Class-based Continuous Query Scheduling for Data Streams

Lory Al Moakar[1] Thao N. Pham[1] Panayiotis Neophytou[1]
Panos K. Chrysanthis[1] Alexandros Labrinidis[1] Mohamed Sharaf[2]

[1]Department of Computer Science, University of Pittsburgh
[2]ECE Department, University of Toronto
{lorym, thao, panickos, panos, labrinid}@cs.pitt.edu, msharaf@eecg.toronto.edu

ABSTRACT

Wireless sensor networks link the physical and digital worlds enabling both surveillance as well as scientific exploration. In both cases, on-line detection of interesting events can be accomplished with continuous queries (CQs) in a Data Stream Management System (DSMS). However, the quality-of-service requirements of detecting these events are different for different monitoring applications. The CQs for detecting anomalous events (e.g., fire, flood) have stricter response time requirements over CQs which are for logging and keeping statistical information of physical phenomena. In this work, we are proposing the Continuous Query Class (CQC) scheduler, a new scheduling policy which employs two-level scheduling that is able to handle different ranks of CQ classes. It provides the lowest response times for classes of critical CQs, while at the same time keeping reasonable response times for the other classes down the rank. We have implemented CQC in the AQSIOS prototype DSMS and evaluated it against existing scheduling policies under different workloads.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – Query Processing

General Terms

Algorithms, Design, Performance

Keywords

Data Stream Management System, Continuous Queries, Operator Scheduling, User-centric, Priority

1. INTRODUCTION

On-line monitoring applications are widely enabled today by Wireless Sensor Networks (WSN), in conjunction with Data Stream Management Systems (DSMS). Environmental monitoring is an example of a monitoring application that combines both surveillance and detection as well as scientific exploration and discovery (e.g., [6], [7], [12] and [14]). Individual sensor devices produce simple periodic readings such as temperature, humidity, and sound measurements which are all combined into streams of data. Depending on the underlying setup of the WSN, these streams could have a steady and/or a bursty rate. Environmental monitoring applications compose these individual readings into more meaningful complex events (e.g., fire, flood). At the same time, these applications also perform aggregations and calculate various useful statistics for future reference and analysis. In both cases, the applications' requests can be expressed in terms of Continuous Queries (CQs) running in a DSMS. These CQs have different response time requirements. For example, CQs that detect hazardous conditions require a lower response time than CQs that collect statistics.

Currently, most DSMSs employ a CQ scheduler to optimize the Quality of Service (QoS) provided by the system. In particular, the CQ scheduler is the DSMS component which decides the execution order of CQs to achieve a certain performance goal such as minimizing response time or maximizing fairness. However, current CQ schedulers assume that all CQs are of the same importance. In particular, existing CQ scheduling policies (e.g., Chain [5], Round Robin (RR), Highest Rate (HR) [10], Highest Normalized Rate policy (HNR) [10]) are oblivious to the different importance levels of different CQs and hence, they just optimize for the overall system performance. Looking back at our environmental monitoring example, this means that the critical queries might be dragged down by the statistics gathering queries. For this reason, the system needs to distinguish between classes of queries, for example, highly-critical, critical and normal queries.

To better understand the multi-class CQ scheduling problem, consider the following example where neglecting the criticality of queries can lead to undesirable response times. In our example, we assume two continuous queries: CQ_1 which detects fire conditions in a forest (e.g. high temperature, low humidity, and strong wind) and as such is high-priority and CQ_2 which records the average temperature and humidity measurements for scientific observation purposes and as such is low-priority. It is a well known fact that RR is oblivious to the priorities and treats CQ_1 and CQ_2 equally. Similarly, schemes that optimize for the average response time such as HR or for the average slowdown time such as HNR are oblivious to query importance. Even worse, they may make decisions that contradict the objective of optimizing the performance of the highly-critical queries. For instance, in our example, CQ_1 is highly-selective since fire conditions are rare. Thus, CQ_1 has a low *output rate* which will lead HR to assign CQ_1 a lower scheduling priority than CQ_2. This conflict between a CQ's importance and its scheduling priority is a problem because CQ_1 is obviously more critical that CQ_2.

To the best of our knowledge, only two other systems (Aurora [1] and RTSTREAM [13]) consider the case of queries associated with importance or weights and incorporate the importance into scheduling decisions. The Aurora scheduler requires the user to provide a soft deadline after which the QoS provided to the query significantly decreases. The RTSTREAM scheduler requires the user to provide a hard deadline after which the query instance is discarded.

In this paper, we focus on system QoS metrics that do not require the user to have any prior knowledge about the query processing requirements or deadlines. We also attempt to support all the high-priority queries without starving low-priority queries and without admission control. Toward this, we have developed a Continuous Query Class (CQC) Scheduler that combines both the RR and HR schedulers.

Specifically, the contributions of this work are :

- We designed a two-level operator scheduler called Continuous Query Class (CQC) Scheduler that considers query classes to offer differentiated levels of response times.
- We implemented the CQC scheduler on our prototype implementation of the AQSIOS stream management system (which is built on top of STREAM [2]).
- We evaluated the CQC scheduler in a scenario with three classes namely Highly-Critical (class H), Critical (class C) and Neutral (class N) which represent hazardous, anomalous, and exploration events, respectively.

Road Map: The rest of the paper is organized as follows: Section 2 surveys the related work. We discuss our system model in Section 3. Section 4 describes the CQC scheduler. We evaluate the system in Section 5. Finally, we conclude in Section 6.

2. RELATED WORK

Efficient CQ scheduling has been one of the main techniques for improving the performance of a DSMS. This motivated the proposal of several policies for scheduling the execution of CQs in a DSMS with the objective of optimizing certain performance goals such as minimizing latency [5, 10, 11] or minimizing memory requirements [3, 4].

In this paper, we also focus on the scheduling of CQs in a DSMS, however, our objective is to optimize DSMS performance in the presence of a multi-class workload where CQs belong to different classes according to their importance.

Related to our work on multi-class CQ scheduling is the work on the Aurora project [1, 5] which considers a set of Quality of Service (QoS) functions including a latency-based one. In particular, under such model, each CQ is associated with a QoS function and the perceived quality of service degrades when the output delay is beyond some threshold δ. Given that model, one can argue that we can specify the importance of a CQ by decreasing its δ and/or by increasing the slope of degradation in its assigned QoS function. However, such mapping between classes and QoS functions is expected to be a daunting task. In addition, under the Aurora model, the objective is to improve the overall DSMS performance, whereas in this paper, we focus on mainly improving the performance of critical CQs while still providing acceptable performance to the other less-important CQs in the DSMS.

The work on the RTSTREAM system [13] considers scheduling classes of CQs based on deadlines. In particular, it assigns to each CQ a deadline and uses those deadlines as priorities for CQ instances during scheduling. However, when a query instance is foreseen to miss its deadline, it will be removed from the system and its input data will be discarded. Hence, the RTSTREAM approach aims at increasing the DSMS success rate by satisfying as many deadlines as possible. This is in contrast to our approach where all CQs are executed to completion without discarding any input data. This is under the goal of minimizing the latency of CQs according to their importance.

The work in [9] also considers scheduling multi-class workloads but in the context of e-commerce OLTP transactions in traditional database management systems. Specifically, it divides transactions to classes of different QoS targets and uses those class-based targets to schedule transactions. However, the scheduler is an external module which non-preemptively dispatches a small set of transactions to the database system for execution, where the size of that set is a system parameter. Hence, improving the system performance relies heavily on figuring out the proper setting of the set size. Additionally, under a non-preemptive dispatcher, a highly important transaction might be blocked waiting for a less important one to finish execution first.

In a similar manner to this work, our group has used the approach of two-level scheduling in the context of web-databases. In particular, the work in [8] deals with the problem of scheduling queries and updates in a web-database system in the presence of Quality Contracts, which specify user-preferences for Quality of Service and Quality of Data for each query. The proposed scheduling algorithm (QUTS) involves two separate queues, one for queries and one for updates, and dynamically assigns CPU time to each according to the expected "profit" in the system through the Quality Contracts framework.

Finally, in-network scheduling techniques have been addressed in the literature. These techniques are specific to sensor networks and are orthogonal to our work. Although inspired by sensor networks, the techniques proposed here are more general, as they apply to the processing of streaming data.

3. AQSIOS PROTOTYPE/SYSTEM MODEL

This work is part of the AQSIOS project, in which we build a new generation of DSMS. AQSIOS is based on the STREAM prototype source code [2], which is currently extended to include new scheduling policies and operators. Figure 1 illustrates an overview of AQSIOS. The query op-

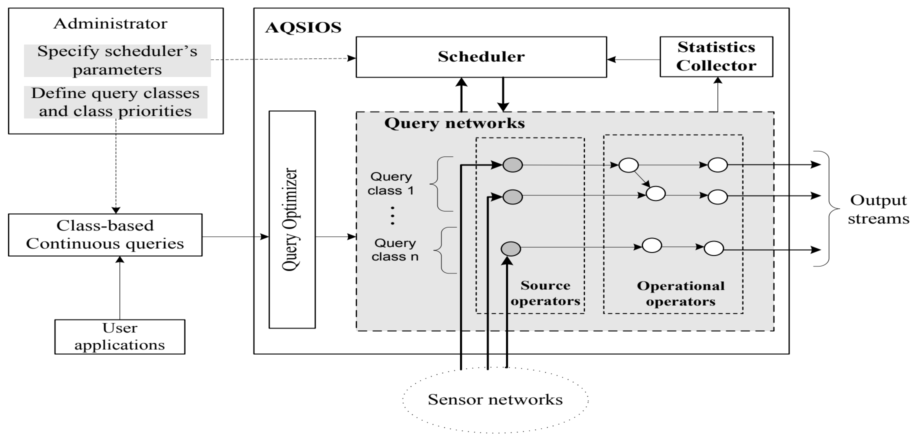

Figure 1: Overview of AQSIOS system

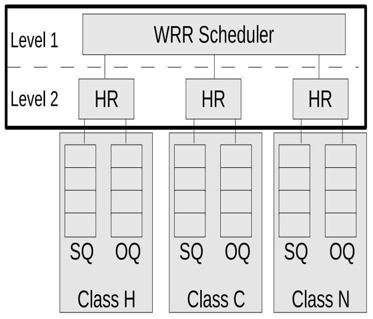

Figure 2: CQC Scheduler

timizer, query execution engine, and a Round Robin (RR) scheduler are inherited from STREAM.

In AQSIOS, all implemented scheduling policies are operator-based, i.e., they consider an operator as the scheduling unit. The scheduling module which sits on top of the query network, selects an operator to run for some specific amount of time before switching to another one. In order to support cost-based scheduling such as HR, we have implemented a Statistics Collector module which collects statistical information about operators' costs and selectivities. For example, HR uses these statistics to set the priority of an operator x to be $\frac{S_x}{C_x}$, where S_x is the expected selectivity and C_x is the expected processing cost of the query fragment downstream from the operator x.

Data streams coming from Sensor Networks are read and translated to an internal representation format by source operators. In the STREAM engine, a source operator (as well as an output operator) is considered part of a query (i.e., database operators) and the query network. As such, source operators are scheduled in the same way as the other operators by the RR scheduler. However, this can result in an unnecessary overhead when a source operator is selected to execute and given the CPU if it has no incoming tuple to read, while some intermediate tuples are still waiting inside the system to be processed. In fact, we have observed that RR's major delays are due to context switching to operators without input tuples. For this reason, in AQSIOS, and in particular under the view of the HR scheduler, source operators are not part of the queries (and query network) and are scheduled separately from the database and output operators which we referred to them as *Operational* operators (see Figure 1). Specifically, AQSIOS groups all source operators together and schedules the source operator set only when there are no tuples waiting in the input queues of the operational operators.

Continuous queries submitted to AQSIOS are classified into several predefined classes of different priorities based on their criticality levels. In this paper, without lost of generality, we assumed only three priority classes. We also assume that there is no sharing of operators across different classes. However, sharing of operators within a class or sharing of synopses is allowed.

4. CONTINUOUS QUERY CLASS SCHEDULER

In this section, we present first the motivation for our two-level scheduler and then the details of our Continuous Query Class scheduler (CQC) which assumes no user-specified QoS requirements (e.g., specific response times or deadlines) and its implementation in AQSIOS.

4.1 Motivation

Our implementation of HR in AQSIOS has confirmed the performance results in [10] that the HR scheduler offers the best average response time in a DSMS. Unfortunately, HR is oblivious to query class priorities and therefore, cannot be used in optimizing the response time of highly critical queries. A simple solution is to extend HR by multiplying the priority of each operator by the priority of its query class. We identified two problems with this extension:

- The original HR scheduler is prone to starve low-priority operators because it schedules high-priority operators first during every round. If high-priority operators always have tuples in their input queues, low-priority operators may never execute. An extended HR might further increase the probability of starvation of those operators if they belong to low class queries. Multiplying their low HR priorities by the low class priority results in a lower relative combined priority and an increased chance of starvation.

- Extended HR is also vulnerable to class-priority inversion. An operator of a query in a higher class H does not always have higher probability to execute than an operator of query of a lower class N. For example, consider two operators Op_1 and Op_2. Op_1 has a low output rate R_1 but belongs to a query in class H with priority P_H. Op_2, on the other hand, has a high output rate R_2 but belongs to a query in class N with priority P_N. Given that the order of execution of Op_1 and Op_2 depends on both their output rates and query class priorities, Op_2 will have higher priority to execute if $R_2 > (P_H/P_N) \cdot R_1$ or Op_1 will have higher priority if and only if $R_2 < (P_H/P_N) \cdot R_1$.

Another alternative is to extend Round Robin (RR) which

Algorithm 1 CQC Scheduling Algorithm: Level 1

INPUT: a set of HR schedulers where each scheduler is responsible for a set of all operators belonging to a specific class.

```
Setup Phase
  1. Sort the scheduler set in decreasing order of class priorities.
  2. For each scheduler $i$, calculate its ideal time slice
     ($T_i = P_i \times k \,/\, \sum_{j=0} P_j$) and its quota ($c_i = T_i$).
Execution Phase
  1. Select the next scheduler $i$ from the scheduler set.
  2. If $i$ has nonpositive quota $c_i$, then Step a, else Step b.
     (a) Add $T_i$ to $c_i$. Go to Step 1.
     (b) Schedule $i$ to run for $c_i$ time units.
  3. After $i$ returns control, determine the number of time
     units ($tu$) it executed for. If $tu \leq c_i$, go to Step a, else
     Step b.
     (a) Reset $c_i$ to $T_i$. Go to Step 1.
     (b) Set $c_i = T_i - (tu - T_i)$. Go to Step 1.
```

already prevents starvation but it does not optimize the performance of highly-critical queries. RR can be extended into a Weighted Round Robin (WRR) scheduler that uses the class priorities to order the operators and to determine their time slices. This extension prevents starvation while optimizing for high-priority query classes. However, WRR does not distinguish between operators in the same class, and thus, does not optimize the average response time within a class as would have been the case of HR.

4.2 Proposed Solution

The last observation above suggested the idea of integrating the WRR and HR schedulers to support class-based continuous query scheduling for data streams. Specifically, we propose a two-level scheduler, namely, Continuous Query Class (CQC) scheduler (Figure 2), that combines the WRR scheduler (level 1) and the HR scheduler (level 2).

Level 2 consists of a set of HR schedulers. Each HR scheduler is responsible for a set of operators that belong to a specific class. In AQSIOS, each HR class scheduler is implemented in a way that distinguishes between source and operational operators and schedules them independently according to their input queues (SQ and OQ in Figure 2). Recall that HR is designed to minimize the average response time. At each scheduling point, an HR class scheduler first sets the priority of operational operators and switches to schedule the source operators only when all the operators in the OQ have empty input queues.

On level 1, a WRR scheduler (Algorithm 1) allocates to each query class i a quota (c_i) equal to a time slice of T_i time units. T_i is the product of the priority of query class i (P_i) and a configurable time period k divided by the sum of all class priorities($\sum_{j=0}^{n} P_j$). For example, in a system with three query classes $\{H, C, N\}$ with priorities of $\{6, 3, 1\}$ and $k = 20$, WRR assigns weighted quotas of $\{12, 6, 2\}$ to each HR class scheduler.

In AQSIOS, WRR does not preempt HR class schedulers but uses a negative credit system (Steps 2 and 3 in the execution phase of Algorithm 1). If any HR class scheduler i exceeds its quota (c_i), WRR deducts the excess amount from its future quotas. This reduces the effect of heavy loaded low-class N queries on high-class H queries. However, unused time quotas do not carry on. If an HR class scheduler is waiting for input tuples, it returns control to WRR after polling its SQ twice.

The CQC scheduler isolates the operators in different query classes. This guarantees a time-slice $k \cdot P_i / \sum_{j=0} P_j$ for operators in class i during each round. Hence, the isolation of operators among classes reduces the probability of starvation. In our example, high-priority operators in classes H and C, cannot starve low-priority operators in class N. The latter compete only with high-priority operators in their own class. Priority class isolation also guarantees a high probability of execution for all queries in class H because their operators do not compete with high-priority operators in class C or N queries. That is, CQC ensures no operator class priority inversion.

System Parameters	
k (time period)	1000 time units
Time unit	1 micro second

Query Load Specifications	
Number of queries	21
Number of query classes	3
Types of queries	Select, Aggr, 2-way Joins
Window Size	10 time units
Selectivity of Selections	$[0.25 - 1]$

Data Stream Specifications	
Humidity (int)	Uniform $[0 - 100]$
Temperature (int)	Uniform $[0 - 40]$
Location (12 chars)	20 locations
Number of input streams	27
Tuple arrival rate/stream	1500 tuples/second
Number of tuples/stream	10,000

Table 1: Experimental Setup

5. PERFORMANCE EVALUATION

5.1 Experimental Setup

To evaluate our proposed solution, we implemented the HR and CQC schedulers in AQSIOS DSMS. AQSIOS is based on the STREAM source code, but is extended to include scheduling policies, and load shedding operators (that are not described in this paper). We compared the performance of CQC to both HR and the STREAM DSMS Round Robin (RR) scheduler. Under RR and HR, each operator processes all the tuples in its input queue before returning control back to the scheduler. Under HR, the operators are scheduled in nonincreasing order of output rate. Table 1 shows the configuration parameters of the system and the workloads.

Workloads: We evaluated the CQC scheduler using three

Workload A			
	Class H	Class C	Class N
Number of queries	7	7	7
Types of queries	all	all	all
Priorities	6	3	1

Workload B			
	Class H	Class C	Class N
Number of queries	2	8	11
Types of queries	join	all	all
Priorities	3	2	1

Workload C			
	Class H	Class C	Class N
Number of queries	2	8	11
Types of queries	join	all	all
Priorities	6	3	1

Table 2: Workloads

workloads: A, B and C. We also ran the HR scheduler with three additional workloads A_H, B_H and C_H that only contain class H queries out of workloads A, B and C, respectively. All workloads consist of nine aggregate queries, six 2-way join queries and six selection queries. We have experimented with various query sets and we chose this one because it brings the system to a highly loaded state without overloading it. Table 2 shows the number of queries and the priorities of each query class under each workload. The queries in workload A are identical in each class while workloads B and C have more queries in classes C and N. Workload B and C have an identical query load and differ only in the priority of the CQ classes. We chose the queries in workloads B and C to satisfy the assumptions made concerning the applications we are considering (i.e. critical monitoring and scientific exploration).

Data Input: We assume a wireless sensor network bandwidth of 250Kbps [14]. Our tuples consist of 3 attributes: location (12 characters), humidity (int) and temperature (int) measurements. The network was simulated by injecting the system with 10,000 tuples per input stream as read from a file, at the highest assumed network bandwidth (1500 tuples/second). We simulate no operator sharing across query classes by duplicating shared input streams and source operators.

Metrics: We measure the average response time for each workload query class and scheduler.

Platform: We ran AQSIOS on a Dell Inspiron E1405 laptop with Intel Core Duo processor T5500 @ 1.66 GHz with 1 GB of RAM running Fedora 10 with the 2.6.27 Linux kernel. AQSIOS executes on only one core of the CPU. We measure response times using the PROCESS_CPUTIME clock, which measures only the process's active execution time.

5.2 Experimental Results

Figure 3 compares the average response times of class H queries for all schedulers on workloads A, B, and C (Table 2) and for the HR scheduler on workloads A_H, B_H, and C_H. In the latter case, class H queries are running by themselves, i.e. there are no class C or N queries. We introduce these workloads to compare with the ideal case. When executing queries of all classes, the CQC scheduler improved the average response time of class H queries as compared to the HR scheduler by a factor of at least 9.4 (the y-axis in Figure 3 is in logarithmic scale). Class H queries running by themselves have an even lower response time because the system is under-loaded.

Figure 4 shows the average response times for workload A query classes under all schedulers. As expected, the CQC scheduler improves the average response time of class H queries, but impairs the response time of class N queries. Class C queries have a higher average response time for this workload because they constitute one third of the system queries and they have a high output rate. As a result, when their HR scheduler executes, class H queries always have tuples and do not return before their quota T_H ends.

CQC improves the average response times of class H and class C queries for both B and C workloads (Figures 5 and 6), but impacts negatively the performance of class N queries. In contrast with workload A, class H queries are fewer, so they do not affect the performance of class C queries in workloads B and C, which are a set of more realistic workloads for the motivating applications we are considering. The response time of class N queries under workload B is lower than under workload C because they are assigned a lower relative priority (1/10 vs. 1/6) under workload C than workload B. Overall, our scheduling policy does extremely well in improving the response time of the most important queries, and penalizes the data gathering queries whose results do not need to be handled immediately.

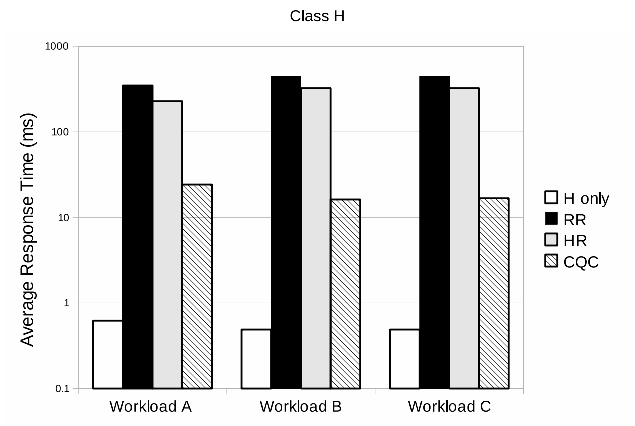

Figure 3: Average Response Time for Class H queries (y-axis is in log-scale). "H only" are the results as the queries of class H were ran by themselves under HR. Those are compared against the response time of class H on the scheduling algorithms as was ran with the corresponding workload mixture.

6. CONCLUSIONS

In this paper, we considered scheduling multiple continuous query classes with different priorities. We developed a new scheduling policy that optimizes the average response time of queries in high-priority classes. Our scheduler consists of two levels. The lower level which schedules queries within a class is using the Highest Rate policy. The top level which schedules which class to run is using the Weighted

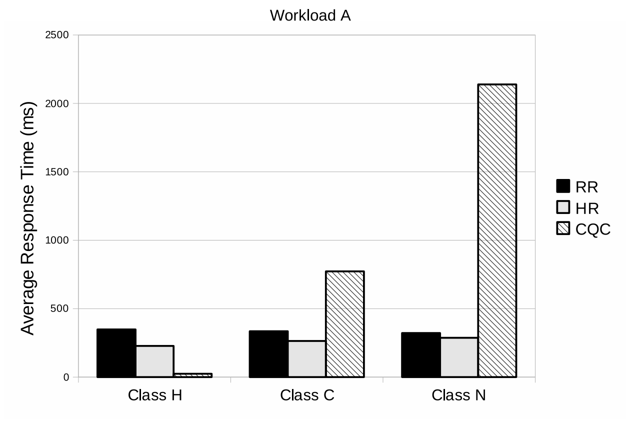

Figure 4: Workload A: All classes have equal query load. CQC provides 9.4 times improvement on class H, compared to HR.

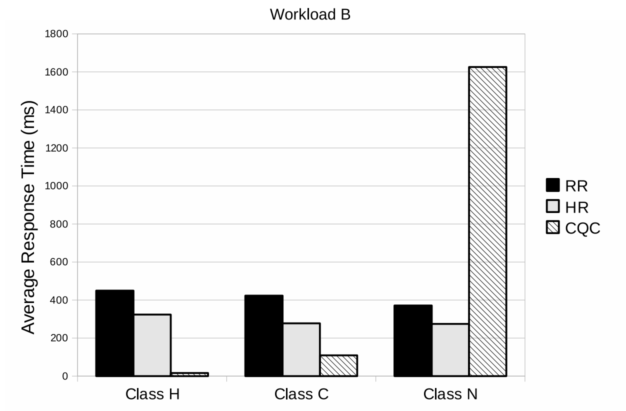

Figure 5: Workload B: Class N has more queries. Class H a couple important join queries. CQC provides 19.8 times improvement on class H, compared to HR.

Round Robin policy. We implemented and evaluated our scheduling scheme on our DSMS prototype code-named AQ-SIOS which is built on top of the STREAM DSMS. We showed that our scheduler improves the response time of critical queries in practical workloads. For class H (highly-critical) by a factor of at least 19 and class C (critical) by a factor of at least 2.5.

In the future, we will modify our scheduler to handle shared operators, first the source operators and then other common operator prefixes within the query plans. Having the prototype system will help us to easily extend our current techniques and explore more constrained and heavily-loaded environments.

Acknowledgements: This research was supported in part by NSF grant IIS-0534531 and NSF career award grant IIS-0746696.

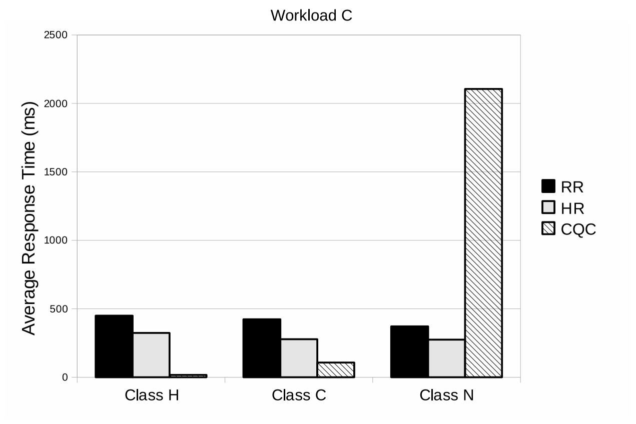

Figure 6: Workload C: Same set of queries as Workload B. Priorities differ by giving higher priority to class H than classes N and C. CQC provides 19.3 times improvement on class H, compared to HR.

7. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[3] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):846–860, 2004.

[5] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, 2003.

[6] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. *IPSN*, 2007.

[7] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *ACM MobiSys*, 2004.

[8] H. Qu and A. Labrinidis. Preference-aware query and update scheduling in web-databases. In *ICDE*, 2007.

[9] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum. Achieving class-based qos for transactional workloads. In *ICDE*, 2006.

[10] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, 2006.

[11] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Transactions on Database Systems*, 2008.

[12] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *ACM SenSys*, 2004.

[13] Y. Wei, S. H. Son, and J. A. Stankovic. Rtstream: Real-time query processing for data streams. In *ISORC*, 2006.

[14] Zigbee specification 053474r06, version 1.0, 2004.