Online View Maintenance Under a Response-Time Constraint [$\star$]

Kamesh Munagala, Jun Yang, and Hai Yu

Department of Computer Science, Duke University
Durham, NC 27708-0129, USA
{*kamesh,junyang,fishhai*}*@cs.duke.edu*

Abstract. A *materialized view* is a certain synopsis structure precomputed from one or more data sets (called *base tables*) in order to facilitate various queries on the data. When the underlying base tables change, the materialized view also needs to be updated accordingly to reflect those changes. We consider the problem of batch-incrementally maintaining a materialized view under a *response-time constraint*. We propose techniques for *selectively* processing updates to some base tables while keeping others batched, with the goal of minimizing the total maintenance cost while meeting the response-time constraint. We reduce this to a generalized paging problem, where the cost of evicting a page is a concave non-decreasing function of the number of continuous requests seen since the last time it was evicted. Our main result is an online algorithm that achieves a constant competitive ratio for all concave cost functions while relaxing the response-time constraint by a constant factor. For several special classes of cost functions, the competitive ratio can be improved with simpler, more intuitive algorithms. Our algorithms are based on emulating the behavior of an online paging algorithm on a page request sequence carefully designed from the cost function. The key novel technical ideas are twofold. The first involves discretizing the cost function, so that there is a collection of periodic paging sequences, with page sizes decreasing geometrically, which approximates the behavior of the original function. The second involves designing an online view maintenance algorithm based on the paging process, by emulating the behavior of the paging scheme in recursively defined phases.

1 Introduction

A *materialized view* is a certain synopsis structure precomputed from one or more data sets (called *base tables*) in order to facilitate various queries on the data [8]. Materialized views have a wide range of traditional and new applications, such as data warehousing, database caching, continuous queries, and publish/subscribe systems, just to name a few. Since a materialized view is a form of derived data, which is computed from the underlying base tables, it needs to be *refreshed* if the base tables change. Instead of recomputing the view from scratch in order to refresh it, we can incrementally maintain the view, i.e., compute and apply only the incremental changes to the view given the base table updates. Furthermore, for many applications, incremental maintenance

* Research is supported in part by NSF CAREER award under grant IIS-0238386.

does not have to be performed eagerly for each base table update; instead, the system can defer maintenance until the view content needs to be accessed. Hence, the system maintains the view in a *batch incremental* fashion: Base table updates are accumulated into a batch and then processed together when needed.

The cost of processing the updates is typically a concave non-decreasing function of the number of updates in the batch. In other words, processing a batch of updates is usually more efficient than processing them one at a time. Therefore, batch incremental maintenance can be used to improve efficiency for many applications where deferred view maintenance is acceptable. For example, recent publish/subscribe systems, e.g., OpenCQ [11], NiagraCQ [4], Xyleme [13], all provide a feature that allows subscribers to specify a *notification condition* in addition to the subscription content query. Examples include: "*report mean and median house prices in North Carolina once every* 200 *house sales*", or "*notify me with a detailed view of my portfolio whenever the price of a stock in my portfolio has changed by more than* 5 *percent since the last notification*". Only when the notification condition is met, the system needs to compute updates to the subscribed content (which can be regarded as a materialized view) and notify the subscriber.

At the same time, in many such applications, it is often desirable to provide a quality-of-service guarantee in the form of a *response-time constraint*. That is, whenever the content of a materialized view is requested, the system should be able to refresh the view under a prescribed time limit. This constraint prevents the system from deferring view refresh indefinitely; if the batch of unprocessed base table updates becomes too big, it may be impossible to process the batch in time upon request. This paper addresses the problem of maintaining a materialized view under a response-time constraint, with the goal of minimizing the total cost of view maintenance over time.

Work in [9] mainly considers the offline version of the problem, where the system has some knowledge of the arrival sequence of future base table updates. In this paper, we propose online algorithms that are constant-competitive against any adversary, assuming the response-time constraint is relaxed by a constant factor. We show that this problem is an interesting generalization of paging with concave cost functions; this connection is of independent interest. We need new ideas to extend paging algorithms to concave functions, as we point out below.

Problem statement. Formally, we want to incrementally maintain a view defined over n base tables, $v_1, v_2, \ldots, v_n$. The cost of refreshing the view by performing x units of updates to base table v_i is $f_i(x)$. Each $f_i(x)$ is a concave, non-decreasing function.[1] By appropriate scaling, we enforce $f_i(1) \geq 1$ for each i. At any time instant t, an update vector $\langle x_{1t}, x_{2t}, \ldots, x_{nt} \rangle$ arrives online, where x_{it} is an integer denoting the number of updates to base table v_i. Let $\langle X_{1t}, X_{2t}, \ldots, X_{nt} \rangle$ denote the total updates pending for each of these base tables respectively after the update vector at the current step arrives. The algorithm can now *schedule* a vector $\langle Y_{1t}, Y_{2t}, \ldots, Y_{nt} \rangle$ of updates on

[1] Strictly speaking, these cost functions are *subadditive*; that is, processing $x + y$ updates together cannot be more expensive than processing x updates in one batch and then y updates in another, since the option of doing the latter is still available given all $x + y$ updates. We note that results for subadditive and concave functions are equivalent within a factor of 2, so the same algorithms can still apply in the subadditive case.

the base tables, and incur an *update cost* of $C_t = \sum_{i=1}^{n} f_i(Y_{it})$, and carry forward the remaining updates as pending to the next time step. Let $\langle Z_{1t}, Z_{2t}, \dots, Z_{nt} \rangle$ denote the vector of updates carried forward, or pending. Note the relations $Z_{it} = X_{it} - Y_{it}$ and $X_{it} = x_{it} + Z_{i,t-1}$ hold for all v_i and all t. There is a bound H on the total update cost that can be pending at any point of time, which we call the *pending update cost constraint*. That is, the constraint on the algorithm is $\sum_{i=1}^{n} f_i(Z_{it}) \leq H$ for all t.

The goal is to design an online algorithm to minimize the total update cost $\sum_t C_t$, when the update vectors $\langle x_{1t}, x_{2t}, \dots, x_{nt} \rangle$ arrive online at every time step.

This problem generalizes the problem of paging with arbitrary page sizes, where the goal is to minimize the total cost of page faults, or equivalently, the total size of pages evicted (under the BIT model of paging [10]). Given a paging problem with n pages, where page p_i has size s_i (an integer), and a cache with size k, the equivalent view maintenance problem has base table v_i for page p_i, with update cost $f_i(x) = s_i$ independent of x, and $H = k$. The constraint that the pending update cost is at most H translates exactly to the cache not overflowing. The update cost at any time step is precisely the size of the pages evicted at that step.

Our results. Our main result in Section 4 presents a constant-competitive online algorithm for the view maintenance problem, with a constant-factor relaxation in the pending update cost constraint. Before that, in Section 2, we present a very simple $O(\log H)$-competitive algorithm, with the pending update cost constraint being relaxed to $8H$. In Section 3, we describe a constant-competitive algorithm for a natural special case, which forms the basis for the more general constant-competitive algorithm of Section 4. Our algorithms are deterministic and work against adaptive adversaries. The competitive ratio is essentially best possible (to within constant factors) if the response-time constraint is allowed to be relaxed by a constant factor.

Our main idea is to emulate the behavior of an online paging algorithm on an appropriately defined paging sequence, and use this behavior to guide the view maintenance algorithm. We first convert an online sequence of updates for the original problem into a page request sequence. Next, we run the online paging algorithm in [5, 16], with the best known competitive ratio of $O(1)$ on this page request sequence using a cache of size $2H$. We then convert the resulting online paging scheme back into an online scheme for view maintenance.

There are constant-competitive paging algorithms known for arbitrary page sizes and eviction costs [5, 10, 16] when the cache size is relaxed. Although we emulate paging on a suitable page request sequence, and hence use these paging algorithms as subroutines, this is in no way straightforward. The main problem is the concave nature of the eviction cost. If we try defining multiple pages with decreasing eviction costs to model concavity, we run into the problem that their evictions have to be correlated: the cheaper page cannot be retained in the cache by the paging algorithm if the more expensive page has been evicted. A problem with using a paging algorithm *as is* is that it becomes non-trivial to convert the behavior of the paging algorithm into a well-defined view maintenance scheme. We show how to tackle both these issues by carefully constructing the paging sequence after discretizing the concave cost function, and by grouping the page evictions into recursive phases and performing view maintenance depending on the behavior of the paging algorithm in each phase. Both these details are non-trivial, requiring new ideas, and are expounded in Section 4.

All our algorithms need to relax the response-time constraint by a constant factor. An interesting open question is to decide the complexity of the problem (especially with respect to oblivious adversaries) if the response-time constraint is not relaxed. We also note that our results can be made bicriteria on the competitive ratio and the pending update cost relaxation in a fairly straightforward manner.

Related work. The classical online paging problem with unit sized pages is well studied in the offline [2] and the online settings [6, 12, 14].

When the pages have arbitrary sizes and eviction cost proportional to size (assuming the smallest page has size 1), the offline paging problem becomes NP-Hard, and the best known polynomial-time approximation algorithm achieves a factor of $O(\log k)$ to the optimal cost [10]. This shows that the offline view maintenance problem is NP-Hard (using the reduction above), with the best approximation algorithm achieving a factor of $O(\log H)$. For the online version of the problem, LRU is k-competitive, and is the best possible deterministic paging scheme [10]. Randomized marking algorithms perform much better; there is a $O(\log^2 k)$-competitive paging scheme against an oblivious adversary, due to Irani [10].

The paging problem can be further generalized to allow both the sizes of the pages and the cost for evicting the pages to be arbitrary. In the offline setting, Albers *et al.* [1] obtained a constant-approximation algorithm that uses an additional amount of memory of size $O(1)$ times the largest page size. In the online setting, Cao and Irani [3] generalized a greedy-dual algorithm of Young [15] and showed that this deterministic online algorithm is k-competitive. Young [16] and Cohen and Kaplan [5] proved that it is $h/(h-k+1)$-competitive, by running the same algorithm on a cache of size $h \geq k$. We use this algorithm as our paging subroutine for $h = 2k$ (meaning we double the cache size used by the offline algorithm in order to be 2-competitive). We note that since the best possible competitive ratio for the paging problem is constant when the cache size is relaxed by a constant factor, the same guarantee is a lower bound for the more general view maintenance problem.

2 Simple $O(\log H)$-Competitive Algorithm

We now show a simple randomized reduction of the online view maintenance problem to the online paging problem. The reduction can be easily made deterministic, but randomization slightly simplifies our analysis. The resulting algorithm achieves a competitive ratio of $O(\log H)$ against an oblivious adversary, provided that the pending update cost constraint can be relaxed to $8H$. This section provides the background for the more involved algorithms in later sections.

The reduction is accomplished in two steps. At the first step, we "translate" an online sequence of updates for the view maintenance problem into a page request sequence, and run a competitive online paging algorithm on this page request sequence. At the second step, we "translate" the resulting online paging scheme back into an online scheme to the original problem.

For each base table v_i, we assume f_i is a continuous function. If not, the proofs below go through at the loss of an additional constant factor. Let $r_0 = 1$, and $r_1, \ldots, r_h$ be the values of update size such that $f_i(r_j) = 2^j f_i(r_0)$. Further, $f_i(r_h) \geq H$, but $f_i(r_{h-1}) < H$, which implies $h \leq \log H$. Corresponding to each r_j, we have a page p_{ij} of size $f_i(r_j)$.

For an online view maintenance problem, we generate an instance of the online paging problem as follows. The size of the cache is $2H$. Whenever one unit of update is input for base table v_i, for each j, we request for page p_{ij} with probability $\min(1, 1/r_j)$. These requests are independent from one unit of update to the next, even within the same time step. Let OPT denote the cost of the optimal offline view maintenance algorithm on a certain update sequence. Our analysis relies on the following lemma.

Lemma 1 ([9]). *There exists a view maintenance scheme such that for all i and t, if $Y_{it} > 0$, then $Z_{it} = 0$, and whose total update cost is at most $2 \cdot \mathrm{OPT}$.*

Intuitively, this lemma implies that one can find a 2-competitive view maintenance scheme so that whenever any update is processed for a base table, the entire pending updates for that base table are processed. The proof of the above lemma proceeds by modifying an optimal offline scheme to process all pending updates of a base table whenever it processes any update of that base table, so that any update operation in the original scheme is charged by at most two update operations in the modified scheme. Details can be found in [9].

Lemma 2. *There is a paging scheme for the page request sequence whose expected cost is $O(\log H) \cdot \mathrm{OPT}$, and which uses $2H$ amount of cache space.*

Proof. We will argue the bound for a certain base table v_i, since the update costs for different base tables are additive. Consider the view maintenance scheme in Lemma 1. Consider a time interval $[t_1, t_2]$ such that $Z_{i,t_1-1} = 0$, $Z_{it_2} = 0$, and $Z_{it} > 0$ for all $t \in [t_1, t_2)$. Note that $Z_{it} = \sum_{t'=t_1}^{t} x_{it'}$, and at time t_2 the view maintenance scheme processes all X_{it_2} pending updates to base table v_i. We will pretend $Z_{it_2} = X_{it_2}$ in the proof below to unify notation.

At any time $t \in [t_1, t_2]$, we allocate space $2f_i(Z_{it})$ to the paging algorithm for caching a subset of the pages $p_{i0}, \cdots, p_{ih}$. More precisely, let p_{ij} be the largest page whose size is at most $f_i(Z_{it})$; we will cache $p_{i0}, p_{i1}, \ldots, p_{ij}$ in this space at time t. Since the page sizes scale by a factor of 2, this space allocated is sufficient to cache all these pages. If there is a page request for a page of size larger than $f_i(Z_{it})$ at time t, we do not cache this page, but evict it as soon as it is encountered, leading to a page fault. At time t_2, we evict all pages p_{ij} in the cache.

Consider a page p_{ij} which is in the cache at time t_2. We brought this page into cache at time t when $Z_{it} \geq r_j$ and $Z_{i,t-1} < r_j$. In the interval $[t_1, t-1]$, the expected number of times page p_{ij} is requested is at most $Z_{i,t-1}/r_j \leq 1$, which means the expected cost for these page faults is at most $f_i(r_j)$. After time t, there are no additional page faults on p_{ij}, until we evict it at time t_2. Let k denote the largest such j, i.e., $f_i(r_k) \leq f_i(Z_{it_2}) < 2f_i(r_k)$. The total expected cost of all page faults for $j \leq k$ in the interval $[t_1, t_2]$ is then at most $2\sum_{j \leq k} f_i(r_j) \leq 4f_i(r_k)$, where there is a factor of 2 since we evict all pages $p_{i0}, \cdots, p_{ik}$ at time t_2.

Consider now a page p_{ij} which is not in the cache at time t_2. Hence $Z_{it_2} < r_j$. The expected number of times page p_{ij} is requested in the time interval $[t_1, t_2]$ is at most Z_{it_2}/r_j. Thus the expected cost of page faults due to this page is at most $(Z_{it_2}/r_j) \cdot f_i(r_j) \leq f_i(Z_{it_2}) < 2f_i(r_k)$, where the first inequality holds because $f_i(x)$ is a concave function. Therefore, the total cost of these page faults is at most $O(\log H) \cdot f_i(r_k)$, as there are at most $\log H$ such pages.

We have thus shown that the paging scheme pays cost at most $O(\log H) \cdot f_i(r_k)$ in the interval $[t_1, t_2]$, while the view maintenance scheme pays cost $f_i(Z_{it_2}) \geq f_i(r_k)$ at time t_2. Therefore, overall, the cost of the paging scheme is at most $O(\log H) \cdot \mathrm{OPT}$.

Note that it does not help the adversary to inject updates costing more than H at time t_2, since doing so would result in the same cost for both. □

The above lemma shows that there is a paging scheme whose cost is $O(\log H)$-competitive with respect to the optimal offline scheme of the view maintenance problem. The online algorithm for the view maintenance problem generates the random page request sequence, and runs the constant-competitive paging algorithm [5, 16] on this sequence using a cache of size $4H$. At any time t, let W_{it} be the size of the largest page corresponding to base table v_i in cache; the algorithm allocates cost at most $2W_{it}$ to the pending updates of v_i. There are two situations where we process all pending updates of v_i: (1) if the total cost of these updates becomes larger than $2W_{it}$; and (2) if the paging algorithm evicts the largest page currently in its cache corresponding to v_i.

Lemma 3. *The online view maintenance algorithm is constant-competitive against the cost of the corresponding online paging scheme.*

Proof. Consider two consecutive time instances t_1 and t_2 when the online view maintenance algorithm processes updates of base table v_i. If the reason for processing updates at time t_2 is because the largest page was evicted from the cache, the cost of the update can be accounted for by the cost of the page evicted (whose size is at least $f_i(X_{it_2})/2$). If the reason is that $f_i(X_{it_2})$ exceeds twice the size of the largest page, we charge the cost of the update to the present or next instant when a page of largest size less than $f_i(X_{it_2})$ is requested (and subsequently evicted); note that the size of this page is at least $f_i(X_{it_2})/2$. Since the behavior of the paging scheme is independent of the distribution of future page requests, the expected number of such charges made to any page is at most one. Therefore, overall, the expected competitive ratio of the online view maintenance algorithm is 2 against the online paging algorithm. □

The above two lemmas immediately imply the following theorem (noting that we lose a factor of 2 in the pending update cost due to the previous lemma).

Theorem 1. *There is an $O(\log H)$-competitive online algorithm for the view maintenance problem that relaxes the pending update cost constraint to $8H$.*

Remark. (1) There is a tradeoff between the relaxation in pending update cost constraint and the competitive ratio. In particular, we can obtain a $O(\frac{1}{\epsilon} \log H)$-competitive algorithm while relaxing the pending update cost to $(1 + \epsilon)H$. We simply use a coarser approximation to f_i by rounding it in powers of $\frac{1}{\epsilon}$.

(2) The page request sequence in the reduction can be made deterministic. Specifically, for each page p_{ij}, we request it once every r_j updates to base table v_i. The competitive ratio remains the same. We omit the details.

3 Improved Algorithms for Special Cases

For several important special cases of cost functions, we can obtain simpler and more intuitive algorithms with better performance guarantees. Here we consider the following three cases: (1) $f_i(x) = \min(a_i x, b_i)$; (2) $f_i(x) = a_i(x-1) + b_i$; and (3) $f_i(x) = \min(a_i(x-1) + b_i, c_i)$.

The first case, $f_i(x) = \min(a_i x, b_i)$, can arise in the following situation. The cost of processing a batch initially increases linearly with the size of the batch. When the size of the batch reaches a certain point, however, it becomes more efficient to simply recompute the view, whose dominating cost becomes independent of the batch size. The second case, $f_i(x) = a_i(x-1) + b_i$, can arise if update processing incurs a fixed amount of startup cost. Details for these two cases will appear in the full paper.

In the remainder of this section, we focus on the third case, $f_i(x) = \min(a_i(x - 1) + b_i, c_i)$. This special case is a generalization of the first two cases, and serves as a preparation for our subsequent discussions in Section 4. The paging sequence that we construct for this case is deterministic and *periodic*. Let us assume c_i is a multiple of b_i, which can be enforced at a loss of factor of 2 in the competitive ratio. We have $k = \frac{c_i}{b_i}$ types of pages corresponding to base table v_i. Let us denote them $p_{i1}, p_{i2}, \ldots, p_{ik}$. We assume $k > 4$; the case for smaller k is simple to deal with, and is therefore omitted.

We maintain two counters, c and r, which are initialized to 0 and 1 respectively. Whenever there is a unit update received for the base table v_i, we request p_{ir} and increment c. If $c \geq \frac{b_i}{a_i}$, we set $c \leftarrow 0$, and $r \leftarrow r \bmod k + 1$. The size of the cache is H. As before, we can prove the following emulation result.

Lemma 4. *There is a paging scheme for the page request sequence whose cost is at most* $2 \cdot \mathrm{OPT}$.

Proof. The paging scheme emulates the view maintenance scheme of Lemma 1 as follows. Consider an interval of time $[t_1, t_2]$ as defined in the proof of Lemma 2. At time t, the space available for pages corresponding to v_i in the cache is at most $f_i(Z_{it}) = f_i(\sum_{i=t_1}^{t} x_{it})$. If the page p_{ir} requested at time t is not present in the cache, it is brought in, and if a page needs to be evicted from the cache due to lack of space, the oldest page corresponding to v_i is evicted. Note that there can be at most one such eviction in $[t_1, t_2)$, and it would be at the instant the second oldest page was requested. For every subsequent time instant, the requested page will either already be in the cache, or there will be sufficient space in the cache for it. At time t_2, the paging algorithm faults on the requested page p_{ir} and evicts all pages corresponding to v_i in its cache.

If $f_i(X_{it_2})$ is at most c_i, the argument from the previous subsection shows that the algorithm is 2-competitive. Otherwise, the paging algorithm will have all k pages in the cache, and the cost of evicting them at time t_2 is c_i, which is equal to the cost paid by the view maintenance scheme at that time for processing all pending updates of v_i. $\square$

The online view maintenance algorithm runs the constant-competitive paging algorithm on this page request sequence using a cache of size $2H$. If the paging algorithm caches a total size x of pages corresponding to base table v_i, the algorithm allocates at most $6x$ pending update cost to v_i. The algorithm proceeds in phases. A phase corresponds to the "period" of the request sequence, that is, the time interval in which all k distinct pages are requested. At the beginning of each phase, all pending updates of v_i are *untagged*, and if the number of pages corresponding to v_i in the cache is less than $k/2$, the algorithm processes all these untagged updates to v_i. During the phase, for each received update of v_i, suppose page p_{ir} is requested and is in or being brought into the cache. The algorithm keeps the corresponding update pending and *tags* this update to that page. If the page is not cached, this update is processed immediately. When a page is evicted, the updates tagged to that page are processed. If the number of cached pages is larger than $k/2$ at the beginning of the phase, but drops below $k/4$ sometime during the phase, the algorithm processes all untagged pending updates to v_i. Finally, all pending updates (either tagged or untagged) are reset to untagged at the end of the phase before entering the next phase.

Theorem 2. *The online algorithm has a competitive ratio of $O(1)$ against the optimal view maintenance algorithm, and needs to relax the pending update cost to $12H$.*

Proof. We lose a factor of 2 in pending update cost upfront simply because the online paging algorithm uses cache $2H$. First note that the total cost of updates that can be tagged to any page is at most $2b_i$, so that the cost of processing these tagged updates can be charged to the eviction of the corresponding page.

Secondly, if there are untagged updates pending, the total cost allocated to the updates of v_i is at least c_i. To verify this claim, observe that the untagged updates must have been from the previous phase, and their presence indicates that the number of pages in the cache must be at least $k/4$, since if there were either less than $k/2$ pages at the beginning of the phase, or less than $k/4$ pages sometime during the phase, these updates would have been processed then. With a factor 4 relaxation in the pending updates cost constraint, the algorithm would allocate cost at least c_i for these updates. The total factor of 6 in the relaxation comes from the sum of the factor 4 for the untagged updates, and the factor of 2 for the tagged updates.

Next, we observe that whenever the algorithm is forced to process all the pending updates (by spending cost of at most c_i) at the beginning of a phase, the number of pages cached is at most $k/2$, which means that the number of pages requested in the phase that will not be present in the cache is at least $k/2$. The cost of fetching these pages (and subsequently evicting them) is at least $c_i/2$. Therefore, the update cost can be charged to the cost of evicting these pages.

If the untagged pending updates are processed during a phase, the number of pages evicted by the algorithm since the beginning of the phase is at least $k/4$, and the cost of these evictions is at least $c_i/4$. The cost for processing the updates can therefore be charged to the cost of evicting these pages. $\square$

Remark. The factor 12 in the relaxation can be improved by choosing when to process the untagged updates better. We omit the discussion of optimizing constant factors.

4 Constant-Competitive Algorithm

In this section, we present an algorithm for general concave functions $f_i(x)$, which achieves a constant competitive ratio, with a constant factor relaxation in the pending update cost constraint. The algorithm is based on the periodic paging sequences constructed in Section 3, combined with an emulation in recursive phases. We first need to discretize the cost function so that a phase-by-phase accounting of cost can be performed. An idea similar to the following lemma appeared in Guha *et al.* [7], and is key to the design of the algorithm. We note that most realistic cost functions would satisfy this lemma upfront.

Lemma 5. *The function $f_i(x)$ can be approximated to a constant factor by a piecewise linear concave function $g_i(x)$ that connects by line segments consecutive points (c_r, d_r) ($c_1 = 1$), so that the points satisfy:*

1. *d_{r+1} is a multiple of d_r;*
2. *$d_{r+1} \geq 2d_r$;*
3. *$\frac{d_r}{c_r} \geq 2\frac{d_{r+1}}{c_{r+1}}$.*

Proof. Let $b = f_i(1)$. Consider those points on the curve $f_i(x)$ with y coordinates $b, 2b, 4b, \ldots$, and denote them by (c_r, d_r), $r \geq 1$. We scan these points in decreasing order of y coordinates. We connect the closest two (in terms of r value) such points whose d_r/c_r values differ by at least a factor of 2 and ignore the intermediate points. The curve ends at $(1, b)$. This new curve $g_i(x)$ is a 2-approximation to the original curve. To see why, consider two points (c_j, d_j) and (c_k, d_k) that are connected (see Figure 1 (a)). For any $x \in [c_{j+1}, c_k]$, by concavity we have $f_i(x) \leq xd_{j+1}/c_{j+1} \leq x(2d_k/c_k)$, and $g_i(x) \geq xd_k/c_k$. For any $x \in [c_j, c_{j+1}]$, we have $f_i(x) \leq f_i(c_{j+1}) = 2b_j$ and $g_i(x) \geq b_j$. Therefore $g_i(x) \geq f_i(x)/2$.

Note that the third condition may not be true at $r = 1$. This problem can be fixed by first approximating $f_i(x)$ by an additional constant factor and then applying the above procedure. We omit the details here. □

In the subsequent discussions, we pretend that $f_i(x) = g_i(x)$. Let (c_r, d_r) be the set of non-smooth points on the curve. For convenience, we further assume $\frac{d_r}{c_r}$ is a multiple of $\frac{d_{r+1}}{c_{r+1}}$; the proof remains the same even if it is not. We have a paging sequence for each "level" r. For $r \geq 1$, we have c_r pages of size $\frac{d_r}{c_r}$. We request a different one every unit update in a cyclic fashion, so that the same page is requested after exactly c_r updates. Let us denote a complete cycle of pages at level r by P_r. Note that at $r = 1$, this process requests the same page of size d_1 every unit update. Note also that this process requests many pages for the same unit update, but these page sizes decrease by a factor of at least 2, so that they sum to at most $2d_1$.

Lemma 6. *There is a paging scheme for the page request sequence that is constant-competitive against the optimal offline view maintenance algorithm, provided that the cache size is $4H$.*

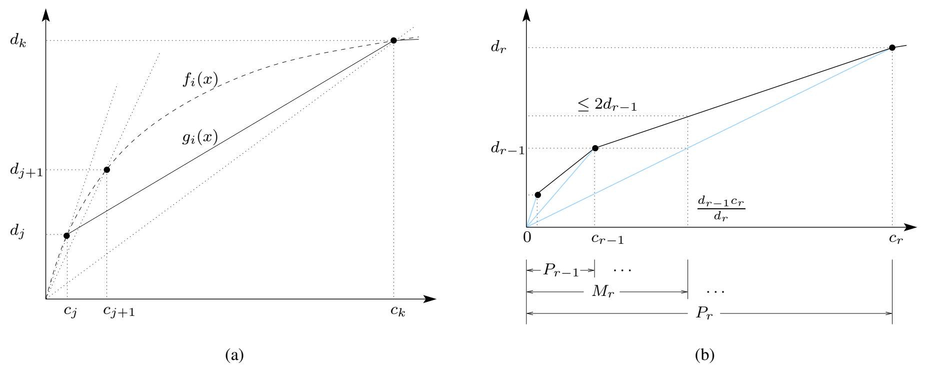

Fig. 1. (a) Approximating $f_i(x)$ by a piecewise linear concave function. (b) Phases and mini-phases.

Proof. The paging scheme emulates the view maintenance scheme of Lemma 1 in the following way. Let $[t_1, t_2)$ be a time interval during which the view maintenance scheme continuously keeps updates to base table v_i pending. At any time, the paging scheme caches all pages requested so far corresponding to v_i. The total size of the cached pages is at most four times the pending update cost of the view maintenance scheme at that time instant. To see why, consider the time when x updates have been pending, and suppose x lies in between c_r and c_{r+1}. The paging scheme has cached the entire set of pages for all levels up to r. The total size of these pages is at most $\sum_{i=1}^{r} d_i \leq 2d_r \leq 2f_i(x)$. For levels greater than r, the total size cached is at most $\sum_{i>r} x \cdot d_i/r_i \leq 2x \cdot d_{r+1}/c_{r+1} \leq 2f_i(x)$. All cached pages corresponding to base table v_i are evicted at time t_2, whose cost can be accounted for by the cost of the view maintenance scheme for processing all the pending updates of v_i at that time. $\square$

As before, we construct the page request sequence and run the constant-competitive online algorithm on the sequence using a cache of size $8H$. We now show how to convert the paging scheme into an online algorithm for the view maintenance problem. The emulation again proceeds in phases as in Section 3, but now consists of recursive phases. Phase J_{rg} marks the end of the g-th complete cycle of the request sequence P_r. We call the value of r the "level" of the phase. Each J_{rg} is composed of consecutive mini-phases M_{rl}, each of which has exactly as many level-r pages as to make the total size exactly d_{r-1}. Note that a mini-phase is composed of many consecutive periods of P_{r-1}. Let B_{rl} be the set of distinct level-r pages corresponding to M_{rl}. Note that $B_{rx} = B_{ry}$ if $y \equiv x \bmod (d_r/d_{r-1})$. The idea behind defining a mini-phase M_{rl} is that the total cost of all updates in this mini-phase is at most $2d_{r-1}$; see Figure 1 (b).

For every unit update, if the $r = 1$ level page is not in the cache, the algorithm processes the corresponding update, else the algorithm pends it. This mechanism corresponds to the base case level $r = 1$. For larger r, the algorithm for the phases is more complicated (see Figure 2). Consider any level r, and a corresponding phase J_{rg}. Suppose the time step is currently in this phase, and in mini-phase M_{rl}. This mini-phase is composed of many sub-phases $J_{r-1,x}$. Suppose the current time step is in phase $J_{r-1,x}$. All updates which arrived within J_{rg} but in $M_{rl'}$ for $l' < l$ are tagged to $B_{rl'}$. Updates arriving in $J_{r-1,x'}$ for $x' < x$, but within M_{rl} are tagged to the set P_{r-1}. This tagging scheme is recursively maintained at all levels r.

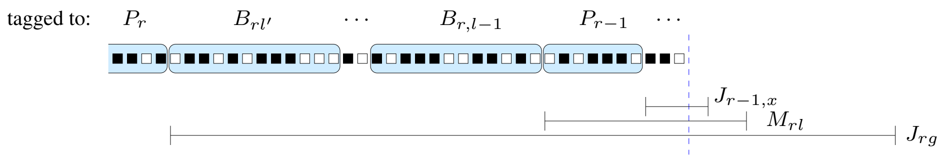

Fig. 2. The recursive tagging scheme. The solid squares represent yet unprocessed (or pending) updates, and the empty squares represent processed updates.

At the end of mini-phase M_{rl}, all pending updates tagged to P_{r-1} are now tagged to B_{rl}. Intuitively, the reason for doing so is that we can no longer afford to tag more updates to P_{r-1}, because otherwise the total cost of these updates may exceed $2d_{r-1}$, while the total size of the pages in P_{r-1} is only d_{r-1}; so we re-tag these updates one level up to B_{rl}, in order to "free" P_{r-1} for tagging future updates.

At the end of every mini-phase M_{rl}, for all $l' \leq l$, if the size of pages from $B_{rl'}$ present in the cache is at most $d_{r-1}/2$, the algorithm processes the updates tagged to $B_{rl'}$. During any mini-phase M_{rl}, if the size of pages for any $B_{rl'}$ for $l' < l$ is at least $d_{r-1}/2$ at the beginning of the phase, but drops below $d_{r-1}/4$ at the current time step, the algorithm also processes all updates tagged to $B_{rl'}$.

At the end of J_{rg}, all updates tagged to each B_{rl} are now tagged to P_r. At the end of J_{rg}, if the size of pages from P_r present in the cache is at most $d_r/2$, the algorithm processes all updates tagged to P_r. If during J_{rg}, this size falls below $d_r/4$, but was larger than $d_r/2$ at the beginning of the phase, the algorithm also processes the pending updates tagged to P_r.

Lemma 7. *The total cost of pending updates is at most* 8 *times the total size of pages at any point during the execution of the algorithm.*

Proof. Any update is tagged either to a P_r or to a B_{rl} for some r and some l. The maximum cost of updates that can be tagged to a P_r before it passes on to a $B_{r+1,l}$ is at most $2d_r$. The maximum cost of updates that can be tagged to a B_{rl} is at most $2d_{r-1}$. We have also maintained the invariant that if the total size of pages in the cache corresponding to P_r is less than $d_r/4$, or those corresponding to B_{rl} is less than $d_{r-1}/4$, there are no updates tagged to these sets. Therefore, the cost of the pending updates can be charged to the size of the pages in the cache. $\square$

Lemma 8. *The cost of processing the pending updates is at most* 12 *times the cost of evictions of the corresponding page sets to which they are tagged, implying the algorithm is constant-competitive against the cost of the online paging scheme.*

Proof. The proof of this claim uses exactly the same argument as the one given in the proof of Theorem 2 in the previous section, charging the update cost to the cost of page evictions in each phase. $\square$

Theorem 3. *The online view maintenance algorithm is constant-competitive, with an* $O(1)$ *relaxation in the pending update cost constraint.*

5 Conclusions

In this paper, we studied the online view maintenance problem, and presented constant-competitive online algorithms against an oblivious adversary, with a constant-factor relaxation in the pending update cost H. The same algorithms can be implemented for more general sub-additive cost functions as well, as they are equivalent to a concave function within a factor of 2. Since our algorithms are based on emulating the behavior of a paging algorithm, they are expected to have much superior performance in practice, and can be designed to exploit specific patterns in the request sequence. We leave finding the best competitive ratio for the problem, when the pending update cost constraint is not relaxed, as an interesting open question.

References

1. S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 31–40, 1999.
2. L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
3. P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 193–206, 1997.
4. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. 19th ACM SIGMOD Intl. Conf. Management of Data*, pages 379–390, 2000.
5. E. Cohen and H. Kaplan. LP-based analysis of greedy-dual size. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 879–880, 1999.
6. A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Competitive paging algorithms. *J. Algorithms*, 12:685–699, 1991.
7. S. Guha, A. Meyerson, and K. Munagala. Hierarchical placement and network design problems. *Proc. 41st IEEE Sympos. Foundations of Comput. Sci.*, pages 603–612, 2000.
8. A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations and Applications.* MIT Press, June 1999.
9. H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *Proc. 21st Intl. Conf. Data Engineering*, pages 106–117, 2005.
10. S. Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.
11. L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowledge and Data Engineering*, 11(4):610–628, 1999.
12. L. McGeoch and D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.
13. B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. 20th ACM SIGMOD Intl. Conf. Management of Data*, pages 437–448, 2001.
14. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
15. N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.
16. N. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002.