TECHNICAL RESEARCH REPORT

Adaptive WebView Materialization

by Alexandros Labrinidis, Nick Roussopoulos

CSHCN TR 2001-14
(ISR TR 2001-22)

Adaptive WebView Materialization[*]

Alexandros Labrinidis   Nick Roussopoulos

labrinid@cs.umd.edu   nick@cs.umd.edu

Department of Computer Science and Institute for Systems Research

University of Maryland, College Park

June 28, 2001

Abstract

Dynamic content generation poses huge resource demands on web servers, creating a scalability problem. WebView Materialization, where web pages are cached and constantly refreshed in the background, has been shown to ameliorate the scalability problem without sacrificing data freshness. In this work we present an adaptive online algorithm to select which WebViews to materialize, that realizes the trade-off between Quality of Service and Quality of Data. Our algorithm performs very close to the static, off-line optimal algorithm, and, under rapid workload changes, it outperforms the optimal.

1 Introduction

Online services, frequently updated content and personalization make dynamic content ubiquitous on the Web today. Unfortunately, the high resource demands of the dynamically generated web

pages create a significant scalability problem. Although web caching was able to address the scalability problem for static web pages, it will not work for dynamic content, since caching cannot provide any guarantees for the freshness of the served data. Serving user requests fast is of paramount importance only if the data is fresh. With many data-intensive web servers being used for critical applications, serving stale data can have catastrophic consequences.

We have showed in [LR00] that *web materialization*, where web pages are cached and constantly refreshed in the background, is a robust solution to the scalability problem. We use the term WebView to refer to the HTML fragments that are the unit of materialization. We presented a detailed cost model that can be used in an off-line fashion to select which WebViews to materialize. However, with the highly dynamic & unpredictable nature of web traffic, an *adaptive* and *online* algorithm is needed to select which WebViews to materialize.

In this work, we present metrics for the Quality of Service (QoS) and the Quality of Data (QoD) at data-intensive web servers. We focus on the *constrained View Materialization problem*: given a limit on the number of WebViews to materialize, select which WebViews to materialize, so that the overall QoS and QoD are maximized. We describe an adaptive online view materialization algorithm that does not rely on a cost model and briefly present the results of our preliminary experiments.

2   Measuring Quality of Service and Quality of Data

We assume a web server architecture similar to that of Figure 1. The web server is the front-end for serving user requests. All requests that require dynamically generated data from the DBMS are intercepted by the *asynchronous cache* and are only forwarded to the DBMS if the data is not cached. Unlike traditional caches in which cached data is invalidated on updates, in the asynchronous cache data elements are *materialized* [LR00] and constantly being refreshed in the background. The *view selection module* collects statistics and determines which WebViews should be kept materialized. All updates in our system must be performed *online*. A separate module, the *update scheduler* is

responsible for scheduling the DBMS updates and the refresh of the WebViews in the asynchronous cache.

Although our work is motivated by database-backed web servers and materialized WebViews, it applies to any system that supports online updates. For the rest of the paper, we will use the more general term *views* instead of *WebViews*.

We assume a database schema with N relations, r_1, r_2, ..., r_N and M views, v_1, v_2, ..., v_M. Views are derived from relations or from other, previously derived views. We distinguish two types of views: *virtual*, which are generated on demand, and *materialized*, which are precomputed, stored in the asynchronous cache and refreshed asynchronously. All user requests

Figure 1: System Architecture

are expressed as view accesses, whereas all incoming updates are applied to relations only and may trigger view refreshes. Finally, we assume that incoming access requests are served in FIFO order, incoming relation updates are also performed in FIFO order, whereas materialized view refreshes can be performed in any order.
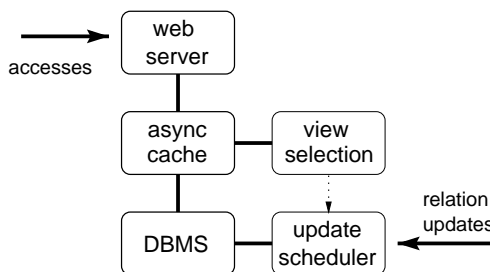
2.1 Quality of Service

Typically, Quality of Service (QoS) is measured through the average *query response time*, which corresponds to the time required to service an access request at the web server. Although, the average query response time is an accurate measure for the performance of a system, it lacks *portability*, since we cannot compare average query response times on different systems or with different workloads. Furthermore, since we want to be able to consider both Quality of Service and Quality of Data, we need to use intuitive [0,1] metrics, with 0 corresponding to the worst and 1 to the best possible performance/data quality for our system.

Performance improves when more views are kept materialized. In fact, the higher the number of requests that can be served by the asynchronous cache, the higher the overall performance.

3

Under this premise, we define Quality of Service as the percentage of accesses that are served by materialized views:

$$\text{QoS} = \frac{\text{number of accesses to materialized views}}{\text{total number of accesses}} \tag{1}$$

This QoS definition is similar to the hit rate definition for traditional caches. However, unlike traditional caches, the asynchronous cache refreshes all its stored objects on updates and does not change its list of objects on cache misses (materialization decisions are made in the background). Finally, it should be noted that our QoS definition is not equivalent to the percentage of views that are materialized, as views are not accessed with the same probability.

2.2   Quality of Data

When an update to a relation is received, the relation and all views 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 definition with the following example.
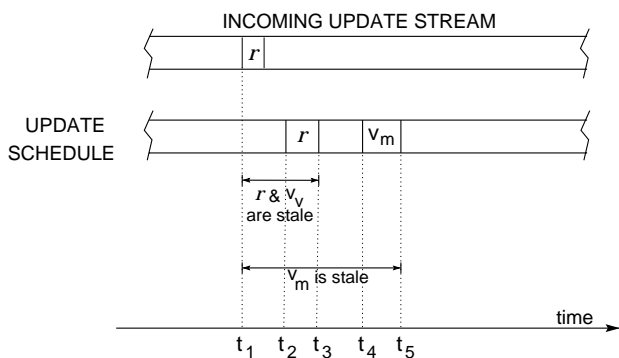

Figure 2: Staleness Example

Assume a database with one relation r and two views: v_v which is virtual and v_m which is materialized. Also assume that at time t_1 an update for relation r arrives (Figure 2). Relation r will become up to date after it is updated. If the update on r starts at time t_2 and is completed at time t_3, then relation r will have been stale from time t_1 until t_3. On the other hand, virtual view v_v will become up to date after all of its parent relations/views are updated. Since relation r was updated at time t_3, view v_v will have been stale from time t_1 until t_3. Finally, materialized view v_m will become up to date after it is refreshed. If the refresh of v_m starts at time t_4 and is completed at time t_5, then view v_m will have been stale from time t_1 until t_5.

4

To measure the Quality of Data (QoD) for the data served to the user, we calculate the percentage of accesses that are performed on *fresh* views, i.e. views that are not stale. Specifically, we consider the freshness status of a view at the start of processing an access request. For the example in Figure 2, an access request for view v_v will be considered fresh if the processing of the request starts before t_1 or after t_3. Overall, we define:

$$\text{QoD} = \frac{\text{number of fresh accesses}}{\text{total number of accesses}} \tag{2}$$

3   Adaptive View Selection Algorithm

Clearly, the choice of views to materialize will have a big impact on QoS and QoD. On the one extreme, materializing all views will give perfect QoS, but very low QoD (i.e. views will be served very fast, but will be very stale). On the other hand, keeping all views virtual will give high QoD, but zero QoS (i.e. views will be as fresh as possible, but the query response time will be high). We refer to the problem of selecting which views to materialize, so that both QoS and QoD are maximized, as the *View Materialization* problem. We will use the term *materialization plan* to denote which views are materialized and which are virtual. In this work, we focus on the *constrained version of the View Materialization problem* : select the materialization plan that maximizes both QoS and QoD, when there is a constraint on the number of views that can be materialized. The constraint can be because of the size of the asynchronous cache (which can fit k materialized views) or because of the update processing speed of the system which can limit the number of materialized views that can be refreshed in the background.

Monitoring Materialization Health   The objective of the View Materialization Problem is to find views that are accessed a lot and have relatively few updates. Since we have limited resources (storage space in the asynchronous cache and processing capacity for performing updates), we want to keep materialized those views that are being *"used"* a lot after each refresh. The amount of view "usage" is simply the number of accesses to that view, since it was last refreshed. In other words,

we want to select for materialization views that have high *refresh utilization*.

A simple way to measure refresh utilization is to instrument each view with a private counter. The refresh utilization counter is incremented every time the view is accessed and is set to zero every time the view is refreshed. If a view is virtual, we reset the counter whenever the virtual view would have been refreshed, if it were materialized. The higher the value of the refresh utilization counter, the more "used" the materialized version of the view is/will be, and therefore the bigger the gains if we keep the view materialized.

The only drawback of the refresh utilization counter is that it does not consider time. For example, we can have two views, v_1 and v_2 with equal refresh utilization counters (assume 5), but v_1 receives 100 accesses per second (and therefore roughly 20 refreshes per second), whereas v_2 might only get 5 accesses per second and only 1 refresh per second on average. Obviously, we should be able to distinguish between the two, and select v_1 for materialization, since it improves QoS, while occupying the same space as v_2. For that reason, instead of the refresh utilization counter, we use the *ratio* of accesses to refreshes over a period of time, which we refer to as the *materialization health* of view v_i, or $mh(v_i)$. We have that:

$$mh(v_i) = \frac{\text{num\_accesses}}{1 + \text{num\_refreshes}} \tag{3}$$

In Eq. 3, num_accesses is the number of accesses to view v_i during the observation period. If view v_i is materialized, then num_refreshes is the number of refreshes that v_i had during the observation period. If view v_i is virtual, then num_refreshes is the number of refreshes that v_i would have had, if it were materialized.

The reason for the $+1$ in the denominator of Eq. 3 is two-fold. First of all, it guarantees a valid $mh()$ value, even when there are no refresh operations during a certain observation interval. This prevents getting a lot of views with $mh() = \infty$ (which would not allow us to compare these views with the rest), even if we have a lot of static views. Secondly, the $+1$ corresponds to the initial generation query of view v_i, since it is already loaded in the asynchronous cache at the beginning

6

of the observation period.

Algorithm   Based on the concept of materialization "health", we present the details of our ADap-tive View Materialization Algorithm (ADVMA). At pre-specified intervals, the current access and refresh counters are consolidated into $mh()$ values for each view (Figure 3). In order to derive the materialization plan with the highest QoS+QoD, under a materialization constraint (assume k), ADVMA selects to **materialize the k views with the highest $mh()$ values** and make the re-maining views virtual. After the decision is made, the system concurrently serves access requests (while maintaining access counters) and updates relations or refreshes materialized views in the background (while maintaining refresh counters). At pre-specified intervals, the current access and refresh counters are consolidated again into $mh()$ values and the decision process is repeated.

1.	compute $mh()$ for all views
2.	materialize k views with highest $mh()$ values
	make remaining views virtual
3.	initialize counters
4.	serve access requests ‖ update relations
	maintain num_accesses ‖ refresh materialized views
	‖ maintain num_refreshes
5.	synchronize at pre-specified time intervals
6.	goto step 1

Figure 3: Pseudo-code for the Adaptive View Materialization Algorithm

ADVMA has many advantages. First of all it is inherently adaptive. It will "pick up" changes in the access & update workloads and react accordingly. Secondly, it does not rely on cost estimates, which makes the algorithm ideal for rapidly changing environments. Thirdly, the algorithm is *on-line*: it does not require the entire access and update stream to decide which views to materialize. Fourthly, it operates in the background, at fixed intervals. This means that a) the selection decision is not on the critical path of serving access requests (which is the case with traditional cache admis-sion and replacement algorithms), b) ADVMA imposes little overhead on the system (runs period-ically and not all the time), and, c) ADVMA has the ability to consolidate statistics and therefore be tolerant to minor fluctuations. Finally, we can tune how adaptive ADVMA will be by modifying

the interval at which ADVMA will re-evaluate the current materialization plan periodically.

4 Experiments

In this section we present results from our experiments using a high-level simulator. The database schema, the update costs for relations, the access and refresh costs for views, the incoming access request stream, the speed in which accesses can be processed, the incoming update stream and the speed in which updates can be performed are inputs to the simulator. The simulator runs in two modes: a) using a specified materialization plan, it can calculate the overall QoS and QoD for the given parameters, and b) it can use ADVMA to adapt the materialization plan at specified intervals and calculate the overall QoS and QoD. Access requests and relation updates are scheduled in FIFO order, whereas view refreshes are scheduled so that QoD is maximized, by scheduling popular or "inexpensive" refreshes first [LR01].

ADVMA vs optimal for static workloads   In the first experiment, we used a database of 16 views and four relations. We created a synthetic access request stream where the view popularity followed the Zipf distribution [BCF+99] and also a synthetic update stream with uniform distribution of updates over the views. The duration of the streams was 60 seconds or 60,000 milliseconds (our simulator's internal clock run at milliseconds). The reason we used such a small database was that we wanted to enumerate all possible materialization plans and identify the selection of materialized views with the highest QoS+QoD value. Assuming a materialization constraint of four views (=25%) the possible materialization plans are 1820.

Figure 4 presents the QoS, QoD values for all materialization plans that have four materialized views. The bottom curve is the QoS, the middle curve is the QoD and the top curve is the sum of QoS and QoD. The plans on the x-axis are listed in "lexicographical" order (first one is 0000 0000 0000 1111, second one is 0000 0000 0001 0111, etc, and the last one is 1111 0000 0000 0000, where the ith bit is 1 if v_i is materialized). Views were named in order of frequency of access (i.e. v_1 is the most popular view and, in general, first bits correspond to most popular views). Although at
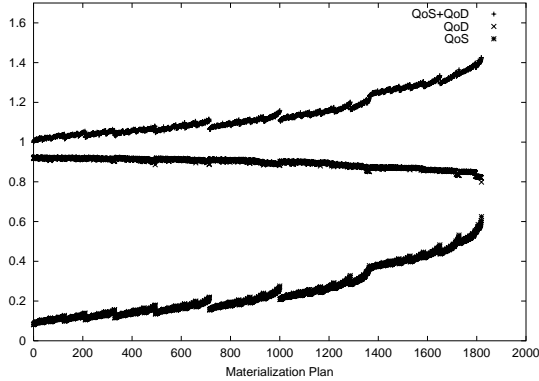
Figure 4: All possible materialization plans

algorithm	QoS + QoD	QoS	QoD
static-optimal	**1.425**	0.605	0.820
worst-static	0.998	0.081	0.917
ADVMA-static	1.423	0.624	0.799
ADVMA (10 sec)	1.423	0.624	0.799
ADVMA (5 sec)	1.423	0.624	0.799
ADVMA (3 sec)	1.424	0.623	0.801
ADVMA (2 sec)	1.422	0.619	0.803
ADVMA (1 sec)	1.412	0.603	0.809

Table 1: Optimal vs ADVMA

all times we have only four materialized views, they clearly correspond to different access request volumes (increasing from left to right) and thus the QoS is improving. On the other hand, more accesses to materialized views means that there is a greater chance for an access to read stale data, and therefore QoD decreases.

Table 1 compares ADVMA with the static algorithms. For all static algorithms we assume that we know the entire access & update stream in advance and we have one materialization plan for the duration of the experiment. On the other hand, our adaptive algorithm, ADVMA, periodically re-evaluates the materialization plan and can modify it in order to improve the overall QoS and QoD. We have enumerated all possible materialization plans for this experiment's workload and ran the simulator for every plan. The plan which gives the highest QoS+QoD corresponds to the *static-optimal* algorithm (Table 1, first row), whereas the plan with the lowest QoS+QoD is the *worst-static* plan (Table 1, second row). *ADVMA-static* is the static version of the ADVMA algorithm, where we decide the materialization plan once, based on the access/update statistics of the entire workload, using the $mh()$ function (Table 1, third row). Finally, the remaining rows, correspond to the fully dynamic version of ADVMA (where knowledge of the future is not required), with varying intervals[1]. We can see that ADVMA-static is very close to the optimal static case (equal to 99.86% of the optimal) and therefore could be used for approximating static-optimal. Furthermore, all dynamic ADVMA cases have given QoS+QoD values that are very close to the static optimal,

---

[1] A 5-second interval means that ADVMA will re-evaluate the materialization plan every five seconds

despite the fact that they do not have knowledge of the future like the static algorithms.

ADVMA vs optimal for semi-static workloads   In the second experiment, we kept the same setup as the first experiment with one variation: we split the access request stream into two segments (of 30 seconds each). The two segments had identical properties, except for the popularities of two frequently requested views from the first segment which were decreased in the second segment, to simulate a shift of interest. Using this "semi-static" workload we aim to expose the inadequacies of static materialization plans under changing conditions.

algorithm	QoS + QoD	QoS	QoD
static-optimal	1.317	0.523	0.794
worst-static	0.985	0.081	0.904
ADVMA-static	1.314	0.519	0.795
ADVMA (6 sec)	1.325	0.542	0.783
ADVMA (4 sec)	**1.332**	0.545	0.787
ADVMA (2 sec)	1.322	0.528	0.794

Table 2: Semi-static experiment results

algorithm	QoS + QoD	QoS	QoD
ADVMA-static	1.465	0.470	0.995
ADVMA (30 sec)	1.482	0.487	0.995
ADVMA (20 sec)	1.518	0.524	0.994
ADVMA (10 sec)	**1.534**	0.540	0.994
ADVMA (5 sec)	1.523	0.528	0.995
ADVMA (2 sec)	1.496	0.502	0.994

Table 3: Results for the dynamic experiment

In Table 2 we compare the static-optimal case with the materialization plans generated by AD-VMA. As was the case with the previous experiment, ADVMA-static is a good approximation of the optimal, since it has QoS+QoD that is 99.77% of the static-optimal case (which was generated by enumerating all possible materialization plans). On the other hand, ADVMA (with 6, 4 and 2-second intervals) outperforms the static algorithms, including the optimal.

ADVMA with dynamic workloads   In the last experiment we increased the size of the database and generated a highly dynamic workload in order to see how well ADVMA can adapt to changing conditions. Using a database of 2000 views and 100 relations, we created a 5-minute access stream, where the popularities of the views changed every minute, to simulate a rapidly changing workload. We report the QoS and QoD for ADVMA-static and ADVMA with multiple re-evaluation intervals in Table 3. Although ADVMA-static had knowledge of the future (and in previous experiments was very close to the static-optimal plan), the highly dynamic nature of this workload dictates using an adaptive algorithm like ADVMA. ADVMA consistently outperformed the static algorithm for all

10

re-evaluation intervals. For example, ADVMA-10 (i.e. ADVMA which re-evaluates the material-ization plan every 10 seconds) was about 5% better than ADVMA-static, which approximated the optimal algorithm in previous experiments.

5   Related Work

Existing work on caching dynamic data has focused on providing an infrastructure to support caching of dynamically generated web pages. The decision of which pages to cache, when to cache them and when to invalidate or refresh them is left to the application program [CIW+00] or the web site designer [YFIV00]. In other words, none of the aforementioned papers deals with the *selection* problem: identifying which dynamic data to cache and which not to cache. In [LR00], we pro-vided a detailed cost model to help with the view materialization problem, but did not provide an online selection algorithm. Finally, the decision whether to materialize a WebView or not, is similar to the problem of selecting which views to materialize in a data warehouse [KR99], known as the *view selection problem*. The most important difference that distinguishes our work from existing view selection literature is that in the web server case, updates are performed *online*, whereas data warehouses are updated in an off-line fashion.

6   Conclusions

We introduced intuitive, normalized Quality of Service (QoS) and Quality of Data (QoD) metrics for data-intensive web servers. We presented ADVMA, an adaptive online algorithm for selecting which views to materialize in order to maximize QoS and QoD. ADVMA poses little overhead to the web server, does not rely on a cost model and is highly adaptive. Through experiments, we illustrated that ADVMA performs very close to the static-optimal case and even outperforms static view selection (which is based on knowledge of the future) under rapidly changing workloads.

For our future work, we plan to build an industrial-strength prototype of a database-backed web server in order to validate the results from our simulation experiments. We also plan to experiment

with alternative measures of overall QoS and QoD, and consider environments with user-triggered updates, like the one in TPC-W.

Acknowledgments We would like to thank the anonymous reviewers for their helpful comments.

Disclaimer The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

References

[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.

[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.

[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.

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

[LR01] Alexandros Labrinidis and Nick Roussopoulos. "Update Propagation Strategies for Improving the Quality of Data on the Web". Submitted for publication, 2001.

[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.