On the Materialization of WebViews*

Alexandros Labrinidis
labrinid@cs.umd.edu

Nick Roussopoulos†
nick@cs.umd.edu

Department of Computer Science and Institute for Systems Research,
University of Maryland, College Park, MD 20742

EXTENDED ABSTRACT

Abstract

A *WebView* is a web page that is automatically created from base data, which are usually drawn from a DBMS. A WebView can be either *materialized* as an html page at the web server, or *virtual*, always being computed on-the-fly. For the materialized case, updates to base data lead to immediate recomputation of the WebView, whereas in the virtual case, recomputation is done on demand with each request. We introduce the *materialize on-demand* approach which combines the two strategies, and generates WebViews on demand, but also stores the results and re-uses them in the future if possible. Deciding on one of the three materialization policies for each WebView is clearly a performance issue. In this paper, we give the framework for the problem and provide a cost model, which we test with experiments on a real web server.

1 Introduction

The web is increasingly being used as the means to do everyday tasks, from reading the newspaper to shopping or paying bills. One common denominator for all these activities is that the corresponding web sites provide some sort of *personalization*, tailored to the style and needs of each individual ([B$^+$98]). Personalized web pages, that are automatically created from base data, are one of the many instances of WebViews. In general, we define *WebViews* as web pages that are automatically constructed from "base data" using a program or a DBMS query.

Similarly to traditional database views, WebViews can be in two forms: *virtual* or *materialized*. Virtual WebViews are computed dynamically on-demand, usually by a CGI script, whereas materialized WebViews are pre-computed and stored as static HTML pages. In the virtual case, the cost to compute the WebView increases the query response time[1]. On the other hand, in the materialized case, every update to the base data leads to a WebView refresh, which increases the server load.

Having a WebView materialized can potentially give significantly lower query response times, provided that the update workload is not heavy. Even if the WebView computation is not very expensive, by keeping it materialized we eliminate the latency of going to the DBMS every time, which could lead to DBMS overloading ([Sin98]). However, if the update workload is heavy, having the WebView materialized can lead to a degradation in performance, as every update will cause a refresh. In this case, *deferring* the updates until the time of query ([RK86]) is the best solution. Clearly, the decision whether to have a WebView materialized or virtual at the server, the *WebView materialization problem*, is a performance issue.

WebView materialization is different from traditional *web caching*: WebView materialization aims at eliminating the processing time needed for repeated generation, whereas web caching strives to eliminate unnecessary data transmissions across the network ([Mal98]). Also, WebView materialization is performed at the web server, whereas web caching is done at the clients or at proxies. However, although different, both techniques improve web server performance.

The WebView materialization problem is similar to that of deciding which views to materialize in a data warehouse ([GM95, Gup97, Rou98]), known as the *view selection problem*. There are however many differences. First of all, although both problems aim at decreasing query response times, warehouse views are materialized in order to speed up the execution of a few & long analytical (OLAP) queries, whereas WebViews are materialized to avoid re-

*Prepared through collaborative participation in the Advanced Telecommunications/Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0002. Also supported in part by DOD Lucite Contract CG9815.

†Also with the Institute for Advanced Computer Studies, University of Maryland.

[1]We use the term *queries* to refer to web page requests for a particular WebView.

peated execution of many small OLTP-style queries. Secondly, since WebViews are defined after user requests, unlike warehouse views, the resulting search space for the decision problem is significantly smaller. Moreover, we have accurate statistics on the access and update frequencies for all the WebViews from the web server logs. Thirdly, WebView materialization means avoiding an extra layer of software (i.e. generating the WebView by executing a program or a DBMS query), whereas in the warehouse case one always has to issue a query to the DBMS. Finally, the general case of the WebView materialization problem has no constraints, whereas most view selection algorithms impose some resource constraints (e.g. space requirement or maintenance time allowed [KR99]).

In the web context, although there is a lot of recent literature on building & maintaining web sites ([AMM98, FFK$^+$98]), on querying the web ([MMM97, FLM98, GW98]) and on integrating heterogeneous data sources ([CDSS98, MZ98]), there is very little work on the performance issues associated with materializing WebViews. [MMM98] provide an algorithm to support client-side materialization of WebViews, and [Sin98], [AMR$^+$98] present algorithms to maintain them incrementally. However, to the best of our knowledge, this is the first attempt to provide a quantitative solution to the problem of deciding the best materialization strategy for WebViews.

In this paper we give the framework for the WebView materialization problem, and also propose a hybrid approach that combines the advantages of both the virtual and the materialized approaches (Section 2). We also present an approximate cost model for the WebView materialization problem in Section 3. Finally, we present the results of our experiments on a real web server in Section 4 and our conclusions in the last section.

2 WebView Materialization Problem

When a WebView is materialized, any update to the base data leads to an *immediate* refresh of the derived WebView (in addition to the update to the underlying DBMS). The refresh can either be incremental or a complete recomputation[2]. Requests for such a WebView, however, are very fast, since they are pre-computed. On the other hand, virtual WebViews are always generated on-the-fly. This means that updates to the base data are only applied to the DBMS, but queries have to wait for the WebView to be recomputed every time. Clearly, both of these approaches can cause significant performance degradation if not used properly (e.g. if materializing a WebView that has a lot of updates and very few requests).

There is another, hybrid alternative: generate the WebView *on demand* (like the virtual approach), but also store the results and re-use them in the future (like the materialized approach), if possible. We call this approach *materialize on-demand*. Under this strategy, an update to the base data must invalidate the derived WebView (but it will not cause a refresh). When the server gets a request for the WebView, it first checks whether it has been invalidated and, if not, simply returns the saved copy. If the view has been invalidated since the last time it was saved, then the server needs to generate the WebView and save it again.

We formulate the *WebView materialization problem* as:

> *For every WebView at the server, select the materialization strategy (virtual, materialized, materialized on-demand) for* **minimizing the average query response time** *on the clients. We assume that there is no storage constraint at the server.*

The assumption that there is no storage constraint on the server is not unrealistic, since, in our case, storage means disk space (and not main memory) and also WebViews are expected to be relatively small[3]. In this paper, we also assume a no staleness requirement, i.e. the WebViews must always be up to date. This is a reasonable requirement, since users would rather access fresh data.

Clearly the WebView materialization decision is heavily dependent on the update and access patterns for the WebViews, whereas the calculation cost and the size could also play some role. There are some classes of WebViews for which a straightforward solution to the materialization problem exists. For example, WebViews with a lot of requests which do not get a lot of updates should be materialized, since keeping them up-to-date "pays off" because of the high access rate. An example for this scenario are the web pages in Yahoo (`http://www.yahoo.com`) which don't get many updates and are thus kept as HTML pages. On the other hand, WebViews with a lot of updates and infrequent access should be virtual, since the overhead of keeping them fresh is not warranted by the number of requests. An example for this case is a personalized stock portfolio page from a web site offering real-time stock market data. Since the update frequency is very high (stock prices can change many times in a second), the corresponding WebView would have to be virtual and generated on-demand using CGI scripts.

Although some classes of WebViews have straightforward solutions to the materialization problem, this is not the case in general. To find an analytical solution to the WebView materialization problem, we have developed a cost model, which we present in the next section.

[2]Since the materialized WebView is in html format, it is difficult to do an incremental refresh, although not impossible. For the remainder of the paper, we assume that a complete recomputation is taking place after every update to the base data.

[3]The average web page is 30KB ([AW97]), so a single 50GB hard disk for example could hold approximately 1.5 million pages.

3 Cost model

We want to compare the three different materialization policies (materialized, virtual, materialized on-demand) for a WebView v_i and decide which one will lead to smaller query response times under given workload conditions. We calculate the cost to the server under each materialization policy for v_i, however we make the distinction between *update cost*, C^U, which is load on the server because of the application of the updates, and the *access cost*, C^A, which is load on the server because of the user requests for v_i. We expected and verified in the experiments that the query response times are going to be more "sensitive" to the access cost, since it is in the "critical path" of each request. Let $f_a(v_i)$ be the *WebView access frequency* given in some unit of time, say minutes, and $f_u(v_i)$ be the *WebView update frequency*. $f_a(v_i)$ includes requests to WebView v_i from all clients. Finally, let the cost to recompute v_i be $c_{gen}(v_i)$.

Materialized Policy If v_i is kept materialized, then the cumulative update cost is

$$C^U_{mat}(v_i) = f_u(v_i) \times (c_{gen}(v_i) + c_w(v_i)) \tag{1}$$

where $c_w(v_i)$ is the cost to write v_i to disk[4].

The cumulative access cost, if v_i is kept materialized is:

$$C^A_{mat}(v_i) = f_a(v_i) \times c_r(v_i) \tag{2}$$

where $c_r(v_i)$ is the cost to read v_i from disk[4].

Since we wish to minimize the average query response time, we must give more weight to the access cost than the update cost, because the access cost has a *direct* effect on the response time, whereas the update cost has only an *indirect* effect (by increasing the server load). This asymmetry is due to the fact that a request for a web page can be serviced while an update on the same page takes place, in other words there is no locking or blocking on typical web servers.

Following this idea, to get the overall cost for keeping v_i materialized, we introduce a weight factor for the access cost, $\alpha \geq 1$, which is expected to be platform-dependent. The total cost is:

$$C_{mat}(v_i) = C^U_{mat}(v_i) + \alpha \times C^A_{mat}(v_i) \tag{3}$$

Virtual Policy In contrast to the materialized strategy, if v_i is kept virtual, there will be no update cost whatsoever[5]. Therefore

$$C^U_{virt}(v_i) = 0 \tag{4}$$

On the other hand, the cumulative access cost if v_i is kept virtual is:

$$C^A_{virt}(v_i) = f_a(v_i) \times c_{gen}(v_i) \tag{5}$$

where $c_{gen}(v_i)$ is the cost to recompute v_i, and, in this case, it is "suffered" by every query.

Like in the materialized case, we have to use α when calculating the total cost for the virtual policy:

$$C_{virt}(v_i) = C^U_{virt}(v_i) + \alpha \times C^A_{virt}(v_i) \tag{6}$$

Materialized On-Demand Policy If v_i is kept materialized on-demand, then the cumulative update cost is only the invalidation cost:

$$C^U_{mod}(v_i) = f_u(v_i) \times c_{inv} \tag{7}$$

where c_{inv} is the cost to invalidate one WebView.

The access cost of v_i under a materialized on-demand policy, has a lookup cost on every request when the server checks to see if the WebView has been invalidated. Furthermore, for every update we also have to include the WebView generation cost plus a small cost to save the WebView to disk. When the access frequency is higher than the update frequency for v_i, we expect to have better performance compared to the virtual strategy, since in that case we only pay the cost of reading the saved WebView from disk for the extra accesses, instead of recomputing it all the time. Here is the upper bound[6] for the access cost to v_i under a materialized on-demand policy:

$$\begin{aligned} C^A_{mod}(v_i) \leq\; & f_a(v_i) \times c_{chk} \\ +\; & f_u(v_i) \times (c_{gen}(v_i) + c_w(v_i)) \\ +\; & b \times (f_a(v_i) - f_u(v_i)) \times c_r(v_i) \end{aligned} \tag{8}$$

where c_{chk} is the cost to check if one WebView has been invalidated, and b is 1 if $f_a(v_i) > f_u(v_i)$, and 0 otherwise.

Finally, the total cost for the materialized on-demand policy is:

$$C_{mod}(v_i) = C^U_{mod}(v_i) + \alpha \times C^A_{mod}(v_i) \tag{9}$$

WebView Materialization Problem Let V be the set of WebViews in our system and let V_{mat} be the subset of WebViews that are materialized, V_{virt} the ones that are virtual, and V_{mod} the ones that are materialized on-demand.

[4]For simplicity one could assume that all WebViews have approximately the same size, kept constant in the presence of updates, and hence the cost to read or write a WebView to disk would be the same for all.

[5]We do not take into account the cost to update the base data, since this will be the same with all three materialization policies.

[6]We cannot calculate the exact cost, as it depends on the interleaving of updates and accesses to v_i.

The total cost would be:

$$\begin{aligned} C_{total} &= \sum_{v_i \in V_{mat}} C_{mat}(v_i) \\ &+ \sum_{v_j \in V_{virt}} C_{virt}(v_j) \\ &+ \sum_{v_k \in V_{mod}} C_{mod}(v_k) \end{aligned} \quad (10)$$

With the help of Equations 1 - 10 we can rephrase the WebView materialization problem as: *partition V into V_{mat}, V_{virt}, V_{mod}, such that C_{total} is minimized.*

By default, web servers log all page requests, so, estimating $f_a(v_i)$ for any WebView v_i is not difficult. Calculating $f_u(v_i)$ and the rest of the costs is easy too, and we can assume that $c_r(v_i)$, $c_w(v_i)$ will be approximately the same for all WebViews to make things even easier.

To get some intuition behind the formulas, we assume that all costs, except for $c_{gen}(v_i)$ and $c_{gen}(v_i)$, are small and constant. Under this assumption, we can get some *approximations* for $C_{mat}(v_i)$, $C_{virt}(v_i)$, $C_{mod}(v_i)$:

$$\begin{aligned} C'_{mat}(v_i) &= f_u(v_i) \times c_{gen}(v_i) \\ C'_{virt}(v_i) &= \alpha \times f_a(v_i) \times c_{gen}(v_i) \\ C'_{mod}(v_i) &= \alpha \times f_m(v_i) \times c_{gen}(v_i) + c_{over} \end{aligned} \quad (11)$$

where $f_m(v_i)$ is $\min(f_a(v_i), f_u(v_i))$, and c_{over}, is a composite cost to reflect the extra disk I/O that the materialize on-demand approach has to make. We expect that the materialize on-demand approach will give better performance than the classic virtual strategy, in cases where $f_a(v_i) > f_u(v_i)$. Furthermore, we can see that in order to choose between the materialize and materialize on-demand policies, we should consult the weighted ratio of accesses to updates, $\lambda = \frac{\alpha \times f_a(v_i)}{f_u(v_i)}$. A ratio $\lambda > 1$ would suggest that the materialized approach is better, otherwise a materialized on-demand or a virtual strategy would be expected to yield better performance.

4 Experiments

For our experiments we used two machines, a SUN UltraSparc-5 with 320MB of memory, running Solaris 2.6 and an AlphaStation 255 with 64MB of memory, running Digital Unix V4.0. The web server, Apache version 1.3.6 (`http://www.apache.org`), ran concurrently with the update process on the SUN machine, while the clients were running on the Alpha. All machines were on the same local area network in order to eliminate (uncontrollable) network latency from our experiments. In every experiment, each client would read a set of queries from a script, send the requests to the web server and wait for the reply, measuring the elapsed time for each query (averaged over multiple runs).

Workload Our workload consisted of 100 WebViews. Their access rates followed the Zipf distribution with a theta of 0.7, as suggested in [BCF+99]. The total accesses to the web server averaged to about 12 requests per second. This should correspond to a quite heavy load on the server, of about 1 million hits per day. For comparison, our departmental web server (`http://www.cs.umd.edu`) gets about 70,000 requests a day which correspond to only about 0.8 requests per second.

While the access rate for each WebView was kept the same for all experiments, we varied the update rate and the materialization policy for 10 out of the 100 WebViews, our *test group*. The remaining 90 WebViews had no updates at all, were always materialized and played the role of a "background" load to the server. The sizes for the WebViews were on average 30KB ([AW97]) and the calculation cost was rather small, 0.5 seconds.

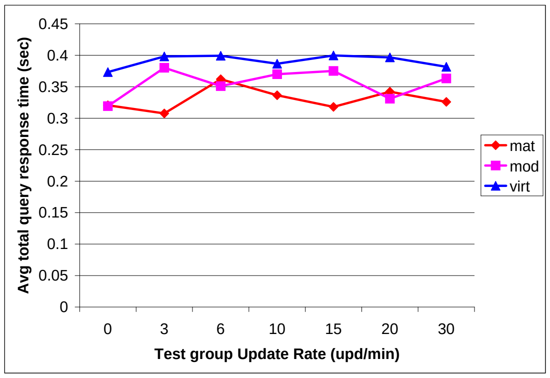

Figure 1: Total avg query response time

Experiments The test group in our experiments consisted of 10 WebViews in the "middle" of the access rate distribution (with access rates about 3-4 requests per minute). We varied the update rate from 0 up to 30 updates per minute for each of the 10 WebViews. For each update rate, we ran three different experiments, one for each materialization policy.

We plot the average query response time for all queries (including the ones in the test group) in Fig. 1. The x-axis is the update rate for the test group, in updates/min, whereas the y-axis is the total average query response time, in seconds. The `mat` line corresponds to the case where all WebViews from the test group are kept materialized (i.e. the updates cause all WebViews to be refreshed in the background), the `virt` line to the virtual case (i.e. the query result is recomputed on every request) and the `mod` line corresponds to the case where all WebViews in the test group are materialized on-demand (i.e. the query result is recomputed on request, but also saved for future use and updates invalidate the saved copy).

We see from Fig. 1 that in the virtual case, the over-

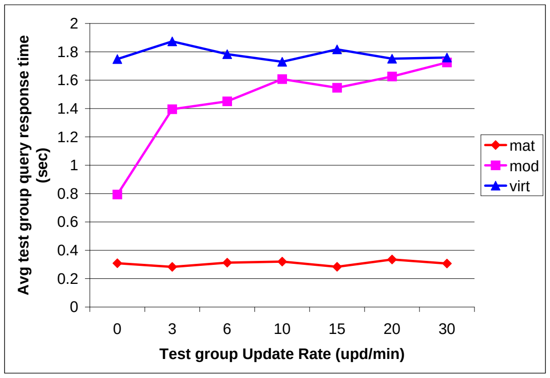

Figure 2: Test group avg query response time

all performance is not affected by the update rate (i.e. the `virt` line is almost straight), as it was expected. On the other hand, the materialize on-demand policy, depending on the interleaving of updates and requests, can have better performance over the virtual approach, since it re-uses pre-computed results as much as possible, whereas the virtual approach blindly recomputes each WebView on every request. Furthermore, if we look at the average query response time for only the WebViews in the test group (Fig 2), we verify that the materialize on-demand policy outperforms the virtual strategy when the update rate is less than the access rate (first two points in the graph).

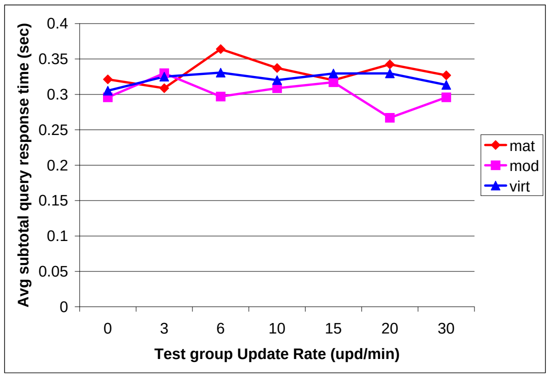

Figure 3: (All$-$Test group) avg query response time

From Figures 1 and 2 we see that the materialized policy performs really well, even for update rates far exceeding the access rate, although, eventually, the virtual & the materialized on-demand strategies are expected to perform better for very high update rates. So where do the savings for the materialized strategy come from? We plot in Fig. 3, the average query response time of all queries **except** for those in the test group. From Fig. 3, we see that the reason for this great performance is that the materialized policy "penalizes" the rest of the views, by slightly increasing their query response times (since the updates done at the background increase the load at the server). On the other hand, the performance of the rest of the views is almost not affected with the materialized on-demand and virtual approaches, since the cost of the updates is inflicted on the query response time of the updated WebView.

5 Conclusions

In this paper, we have introduced the materialize on-demand policy for WebViews, that combines the materialized and virtual strategies. We also formulated the WebView materialization problem, and described a cost model to help decide among the three materialization strategies (materialized, virtual, materialized on-demand). Our experiments showed that the materialized policy usually leads to better performance, at the expense, however, of the other WebViews. On the other hand, if the update rate is really high compared to the access rate, the virtual and materialized on-demand strategies have better overall performance than the materialized policy, since they defer the updates till the time of the query. Finally, the materialized on-demand strategy outperforms the virtual policy when the access rate is higher than the update rate, since it avoids recomputation of the WebView when there are no updates.

We are currently implementing the materialized on-demand policy for the web server of the *AMASE* project (`http://amase.gsfc.nasa.gov`). As part of our future work, we want to drive our experiments with trace data from a commercial web server.

Acknowledgements We would like to thank Kostas Stathatos and Ugur Cetintemel for their suggestions, and the reviewers for their useful comments.

References

[AMM98] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. “Design and Maintenance of Data-Intensive Web Sites”. In *Proc. of the 6th International Conference on Extending Database Technology (EDBT’98)*, pages 436–450, Valencia, Spain, March 1998.

[AMR+98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. “Incremental Maintenance for Materialized Views over Semistructured Data”. In *Proc. of the 24th VLDB Conference*, pages 38–49, New York City, USA, August 1998.

[AW97] Martin F. Arlitt and Carey Williamson. “Internet Web Servers: Workload Characterization and Performance Implications”. *IEEE/ACM Transactions on Networking*, 5(5), October 1997.

[B+98] Phil Bernstein et al. “The Asilomar Report on Database Research”. *SIGMOD Record*, 27(4), Dec. 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 INFOCOMM’99*, New York, USA, March 1999.

[CDSS98] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. “Your Mediators Need Data Conversion!”. In *Proc. of the ACM SIGMOD Conference*, Seattle, Washington, June 1998.

[FFK+98] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. “Catching the Boat with Strudel: Experiences with a Web-Site Management System”. In *Proc. of the ACM SIGMOD Conference*, pages 414–425, Seattle, Washington, June 1998.

[FLM98] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. “Database Techniques for the World-Wide Web: A Survey”. *SIGMOD Record*, 27(3):59–74, September 1998.

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

[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.

[GW98] Roy Goldman and Jennifer Widom. “Interactive Query and Search in Semistructured Databases”. In *Proc. of the 1st International Workshop on the Web and Databases*, Valencia, Spain, March 1998.

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

[Mal98] Susan Malaika. “Resistance is Futile: The Web Will Assimilate Your Database”. *Data Engineering Bulletin*, 21(2):4–13, June 1998.

[MMM97] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. “Querying the World Wide Web”. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[MMM98] Giansalvatore Mecca, Alberto O. Mendelzon, and Paolo Merialdo. “Efficient Queries over Web Views”. In *Proc. of the 6th International Conference on Extending Database Technology (EDBT’98)*, pages 72–86, Valencia, Spain, March 1998.

[MZ98] Tova Milo and Sagit Zohar. “Using Schema Matching to Simplify Heterogeneous Data Translation”. In *Proc. of the 24th VLDB Conference*, pages 122–133, New York City, USA, August 1998.

[RK86] Nick Roussopoulos and Hyunchul Kang. “Principles and Techniques in the Design of ADMS $\pm$”. *Computer*, pages 19–25, December 1986.

[Rou98] Nick Roussopoulos. “Materialized Views and Data Warehouses”. *SIGMOD Record*, 27(1), March 1998.

[Sin98] Giuseppe Sindoni. “Incremental Maintenance of Hypertext Views”. In *Proc. of the 1st International Workshop on the Web and Databases*, Valencia, Spain, March 1998.