.}) - \mathit{cost}(W_i \text{ is mat.}) \\
&= N_{hits}(W_i) \times A_{mat}(W_i) \\
&+ (N_{acc}(W_i) - N_{hits}(W_i)) \times A_{virt}(W_i) \\
&- N_{acc}(W_i) \times A_{mat}(W_i) \qquad (4)
\end{aligned}$$

In the estimation of the cost of accessing a materialized WebView, we do not consider the cost for refreshing the WebView in the background upon updates. When estimating the cost of accessing a non-materialized WebView, we use $N_{hits}(W_i)$ to determine how many times the cached version was used, whereas for the rest of the times, $(N_{acc}(W_i) - N_{hits}(W_i))$, a fresh version needs to be generated at a greater cost.

The benefit on QoS from dematerializing W_i is

$$\Delta_{cost}(\text{demat}) = -\Delta_{cost}(\text{mat}) \qquad (5)$$

Calculating the QoD Benefit Having kept statistics on the freshness of the served data, we can accurately determine the change in QoD induced by materializing or dematerializing a WebView W_i.

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. Clearly, $\sum_j N_{fresh}(W_i \mid P_j) = N_{fresh}(W_i)$, for each WebView W_i. The overall QoD definition from Eq. 3 can be rewritten as follows:

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

where n is the total number of page accesses.

Instead of keeping separate $N_{fresh}(W_i \mid P_j)$ counters for all (WebView, page) combinations, we maintain **one**, weighted counter, $N_{fresh/a}(W_i)$. Instead of incrementing by one, we increment $N_{fresh/a}(W_i)$ using the weight $a_{i,j}$ in order to record a fresh access on W_i originating from page P_j. We have that $N_{fresh/a}(W_i) = \sum_j [a_{i,j} \times N_{fresh}(W_i \mid P_j)]$. Therefore, the QoD definition can be simplified as follows:

$$\mathit{QoD} = \frac{1}{n} \times \sum_i N_{fresh/a}(W_i) \qquad (6)$$

Based on Eq. 6, we compute the impact on QoD from materializing or dematerializing one WebView, as follows:

$$\Delta_{QoD}(W_i) = \frac{1}{n} \times [N_{fresh/a}(W_i) - N^{est}_{fresh/a}(W_i)] \qquad (7)$$

where $N_{fresh/a}(W_i)$ is the current weighted counter and $N^{est}_{fresh/a}(W_i)$ is the estimate for the counter **after** materializing or dematerializing WebView W_i.

If we assume that the access and update patterns of the immediate past hold for the immediate future, when materializing a previously non-materialized WebView W_i, the number of fresh accesses on W_i will be equal to the number of attempts to access a fresh version of W_i, or $N^{est}_{\mathit{fresh/a}}(W_i) = N_{\mathit{hits/a}}(W_i)$. Thus Eq. 7 is written:

$$\Delta_{QoD}(\text{mat } W_i) = \frac{1}{n} \times [N_{\mathit{fresh/a}}(W_i) - N_{\mathit{hits/a}}(W_i)] \tag{8}$$

where $N_{\mathit{hits/a}}(W_i)$ is the weighted version of $N_{\mathit{hits}}(W_i)$.

On the other hand, if dematerializing a WebView W_i, it will become non-materialized. In that case, the number of fresh accesses on W_i will be equal to the number of accesses to W_i, or $N^{est}_{\mathit{fresh/a}}(W_i) = N_{\mathit{acc/a}}(W_i)$. Thus Eq. 7 is written:

$$\Delta_{QoD}(\text{demat } W_i) = \frac{1}{n}[N_{\mathit{fresh/a}}(W_i) - N_{\mathit{acc/a}}(W_i)] \tag{9}$$

where $N_{\mathit{acc/a}}(W_i)$ is the weighted version of $N_{\mathit{acc}}(W_i)$.

3.2 OVIS(θ) Algorithm

The OVIS(θ) Algorithm constantly monitors the QoD of the served data and adjusts the materialization plan (i.e. which WebViews are materialized and which ones are non-materialized). By maintaining the statistics presented in the previous section, OVIS(θ) has a very good estimate of how big an effect on the overall QoD/QoS the changes in the materialization plan will have.

QoD surplus When the observed QoD Q is higher than the 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 non-materialized (but in the cache). For the algorithm to take the most profitable decision, we just need to maximize the QoS benefit, Δ_{cost}, for the WebViews that become materialized, while the QoD "losses", Δ_{QoD}, remain less than $Q - \theta$. A greedy strategy, that picks WebViews based on their $\Delta_{cost} / \Delta_{QoD}$ ratio, provides a good solution.

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 dematerialize 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 minimize the QoS "losses", Δ_{cost}, for the WebViews that become cached, while the QoD benefits (Δ_{QoD}) eliminates $\theta - Q$. A greedy strategy, that picks WebViews based on their $-\Delta_{cost} / \Delta_{QoD}$ ratio, provides a good solution.

Pseudocode The OVIS(θ) algorithm is in Passive Mode most of the time, collecting statistics. Periodically, OVIS(θ) enters Active Mode in order to make changes to materialization decisions. Figures 6 and 7 present the active mode of the OVIS(θ) algorithm under QoD surplus and QoD deficit conditions.

3.3 Detecting Server Lag

From elementary queueing theory [Jai91] we know that system performance worsens significantly as we approach 100% utilization. In practice, there can be cases where the incoming access and update workload

QoD surplus: $QoD > \theta$

0. qod_diff = $QoD - \theta$
1. select cached WebViews with $\Delta_{cost}(\text{mat}) > 0$
2. materialize all WebViews with $\Delta_{QoD}(\text{mat}) = 0$
3. find W_i with max $\Delta_{cost}(\text{mat}) \,/\, \Delta_{QoD}(\text{mat})$
4. if ($\Delta_{QoD}(\text{mat}) <$ qod_diff)
5.   qod_diff $-=$ $\Delta_{QoD}(\text{mat})$
6.   materialize W_i
7.   goto 3
8. else
9.   stop

Figure 6: Pseudocode for OVIS(θ) - QoD surplus

QoD deficit: $QoD < \theta$

0. qod_diff = $\theta - QoD$
1. select materialized WebViews
2. find W_i with max $-\Delta_{cost}(\text{demat}) \,/\, \Delta_{QoD}(\text{demat})$
3. if ($-\Delta_{QoD}(\text{demat}) <$ qod_diff)
4.   qod_diff $+=$ $\Delta_{QoD}(\text{demat})$
5.   stop materializing W_i
6.   goto 2
7. else
8.   stop

Figure 7: Pseudocode for OVIS(θ) - QoD deficit

generate more load than what the server can handle, resulting in backlog, which we refer to as *server lag*. Figure 8 has an example of server lag, which is visible on the access stream.

It is crucial to detect server lag in an online system. Failure to identify server lag can lead to long, ever-increasing backlogs which will eventually bring the server to a halt with catastrophic consequences. In our system, we detect server lag by instrumenting the incoming access stream with special markers which are inserted in the access stream at regular intervals.

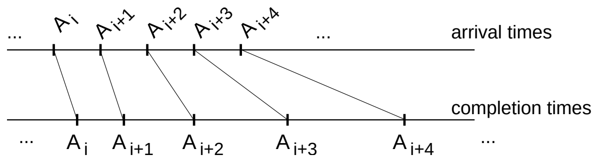

Figure 8: Server Lag Example

In order to identify server lag for an observation window T, we compare the arrival and completion times of the markers closer to the observation start/end-points. If the beginning marker is A_i and the end marker is A_{i+n}, we have that:

$$lag = \frac{t_a(A_{i+n}) - t_a(A_i)}{t_c(A_{i+n}) - t_c(A_i)} \tag{10}$$

where t_a is the arrival time for the marker, and t_c is the completion time. Markers have zero processing cost, so $t_c - t_a$ is just the current queueing time for access requests. In a stable system, we expect *lag* to be equal or very close to 1. When the processing demands exceed the processing capacity, *lag* will be significantly lower than 1 and eventually approach 0.

Server lag can be used to detect *infeasible QoD thresholds*. For example a QoD threshold very close to 1 will most likely lead to a server meltdown, since no WebView can be materialized (for the system to support such a strong freshness guarantee).

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.

Our online simulator, `osim`, runs in two modes: *static mode* and *adaptive mode*. In static mode, the materialization plan is pre-specified and is fixed for the duration of the simulation. In adaptive mode, the materialization plan is modified at regular intervals using the OVIS(θ) algorithm (Figure 6 and Figure 7). For each experiment we report the average response time and the observed QoD.

In order to detect server *lag* and be able to prevent server backlogs, we instrument the access stream with special markers as explained in Section 3.3. These markers are inserted at regular intervals in the incoming access stream as access requests for a special web page. In a real system, the markers can be added at run-time without the need to modify the incoming access stream.

Workload Characteristics In all experiments we used synthetic workloads. The accesses were distributed over web pages following the Zipf distribution [BCF+99] and the updates were distributed uniformly among relations. Interarrival rates for the access and the update stream approximated a uniform distribution. The cost to update a relation was 100 ms, the cost to access a WebView from the Asynchronous Cache was 10 ms and the cost to generate/refresh a WebView was 100 ms in all experiments. The database size, the number of accesses, the number of updates and the duration varied in each experiment.

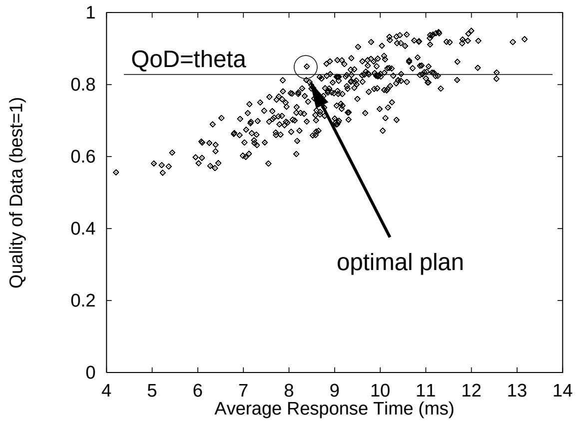

Figure 9: Off-line Optimal Computation

4.1 Computing the Off-line optimal

In order to compute the off-line optimal materialization plan for the given workload, we first enumerate all possible materialization plans. Each WebView can either be materialized or non-materialized, therefore we have 2^N possible plans, where N is the number of WebViews in the system (Figure 9). Out of the 2^N plans, we pick the ones that have QoD $> \theta$, where θ is the user-specified QoD threshold. These plans are the ones above the $QoD = \theta$ line in Figure 9. Finally, out of the plans with QoD $> \theta$ we identify the one with the lowest average response time, which will be the optimal plan.

4.2 Comparing the Off-line optimal to OVIS(θ)

In order to compare our algorithm with the off-line optimal (OPT), we used a relatively small database schema, of 5 relations, 12 WebViews, and 30 web pages. This configuration generates $2^{12} = 4096$ materialization plans which the optimal algorithm must check, which makes the solution tractable. The workload consisted of 10,000 web page accesses and 3,000 relation updates over a period of 140,000 ms.

	Optimal		OVIS(θ)	
θ	QoD	resp. time	QoD	resp. time
0.95	-	-	0.94	11.7 ms
0.90	0.90	9.5 ms	0.90	10.0 ms
0.85	0.85	8.4 ms	0.85	8.9 ms
0.80	0.81	7.8 ms	0.80	8.2 ms
0.70	0.71	6.5 ms	0.72	6.8 ms
0.60	0.61	5.4 ms	0.65	5.9 ms
0.50	0.55	4.2 ms	0.59	4.8 ms

Table 2: Comparison with Optimal

Table 2 has the results of our experiments for various QoD thresholds. For example, for a QoD threshold of 0.70, the optimal materialization plan would have QoD of 0.71 and an average response time of 6.5 ms, whereas OVIS(θ) has QoD 0.72 and a slightly worse average response time of 6.8 ms.

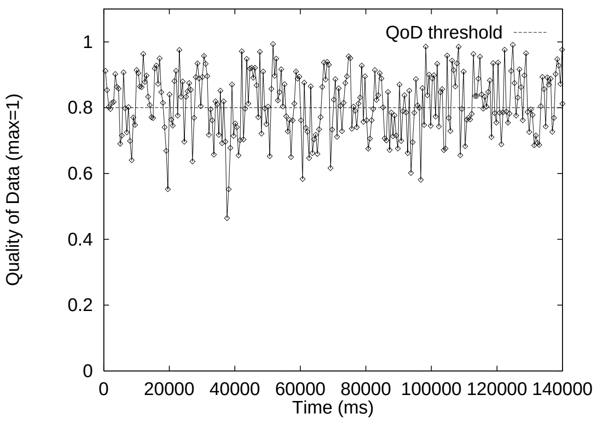

Figure 10: QoD over time for the Optimal Plan

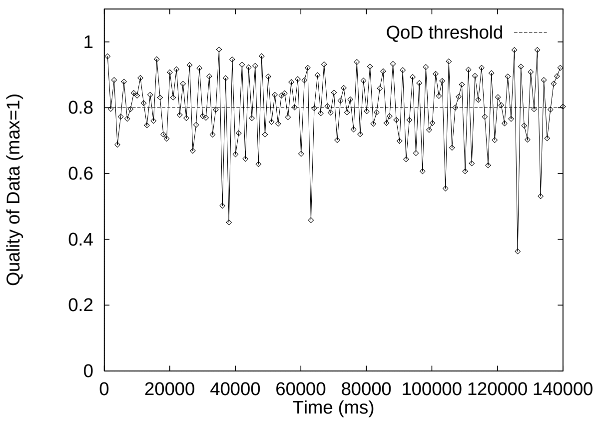

Figure 11: QoD over time for OVIS(0.8)

Overall, we see that OVIS(θ) produces QoD values very close to those of the optimal materialization plan. Also, the average response times under OVIS(θ) are within 10% of the optimal, which is respectable if we consider that OVIS(θ) decides on the materialization plan in an online fashion without prior knowledge of the access/update stream like OPT does. Table 2 clearly illustrates the trade-off between QoD and average response time: when the QoD increases the average response time also increases and vice-versa.

We plot the QoD over time for the Optimal and OVIS(θ) algorithm when $\theta = 0.8$ in Figures 10 and 11. In both cases there is a fluctuation of the QoD, around the required threshold θ. However, there is one important difference between the two graphs. In Figure 11 the QoD always alternates from QoD surplus to QoD deficit states, under OVIS(θ), in its goal of reaching the QoD threshold θ. There is no similar behaviour for the Optimal case (Figure 10).

4.3 OVIS(θ) study

For this set of experiments, we used a moderate-sized database of 100 base relations, 500 WebViews and 2000 web pages. The access workload composed of 50,000 web page accesses and the update workload had 10,000 relation updates. The duration of this experiment was 600,000 ms.

	OVIS(θ)	
θ	QoD	resp. time
0.97	0.968	15.1 ms
0.95	0.951	15.8 ms
0.92	0.917	14.0 ms
0.90	0.899	12.0 ms
0.85	0.867	9.6 ms
0.80	0.856	9.4 ms

Table 3: QoD for OVIS(θ)

Table 3 has the results of our experiments for various threshold values. In all cases, OVIS(θ) maintained the QoD guarantee. We did not compute the optimal plan for this workload, since we would need to enumerate and select the best out of 2^{500} plans.

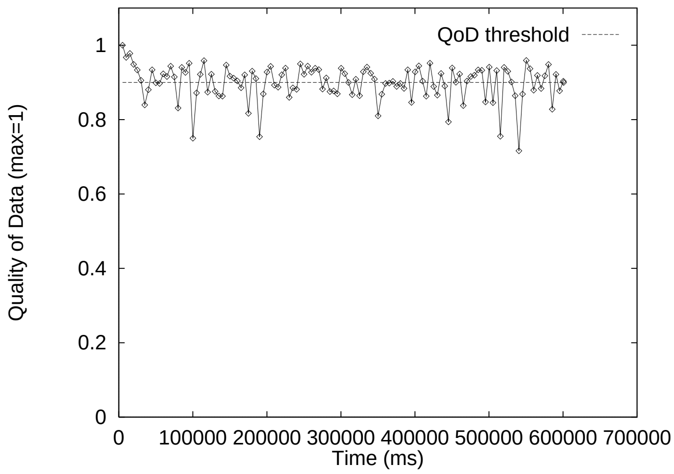

Figure 12: QoD over time for OVIS(0.9) - Static workload

Figure 12 plots the changes of QoD over time for the OVIS(0.9) experiment. Clearly, despite the high number of WebViews, OVIS(θ) manages to maintain a QoD around the user-specified threshold of 0.9

We ran a variation of the same workload, to simulate changing access and update patterns. Instead of generating the entire access and update stream with one distribution function, we used two seperate functions to generate the first and second half of the streams. In other words, the access and update frequencies between the two halfs were totally independent. Figure 13 plots the changes of QoD over time for the OVIS(0.8) experiment with the dynamic workload. Despite the drastic change in the access and update patterns, OVIS(θ) rapidly adapts to the new workload and manages to maintain the QoD guarantee.

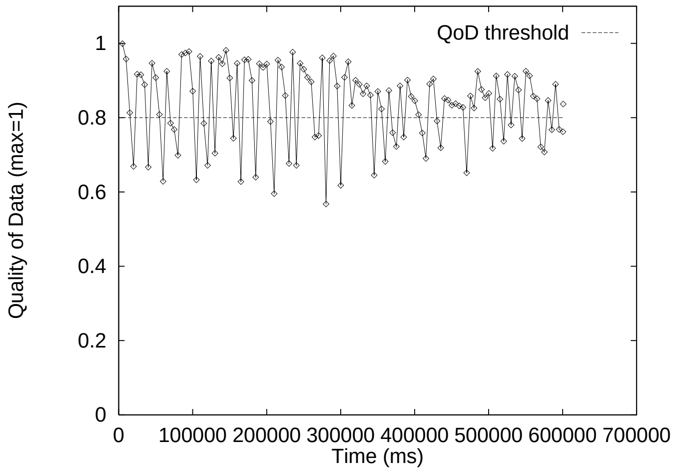

Figure 13: QoD over time for OVIS(0.8) - Changing workload

4.4 QoS vs. QoD trade-off

For the last set of experiments, we used a moderate-sized database of 100 base relations, 500 WebViews and 2000 web pages. The access workload composed of 50,000 web page accesses and the update workload had 10,000 relation updates. The duration of this experiment was 600,000 ms.

We compared three different policies: materializing all WebViews (`all-mat`), not materializing any WebView (`none-mat`), and our adaptive algorithm for dynamically selecting which WebViews to materialize with a QoD threshold of 0.7 (`ovis(0.7)`).

In Figure 14 we plot the QoD over time for the three policies and in Figure 15 we plot the QoS over time. When we do not materialize any WebView (top curve in both plots), we get the highest QoD, but also get the highest response times, and thus the lowest QoS. On the other hand, if we materialize all WebViews, then the average response time will be the lowest, but the QoD can be below the user-specified threshold. The OVIS(θ) algorithm produces QoD that is always near the user-specified QoD threshold and has fairly constant and small average response times (near those of the `all-mat` policy). In other words, OVIS(θ) is able to automatically balance the trade-off between QoS and QoD.

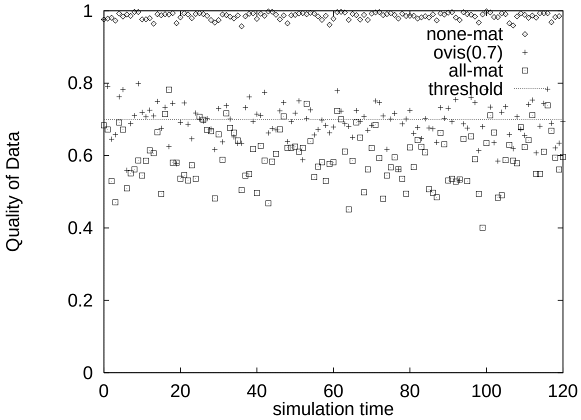

Figure 14: QoD over time

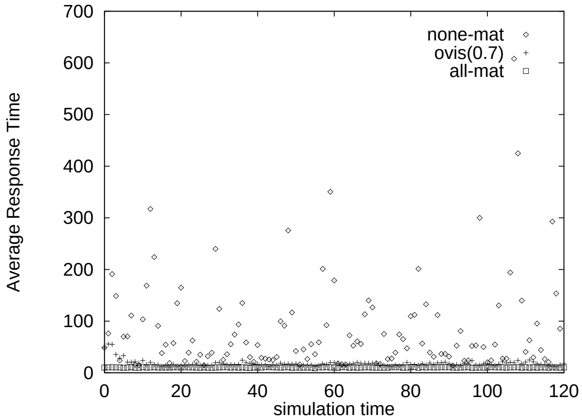

Figure 15: QoS over time

5 Related Work

To the best of our knowledge this is the first paper that attempts to solve the Online View Selection problem. There is a lot of research on the off-line version of the view selection problem in Data Warehousing literature [Sel88, GM95, SSV96, Gup97, GM99, KR99]. Materialization has also been explored in [YFIV00] where multiple maintenance policies were supported, but the selection problem was not addressed. [LR00] presented a cost model for the view selection problem, but did not provide an algorithm. [CIW$^+$00] described a system to support caching of dynamic content, but the application or the user has to decide which data should be cached. [LR01a] provided an algorithm for solving the constrainted version of the online view selection problem, i.e. only when the number of views is given.

6 Conclusions

In this work, we have defined the Online View Selection problem: given a Quality of Data (QoD) threshold θ, choose which WebViews to materialize so that QoD remains above θ and the average response time is minimized. We have also presented OVIS(θ), an adaptive algorithm for solving the Online View Selection problem. OVIS(θ) recognizes QoD surplus or deficit situations and evolves the materialization plans to improve the average response time. OVIS(θ) is highly adaptive and does not require advance knowledge of the access / update patterns. Simulation experiments have shown that OVIS(θ) comes close to the behavior of the optimal off-line algorithm.

One of the main advantages of OVIS(θ) is that it does not require human intervention. The algorithm automatically selects the best materialization plan under the specified QoD constraints. This makes view materialization (which has up till now only been used in data warehousing environments) an attractive method of increasing the scalability of data-intensive web servers.

References

[BBC+98] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, H. V. Jagadish, Michael Lesk, Dave Maier, Jeff Naughton, Hamid Pirahesh, Mike Stonebraker, and Jeff Ullman. "The Asilomar Report on Database Research". *SIGMOD Record*, 27(4), December 1998.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. "Web Caching and Zipf-like Distributions: Evidence and Implications". In *Proc. of IEEE INFOCOM'99*, New York, USA, March 1999.

[BM90] José A. Blakeley and Nancy L. Martin. "Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis". In *Proc. of the Sixth International Conference on Data Engineering*, pages 256–263, Los Angeles, California, USA, February 1990.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. "An Overview of Data Warehousing and OLAP Technology". *SIGMOD Record*, 26(1):65–74, March 1997.

[CIW+00] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. "A Publishing System for Efficiently Creating Dynamic Web Content". In *Proc. of IEEE INFOCOM'2000*, Tel Aviv, Israel, March 2000.

[GM95] Ashish Gupta and Inderpal Singh Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". *Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GM99] Ashish Gupta and Inderpal Singh Mumick, editors. *"Materialized Views: Techniques, Implementations, and Applications"*. MIT Press, June 1999.

[Gup97] Himanshu Gupta. "Selection of Views to Materialize in a Data Warehouse". In *Proc. of the 6th International Conference on Database Theory (ICDT '97)*, pages 98–112, Delphi, Greece, January 1997.

[Han87] Eric N. Hanson. "A Performance Analysis of View Materialization Strategies". In *Proc. of the ACM SIGMOD Conference*, pages 440–453, San Francisco, California, May 1987.

[Jai91] Raj Jain. *"The Art of Computer Systems Performance Analysis"*. John Wiley & Sons, 1991.

[KR99] Yannis Kotidis and Nick Roussopoulos. "DynaMat: A Dynamic View Management System for Data Warehouses". In *Proc. of the ACM SIGMOD Conference*, Philadelphia, USA, June 1999.

[LR99] Alexandros Labrinidis and Nick Roussopoulos. "On the Materialization of WebViews". In *Proc. of the ACM SIGMOD Workshop on the Web and Databases (WebDB'99)*, Philadelphia, USA, June 1999.

[LR00] Alexandros Labrinidis and Nick Roussopoulos. "WebView Materialization". In *Proc. of the ACM SIGMOD Conference*, Dallas, Texas, USA, May 2000.

[LR01a] Alexandros Labrinidis and Nick Roussopoulos. "Adaptive WebView Materialization". In *Proc. of the ACM SIGMOD Workshop on the Web and Databases (WebDB'2001)*, Santa Barbara, California, USA, June 2001.

[LR01b] Alexandros Labrinidis and Nick Roussopoulos. "Update Propagation Strategies for Improving the Quality of Data on the Web". In *Proc. of the 27th VLDB Conference*, Rome, Italy, September 2001.

[Mar01] Evangelos Markatos. "On caching search engine query results". *Computer Communications*, 24(2), February 2001.

[PQ00] Venkata N. Padmanabhan and Lili Qiu. "The Content and Access Dynamics of a Busy Web Site: Findings and Implications". In *Proc. of the ACM SIGCOMM Conference*, Stockholm, Sweden, August 2000.

[RCK+95] Nick Roussopoulos, Chungmin Melvin Chen, Stephen Kelley, Alex Delis, and Yannis Papakonstantinou. "The ADMS Project: Views R Us". *Data Engineering Bulletin*, 18(2):19–28, June 1995.

[Rou82] Nick Roussopoulos. "View Indexing in Relational Databases". *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[Sel88] Timos Sellis. "Intelligent caching and indexing techniques for relational database systems". *Information Systems*, 13(2), 1988.

[SSV96] Peter Scheuermann, Junho Shim, and Radek Vingralek. "WATCHMAN : A Data Warehouse Intelligent Cache Manager". In *Proc. of the 22nd VLDB Conference*, Bombay, India, September 1996.

[YFIV00] Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. "Caching Strategies for Data-Intensive Web Sites". In *Proc. of the 26th VLDB Conference*, Cairo, Egypt, September 2000.