Update Propagation Strategies for Improving the Quality of Data on the Web*

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

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

Abstract

Dynamically generated web pages are ubiquitous today but their high demand for resources creates a huge scalability problem at the servers. Traditional web caching is not able to solve this problem since it cannot provide any guarantees as to the freshness of the cached data. A robust solution to the problem is web materialization, where pages are cached at the web server and constantly updated in the background, resulting in fresh data accesses on cache hits. In this work, we define Quality of Data metrics to evaluate how fresh the data served to the users is. We then focus on the update scheduling problem: given a set of views that are materialized, find the best order to refresh them, in the presence of continuous updates, so that the overall Quality of Data (QoD) is maximized. We present a QoD-aware Update Scheduling algorithm that is adaptive and tolerant to surges in the incoming update stream. We performed extensive experiments using real traces and synthetic ones, which show that our algorithm consistently outperforms FIFO scheduling by up to two orders of magnitude.

1 Introduction

The World Wide Web has seen tremendous growth in the years since its inception, accompanied by a big transformation in its nature, from purely static HTML pages in the early 90s to most web pages having some dynamic content today. Online services, frequently updated content and personalization[BBC+98] are the main reasons behind dynamically generated web pages. Unfortunately, dynamic content requires far greater resources from web servers than static pages do and does not scale.

Although web caching has addressed the scalability problem for static pages, it cannot be directly applied to dynamically generated pages, since it does not deal with continuously changing data and, therefore, cannot provide any guarantees for the freshness of the cached data. Web caching also helps in serving user requests fast. This is certainly important, but only if the data is fresh and correct, otherwise it may be more harmful than slow or even no data service. For example, a lightning fast web site with 20 minute delayed stock information is of very little use to those investors who want to know what the market is doing right now. A slightly slower site that can display up to the second stock information would be more valuable.

In general, when evaluating the quality of a web server, one must evaluate both its *Quality of Service* (QoS), or how fast it services user requests, and, its *Quality of Data* (QoD), or how "good" the served data are. Goodness of data can be measured in freshness, accuracy, and other metrics that need to be defined from the semantics of the application. It is unfortunate that most web servers do not provide to the users any means of knowing about the QoD they serve, how fresh the data is, or the reliability of the sources. It is prudent to include in the business model the QoD guarantees, especially for those web sites whose sole or primary business is serving data.

In [LR99, LR00a] we have showed that *materialization* of dynamically generated web pages is a robust solution to the scalability problem. With materialization, web pages are cached and constantly kept up to date in the background, resulting in fresh data accesses on cache hits. We use the term *WebView* to refer to the unit of materialization, which is a page that contains dynamic *HTML fragments* generated from a DBMS. Having a WebView materialized can potentially give significantly lower service times, compared to a virtual (un-materialized) WebView. Although the selection

* 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.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

of WebViews to materialize will have important implications on both the QoS and QoD, the order by which materialized WebViews are refreshed plays an even more crucial role in the overall Quality of Data. For example, we want to update popular WebViews first, since, we expect that overall they will contribute to higher freshness of the data served.

In this paper we focus on the *update scheduling problem* as it relates to QoD: given a set of WebViews that are materialized, find the best order to refresh them, so that the overall Quality of Data is maximized. Our work is motivated by materialized WebViews in data & update-intensive web servers, but it can be applied to any environment that has continuous online updates. We demonstrate that a FIFO schedule for the WebView updates can have disastrous effects on QoD. Except for ignoring the popularity of the WebViews, a FIFO schedule also ignores the cost to update each WebView. Scheduling the refresh of "cheaper" WebViews ahead of "expensive" ones, could also lead to higher QoD.

We performed a workload study on Quote.com, a popular update-intensive web server with online stock information. We found that both access and update workloads are highly skewed, with a small percentage of the stocks being responsible for a big percentage of the accesses and the updates. Moreover, we found that access and update patterns are correlated. The results of this study were used in the release of the current Quote.com server.

Based on the workload analysis, we developed an adaptive QoD-aware update scheduling algorithm (QoDA) that takes into consideration the popularity and the update cost of the views. Our algorithm unifies the scheduling of relation and view updates under one framework, takes advantage of temporal locality in the incoming update stream and is tolerant to update surges. QoDA also takes into account the database schema and can support any type of views and arbitrary view hierarchies.

We implemented an update scheduling simulator and ran extensive experiments using real traces (from Quote.com and the NYSE) and synthetic ones. Our experiments clearly show that QoDA update schedules consistently outperform FIFO schedules by up to two orders of magnitude. One of the main advantages of QoDA scheduling is that it can maintain a high level of QoD, even when the update processing capacity is not enough or when there are surges in the incoming update rate. The most important discriminator of QoDA over FIFO is the speed of restoring QoD after update surges. QoDA rapidly restores QoD while FIFO does it slowly and sometimes never recovers.

In the next section we summarize the results from a workload study that we recently performed on Quote.com. In Section 3 we define the QoD metrics, and in Section 4 we present the QoD-aware update scheduling algorithm. Section 5 contains the experiments we performed using real and synthetic trace data. Section 6 briefly presents the related work. We conclude in Section 7.

2 A Workload Study of a Web Server with Continuous Updates

We recently performed a workload study on Quote.com [LR00b], one of the most popular stock quote servers. We focused our study on the stock information pages of about 9000 stocks and used the server log traces from Quote.com to explore the access workload. We then correlated these logs with the Trade and Quote Database from the New York Stock Exchange (NYSE), which contains all the stock "ticks" (buy or sell activity), i.e. the update logs.

Accesses			**Updates**	
number of symbols		% of total requests	number of symbols	
1		15%	10	
2		25%	25	
10		40%	81	(0.9%)
25		50%	153	(1.7%)
70	(0.8%)	60%	287	(3.1%)
190	(2.1%)	70%	529	(5.7%)
442	(4.8%)	80%	963	(10.6%)
1081	(11.8%)	90%	1833	(20.0%)

Table 1: Number of symbols vs total request load

We found that, as is the case with static web pages [BCF+99], the access request workload of dynamically generated pages is highly skewed: a small percentage of the web pages is responsible for a big percentage of the overall request load (Table 1). For example, the 25 most popular stock symbols generate over 50% of the total request load for Quote.com. The update workload is also highly skewed, although not as much as the access workload. For example, the ten most update-intensive stocks receive 15% of the updates (Table 1). The highly skewed access and update workloads mandate the use of popularity when measuring the QoD for the accessed data and also when scheduling the updates.

We looked at the access rates for the Quote.com web server and the update rates from the NYSE trace. In addition to the big, but predictable, variations at the beginning or end of each day's session, there is significant variance in the access and update rates during market hours. WebView Materialization inherently deals with access surges. However, for update surges, it is the update scheduling algorithm that must tolerate them and rapidly adapt to the incoming update rates. For example, the NYSE workload had update rates of up to 696 updates per second. Furthermore, since access rate variations can lead to increased load in the system, the update scheduling algorithm must also tolerate decreased server capacity. In general, we need an *adaptive* algorithm that will react rapidly to changes in the update workloads and system conditions.

We found that sudden increases in the incoming update rate, or *update surges*, are frequent in the update workload. We modeled "sudden" to be within a ten-second sliding window, during which we compute the maximum positive update rate difference and report it as a percentage increase.

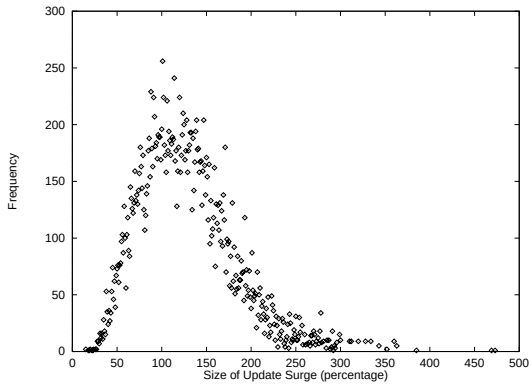


Figure 1: Surges in Update Rate for April 4, 2000

Therefore a 200% percentage increase would correspond to a three-fold update surge. We measured the intensity and the frequency of update surges in the NYSE workload and plot them in Figure 1. Clearly, the peak of the curve is around the 100% mark, which suggests that two-fold update surges are the most common. However, there are also a lot of cases with higher update surges, up to 500% or six-fold.

Finally, we found that both the update stream exhibits *temporal locality*: recently updated views are more likely to be updated again. This suggests that the update scheduling algorithm must exploit it and attempt to "merge" consecutive updates to the same WebView.

3 Quality of Data for Materialized Views

In this section we present a probabilistic model for measuring the Quality of cached data that are derived from a DBMS. We assume a web server architecture similar to that of Figure 2. 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* [LR00a] and constantly being refreshed in the background. A separate module, the *update scheduler* is responsible for scheduling the DBMS updates and the refresh of the data in the asynchronous cache. Updates must be performed *online* and our goal is to serve content with as high QoD as possible to the users. The system architecture in Figure 2 implies that the updates arrive at the back-end and that the requests at the web server are read-only. In general, this is not a requirement for our work and we do allow updates to originate at the web server, provided that there are no consistency issues and the updates still go through the update scheduling module before being applied to the DBMS.

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

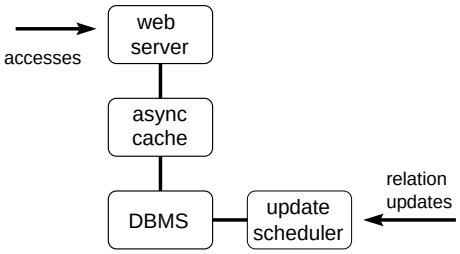


Figure 2: System Architecture

derived from relations or from other, previously derived views. There is no restriction on the types of views or their complexity. We distinguish two types of views: *virtual*, which are generated on demand from relations or other views, and *materialized*, which are precomputed, stored in the asynchronous cache and refreshed asynchronously. All user requests are expressed as view accesses, whereas all incoming updates are applied to relations only and schedule view refreshes. Finally, we assume that incoming relation updates must be performed in the order received, whereas materialized view refreshes can be performed in any order.

3.1 Data Freshness

The *incoming update stream* contains relation updates which modify one or more tuples. The *update schedule* lists the order in which the relation updates along with materialized view refreshes are to be performed. A *valid update schedule* in our framework must have the following three properties: (1) relation updates or view refreshes cannot overlap, (2) all relation updates must be performed in the order of arrival, and, (3) stale materialized views must be refreshed.

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. Note that we start counting staleness at the earliest possible point to bring the QoD staleness metric as close as possible to the time of the originating source of the update. We illustrate this definition with the following 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 3). 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. Virtual view v_v will become fresh after its parent relations/views are updated (r in this example). Since relation r was updated at time t_3, view v_v inherits its staleness from r, and thus 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. Clearly, the total time that relation r and views v_v & v_m are stale will be minimized if there is no "wait" time, i.e. when $t_1 = t_2$ and $t_3 = t_4$.

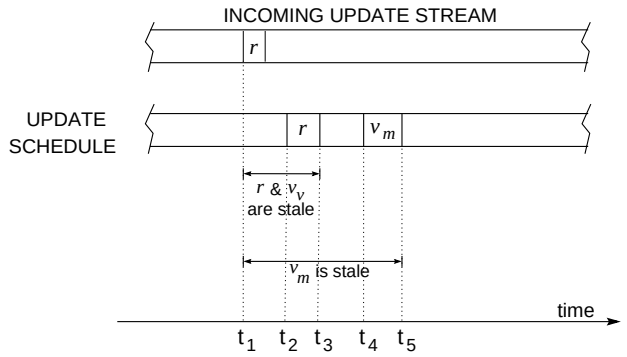


Figure 3: Staleness Example

A database object d_i is considered to be *fresh*, when it is not *stale*. We define the *freshness function* for object d_i, $b_{fresh}(d_i)^t$, as following:

$$b_{fresh}(d_i)^t = \begin{cases} 0, & \text{if object } d_i \text{ is stale at time } t \\ 1, & \text{if } d_i \text{ is not stale at time } t \end{cases} \quad (1)$$

The definition implies a boolean treatment of staleness: data objects are marked as stale because of at least one unapplied update. In other words, if multiple consecutive relation updates render a materialized view v_m stale, view v_m will be fresh only after the last refresh is performed, even if we refresh v_m multiple times. This may penalize materialized views affected by frequently-updated relations, but also gives the opportunity to perform other relation updates or view refreshes instead.

If we want to measure the freshness of a database object d_i over an entire observation time period $T = [t_i, t_j]$, we have that

$$b_{fresh}(d_i)^T = b_{fresh}(d_i)^{[t_i,t_j]} = \frac{1}{T} \times \int_{t_i}^{t_j} b_{fresh}(d_i)^t \quad (2)$$

This definition is equivalent to computing the percentage of time during the observation period that the database object d_i is stale. Since we are mostly interested in continuous update streams, T is expected to be a sliding time window which ends at the current point of time.

So far the definitions of freshness for database objects did not consider accesses to them. In order to measure the quality of the accessed data, we need a "normalized" metric that will account for the probability of accessing a fresh version of a view.

Definition 1 *We define the **Freshness Probability** for a view v, $p_{\text{fresh}}(v)$, as the probability of accessing a fresh version of view v during the observation interval T.*

If we assume a uniform probability of accessing view v during the observation interval, then the probability of accessing a fresh version of v is equal to the percentage of time that the view is fresh, or the freshness as was defined in Eq. 2. In other words,

$$p_{fresh}(v) = b_{fresh}(v)^T = \frac{1}{T} \times \int_{t_i}^{t_j} b_{fresh}(d_i)^t \quad (3)$$

The higher the values for $p_{fresh}(v)$ the higher the quality of the accessed data.

3.2 Overall QoD based on Freshness

Although the freshness probability for a given view v_i, $p_{fresh}(v_i)$, is an accurate QoD metric for that view, we need to be able to aggregate QoD over the entire database. In other words we want to estimate the *probability that a database access returns fresh data*, which we will denote as $p_{fresh}(db)$.

In order to calculate the aggregate QoD over the entire database, $p_{fresh}(db)$, we could just add the freshness probabilities for all views. However, since views are accessed with different frequencies and web workloads exhibit highly skewed access patterns (Section 2), we must take into account the access frequency of each view when aggregating QoD over the entire database. We want the overall QoD to be influenced more by the freshness probabilities of popular views than those of unpopular views.

Let us assume that $f_a(v_i)$ is the access frequency of view v_i, expressing the ratio of v_i requests over the total number of requests. We have that $\sum_{v_i} f_a(v_i) = 1$. We compute the overall QoD as the weighted sum of the freshness probabilities of all views, as follows:

$$p_{fresh}(db) = \sum_{v_i \in V} f_a(v_i) \times p_{fresh}(v_i) \quad (4)$$

This definition implies that $0 \leq p_{fresh}(db) \leq 1$. We associate QoD with freshness, so the higher the value of $p_{fresh}(db)$ the better the overall QoD.

4 Update Scheduling

Given a database schema, the set of views that are materialized and an incoming relation update stream, the ***Update Scheduling Problem*** consists of determining the update schedule which maximizes the overall Quality of Data (QoD). We assume a database with n relations, r_1, r_2, . . ., r_n and m views, v_1, v_2, . . ., v_m. We use a directed acyclic graph, the *View Dependency Graph*, to represent derivation paths for the views. Views can be of arbitrary complexity, including SPJ-views, aggregation views, etc. The nodes of the view dependency graph correspond to either relations or views, and are marked to distinguish between virtual and materialized views. Nodes that have zero in-degree correspond to relations. An edge from node a to node b exists only if node b is derived directly from node a. No other views can be derived from virtual views. Finally, we assume that for each relation we know the cost to update it and for each materialized view the cost to refresh it. We do not actually need the real update costs for relations and materialized views, but rather the relative update costs.

The incoming update stream contains relation updates that trigger materialized view refreshes. The definition of valid update schedules on page 3 implies that we can (1) delay relation updates, as long as we perform all of them and in the order received, (2) postpone materialized view

refreshes and not necessarily perform them immediately after they were triggered, and (3) reorder materialized view refreshes.

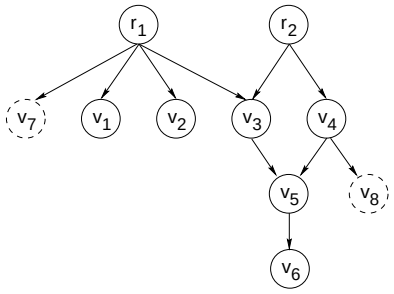


Figure 4: View Dependency Graph for the Motivating Example

4.1 Motivating Example

Let us assume a database with two relations r_1, r_2 and eight views v_1, ..., v_8. Views v_1 through v_6 are materialized, whereas views v_7 and v_8 are virtual. Figure 4 displays the view dependency graph for this example. Table 2 has the access frequencies for this example (which follow the Zipf distribution) and the cost to update each relation or materialized view. For simplicity, we will assume that all update and refresh operations take one time unit except for views v_2 and v_3. Finally in this example, we only have two updates, one for r_1 that arrives at time 0, and one for r_2 that arrives at time 3.

object	$f_a()$	cost	type
r_1	0	1	relation
r_2	0	1	relation
v_1	0.12	1	mat. view
v_2	0.37	2	mat. view
v_3	0.19	3	mat. view
v_4	0.09	1	mat. view
v_5	0.07	1	mat. view
v_6	0.06	1	mat. view
v_7	0.05	-	virtual view
v_8	0.05	-	virtual view

Table 2: Access Frequencies

Under a FIFO update propagation schedule, we should perform the refresh of all the affected views right after the update to the parent relation is completed. When we have multiple derivation paths for one view, we must avoid scheduling *unnecessary refreshes*. For example, once we receive an update for r_2, we would rather use schedule r_2 v_3 v_4 v_5, instead of r_2 v_3 v_5 v_4 v_5, thus avoiding refreshing v_5 twice. Our implementation of the FIFO update schedule avoids unnecessary refreshes, by performing a breadth-first traversal of the view dependency graph to compute the refresh order.

Figure 5 has the FIFO update schedule for the motivating example. If we calculate the overall QoD for this schedule

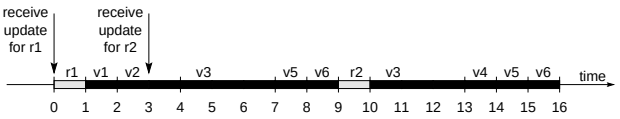


Figure 5: Motivating Example - FIFO Update Schedule

($T = 16$), we have $p_{fresh}(db)$ = 0.513125. We also consider a variation of the FIFO schedule, where we refresh the most popular views first, which we refer to as FIFO Popularity-Aware Schedule. The overall QoD for this schedule, is $p_{fresh}(db)$ = 0.49875, slightly worse than the simple FIFO Schedule. In other words, there are cases when blindly refreshing the most popular views first will not lead to higher QoD.

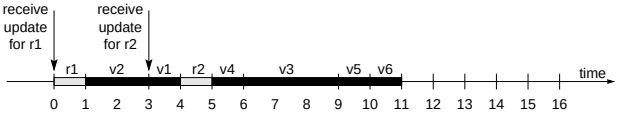


Figure 6: Motivating Example - Optimal Update Schedule

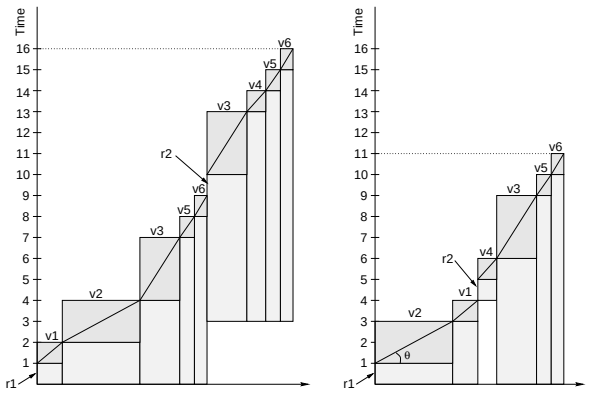
Finally, we consider the optimal off-line update schedule for this example. Assuming that we have the entire incoming update stream in advance, we compute the optimal update schedule by enumerating all possible schedules and finding the one with the highest QoD. Figure 6 has the optimal off-line update schedule for the motivating example. The overall QoD for this schedule ($T = 16$, same as in the FIFO schedules), is $p_{fresh}(db)$ = 0.679375, a 32.4% improvement over the FIFO Schedule.

We see that even with a simple two-update example, there is a lot of room for improvement over the FIFO refresh schedule (more than a 32% QoD gain). As we will demonstrate in the experiments, the scheduling of the updates has a dramatic impact on the overall QoD.

4.2 Visualizing Quality of Data

We use a two-dimensional plot of view staleness to illustrate the difference in the overall Quality of Data among the various refresh schedules. On the X-axis we list all views in the order they appear in the update schedule, spacing them proportionally to their frequency of access (relations have zero frequency of access). On the Y-axis we report the amount of time that each view was stale before it was refreshed (Figure 7). The dark-shaded area for each view (the box with the diagonal) corresponds to staleness because of the cost to refresh or generate the view, whereas the light-shaded area underneath corresponds to view staleness because of scheduling delay. The smaller the overall shaded area is, the less staleness we have and therefore the higher the overall QoD is.

Figure 7a is the QoD visualization of the FIFO schedule (Figure 5) for the motivating example. Figure 7b is the QoD visualization of the optimal update schedule (Figure 6) for the same example and illustrates a smaller shaded area compared to that of Figure 7a, which agrees with the QoD calculation for the two cases that was performed earlier.



(a) FIFO Schedule (b) Optimal

Figure 7: Staleness Visualization for a) the FIFO Update Schedule and b) the Optimal Update Schedule

By observing the QoD visualization of the optimal update schedule (Figure 7b) we see that the views are scheduled by increasing *slope*, where slope is the angle θ between the diagonal of the dark-shaded box for each view and the X-axis. If for each view v_i, we define the *rank* $R(v_i) = \frac{f_a(v_i)}{c(v_i)}$, where $f_a(v_i)$ is the frequency of v_i accesses and $c(v_i)$ is the time required to refresh view v_i, then the optimal update schedule refreshes the views in decreasing rank order (since slope is proportional to $\frac{1}{\text{rank}}$). This is intuitive because in order to minimize the shaded areas in the staleness plot and thus get better QoD, we should choose to schedule for refreshment views that are either popular or cheap to refresh. We examine this idea in detail in the next section.

4.3 QoD-Aware Update Scheduling Algorithm

In this section we present the QoD-Aware update scheduling algorithm, **QoDA** (pronounced *koda*). QoDA unifies the scheduling of relation updates and view refreshes under one framework. We saw in the previous section that in the optimal update schedule views were refreshed by order of rank $R(v_i) = \frac{f_a(v_i)}{c(v_i)}$. This idea cannot be applied directly in our framework for two reasons. First of all, we have a view derivation hierarchy which forces precedence constraints. Secondly, we need to be able to also schedule relation updates in the same way, although $f_a(r_i)$ = 0, for all r_i, since relations are not accessed directly in our framework.

In order to estimate the rank for a relation or a view, we consider its popularity along with the popularities of all the views that are derived from it, either directly or indirectly. With this approach we can assign popularity estimates to relations even though they are not accessed directly.

Definition 2 *The **Popularity Weight** of a relation or view s_i,* pop(s_i)*, is defined as the sum of the access frequencies of s_i and all of its descendants.*

We have that:

$$pop(s_i) = f_a(s_i) + \sum_{v_j \in desc(s_i)} f_a(v_j) \qquad (5)$$

where *desc*(s_i) is the set of descendants of s_i in the view dependency graph. Note that if more than one path exist from s_i to a view v_j, only one instance of v_j is inserted in *desc*(s_i). If s_i is a relation, then $f_a(s_i)$ = 0, whereas when s_i is a virtual view, *desc*$(s_i) = \emptyset$. The popularity weight calculation is similar to that for the LAP Schema [Rou82]. The main difference is that we use a generic view dependency graph which does not differentiate over the type of operation that generates each view (e.g. select, join, union, etc) as is the case for LAP Schemas.

The intuition behind the QoD-Aware update scheduling algorithm is that in order to improve QoD, we schedule to update the relation or materialized view that will have the biggest negative *impact* on QoD, if it is not updated. The impact of delaying the refresh of a relation or view is the modified rank value, as follows: $impact(d) = \frac{pop(d)}{c(d)}$. This implies that between two objects with the same popularity values, QoDA will select the one with the smallest refresh cost, since it will have the biggest impact value. By selecting the database object with the smallest refresh cost, we will be able to "squeeze" in more refresh operations and thus improve the overall QoD.

Dirty Counting

We want to enforce a topological sort order on the view refresh schedule, based on the view dependency graph. In other words we want to guarantee that a materialized view will only be refreshed after all of its parent views or relations that were stale have been updated. Consider for example the view dependency graph of Figure 4. Regardless of the view popularities, it makes no sense to refresh v_5 before v_3, since v_5 will be recomputed from an old version of v_3 and therefore will remain stale. To implement this idea we instrument all views with *dirty counters*, which correspond to the number of stale ancestors each view has. Views are allowed to be refreshed only when their dirty counters reach zero.

```
QoD-aware Update Scheduling algorithm
(1)  candset = ∅
(2)  schedule = ∅
(3)  while (candset != ∅ and ∃ incoming_updates)
(4)       foreach r_i in (incoming_updates)
(5)            candset.append(r_i)
(6)       select d from candset with max impact(d)
(7)       schedule.append(d)
(8)       candset.remove(d)
```

Figure 8: Pseudo-code for the QoDA Algorithm

QoDA Algorithm

In Figure 8 we present the pseudo-code for the QoDA algorithm, which schedules relation updates and view refreshes in order to maximize QoD. The algorithm maintains a set of stale database objects, the *Candidates Set* (candset), and at each step it selects the object with the maximum impact

value (which will have the biggest negative effect on QoD if not scheduled). In order to use dirty counters, the impact value of an object d is calculated using the following formula:

$$\textit{impact}(d) = \begin{cases} \frac{pop(d)}{c(d)}, & \text{if dirty counter for } d = 0 \\ 0, & \text{if dirty counter for } d \neq 0 \end{cases} \quad (6)$$

When an object is appended to the Candidates Set, (Figure 8, line 5), all of its descendants are appended to the Candidates Set, and their dirty counters are updated. Note, that if a node already exists in candset it is not appended, i.e. we allow no duplicates. On the other hand, when an object is removed from the Candidates Set (Figure 8, line 8), the dirty counters of its descendants are decremented. The algorithm terminates when the candidate set is empty and there are no more incoming updates. Note that the implementation of the QoDA update scheduling algorithm can be very fast, adding very little overhead to the system, since it has no time-dependent computation.

Example

In order to illustrate the way QoDA works, we will go through its execution on the motivating example (Section 4.1). The schema for this example is in Figure 4, and the update costs are listed in Table 2. The first update, on r_1, arrives on time $t = 0$, whereas the second update, on r_2, arrives on time $t = 3$. With the arrival of the first update, the following data objects are inserted in the candidates set: r_1 (0), v_1 (1), v_2 (1), v_3 (1), v_5 (2), v_6 (3), v_7 (1), where the numbers in parentheses are the dirty counters for each view. Since only relation r_1 has a zero dirty counter, it has the highest impact value and thus is scheduled to be updated first. After relation r_1 is updated, all the dirty counters of all its descendants are decreased by one. v_2 has the highest impact value from the remaining views, so it is scheduled next. At time $t = 3$, the update on relation r_2 arrives, so r_2, v_4, v_8 are added to the candidate set. The candidate set now has the following items: r_2 (0), v_4 (1), v_5 (3), v_1 (0), v_3 (1), v_6 (4), v_8 (1). With the arrival of the update on r_2, only r_2 and v_1 have zero dirty counters and since impact(r_2) $>$ impact(v_1), r_2 is scheduled to be updated next. The process of updating the dirty counters and picking the element with the highest impact value is repeated until the candidate set is depleted.

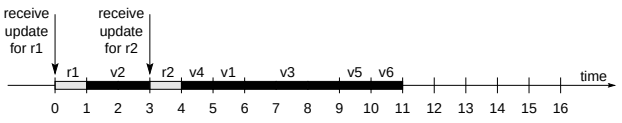


Figure 9: Motivating Example - QoDA Refresh Schedule

Figure 9 has the QoDA update schedule for the motivating example of Section 4.1. The QoD metric for this schedule, is $p_{fresh}(db)$ = 0.673125. This is a 31.2% improvement over the best FIFO schedule (Figure 5) and corresponds to 99% of the QoD for the off-line optimal schedule (Figure 6).

5 Experiments

We implemented a high-level update scheduling simulator which takes as input the database relations, the materialized and virtual views, the access frequencies for the views, the update cost for the relations, the refresh cost for the views and the incoming update stream. The simulator generates the update schedule under the specified algorithm (FIFO or QoDA) and also reports the Quality of Data at each time instant of the simulation. The simulator's internal clock ran at the milli-second level, but we report our findings rounded up to seconds.

An important parameter in the simulator is the *update processing speed* which is the number of updates per second that the simulated system can process. Note that this speed implicitly measures the hardware and software capacity to process updates and respond to queries. In other words, an increase in the access request rate is expected to decrease the update processing speed. When the update processing speed is more than the incoming update rate, we have *extra update capacity* in the system. For example, if we have 1000 updates/sec for the update processing speed and the incoming update rate is 800 updates/sec, then we have extra update capacity of 20%.

Due to space limitations we only present two experiments, the rest can be found in [LR01].

5.1 Experiments with Real Workloads

We used the web logs from the Quote.com server as our access workload and a 10-minute interval of the Trade and Quote Database from NYSE as our update workload. We assumed a database schema where all stock information is stored in one table, and each stock has four materialized views (simple projections and selections on the stock symbol), which have the same refresh cost. The access frequencies were derived from the Quote.com traces. The updates correspond to any buy or sell activity on the stock symbol (which would render the views stale). It should be noted that the NYSE update workload is very intensive (average incoming update rate of 652 updates/second), which dictates an efficient update scheduling algorithm. Also there is great variability in the update rates with values between 40% and 160% of the average rate.

In our experiments we measure the QoD and how it varies during the simulation under the FIFO and QoDA update scheduling algorithms. When the update processing speed of the server is equal to the average incoming update rate, the QoDA schedule consistently gives better QoD over the FIFO schedule, about 6% on average. Furthermore, the QoD under the QoDA schedule is almost constant, whereas in the FIFO schedule the QoD is fluctuating.

In Figure 10 we plot the QoD under the FIFO and QoDA update scheduling algorithms when the processing speed is less than the average incoming update rate. There are two reasons for something like that to happen: (a) because of a surge in the update rate, or (b) because of an increase in the access rate that would increase the load at the server and thus decrease the effective update processing speed. In

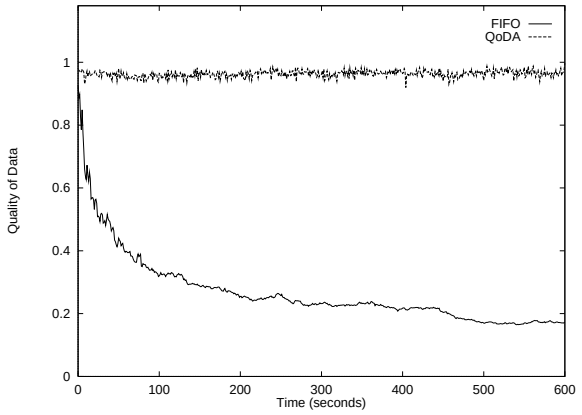


Figure 10: Real workload experiment: update processing speed is 70% of the average incoming update rate

the case where the update processing speed is 70% of the average incoming update rate (Figure 10), the QoD for the QoDA schedule (top line) is on average 3.6 times better than the QoD for the FIFO schedule (bottom line). At this rate, FIFO does not recover and its QoD continuously deteriorates.

In Table 3 we list the overall QoD for the real workload experiment for both the FIFO and QoDA update schedules when the update processing speeds ranges from 300 updates/sec (or 46% of the average incoming update rate) to 1200 updates/sec (or 185% of the average incoming update rate). The first row of the table is the update processing speed in updates/second for each experiment, and the second row lists what percentage of the average incoming update rate the processing speed corresponds to. We clearly see that the QoDA schedule consistently outperforms the FIFO schedule, even when the processing speed is more than the average incoming update rate. Moreover, the QoD under the QoDA schedule remains high, even at low update processing speeds. For example, when the update processing speed is at 46% of the average incoming update rate, the QoD under QoDA is 0.82. For the same case, the FIFO schedule gives a 0.13 QoD, which is six times worse than the QoDA schedule. A 0.13 QoD means that an estimated 87% of the accesses will be served with stale data. In general, under medium update processing speeds the FIFO schedule gives really poor QoD.

5.2 Tolerance to Update Surges

Tolerance to surges in the update rate is crucial for any update scheduling algorithm. Users should not have to suffer poor QoD if there is a surge in the update volume. As we saw in the workload study (Section 2), update surges occur often in the incoming update stream. Furthermore, the effects created by update surges can also be created indirectly, by surges in the incoming access rate, which increase the overall load at the server and decrease the update processing speed of the system.

We created a synthetic database of 1000 relations with 20 materialized views each (i.e. we had a total of 20,000 views). The view refresh cost was uniform, the frequency of access followed the Zipf distribution and the average incoming update rate was 1050 updates/second. The incoming update stream was 120 seconds long and consisted of three parts. The first 20 seconds had a "regular" incoming update rate, the next 10 seconds had a "surge" during which the incoming update rate jumped to five times that of the regular update rate, and, finally, the remaining 90 seconds also had the same "regular" update rate as the first part.

Figure 11 has the results from a two-fold, five-fold and ten-fold update surge with a 20% extra update processing capacity for both FIFO scheduling (Figure 11a) and QoDA scheduling (Figure 11b). QoDA outperforms FIFO, especially under high update surges, since it is able to identify the views to refresh so that the QoD is maximized. For example, for the ten-fold surge, the QoDA schedule manages to recover in approximately the same time as in the five-fold surge. On the other hand, the FIFO schedule never recovers from the ten-fold surge and, after the surge, the QoD drops to unusable levels (less than 0.04) and remains practically fixed. In this case, the QoD for the QoDA schedule is about two orders of magnitude higher than the QoD for the FIFO schedule.

Note the shape of the QoD curves for QoDA after the update surges (Figure 11b). This is attributed to the fact that the QoDA update schedule refreshes first the views that will have the greatest impact on the QoD, therefore the rate of improvement on the QoD decreases during the recovery from the update surge. In practice, the limiting factor to how quickly the QoDA schedule recovers from update surges is the amount of relation updates that have to be performed (and cannot be postponed like materialized view refreshes).

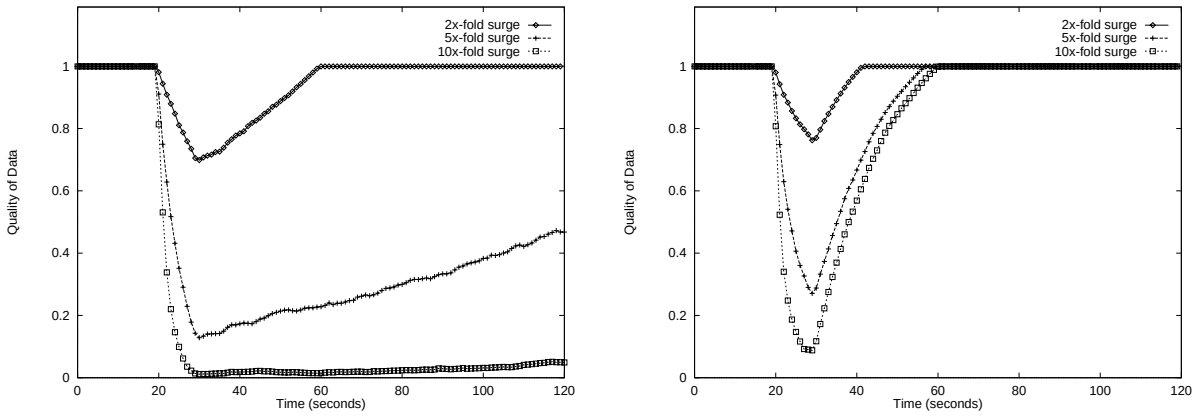
Unfortunately, we cannot compare QoDA with the off-line optimal update scheduling algorithm (as we did for the motivating example), because the off-line optimal algorithm would require enumerating all the possible update schedules, which is infeasible for more than 15-20 updates.

6 Related Work

The update scheduling problem is to some extent similar to the problem of scheduling tasks on a single machine in order to minimize the *weighted completion time* under precedence constraints [CM99], which has been proven to be NP-hard for the general case [Law78]. In the update scheduling problem the objective is to maximize QoD, which can be translated to minimizing the weighted staleness for all views. There are, however, a lot of differences between the two problems. First of all, the update scheduling problem has multiple classes of "tasks": updates for relations, materialized views, and virtual views, as opposed to one type for the weighted completion time problem. Relation updates must be scheduled in the order of arrival which is not the case for any of the tasks in the weighted completion time problem, and also, their staleness is not measured in the overall QoD metric. Furthermore, we have the option of postponing materialized view refreshes in the update

processing speed (upd/sec)	300	400	450	550	600	650	750	1200
% of average incoming rate	46%	61%	70%	85%	92%	100%	115%	185%
QoD for FIFO schedule	0.135	0.199	0.268	0.892	0.913	0.921	0.932	0.957
QoD for QoDA schedule	0.821	0.935	0.963	0.975	0.977	0.978	0.981	0.988

Table 3: Average QoD for real workloads under various update processing speeds



(a) FIFO schedule

(b) QoDA schedule

Figure 11: Effect of update surges on scheduling algorithms with 20% extra update processing capacity

scheduling case, whereas in the weighted completion time problem all tasks must be performed. Also, although virtual views do not have to be refreshed, they are "counted" when reporting staleness. Finally, for the update scheduling problem we must have an *online* algorithm, whereas most approximation algorithms for the weighted completion time problem are off-line.

[PSM98, YV00, OLW01] deal with consistency issues for update propagation in the context of replicated databases. [AGMK95] and [AKGM96] deal with the scheduling of updates in the context of real-time databases, where update operations have to compete with transaction processes that have soft deadlines. [AGMK95] considers the scheduling of updates only and suggests algorithms to improve transaction timeliness without sacrificing data timeliness. [AKGM96] focuses on recomputation strategies: how and when to perform refreshes of derived data in order to reduce cost without sacrificing data timeliness. They propose delaying recomputations slightly (forced delay), so that several related source updates can be combined in a single step. However they do not provide the means to determine the forced delay interval. Moreover, they do not consider view derivation hierarchies, differences in update costs or allow for the ability to reorder view refreshes and tolerate update surges like we do.

[CGM00] deal with the issue of when to poll remote sources in order to update local copies and improve the database freshness. They provide synchronization policies which are mostly suited for web crawlers. However, their freshness metric is not popularity-aware. With the highly skewed access patterns on web servers (as we saw in Section 2, even one page can correspond to as much as 17% of the entire web server traffic), we must weigh the freshness of each page accordingly when reporting the overall freshness.

The update scheduling problem is somewhat similar to data broadcast scheduling. [SRB96] adapt the broadcast content based on the "misses" using a temperature-based model. [AF99] schedule data items for broadcast based on the product of the number of requests for an item times the amount of wait time since it was first requested. An important difference between update scheduling and data broadcast scheduling is that update scheduling must handle precedence constraints (as a result of a view computation hierarchy) and non-uniform view refresh costs which are not considered in data broadcast scheduling algorithms.

7 Conclusions

We studied the workload of a commercial update-intensive web server and found highly skewed access & update patterns, as well as frequent surges in the update load. Inspired by this study, we have developed a framework for measuring Quality of Data in web server caches, which is based on freshness. We focused on the *update scheduling problem*: ordering the relation and materialized view updates to maximize QoD. We introduced QoDA, a QoD-Aware update scheduling algorithm that unifies the scheduling of relation updates and materialized view refreshes under one framework.

We compared QoDA update scheduling to FIFO scheduling through extensive experiments on real and synthetic workloads. QoDA schedules consistently exhibited higher QoD than FIFO schedules by up to two

orders of magnitude. Especially for update surges, FIFO schedules degenerate to unusable QoD levels, whereas QoDA schedules quickly recover and maintain high QoD.

Update surges are a fact of life and in panic situations they exceed any server capacity. Brute-force solutions of increasing hardware and software capacity to tolerate surges are not financially sound. Instead, we envision QoDA acting as an "*update surge protector*" for guaranteeing high Quality of Data under rapidly changing load conditions, in the same way that caching of static web pages is used to guarantee high QoS under access surges.

Acknowledgments

We would like to thank Damianos Karakos, Yannis Sismanis, Manuel Rodriguez and the anonymous reviewers for their helpful comments.

References

[AF99] Demet Aksoy and Michael Franklin. "RxW: A Scheduling Approach for Large-Scale On-Demand Data Broadcast". *IEEE/ACM Transactions on Networking*, 7(6), December 1999.

[AGMK95] Brad Adelberg, Hector Garcia-Molina, and Ben Kao. "Applying Update Streams in a Soft Real-Time Database System". In *Proc. of the ACM SIGMOD Conference*, pages 245–256, San Jose, California, June 1995.

[AKGM96] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. "Database Support for Efficiently Maintaining Derived Data". In *Proc. of the 5th International Conference on Extending Database Technology (EDBT'96)*, Avignon, France, March 1996.

[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.

[CGM00] Junghoo Cho and Hector Garcia-Molina. "Synchronizing a database to improve freshness". In *Proc. of the ACM SIGMOD Conference*, Dallas, Texas, USA, May 2000.

[CM99] Chandra Chekuri and Rajeev Motwani. "Precedence Constrained Scheduling to Minimize Weighted Completion Time on a Single Machine". *Discrete Applied Mathematics*, 98(1-2), October 1999.

[Law78] E. L. Lawler. "Sequencing jobs to minimize total weighted completion time". *Annals of Discrete Mathematics*, 2:75–90, 1978.

[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.

[LR00a] Alexandros Labrinidis and Nick Roussopoulos. "WebView Materialization". In *Proc. of the ACM SIGMOD Conference*, Dallas, Texas, USA, May 2000.

[LR00b] Alexandros Labrinidis and Nick Roussopoulos. "Workload Study of a Popular Web Server with Dynamic Content". Technical Report submitted to Quote.com, June 2000.

[LR01] Alexandros Labrinidis and Nick Roussopoulos. "Update Propagation Strategies for Improving the Quality of Data on the Web". Technical report, Institute for Systems Research, June 2001.

[OLW01] Chris Olston, Boon Thau Loo, and Jennifer Widom. "Adaptive Precision Setting for Cached Approximate Values". In *Proc. of the ACM SIGMOD Conference*, Santa Barbara, California, USA, May 2001.

[PSM98] Esther Pacitti, Eric Simon, and Rubens N. Melo. "Improving Data Freshness in Lazy Master Schemes". In *Proc. of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[Rou82] Nick Roussopoulos. "The Logical Access Path Schema of a Database". *IEEE Transactions on Software Engineering*, 8(6):563–573, November 1982.

[SRB96] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. "Adaptive Data Broadcasting Using Air–Cache". In *First International Workshop on Satellite-based Information Services*, Rye, New York, November 1996.

[YV00] Haifeng Yu and Amin Vahdat. "Design and Evaluation of a Continuous Consistency Model for Replicated Services". In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.