Caching Queues in Memory Buffers

Rajeev Motwani[*] Dilys Thomas[†]

Abstract
Motivated by the need for maintaining multiple, large queues of data in modern high-performance systems, we study the problem of caching queues in memory under the following simple, but widely applicable, model. At each clock-tick, any number of data items may enter the various queues, while data-items are consumed from the heads of the queues. Since the number of unconsumed items may exceed memory buffer size, some items in the queues need to be spilled to secondary storage and later moved back into memory for consumption. We provide online queue-caching algorithms under a number of interesting cost models.

1 Introduction

Building infrastructure and middleware for high-performance systems dealing with real-time and streaming data is an important problem in a number of disparate settings such as data stream systems [1, 7, 8, 9, 11], networking [15, 20, 21] and distributed transaction processing [14]. Queues are used by these systems to buffer incoming data items, to transfer intermediate data across internal modules and to temporarily buffer the output produced. While the size of the queues may entail the use of large but slow secondary storage, performance benefits motivate the use of much faster but comparatively smaller memory caches. "Queue spilling" in memory hierarchies is an important problem in all the above-mentioned data flow management systems including data stream systems, packet processing in network routers and switches, and distributed messaging and transaction-processing systems.

Irregularity and burstiness in arrival rates of data-items for the above systems have been well studied and documented [19, 16, 17, 23]. Due to bursts in arrival of data, the number of unprocessed data items at any instant may well exceed the size of the available memory buffer. Over-provisioning memory for periods of high load is not economically sensible, as most often a small memory buffer suffices, and for very short bursts of time the number of unconsumed items in the queues may exceed the average number of unconsumed items in the queues by over an order of magnitude. Some systems resort to load-shedding [2, 12, 5, 22], i.e., dropping excess data items in times of high load. However, a large class of applications, e.g., transaction processing [14] and real-time stock transactions, have integrity requirements which do not allow load-shedding. We take the position that it is preferable to sacrifice performance by spilling data items to disk when the memory buffer overflows, rather than sacrifice data integrity for the sake of performance. The problem then amounts to designing queue spilling strategies for maximum performance.

In the context of maintaining packet queues in network routers/switches, recent work [15, 20] have shown that a small amount of fast expensive SRAM along with slower inexpensive DRAM and queue-spilling algorithms, provides substantial benefits over architectures using a large amount of DRAM alone. IBM MQSeries [14], a distributed messaging system, which has to be resilient to bursty traffic, employs heuristics for the queue spilling problem. Unfortunately, current data stream systems do not use secondary storage (disk) efficiently. An experimental study in the context of the Gigascope system [10] demonstrated a sudden degradation in performance and throughput when the system started spilling to disk storage.

We formulate a model for the problem of spilling data queues. Our model and algorithms are applicable to a wide variety of data stream systems. We provide competitive online algorithms for the efficient use of secondary storage to maintain queues in memory buffers.

1.1 Framework In our model, at any clock-tick an arbitrary number of data items (called "tuples") may enter their respective queues, but only a single tuple is consumed from the head of one of the FIFO (first-in-first-out) queues[1]. We assume that the tuples have uniform size and n different queues (corresponding to different data streams) are being maintained in a memory buffer of a fixed size, say M tuples. Since the total number of tuples in the queues may exceed the size

[*]Department of Computer Science, Stanford University, Stanford, CA 94305. Email: rajeev@cs.stanford.edu. Supported by NSF Grant IIS-0118173, NSF ITR Award Number 0331640, an SNRC Grant, and grants from Microsoft and Veritas.

[†]Department of Computer Science, Stanford University, Stanford, CA 94305. Email: dilys@cs.stanford.edu. Supported by NSF Grant EIA-0137761 and NSF ITR Award Number 0331640.

[1]Our algorithms work for arbitrary number of tuples being consumed at every clock-tick, but we make this assumption only for ease of exposition.

of the memory buffer, some of the tuples in the queues may need to be spilled to secondary storage, which is assumed to be unbounded. We require that the tuple for consumption at the current clock-tick must be read back to memory if it is presently on secondary storage, since tuples can only be consumed from memory. The queue-spilling algorithm must decide in an online manner which (and how many) tuples to write or read, as new tuples arrive and old tuples are consumed.

The following aspects of our model and algorithms deserve to be highlighted:

- *Acyclicity:* A common problem in using a memory hierarchy is that of thrashing – where due to some pathological cases, data is repeatedly moved up and down the hierarchy. This, we suspect, is one of the main reasons for the sudden decrease in performance upon using disk storage in Gigascope [10]. It should be intuitively clear that thrashing is aggravated in stream systems since for FIFO queues the disk blocks that are to be consumed earliest are precisely the ones that are the oldest in the system; consequently, conventional paging algorithms such as LRU will always write exactly the wrong set of blocks onto disk.

  Formally speaking, we say that an algorithm *thrashes* if it writes out blocks to slower storage, and then reads it back for consumption into faster storage, only to be written back to disk once again before it is actually consumed. We desire algorithms that do not thrash, formally defined as those which have a property we call *acyclicity* – the algorithm must ensure that it moves each tuple to disk (and back) at most once.

- *Cost Model:* The cost of an algorithm is defined to be the sum of the cost it incurs for its reads and writes on disk. To a first approximation, disk access times can be decomposed into *seek time* (typically 5-10 ms) to position the disk-head, and *transfer time* to actually read/write the data. Current disk transfer speeds are fairly high, e.g., SCSI disks support 10-160MBps. For moderate size data transfers, it is usually accepted that a good way to account for disk I-O cost is to model each read/write to disk as having unit cost, irrespective of the amount read or written. Therefore, we define the *unit-cost model* where each read/write of *contiguous* tuples in the queue is assumed to have the same cost, regardless of the number of tuples being transferred. Note that in the case of multiple queues, for data locality purposes on disk, we will assume that each read/write on disk involves a *contiguous* block of tuples from a *single* queue. Non-contiguous blocks, or tuples from different queues, will require multiple disk accesses with a higher cost. For large data transfers, the cost of writing t blocks is better modeled as $c_0 + c_1 \times t$, where c_0 and c_1 are constants with $c_0 \gg c_1$. We refer to this model as the *extended cost model*.

- *Queue Updates:* Unless explicitly stated otherwise, we assume that at any clock-tick, at most $M/2n$ new tuples enter the system. In other words, we assume that the input arrival is slow enough that there is at least time to write the excess tuples to disk. This also ensures that tuples currently entering the system first get placed in memory, which is required in many settings.

- *Queue Depletion:* For every queue in the system, the order in which tuples are removed from the queues is determined completely by their arrival order (FIFO). In the case of multiple queues, there is a degree of freedom in choosing the queue for consumption. We consider two different queue depletion scenarios: *adversarial* and *round-robin*. In *adversarial* we do not make any assumption about the order or rate at which the scheduler depletes the different queues. In *round-robin* the scheduler consumes the heads of the queues in a round-robin fashion.

- *Competitiveness:* We desire *online* algorithms that incur cost close to that of any *offline* or *clairvoyant* algorithm which knows the future arrival pattern of tuples. The *competitive ratio* of an *online* algorithm is defined to be the maximum ratio, over all arrival patterns, of its cost and that of an optimal offline algorithm [6]. We will provide competitive online algorithms for both cost models discussed above.

1.2 Summary of Results We now summarize our results and present a road-map of the rest of the paper. In Section 2, for a single queue ($n = 1$) in the unit-cost model, we provide a 2-competitive online algorithm and establish a matching lower bound. Then, in Section 3, we consider the extended cost model where the cost of a disk access depends on the number of blocks accessed. We present a 4-competitive algorithm which can also be extended to the case of multiple queues. For the n-queue problem in Section 4, if the scheduler consumes the heads of the different queues in a round-robin fashion, we have a $2n$-competitive online acyclic algorithm and a $\Omega(\sqrt{n})$ lower bound. On the other hand, in the case of adversarial schedulers, we show that it is not possible to have an *acyclic* online algorithm which is $o(M)$-competitive, where the memory size M is much larger than the number of queues n. However if the online algorithm has an extra $M/2$ memory (vis-a-vis the offline algorithm), we provide a $2n$-competitive algorithm, even for adversarial schedulers. We end the paper in Section 5 with some concluding remarks.

2 Single Queue and Unit Cost Model

In this section, we consider the problem of maintaining a single queue in a memory buffer of size M, under the unit-cost model. We present Algorithm HALF for this version of the problem. This algorithm will form the basis for later algorithms maintaining multiple queues. Recall that each read/write transfers some set of contiguous tuples and has unit cost. The cost of the algorithm is the total number of such reads and writes.

Algorithm HALF keeps the two active ends of the queue buffered in memory. At any point during its execution, HALF divides the set of unconsumed tuples at that instant into three parts: TAIL, SPILLED, and HEAD. The HEAD portion of the queue contains the oldest tuples, while TAIL portion contains the most recent tuples. The TAIL and HEAD portions are in memory, while the SPILLED portion resides on disk. Throughout the algorithm, the oldest tuple in HEAD is the one that is consumed at the next clock-tick. Initially, all unconsumed tuples are in the HEAD portion, and both TAIL and SPILLED portions are empty. As new tuples arrive, they are appended to the HEAD portion. The first write is forced when the memory buffer of size M is full. At this time, HALF writes out $M/2$ of the newest tuples from HEAD to SPILLED. The sizes of HEAD and SPILLED are both exactly $M/2$ at this point. Any new incoming tuples now become a part of the TAIL portion. Note that if SPILLED is empty, the incoming tuples are placed into HEAD whereas if SPILLED is non-empty, the incoming tuples are placed into TAIL. Algorithm HALF will schedule writes/reads to ensure that the following invariants are maintained.

Invariant 1: The tuples in HEAD are always older than the tuples in SPILLED, which in turn are always older than the tuples in TAIL.

Invariant 2: Whenever SPILLED is empty, TAIL is empty too.

Invariant 3: When SPILLED is nonempty, the number of HEAD tuples and the number of TAIL tuples are each at most $M/2$.

The actions of the algorithm to maintain these invariants can be summarized as below.

- **[Write-Out]:** If the number of TAIL tuples reaches $M/2$, while SPILLED is nonempty, the algorithm will write-out all $M/2$ tuples from TAIL to SPILLED.
- **[Read-In]:** If HEAD becomes empty before TAIL reaches M/2, the algorithm will read-in $M/2$ oldest tuples from SPILLED to HEAD.
- **[Transfer]:** If after a read-in, SPILLED becomes empty, the tuples in TAIL are moved to HEAD. The resulting configuration is exactly the one we had at the beginning of the algorithm, before any reads/writes were made and TAIL and SPILLED were empty.

It should be clear that all reads/writes involve exactly $M/2$ tuples, implying that the size of SPILLED is always a multiple of $M/2$. Note that since the TAIL portion is at most $M/2$ in size, there will always be sufficient space for the tuples being read-in. From Invariant 1, it can be seen that Algorithm HALF implements FIFO semantics. HALF also stays within the desired memory bound, and in fact, maintains the invariants enumerated above. We now focus on the performance analysis of the algorithm, starting with a few simple observations.[2]

LEMMA 2.1. *In Algorithm* HALF*, all writes/reads are exactly $M/2$ in size and when a write is performed the size of the queue is more than M.*

LEMMA 2.2. *Algorithm* HALF *is acyclic.*

Proof. The $M/2$ oldest tuples, which contain all the read-in tuples are never written-out. □

THEOREM 2.1. *Algorithm* HALF *is 2-competitive.*

Proof. Let us number the tuples in the order in which they arrive into the queue. Suppose HALF

[2] An alternate version of Algorithm HALF writes not when the TAIL portion becomes $M/2$ in size but rather waits until the HEAD and TAIL tuples fill up memory, and then writes all the TAIL tuples. In this version, whenever SPILLED is nonempty, HEAD is at most $M/2$ in size as before, but TAIL may exceed $M/2$ in size. The sum of sizes of the TAIL and HEAD portions is at most M, as before. Since the number of HEAD tuples is at most $M/2$ when SPILLED is nonempty, in this case each write will involve at least $M/2$ tuples, but can also involve more than $M/2$ tuples. For this modified algorithm the reads need not be exactly $M/2$ in size. It will have to read the $M/2$ tuples to be consumed next from SPILLED, or all of the tuples on disk, whichever is smaller. This is because the last read will read the residual tuples when the amount written is not an integral multiple of $M/2$. For this modified version of HALF, it may be the case that a read may not be directly feasible, since the number of tuples read from disk may not fit in memory along with the TAIL tuples, which may exceed $M/2$ in size. Then, the modified algorithm will need to write the TAIL tuples onto disk, adding them to SPILLED, before actually performing the desired read. Both variants of the algorithm are 2-competitive, but the modified version will have fewer writes than HALF. We only present the analysis of the simpler version of HALF, leaving the analysis of the other version to the full version of the paper.

performs writes upon the arrival of tuples numbered $w_1, w_2, w_3, \ldots, w_i$. Upon the arrival of each such tuple w_j, the $M/2$ newest tuples in memory are written out. Since the algorithm is acyclic, no two writes have a common tuple. Therefore the number of tuples between w_j and w_{j+2} is at least M. Also at each w_j, by Lemma 2.1, there are at least $M+1$ unconsumed tuples, including the set of M tuples just preceding w_j, i.e., $w_j - M + 1, w_j - M + 2, \ldots, w_j - 1, w_j$. As the oldest unconsumed tuple, which is not amongst the M tuples preceding w_j, must be in memory for consumption at the current clock-tick, at least one of the M tuples just preceding w_j must be written to disk by any algorithm, including an offline one. Since the various windows of the M tuples preceding each of the odd numbered tuples $w_1, w_3, w_5, \ldots$ are disjoint, they would cost any offline algorithm distinct writes, thus establishing the 2-competitiveness of Half with respect to the writes.

The 2-competitiveness of reads can be established by a similar argument. Note that if an algorithm attempts to read tuples from among the windows of the M tuples preceding both w_j and w_{j+2} in a common read, then it will be forced to perform another write of tuples among the M tuples preceding w_{j+2} after the common read, and then will have to read them in again, making the situation worse. □

We now establish the optimality of Algorithm Half by showing that no online algorithm can achieve a competitive ratio better than 2.

Theorem 2.2. *There does not exist any online algorithm with competitive ratio smaller than 2.*

Proof. Assume, for contradiction, that we have an online algorithm A with competitive ratio smaller than 2. Consider an arrival pattern where when the first write is required, at that instant (say time τ) the queue has just 1 tuple exceeding the memory buffer size, i.e., there are $M+1$ unconsumed tuples. We assume that the online algorithm does not perform a write before time τ, since otherwise it cannot be competitive. (Note that the optimal offline algorithm will not perform any writes. Now, we can wait for the queue to flush out without any new arrivals, and repeat the same pattern.) The proof idea is to first establish that within the first couple of writes, algorithm A must write out a number of tuples p which is at least $2x+1$, even though the size of the queue exceeds the memory buffer size by only x; otherwise, A's competitive ratio cannot be less than 2. Then, we will show that having written excessively to disk, algorithm A can be forced into a situation where it performs much worse than an optimal offline algorithm, and hence is not better than 2-competitive.

At time τ, suppose the online algorithm writes out 3 tuples, we are done since we have established our first goal with $x = 1$. Suppose, however, that the online algorithm writes at most 2 tuples at time τ. In the next instant, at time $\tau + 1$, suppose 3 new tuples arrive into the queue. Given that the algorithm had made a smaller write earlier, it will be forced to write some tuples at this time. If the online algorithm now writes out 7 tuples, again we are done with our first goal. Otherwise, we inject 7 new tuples into the queue at time $\tau + 2$ which is just enough to force a third write. At this point we stop giving input. The online algorithm has now made 3 writes and will need at least 1 read. In contrast, an optimal offline algorithm for this arrival pattern would have written more tuples at time τ, and would not need any further writes, although it would need to perform a single read later. Since we could wait till the queue is flushed out and then repeat the entire arrival pattern, it follows that the online algorithm is not better than 2-competitive, a contradiction.

We still have to consider the two scenarios: 1) where at time τ, the online algorithm wrote at least 3 tuples onto the disk; and 2) where at time τ the online algorithm wrote at most 2 tuples, but when at time $\tau+1$ there was an arrival of 3 new tuples, it wrote at least 7 tuples onto the disk. We will extend the arrival pattern in both cases such that: in the first scenario, an optimal offline algorithm would have written exactly 1 tuple at time τ; and, in the second scenario, an optimal offline algorithm would have written exactly 3 tuples at time τ. In either case, we have arranged a situation where the online algorithm writes out p tuples but an optimal offline algorithm writes out $\lfloor \frac{p-1}{2} \rfloor$ tuples only. Note that in the second scenario the online algorithm has performed 2 writes, while the optimal offline algorithm performs only 1 write in either case. In what follows, we will establish the negative result for the first scenario only, a similar argument works for the second scenario even if we do not charge the online algorithm for the second write it has already performed.

Suppose the arrivals stop at this point and only the queue depletion carries on at each clock-tick. As the online algorithm has written out more, it will have to perform a read before one is required by the offline algorithm. If the online algorithm reads in the p tuples in 3 or more read operations, its competitiveness is no better than 2. Therefore, it must use at most 2 reads and in one of them read in at least $\frac{p}{2}$ tuples. Just after this large read, the online algorithm has more tuples in memory than the offline algorithm, regardless of when the offline algorithm performs its read. At this time, inject enough new tuples to just exceed the memory buffer for the online algorithm. The online algorithm is forced to write, but the offline algorithm does not need to do so. The online algorithm must perform a total of 2 reads and 2 writes, as opposed to the single read and single write required by the offline algorithm. After waiting for the queue to empty out, we can repeat the same sequence, and keep doing so indefinitely. Thus, algorithm A cannot be better than 2-competitive. $\square$

Note that the argument for the lower bound of 2 does not need a large number of tuples to enter the queue at any instant. It uses arrival patterns where only $O(1)$ tuples enter the queue at any clock-tick.

3 Single Queue and Extended Cost Model

For large memory buffers the cost of reading/writing t disk blocks is better modeled as $c_0 + c_1 \times t$, with $c_1 \gg c_0$. Under this extended cost model, we provide a 4-competitive algorithm for maintaining a single queue.

To understand the solution for this cost model, consider the natural greedy algorithm which attempts to minimize the number of tuples written to disk. Whenever the memory buffer overflows upon the arrival of new tuples, write the minimum number of tuples to disk needed to correct the overflow; furthermore, for consuming tuples that have been written onto disk, read them in, a single tuple at a time, just when they have to be consumed. Each read can cause a write of at most one tuple. The following theorem is easily verified.

THEOREM 3.1. *The greedy algorithm is optimal from the perspective of minimizing the number of tuples written to disk.*

Let $T = c_0/c_1$ and assume that $T < M/2$. Let Algorithm GREEDYCHUNK be the variant of the greedy algorithm which operates on blocks of T tuples, i.e., always reads/writes the minimum possible number of blocks of T tuples.

THEOREM 3.2. *Algorithm* GREEDYCHUNK *is acyclic and 4-competitive, provided $T = c_0/c_1 < M/2$.*

Proof. Since $T < M/2$, the oldest T tuples are never written from memory and upon overflow there are at least T new tuples that can be written. GREEDYCHUNK is hence acyclic.

At any instant of a queue-caching algorithm, let D denote the sets of contiguous tuples on disk and M denote the sets of contiguous tuples in memory. Let us define the state of the algorithm at that instant to be the given by $S = (D, M)$. Note for all algorithms with the same memory buffer size and for the same tuple arrival history, $D \cup M$ is the same at a given instant and is the set of unconsumed tuples. Given two algorithms, A_1 and A_2 in states S_1 and S_2 respectively, we say $S_1 = (D_1, M_1)$ subsumes $S_2 = (D_2, M_2)$ if there is an injection from D_1 to D_2, so that the preimage of contiguous tuples are contiguous tuples, and every tuple is mapped to an older tuple. It can be easily shown that if S_1 subsumes S_2 and if A_2 pays cost c for its future arrival pattern, there exists an algorithm that starts at state S_1 and pays cost at most c for the same future arrival pattern. We can relax the subsumption property for an acyclic algorithm A so that D_1 has extra t tuples, the newest t in D_1, which need not participate in the function (or the function values on these points are some constants) if the cost of reading these extra tuples has been accounted for. Non-contiguity in the preimage can also be allowed if the overhead of a new read (c_0) has already been accounted for each non-contiguity.

Suppose now that the optimal offline algorithm performs a write of $aT + b$ tuples, where $a, b \in \mathcal{N}$ and $0 \leq b < T$. This will cost the offline algorithm at least $(a+1)c_0$. For the same set of tuples, Algorithm GREEDYCHUNK will need at most $a + 1$ writes of T tuples each, with cost $2(a+1)c_0$ (which is within factor 2 of the offline cost) to be in a state that subsumes the optimal offline if you do not consider the non-contiguity and $T - b$ (newest) extra tuples in the last write. However, since GREEDYCHUNK will read in blocks of T tuples at a time, reading in more tuples than the optimal offline algorithm may cause it to have more tuples in the buffer than the offline algorithm at some intermediate instants. This may cause GREEDYCHUNK to perform an extra write of a block of T tuples, due to incoming tuples (so the subsumption property is broken by at most this extra batch of T tuples written out by GREEDYCHUNK), at an additional cost of $2c_0$ for the write and $2c_0$ later for reading back these tuples when they need to be consumed. But this single extra write by GREEDYCHUNK will indemnify it from further writes caused and there will be no further breakage of the subsumption property within this batch of write-outs due to larger reads of tuples corresponding to the $aT+b$ tuples in this batch. GREEDYCHUNK has T fewer tuples in the memory buffer as compared to the offline algorithm after this extra write. Note that this extra space of T can hold the further reads of size T in this batch. The cost paid by the online algorithm for reads and writes is thus at most $2(2a+4)c_0$, while the optimal offline algorithm pays at least $2(a+1)c_0$. This argument holds for every read done by offline. Thus every read-write pair that costs offline at least $2(a + 1)c_0$ can cost GREEDYCHUNK at most $2(2a + 4)c_0$. This implies 4-competitiveness of GREEDYCHUNK. $\square$

It can be shown that the competitive ratio for Algorithm GREEDYCHUNK has a lower bound of 4, using techniques from Theorem 2.2.

THEOREM 3.3. *For $T \geq M/2$, Algorithm* HALF *is 4-competitive under the extended cost model.*

Proof. We have already seen that Algorithm HALF performs at most twice as many reads and writes as compared to any offline algorithm. Furthermore, since it never reads or writes more than $M/2$ tuples at a time, under the extended cost model, for each read/write this algorithm pays at most $c_0 + c_1 \times M/2$ which is bounded by $2c_0$ for $T \geq M/2$. Since any algorithm has to pay at least cost c_0 for each read/write it performs, this implies the 4-competitiveness of Algorithm HALF. □

COROLLARY 3.1. *There is an acyclic 4-competitive online algorithm for the extended cost model.*

Proof. The following hybrid algorithm suffices: Use Algorithm HALF when $T > M/2$, and use Algorithm GREEDYCHUNK otherwise. □

Note that for current disk technology, the value of T lies between 100,000 and 2,000,000. Our results above suggest that we must read/write blocks of size much larger (100KB-2MB) than the standard disk-block size of 4KB in order to minimize the access time costs due to seeking. Conventionally, smaller blocks are used to avoid memory wastage due to fragmentation. But in data stream systems, bigger blocks are more suitable. Our results are corroborated by the observations[3] of Patterson and Gray [13] on how disks of the future should be looked upon as sequential devices and not random access devices.

4 Multiple Queues

We now consider the general version of the problem in which the memory buffer of size M is shared by n independent queues. The cost model is once again the sum of the number of reads and the writes, i.e., the unit-cost model. As discussed in Section 1, a single read/write can only involve *contiguous* tuples from a *single* queue. We will assume, as before, that the input to the various queues is adversarial. The new issue in the case of multiple queues is the choice by the scheduler as to which of the queues' head is to be consumed next. Round-robin schedulers cycle through the queues, consuming the oldest tuple in each queue in turn, skipping over any empty queues. This model makes sense when the goal is to provide fair and equitable service to each queue. Under round-robin schedulers, we provide a $2n$-competitive, acyclic algorithm and also provide a lower bound of $\Omega(\sqrt{n})$ on the competitive ratio of any acyclic online algorithm. We also consider adversarial schedulers, which at each clock-tick picks up an arbitrary nonempty queue and consumes the oldest tuple from that queue. Here we establish a negative result that it is not even possible to have an $o(M)$-competitive acyclic algorithm. However, if we allow the online algorithm an extra $M/2$ units of memory (i.e., the online algorithm has memory size $3M/2$ while competing with an offline algorithm with memory size M), we can devise an online algorithm which is $O(n)$-competitive.

We begin by pointing out that we cannot use the obvious approach of statically allocating M/n units of memory to each queue and running an algorithm such as HALF for each queue. To see this, consider the case when essentially all the tuples in the system belong to a specific queue and the number of unconsumed tuples of that queue is close to M, say $M - 1$, by having a new tuple arrive every clock-tick when a tuple is consumed. In this case the optimal offline algorithm allocates the entire memory to the active queue and has cost 0, while the online algorithm, which allocated only M/n memory to that specific queue will keep making writes and reads, and will be unable to provide any reasonable competitive ratio.

4.1 Algorithm BufferedHead We first describe an acyclic algorithm, BUFFEREDHEAD, and then show that it can be used to derive the positive results for the two types of schedulers discussed above. Our algorithm will have a *protected head*, i.e., some fraction of the oldest tuples from every queue will never be written out to disk to make space for the incoming tuples. We maintain a protected head of size up to $M/2n$ for each queue in the system.

Algorithm BUFFEREDHEAD is a generalization of Algorithm HALF and works as follows. When tuples enter the system they are placed in their respective queues. The memory is not statically partitioned between the queues and the queues dynamically change in size as tuples enter and leave. As before, we will only analyze a simple version[4] of BUFFEREDHEAD

[3] Patterson and Gray [13] mention that disks providing 200 accesses per second, each involving data blocks of size a few KB, will give a throughput of few MBps and this will take a year to read a 20-Terabyte disk of the future. On the other hand, sequential scans avoiding random seeks will give 500 times more bandwidth, making it possible to read the entire disk in one day. This is because disk density and capacity is increasing much more rapidly than their access speeds.

[4] As discussed for Algorithm HALF in Footnote 2, there are modified versions of BUFFEREDHEAD that perform more relaxed form of writes. At a clock-tick, if there is excess input that does not fit into the available memory, select the largest queue in memory, and write all but its $M/2n$ oldest tuples to disk, or say instead write-out the newest half of the largest queue in memory

which writes exactly $M/2n$ of the newest tuples of the largest queue in memory when there is a memory buffer overflow (by our arrival rate assumption, it never needs to write more than these many tuples). It also reads tuples from disk in chunks of size $M/2n$. Note that in both versions of this Algorithm, read-ins may also cause write-outs to make space for the tuples being read-in.

THEOREM 4.1. *Algorithm* BUFFEREDHEAD *is acyclic.*

Proof. The oldest $M/2n$ tuples of any queue are never written-out to disk in all versions of BUFFEREDHEAD. Since tuples read-in from disk will always be the oldest tuples in their queue, and read-ins are done in chunks of $M/2n$, it follows that no tuple is written-out and then read-in back more than once. □

We analyze this algorithm for the two kinds of schedulers in the following subsections.

4.2 Round Robin Schedulers Under round robin schedulers, the oldest tuple of each queue is consumed before the second oldest of any queue. This can be modeled as having to consume the oldest tuple of *every* nonempty queue at *each* clock-tick, which requires having the oldest tuple of each queue in memory at every clock-tick. Note that under round-robin schedulers, the $M/2n$ tuples read-in for any queue will be among the next $M/2$ tuples consumed over all queues, as only at most $M/2n$ from each of the n queues will be consumed before them. Also note that, for every batch of $2n$ writes by BUFFEREDHEAD atleast M new tuples would have entered the system, as those written-out tuples that were read-in are now in the *protected head*.

THEOREM 4.2. *Algorithm* BUFFEREDHEAD *is a* $2n$*-competitive algorithm for round-robin schedulers.*

Proof Sketch. We give only a brief sketch of the proof here, deferring the detailed proof to the extended version of the paper. The intuitition is that, as write sizes are at least $M/2n$, at most a $2n$ factor is lost for larger writes performed by the optimal offline algorithm (they cannot exceed M in size). For smaller writes by the optimal offline algorithm, BUFFEREDHEAD would have to do an early read, but this can cause at most one extra write after which the memory buffer for the online algorithm would have newer tuples on disk as compared to the offline algorithm. The detailed proof uses a natural generalization of the subsumption property (see Theorem 3.2) to multiple queues to show for every batch of $2n$ write-outs performed by BUFFEREDHEAD, any offline algorithm would have to perform atleast one write-out. Reads as usual can be accounted to writes or analyzed similarly.

□

While we do not have a matching lower bound, the following indicates that BUFFEREDHEAD may not be far from the best possible acyclic algorithm.

THEOREM 4.3. *Under round-robin schedulers, the competitive ratio of an acyclic online algorithm must be* $\Omega(\sqrt{n})$.

Proof. Let tuples be provided as input only to queue 1 at the maximum possible rate, until the number of unconsumed tuples reaches $2M$. We assume that the rate is large enough that the optimal offline algorithm will incur only $O(1)$ writes. At this time, any algorithm should have at least M tuples on disk. Stop giving any input now, until the algorithm performs a "big" read, as described next. Clearly, a $\sqrt{n}$-competitive algorithm must perform at least one read of size $M/\sqrt{n}$ if no new input tuples arrive, since an offline algorithm could read in chunks of $\Omega(M)$ tuples. Once this big read has been performed, inject new tuples at the maximum possible input rate so as to add M/n new tuples into each queue, for a total of M tuples. Since the online algorithm is acyclic, it cannot write back the $M/\sqrt{n}$ tuples from the big read back to disk, so it will have to write-out at least $M/\sqrt{n}$ of the arriving tuples which amounts to $\sqrt{n}$ queues, as each queue has size M/n. The optimal offline algorithm, foreseeing this sudden input spurt, would have read-in only $M/2n$ tuples of queue 1, just before this new batch of M tuples arrived. Thus, the offline algorithm can have $O(1)$ cost, while any online algorithm will have cost $\Omega(\sqrt{n})$. □

4.3 Adversarial Schedulers We first show that there are no $o(M)$-competitive acyclic algorithms under adversarial schedulers. We then use Algorithm BUFFEREDHEAD to show that if we give the online algorithm a larger memory buffer size vis-a-vis an offline algorithm, then it is an $O(n)$-competitive algorithm.

Since $M \gg n$, we are interested only in competitive ratios that are bounded independent of M. The following result shows that under adversarial schedulers, we cannot achieve this goal.

THEOREM 4.4. *Under adversarial schedulers, there is no acyclic online algorithm with competitive ratio* $f(n)$*, for any bound* $f(n)$ *independent of* M*.*

to disk. When tuples have to be read from disk for consumption in a particular queue, read $M/2n$ tuples for that queue, or the entire portion of that queue on disk, whichever is smaller. We defer the analysis of these forms of the algorithm to the extended version of the paper.

Proof. Let tuples be provided as input only to queue 1 at the maximum possible rate, until the number of unconsumed tuples reaches $2M$. At this point, any algorithm should have at least M tuples on disk. We assume that the rate is large enough that offline will incur only $O(1)$ writes. Stop giving any input now, until the algorithm makes a big read, as described next. For an $f(n)$-competitive algorithm to read-in the M tuples on disk, there must be at least one read of size $M/f(n)$ if no new tuples arrive, since an optimal offline algorithm could read in chunks of size $\Omega(M)$. As the tuples in the system are all from a single queue, the adversarial scheduler is forced to consume only from this queue up to this point in time. Once this big read has been performed, inject tuples for queue 2 into the system, until the number of unconsumed tuples in queue 2 reaches $M - 1$. Also from now on suppose the scheduler only consumes tuples from queue 2, and after this at every instant in the future exactly one tuple for queue 2 enters the system to replenish the tuples of queue 2 being consumed. Thus $M - 1$ unconsumed tuples of queue 2 are constantly being maintained in the system. Observe that the space occupied by the $M/f(n)$ tuples of queue 1, from the large read, can never be reclaimed. Therefore, an acyclic online algorithm will incessantly be performing writes and reads, while an offline algorithm foreseeing this, could have performed a read of just a single tuple for queue 1, just before this deluge of queue 2 tuples began. Since the number of writes and reads of the online algorithm in this case is unbounded, while the offline does not incur any more writes, the competitive ratio of the online algorithm is unbounded. □

Actually, using the preceding argument we can also show an even more negative result as indicated by the corollary below.

COROLLARY 4.1. *If up to k tuples can enter the system at any instant, then no acyclic online algorithm can be k-competitive for adversarial schedulers.*

The reason why a result similar to Theorem 4.2 does not hold for adversarial schedulers is that, under round-robin schedulers, the oldest unconsumed tuple amongst all the queues will be consumed at the next clock-tick and hence has to be in memory. But an adversarial scheduler may not select the queue corresponding to the oldest unprocessed tuple for consumption and therefore it need not be in memory. In other words, it may be possible that at an instant when there are more than M unprocessed tuples, an offline algorithm maintains the newest M tuples in memory and does not pay for any reads/writes.

In spite of the previous result, it turns out that BUFFEREDHEAD is still applicable under adversarial schedulers, provided we compare its performance to that of offline algorithms with smaller memory size.

THEOREM 4.5. *Under adversarial schedulers, Algorithm* BUFFEREDHEAD *is an acyclic $2n$-competitive algorithm when provided with extra $M/2$ memory.*

Proof Sketch. We give only a brief sketch of the proof here, deferring the detailed proof to the extended version of the paper. The protected region (the oldest parts of all queues that cannot be written to disk) occupy at most $M/2$ space and all the writes happen from the unprotected region of size at least M. Just as in Theorem 4.2, between $2n$ writes of BUFFEREDHEAD there must be atleast one write by the optimal offline algorithm. □

We can improve the competitive ratio to n for writes when given memory of total size $3M/2$, by changing BUFFEREDHEAD to write out all except the head ($M/2n$ oldest tuples) of the largest queue in memory, or for a simpler analysis, M/n of the newest tuples from the largest queue in memory. Algorithm BUFFEREDHEAD remains acyclic, but now writes have size at least M/n.

4.4 Extended Cost Model The single-queue algorithm for the extended cost model in Section 3 can be easily extended to maintaining multiple queues under the extended cost model to give a 4-competitive algorithm for round-robin consumption. We defer the details to the extended version of this paper.

5 Conclusion

We studied the problem of maintaining queues in a memory cache, which arises in a number of important applications such as data stream systems, networking, and distributed messaging services. We analyzed why data stream systems built on top of buffer managers that use traditional caching algorithms like LRU perform badly as noticed elsewhere [10]. We provided online competitive algorithms for this problem under different interesting cost models. These algorithms will be implemented in the Stanford Stream system [1].

Acknowledgements

We thank Gurmeet Manku for his comments in improving the exposition of the paper.

References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas,

R. Varma, and J. Widom: "STREAM: The Stanford Stream Data Manager." *IEEE Data Engineering Bulletin* 26(1):19-26, 2003.

[2] Yossi Azar and Yossi Richter. "Management of multi-queue switches In QOS networks." In: *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pp. 82–89, 2003.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. "Models and issues in data stream systems." In: *Proceedings of the Twenty-First ACM SIGMOD Symposium on Principles of Database Systems*, pp. 1–16, 2003.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. "Chain: Operator Scheduling for Memory Minimization in Stream Systems." In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 253–264, 2003.

[5] Brian Babcock, Mayur Datar, and Rajeev Motwani. "Load shedding techniques for data stream systems." In: *Proceedings of the 20th International Conference on Data Engineering*, 2004 (to appear).

[6] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[7] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. "Monitoring streams - A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 215–226, 2002.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, V. Raman, F. Reiss, and M.A. Shah. "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World." In: *First Biennial Conference on Innovative Data Systems Research*, 2003.

[9] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. "NiagaraCQ: a scalable continuous query system for Internet databases." In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 379–390, 2000.

[10] Charles Cranor, Theodore Johnson, Oliver Spatsheck, and Vladislav Shkapenyuk. "Gigascope: A Stream Database for Network Applications." In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 647–651, 2003.

[11] Charles Cranor, Theodore Johnson, Oliver Spatsheck, and Vladislav Shkapenyuk. "The Gigascope Stream Database." *IEEE Data Engineering Bulletin* 26(1):27-32, 2003.

[12] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. "Approximate join processing over data streams." In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 40–51, 2003.

[13] Jim Gray and David Patterson. "Storage: A Conversation with Jim Gray." *ACM Queue* 1 (2003).

[14] IBM Corporation. *IBM MQ Series*. `http://www-3.ibm.com/software/integration/wmq/`.

[15] Sundar Iyer, R.R. Kompella and Nick McKeown. "Analysis of a Memory Architecture for Fast Packet Buffers." In: *IEEE Workshop on High Performance Switching and Routing*, 2001.

[16] Jon Kleinberg. "Bursty and hierarchical structure in streams." In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 91–101, 2002.

[17] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson "On the self similar nature of ethernet traffic." *IEEE/ACM Transactions on Networking*, 2(1):1–15, 1994.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. "Query Processing, Approximation, and Resource Management in a Data Stream Management System." In: *First Biennial Conference on Innovative Data Systems Research*, pp. 245–256, 2003.

[19] Vern Paxson and Sally Floyd. "Wide-area Traffic: The failure of Poisson modeling." *IEEE/ACM Transactions on Networking*, 3(3):226-244, 1995.

[20] Devavrat Shah, Sundar Iyer, Balaji Prabhakar, and Nick McKeown. "Maintaining statistics counters in line cards." *IEEE Micro*, pp. 76–81, 2002.

[21] Sandeep Sikka and George Varghese. "Memory-efficient state lookups with fast updates." In: *Proceedings of the ACM SIGCOMM Conference*, pp. 335–347, 2000.

[22] Nesime Tatbul, Ugur Cetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. "Load Shedding in a data stream manager." In *Proceedings of the 29th International Conference on Very Large Data Bases*, pp. 309–320, 2003.

[23] Walter Willinger, Murad S. Taqqu, and Ashok Erramilli. "A Bibliographical Guide to Self-Similar Traffic and Performance Modeling for Modern High-Speed Networks." In: *Stochastic Networks: Theory and Applications*, F.P. Kelly, S. Zachary, and I. Ziedins (Eds.), Oxford University Press, pp. 339-366. 1996.