2012 IEEE 28th International Conference on Data Engineering Workshops

Adaptive Class-Based Scheduling of Continuous Queries

Lory Al Moakar, Alexandros Labrinidis, Panos K. Chrysanthis

Computer Science Department, University of Pittsburgh
{lorym, labrinid, panos}@cs.pitt.edu

***Abstract*— The emergence of Data Stream Management Systems (DSMS) facilitates implementing many types of monitoring applications via continuous queries (CQs). However, these applications usually have different quality-of-service requirements for different CQs. In this work, we are proposing the Adaptive Broadcast Disks (ABD) scheduler, a new scheduling policy which employs two-level scheduling that can handle different ranks of CQ classes. The ABD scheduler optimizes the weighted average response time of the CQ classes while still preserving the relative importance of each class. We demonstrate that ABD outperforms state-of-the-art schedulers and adapts to changes in the workload without manual intervention.**

I. Introduction

Motivation: Data Stream Management Systems (DSMS) are at the heart of any monitoring application. Users submit monitoring requests in the form of Continuous Queries (CQs) which could run indefinitely. In many applications, some of the CQs submitted to the DSMS are more critical than others. For example, a CQ that detects flooding inside a warehouse is very critical and requires short response time whereas CQs that report average conditions are not as critical.

A traditional DSMS treats all the CQs as having equal priority in the system and attempts to optimize their overall performance. In particular, it employs a CQ scheduler to optimize the Quality of Service (QoS) provided by the system. 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. Although there have been multiple scheduling policies proposed (e.g., Round Robin (RR), Chain [1], Highest Rate (HR) [2], Highest Normalized Rate (HNR) [2]), these are oblivious to the different importance levels of different CQs.

In our previous work [3], we proposed using the Continuous Query Class (CQC) scheduler to schedule queries of different importance by grouping them into query classes, where each query class has a user specified priority value that indicates its relative importance. CQC is a two level scheduler that uses a weighted RR scheduler on the top level with multiple HR schedulers on the lower level. It requires a manual scheduling period K for the weighted RR scheduler. Determining the correct value of K depends on the workload characteristics and is not easy to derive ahead of time. An incorrect value of K could result in starving low priority classes or in the policy degenerating as round robin.

In this paper, we assume that the system is running on a dual-core machine. In order to efficiently utilize both cores, we divide the execution into two parts: One core is used to listen on the network for any incoming tuples and put them in the input queues of the operators. The second core is used to run the CQs registered by the user. The separation between these two modules allows the system to process data tuples as they arrive without waiting for all the older tuples in the system to finish their execution. Also, this model is better suited at our target application because it removes the dependency between critical CQs and non-critical CQs at the input level. However, under this model, CQC no longer works as expected which motivated us to investigate alternative scheduling schemes.

To overcome the problems of CQC, we designed *Adaptive Broadcast Disks scheduler (ABD)*, a novel scheduler that optimizes the weighted average response time of the query classes and respects user priorities by preventing priority inversion. ABD is designed to run with the expectation that an operator can receive tuples at any time during the scheduling cycle. Like CQC, ABD is a two-level scheduler, however, unlike CQC it is not dependent on manual parameters.

Contributions: Specifically, the contributions of this work are:

- We designed a novel two-level scheduler namely Adaptive Broadcast Disks scheduler (ABD) that optimizes the weighted average response time of the CQs while providing better response time for the critical classes.
- Compared to the state of the art on class-based scheduling, this work has the following characteristics:
  - It works in dual-core environments.
  - It is adaptive to changes in the workload and does not depend on parameters that require manual tuning.
  - It does not exhibit priority inversion.
  - It does not exhibit starvation.
- We evaluated ABD under multiple workloads and show that it is adaptive and parameter-free.

Road Map: We discuss our system model in Section II. Section III reviews the CQC scheduler. Section IV describes the ABD scheduler. We present our experimental evaluation results in Section V. Section VI surveys the related work. Finally, we conclude in Section VII.

II. 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 [4], which we have extended to include new scheduling policies (such as CQC, HR) and

978-0-7695-4748-0/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDEW.2012.38

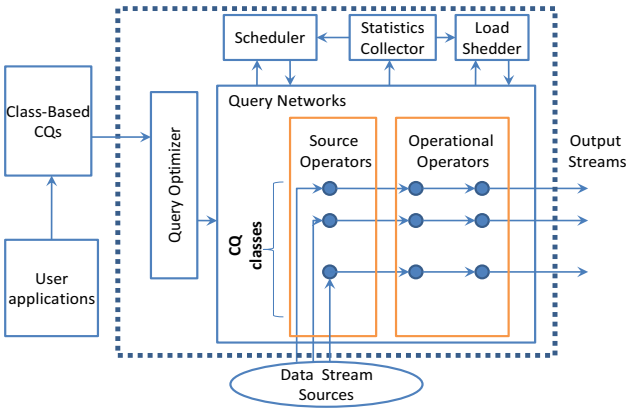

Fig. 1. Overview of AQSIOS

load shedders. Figure 1 illustrates an overview of AQSIOS, in which the query optimizer and the query execution engine are inherited from STREAM.

AQSIOS allows the user to specify in addition to their CQ its importance or criticality in the form of a numerical priority. This priority indicates the relative importance of the results of this CQ relative to the other CQs in the system.

Inside AQSIOS, once the user submits a CQ, the query optimizer translates the CQ into an efficient query plan composed of operators connected via queues. Our system supports three types of operators namely source, operational, and output operators (Figure 1). The source operators read and translate the data streams coming from the data sources to an internal representation format in the form of data tuples. They also record the time the tuple arrived at the DSMS as a timestamp. The operational operators process the data according to the CQ specified by the user. The Join, Select, and Project operators are examples of operational operators. Once the operational operators have finished processing the tuples, the results are put in the input queues of the output operators. The output operators then disseminate the results to the end users. They also calculate the *response time* of the tuple by subtracting the timestamp from the time the tuple was sent to the user.

A. Scheduling Model

The new version of AQSIOS that we are developing would run on two processing cores to better utilize today's machines. Our prior versions were close to the original STREAM prototype and would not utilize multiple cores. In fact, the system would leave one of the cores idle while delaying the processing of the heavy source operators until needed. Under the point of view of the scheduler in the new version of AQSIOS, the source operators are not part of the query network and are scheduled separately from the operational and output operators (see Figure 1). Thus, the source operators are scheduled and run on a separate core than the operational and output operators. A simple round robin scheduler is used to schedule the sources. However, another scheduler is needed to allocate resources to the operators. At every scheduling point, the scheduler selects the next operator to execute and allows it to process its tuples for a specified amount of time/tuples.

The user-specified priorities are used by the scheduler to classify CQs into several predefined classes of different priorities based on their criticality levels. The goal of the scheduler in this case is to optimize the weighted average response time instead of the average response time. In other words, it needs to optimize the response time of the higher-priority query classes without starving the lower-priority query classes and without priority inversion. In this paper, we assume no sharing of operators across different classes. However, sharing of operators within a class or sharing of synopses is allowed. We assume that admission control on the level of the CQs has already taken place (e.g. [5]) and that the load shedding module is disabled, so that we can focus on just the behavior of the scheduling. We have independently considered the synergy between scheduling and load shedding in our previous work [6].

III. State of the art: CQC scheduler

In our previous work [3], we proposed the Continuous Query Class scheduler (CQC) which is composed of two levels: a Weighted Round Robin (WRR) scheduler which is responsible to schedule multiple Highest Rate (HR) schedulers. Each HR scheduler schedules operators that belong to one class. CQC works very well when the system is running on one core. However, it has many side-effects under our dual-core system model.

First, HR is proven to optimize the average response time of a set of CQs when all they have equal priority. However, once it is used in a dual-core environment on the lower level, it has higher chances of starving the operators at the end of the priority queue. *Starvation* happens when the size of the scheduling period is not enough to execute all of its operators in one round. In this case, the semantics of HR's priority queue are violated because there is an interruption of execution when control is switched to another HR scheduler. During this interruption, given that the source operators are executing on another core, new tuples could arrive at the operators at the head of the priority queue. Thus even when the input rate is low, the operators at the end of the priority queue might not execute for many rounds.

Second, we may have *priority inversion* between the query classes where the operators at the head of the priority queues of all the classes could execute more often than the operators at the end of the priority queue of the most critical class.

Third, CQC relies heavily on the size of the scheduling period K which is a manual input to the scheduler. As mentioned previously, the size of the scheduling period allocated to a scheduler is very important. A small value for K could result in *starvation*. A large value of K could result in RR scheduling. Finding the perfect value of K ahead of time is not trivial because it depends on the load in the system which can change during runtime.

Last, CQC increases the wait time and thus the response time of the tuples. All the HR schedulers including the critical

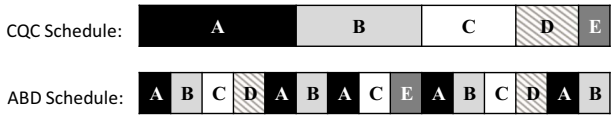

Fig. 2. A comparison between ABD and CQC schedules for a workload with 5 classes A, B, C, D & E with their respective priorities 5, 4, 3, 2 & 1. ABD divides the scheduling period into slices in order to reduce the wait time for A and B.

class HR scheduler have to wait for all the other schedulers to use their allocated time in order to execute. This is especially bad when the load in the lower priority classes is considerably heavier than the load in the critical classes because the latter have to wait a long time in order to execute.

IV. Proposed ABD scheduler

To solve all the above problems with the CQC scheduler, we propose a novel scheduling policy called ABD (Adaptive Broadcast Disks) scheduler. Like CQC, ABD is also a two-level scheduler with one scheduler per class on the lower level and one scheduler on the higher level that determines which class to run next. Similar to CQC, ABD attempts to allocate to each class an amount of time proportional to its priority. However, ABD splits the amount of time allocated to a scheduler into *quota slices* or equal time intervals. In order to reduce the waiting times, ABD builds a schedule so that a scheduler gets a number of quota slices distributed over the entire *scheduling period*.

The ABD schedule is built using binary manipulation where the priority is converted to a bitmap. The bitmap is broken down into individual bits. Then, the bits are compared against pre-generated masks of powers of 2 to determine when to place a class inside the schedule. Figure 2 shows an example schedule under CQC and ABD for a workload with 5 classes: A, B, C, D and E with priorities 5, 4, 3, 2 and 1 respectively. CQC allocated all of the quota slices allocated to class A at once. ABD still gave class A the same amount of time, but reduces its wait time by having it wait for a maximum of 4 quota slices before executing again versus the equivalent of 10 quota slices under CQC.

In order to avoid starving operators within a class, ABD uses a round robin scheduler (RR) on the lower level instead of HR. Each RR scheduler is responsible for scheduling operators in its assigned class. It needs to avoid going over the quota slice by as little as possible. Thus, before it schedules an operator, it estimates the amount of time it would take to execute all the tuples in its queue. If the time left until the end of the quota slice is not enough, then the RR scheduler determines how many tuples the operator can execute so that it doesn't go over the allocated quota slice. In cases when the time left is not enough to process any tuples, then the RR scheduler records which operator it stopped at and returns control back to the ABD scheduler. When the RR scheduler is scheduled next, it starts executing at this operator. This ensures fairness to all the operators in the class and avoids priority inversion between operators at different positions among classes. At the end of its execution, an RR scheduler also records the amount of time it needed to finish executing all the tuples in the last operator's queue.

Dynamic Quota Slice: After running experiments with CQC, we realized that its performance is highly dependent on the size of K. Measuring the workload during runtime is a costly process to determine the correct value. In order to keep the scheduling overhead small, ABD monitors the amount of time each RR scheduler runs for. If RR goes over its quota slice, ABD uses this as an indication that the quota slice is too small and needs to be increased. The quota slice is then incremented by the amount over the quota slice plus the amount of time RR needs to finish all the tuples in the last operator. This new quota slice is then allocated to all the RR schedulers.

However, the workload could fluctuate during runtime and the system could end up with a quota slice that is too big such that the schedule is not used anymore and it is essentially running round robin also on the top level which would be unaware of class priorities. To avoid this situation, ABD attempts to decrease the size of the quota slice if it was not increased in the last schedule. It decreases gradually by the same amount it was last increased with. This insures that the quota slice would adjust in small steps both in increasing and decreasing directions.

Response Time Monitoring: It is important to respect the priorities throughout execution even in the presence of fluctuations in the workload. Towards this, ABD monitors the average response time of each class to detect any priority inversion. Whenever priority inversion is detected, the priorities are adjusted in order to correct it. Priority inversion happens when the priority of class A is larger than that of class B but the response time of class A is higher than that of class B. In this case, ABD would decrease the priority of B if it is greater than 1. If B's priority is equal to 1, then ABD would increment the priority of A by 1. ABD then generates a new schedule. The process of generating a new schedule and detecting priority inversion is costly. Thus, this is done only as needed at the end of the schedule and average response times are used to guard against sudden fluctuations and frequent schedule rebuilding.

V. Experimental Evaluation

A. SimAQSIOS

The newest version of AQSIOS which runs in a dual core environment is still under development. Thus, to evaluate our new scheduler, we developed a simulator called SimAQSIOS using the SimPy Simulation Package [7]. SimAQSIOS models all the operators that are supported by AQSIOS.

The selectivities of the operators are implemented probabilistically. However, the cost of the operators i.e. the amount of time it takes to execute an operator and its tuples as well as the scheduling and the statistics collection overhead is modeled after careful profiling of AQSIOS. The cost of executing an operator takes into account the cost to call the operator's run

TABLE I

EXPERIMENTAL SETUP

Query Load Specifications	
Types of queries	Select, Aggr, 2-way Joins
Window Size	10 micro seconds
Selectivity of Selections	$[0.25 - 1]$

Data Stream Specifications	
Humidity (int)	Uniform $[0 - 100]$
Temperature (int)	Uniform $[0 - 40]$
Location (8 chars)	20 locations

Scheduler Parameters	
BDS initial quota slice	50 micro seconds
CQC scheduling period K	30,000 micro seconds

TABLE II

WORKLOAD SPECIFICATIONS

3A

	Class 1	Class 2	Class 3
Number of queries	7	7	7
Types of queries	all	all	all
Priorities	30	20	10
Input rate (tup/sec)	1600	1600	1600

3B

	Class 1	Class 2	Class 3
Number of queries	2	8	11
Types of queries	join	all	all
Priorities	60	30	10
Input rate (tup/sec)	1500	1500	1500

3C

	Class 1	Class 2	Class 3
Number of queries	11	8	2
Types of queries	all	all	join
Priorities	60	30	10
Input rate (tup/sec)	1500	1500	1500

5D

	Class 1	Class 2	Class 3	Class 4	Class 5
Number of queries	3	5	7	6	4
Types of queries	join & aggr	all	all	all	all
Priorities	50	40	30	20	10
Input rate (tup/sec)	1200	1200	1200	1200	1200

function and set it up, which is the cost the system pays regardless of how many tuples it executes. It also considers the cost of executing each tuple inside the operator. We added fluctuations of +/- 5% on average to all the costs to model real system fluctuations.

We implemented under SimAQSIOS: ABD, CQC, and Weighted HR schedulers. The Weighted HR scheduler (WeHR) assigns to each operator a priority equal to the product of its output rate and the priority of the class it belongs to.

B. Experimental Setup

Tables I and II show the configuration parameters for the schedulers and the workloads.

Scheduler-specific parameters: For CQC, we choose the value for K to be 30,000 so that none of the classes is starving or overloaded. For ABD, we choose an initial quota slice size of 50. We show later in this section that this initial value has no effect on the performance of ABD.

Workloads: We evaluate ABD using four workloads. Workloads 3A, 3B and 3C have three query classes. Workload 3A has the same identical CQs in each class. Workloads 3B and 3C have the same CQs, but the CQs for Class 1 under 3B are assigned to Class 3 under 3C. Workload 5D has five query classes; the heaviest class is Class 3. We choose the input rates for the streams so that the system is loaded, but not overloaded. We choose the workloads so that they have a variety of priorities and load distributions among classes. Table II shows the details of the workloads.

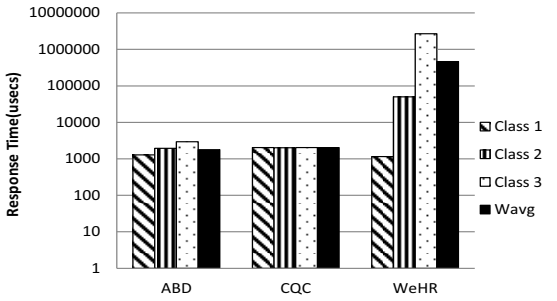

Fig. 3. The weighted average response time (Wavg) for Workload 3A under ABD is lower by 12.16% than under CQC. The response time of Class 1 is 36.6% lower under ABD as well. Under WeHR, Class 1 has a 10% lower response time than under ABD, however, Class 3 starves: its response time is 2.6 sec and Wavg is 466 msecs.

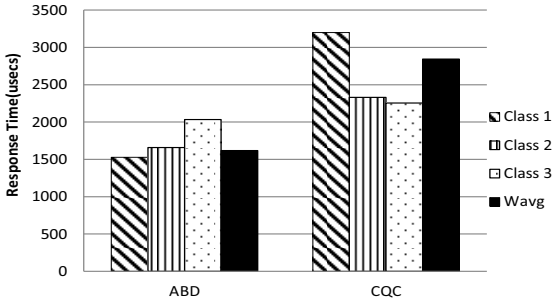

Fig. 4. The weighted average response time (Wavg) for Workload 3B under ABD is lower by 43.1% than under CQC. The response time of Class 1 is 52.2% lower under ABD as well.

Metrics: We report the weighted average response time (Wavg) and the average response time of each class.

Overheads: All of the overheads including the context switches to schedulers and to operators are part of the response time calculation to better compare the schedulers.

Data Input: Our tuples consist of three attributes: location (12 characters), humidity (int) and temperature (int) measurements. The input streams were simulated by injecting the system with 10,000 tuples per input stream.

C. Experimental Results

Comparison of WeHR, CQC, and ABD (Figure 3): Figure 3 shows the performance of the system while running Workload 3A. Notice that the y-axis is in log-scale. ABD lowers the weighted average response time (Wavg) by 12.16% compared

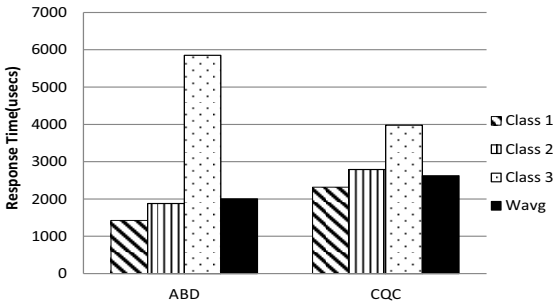

Fig. 5. The weighted average response time (Wavg) for Workload 3C under ABD is lower by 23.7% than under CQC. The response time of Class 1 is 38.6% lower under ABD as well.

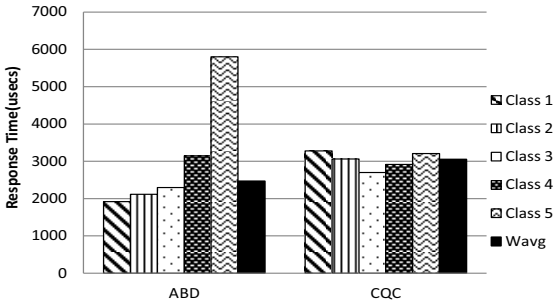

Fig. 6. The weighted average response time (Wavg) for Workload 5D under ABD is lower by 19.1% than under CQC. The response time of Class 1 is 41.5% lower under ABD as well.

to CQC and by 26 times compared to WeHR. Even though WeHR improves the response time of Class 1 by 10%, this comes at the expense of starving Class 3 making its response time over 2 seconds. WeHR has similar performance for all of the workloads we ran, so we omit it from further graphs to improve their readability. The CQC scheduler for workload 3A runs close to how round robin would run in that the response times of the three classes are very close. ABD on the other hand, starting from a very low quota slice adapts to the workload and respects the priorities of the three classes.

No Priority Inversion (Figures 4, 5, 6): Figures 4 and 5 show Workloads 3B and 3C under ABD and CQC. In both workloads, ABD optimizes Wavg (43.1% for 3B and 23.7% for 3C) and improves the response time of Class 1 by 52.2% for 3B and 38.6% for 3C over CQC. CQC results in priority inversion for workload 3B due to the accumulated wait time for Class 1. ABD by slicing the quota reduces the wait time for this class and maintains the relative priorities of the classes.

Figure 6 shows the performance of CQC and ABD while running Workload 5D. CQC again results in priority inversion for classes 1 and 2. ABD preserves these priorities with 5 classes and improves Wavg by 19.1% and the response time for Class 1 by 41.5%.

Sensitivity to Initial Quota Slice (Table III) : We ran Workload 3A under ABD with different initial quota slices.

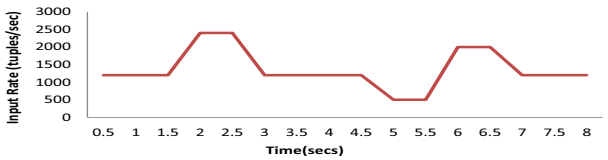

Fig. 7. The input rate used for Class 1 in the adaptive experiment

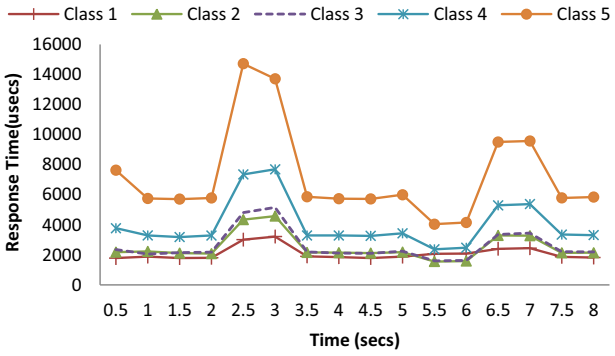

Fig. 8. Adaptive Experiment: shows that ABD adapts as the input rate of Class 1 changes

TABLE III
SENSITIVITY ANALYSIS

Initial Quota Slice	20	50	100	500	1000
Wavg	1749	1788	1776	1775	1768

Table III shows the average Wavg for each quota slice. Wavg stayed more or less constant throughout the runs and the slight differences are due to fluctuations in the system.

Adaptivity (Figures 7, 8) : To evaluate how well ABD adapts to changes in the workload, we modified the input stream for Class 1 under Workload 5D so that it follows the input rate shown in Figure 7. The results are shown in Figure 8. ABD adapts fairly well to the changes in the input rate by keeping the response time for Class 1 lower than all the other classes except for a brief period between 5.5 and 6 seconds. The overall average response times for the classes 1, 2, 3, 4 and 5 are 2.2, 2.46, 2.6, 3.8 and 6.9 msecs respectively. Consequently, ABD preserves the priorities of the classes even in the presence of changes in the workload.

VI. RELATED WORK

Several policies for scheduling the execution of CQs in a DSMS have been proposed to optimize specific performance goals such as latency [1], [2], [8] or memory usage [9], [10].

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. In our previous work [3], we proposed the Continuous Query Class (CQC) scheduler for scheduling CQs of different CQ

classes. However, CQC does not work as expected in a dual-core system. Our experiments show that ABD outperforms CQC without requiring any manual input. In [6], we extended our work on CQC by exploiting the synergy between the scheduler and the load shedder. In this case, the load shedder estimates the load in each class so that the scheduler can adjust the scheduling period accordingly. As part of our future work, we plan to extend the synergy to use ABD.

Related to our work on multi-class CQ scheduling is the work on the Aurora project [11], [1] which considers a set of Quality of Service (QoS) functions including a latency-based one. In particular, under their 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. 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 optimizing the weighted average response time.

The work on the RTSTREAM system [12] 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 is removed from the system and its input data is discarded. This is in contrast to our approach where all CQs are executed to completion without discarding any input data.

The work in [13] 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. 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. However, under a non-preemptive dispatcher, a highly important transaction might be blocked waiting for a less important one to finish execution first.

The work in [14] considers scheduling queries with different priorities in a parallel DBMS. They propose a mechanism to balance the resource allocation for queries based on their priorities. The objective is to allocate to each query a portion of the CPU at least proportional to its priority while maximizing resource utilization.

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 [15] deals with scheduling queries and updates in a web-database system in the presence of Quality Contracts. 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.

VII. Conclusion

In this paper, we considered scheduling multiple continuous query classes with different priorities under a DSMS in a dual-core environment. We developed a new scheduling policy that optimizes the weighted average response time of CQs while improving the response time of critical classes. Our scheduler consists of two levels: the lower level which schedules queries within a class is using a round robin policy, the top level allocates to each class a number of quota slices proportional to its priority and then builds a schedule that determines the execution order of the classes. We implemented and evaluated our scheduling scheme on SimAQSIOS which is built to be as close as possible to our AQSIOS prototype but supports dual-core execution. We showed that our scheduler optimizes the weighted average response time by 12% to 43% compared to the state of the art.

Acknowledgments

This research was supported in part by NSF grants IIS-0534531 and OIA-1028162, and NSF career award grant IIS-0746696. We would like to thank Thao Pham for her help with the AQSIOS prototype and the anonymous reviewers for their feedback.

References

[1] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator scheduling in a data stream manager," in *VLDB*, 2003.
[2] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs, "Efficient scheduling of heterogeneous continuous queries," in *VLDB*, 2006.
[3] L. A. Moakar, T. N. Pham, P. Neophytou, P. K. Chrysanthis, A. Labrinidis, and M. Sharaf, "Class-based continuous query scheduling for data streams," in *DMSN*, 2009.
[4] 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," Stanford InfoLab, Technical Report 2004-20, 2004. [Online]. Available: http://ilpubs.stanford.edu:8090/641/
[5] L. A. Moakar, P. K. Chrysanthis, C. Chung, S. Guirguis, A. Labrinidis, P. Neophytou, and K. Pruhs, "Admission control mechanisms for continuous queries in the cloud," in *ICDE*, 2010.
[6] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis, "Dilos: A dynamic integrated load manager and scheduler for continuous queries," in *SMDB*, 2011.
[7] "Simpy simulation package," http://simpy.sourceforge.net.
[8] 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.
[9] B. Babcock, S. Babu, M. Datar, and R. Motwani, "Chain: Operator scheduling for memory minimization in data stream systems," in *SIGMOD*, 2003.
[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling in data stream systems," *The VLDB Journal*, vol. 13, no. 4, 2004.
[11] 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*, vol. 12, no. 2, 2003.
[12] Y. Wei, S. H. Son, and J. A. Stankovic, "Rtstream: Real-time query processing for data streams," in *ISORC*, 2006.
[13] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum, "Achieving class-based qos for transactional workloads," in *ICDE*, 2006.
[14] F. Narayanan, S.; Waas, "Dynamic prioritization of database queries," in *ICDE*, 2011.
[15] H. Qu and A. Labrinidis, "Preference-aware query and update scheduling in web-databases," in *ICDE*, 2007.