RTSTREAM: Real-Time Query Processing for Data Streams

Yuan Wei Sang H. Son John A. Stankovic
Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22904-4740
E-mail: {yw3f, son, stankovic}@cs.virginia.edu

Abstract

Many real-time applications, such as traffic control systems, surveillance systems and health monitoring systems, need to operate on continuous unbounded streams of data. These applications also have inherent real-time performance requirements that have to be met under high-volume, time-varying incoming data streams. In this paper, we present a real-time data stream query model named PQuery, which provides periodic real-time queries on data streams for the aforementioned real-time applications. To support the PQuery model, a real-time data stream management prototype system named RT-STREAM is developed to provide deadline miss ratio guarantees for periodic queries over continuous and unbounded data streams. We describe the periodic query semantics and discuss why the periodic query model is appropriate for real-time applications. To handle irregular data arrival patterns and query workloads, we propose data admission as an overload protection mechanism. We conduct performance studies with synthetic workloads as well as real workloads from network traffic monitoring applications. The experimental results show that the proposed periodic query model suits the need of the real-time applications and the data admission overload protection approach is effective in managing the workload fluctuations.

1 Introduction

With the emergence of a large number of more powerful, but ever smaller sensors, combined with the improved wired and wireless communication technology, the computing systems nowadays are equipped with great power in monitoring the physical world. As a result, there is a growing need for managing the data generated by the physical sensors. Many new applications need to query on series of sensor data to make proper decisions. For example, in security surveillance systems, a surveillance query may need more than just the most recent sensor readings. It needs to read a series of sensor readings or even correlate data series from several different sensors. Therefore, stream data management capability becomes an essential requirement of these applications.

These applications also have inherent real-time requirements, i.e., their queries on streaming data should be finished within their specific deadlines. Consider the surveillance systems as an example. The system has to alert the controlling party (e.g., human operators) within certain deadlines or the target may be missed.

Real-time query processing on data streams is challenging for several reasons. First, the incoming data streams can be irregular, with unpredictable peaks which may overload the system. When the data arrival rate is too high, the system may become *CPU-constrained* and many queries will miss their deadlines. Second, the execution time of the queries depends not only on the data volume, but also on the contents of the data streams. The queries registered in the databases (especially those with *join* operations) may have execution time that varies dramatically with the contents of their inputs. Third, in addition to being CPU-constrained, the system can also be *memory-constrained*. The total storage space required by all registered queries may exceed the size of the physical memory and the system performance degrades drastically when the intermediate query results are forced to be swapped out of the physical memory.

In this paper, we mainly deal with the CPU time constraints. Addressing the memory constraints in data stream query processing is an important yet complicated research problem. We plan to address the memory issues in future work. In the rest of this paper, we assume that there is always enough main memory space for query processing. The remaining parts of this paper are organized as follows. Sections 2 gives an overview of the data stream management systems. Section 3 introduces the periodic query model and compares it with the existing continuous query model. Section 4 presents our Quality-of-Service (QoS) metrics and our data admission overload protection mechanism. The prototype implementation details are described in section 5. Our performance

```
Stream S (int ID, float value, char[8] type);
Relation R (int ID);

Select avg (S.value)
From S [range 5 second], R
Where S.ID = R.ID and S.type = Magnetic;
```

Figure 1. An Example Query in CQL

evaluation and experimental results are presented in section 6. Section 7 discusses the related work and section 8 summarizes the paper.

2 Data Stream Management System

A data stream is defined as a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamps) sequence of items [14]. Due to the high volume of data streams, it is often assumed that it is not possible to store a stream in its entirety, nor is it feasible to query the whole stream history (often summaries or approximated results are used).

A number of data stream query languages were developed in several projects [18][13][9]. In this paper, we choose the *Continuous Query Language* (CQL) to specify queries on data streams. CQL is a SQL-based declarative language for registering continuous queries against streams and updatable relations. CQL expands the basic SQL with a number of operators that converts data streams to relations and vice versa. We choose CQL because it has a comprehensive set of operators and is more expressive than other existing data stream query languages. More information about CQL and comparisons against other related query languages are provided in [2].

An example of query written in CQL is shown in Figure 1. In this example, there are a data stream S and a relation R registered with the system. The data stream is the sensor reading stream from the sensing system. The data tuple in the stream has three fields: an *ID* field of integer type, a *value* field of float type and a *type* field of string type. The relation R contains all the sensor IDs of a target area. The query operates on data tuples arriving in stream S during the last 5 seconds and calculates the average value of the magnetic sensor readings in the target area during that time period.

3 Real-Time Periodic Query

Many existing data stream management systems support the *continuous query* model [8][2]. In the continuous query model, the query instances are triggered by the incoming streaming data. When a new tuple arrives, a new query instance is initiated and processed. The query results are updated with the results from the newest input. The continuous model performs well when the system workloads are moderate and the system has resources to finish all query instances triggered. However, because the number of query instances and system workloads depend directly on the unpredictable input, it is not appropriate for real-time applications that need predictable responses.

Another drawback of the continuous query model is that the application can not control how often each query is executed. In some applications, some queries are more interesting to the application and thus need to be executed more often. With continuous query model, this can not be easily achieved. Statically allocating system resource to different queries does not work because the query execution time changes with the system input.

We propose the *periodic query* (PQuery) model for real-time applications that need predictable query response. In the PQuery model, once a query is registered with the DSMS system, its query instances are initiated periodically by the system. Upon initialization, a query instance takes the snapshot from data streams as its input. The input does not change throughout the course of the instance execution even when there are newer data tuples in the data streams. Instead, the newly arrived data tuples are processed by the next query instance. Using this semantics, the query execution is not interrupted or aborted by the new incoming data. When an application gets the results of a periodic query instance, it knows that the results reflect the state of the system when the query instance is initiated. In the periodic query model, the query frequencies and deadlines are specified by applications and enforced by the system. Compared to the continuous query model, the workloads of the periodic queries are easier to estimate since at any given time, there are a fixed number of query instances in the system.

4 Query QoS and Overload Protection

Given the fact that the stream data and query execution time are bursty, the system may experience transient overload at run time. When the system is overloaded, QoS adaptations are needed to reduce the system workloads and restore the system to stable states.

4.1 Periodic Query QoS Metrics

In this paper, the quality of stream data query is measured by the following QoS metrics:

- Data Completeness: The data completeness measures the percentage of incoming stream data that are used to compute the query results. Due to the bursty nature of stream data, techniques like *sampling* and *load shedding* [5] [19] may be used to reduce the system workload in case of overload.

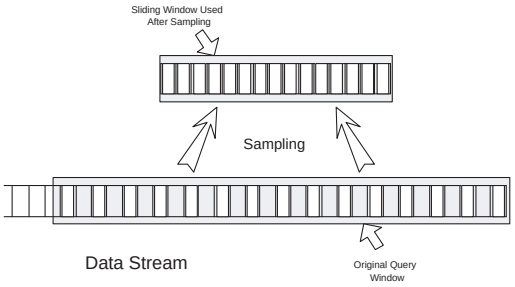

Figure 2. A Data Completeness Example

The data completeness metric quantitatively measures the effects of these operations. An example is given in Figure 2 to show the concept of data completeness. In Figure 2, 15 tuples are sampled from a larger sliding window which contains 30 tuples. In the ideal case, the query should be executed against the larger sliding window with 30 data tuples. Due to the resource constraints, the system can only process 15 tuples. The sampling algorithm selects every other tuple along the data stream and the query is executed on the newly created data window. The sampling process reduces the data completeness of the query to 50%. The data completeness notion applies to both the *raw data* (i.e., the incoming sensor data streams) and the *intermediate results* (e.g., the output of a *join* operator).

- Miss Ratios: Miss ratios measure the percentage of queries that are not finished within their deadlines.

4.2 Data Admission Controller

In many real-time applications, partial results are more desirable than queries missing their deadlines. Therefore, the data completeness may be traded off for better query miss ratios at run time. We propose an overload protection mechanism called *data admission*, which trades data completeness for better query miss ratios. The basic approach is to reduce the incoming data volume when the system is overloaded. Based on the previous research results [5], which indicate that load shedding is more effective when performed at the earlier stages of the query plan, we design our load shedding process to be performed before data stream tuples are processed by the system. Considering that certain data stream tuples are more important than others, we allow the data stream sources to mark their important data tuples with high importance flags. All data tuples with flags are admitted by the system.

We use a Proportional-Integral (PI) controller to control the data admission process as it is simple to use and it provides acceptable response time to workload

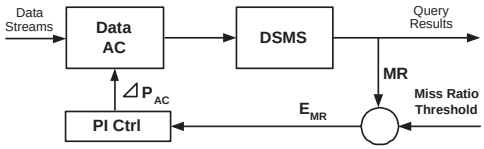

Figure 3. Data Admission Controller

fluctuations. We do not choose proportional-Integral-Derivative (PID) controller because the workloads of query system vary dramatically from one sampling period to another (due to irregular data arrival rate or time-varying query selectivity). Adding a derivative control signal amplifies the random fluctuations in the system workloads. The data admission control architecture is shown in Figure 3. As shown in the figure, the query miss ratios (MR) are sampled periodically and compared against the miss ratio target. The differences are passed to the PI controller to generate the data admission control signal ΔP_{AC}, which is subtracted from the current data admission ratio. The ΔP_{AC} is derived using the following equation.

$$\Delta P_{AC} = P_{MR} \times (MR_{ST} - MR_{threshold}) + I_{MR} \times (MR_{LT} - MR_{threshold})$$

MR_{ST} and MR_{LT} are the short-term and long-term query miss ratios sampled in the last sampling period. $MR_{threshold}$ is the specified maximum miss ratio allowed by the application; P_{MR} and I_{MR} are two controller parameters which control the weights that short-term and long-term query miss ratios have on the data admission control signal. In this paper, the two controller parameters are hand picked to give the best system response. How to tune the controller to suit different system responses is not addressed in this paper and is left for future work.

To provide service differentiation, multiple data admission controllers can be used, with one data admission controller corresponding to one service class.

5 System Implementation

To evaluate the real-time periodic query model, we have developed a data stream management system prototype named RTSTREAM. The system is developed on top of STREAM data stream management system prototype developed at Stanford University [16]. In the prototype system, we extended the query specification language and its parser, added an EDF operator scheduler and data admission controller. The original STREAM has about 32k lines of code and 3k lines of code are added or modified to support periodic query model.

5.1 Query Language Extensions

As mentioned earlier, the CQL query language supported by STREAM project has a comprehensive set of operators and is more expressive than other existing stream query languages [2]. However, CQL can not specify the timing requirements of the queries. In RT-STREAM, in order to specify periodic real-time queries, we added the following constructs to CQL query language:

- STIME: Starting time of a periodic query;
- ETIME: Ending time of a periodic query;
- PERIOD: Period of the query (if applicable);
- DEADLINE: Deadline of the query (relative to the query starting time);
- IMPORTANCE: Importance level of the query;

The new query language can specify different types of queries. It can specify real-time and non-real-time queries depending on how the DEADLINE fields are set (non-real-time queries have infinitely large deadline values). It can specify both periodic and aperiodic queries depending on how the PERIOD fields are set (aperiodic queries have infinitely large period value). To provide service differentiation, the query language allows applications to divide their queries into different service classes by specifying different importance values.

5.2 Query and Scheduler

We have implemented an EDF scheduler to replace the original round-robin scheduler. After the specification of a periodic query is parsed by the query parser, it is stored in an internal object that represents the query. The query instances are initialized periodically according to the specification. The queries that are ready to run are stored in a linked list called *active query list*. The list is sorted based on their importance levels and deadlines. The queries that do not have active query instances or have instances that are already finished are stored in a separate linked list called *dormant query list*. The queries in the dormant query list are sorted by the starting times of their next query instances.

The pseudo code of the EDF scheduler is shown in Figure 4. The CPU time is divided into small time slots (currently 10 ms). At each slot, the EDF scheduler first inspects the dormant query list for query instances that need to be activated. If there are such query instances, the query instances are activated and inserted to the active query list. The active query list is then inspected for query instances that have already missed their deadlines. If such tardy query instances exist, they are immediately terminated and removed from the active query

```
EDFScheduelr::Run ()
{
  Get_Current_System_Time();

  Inspect_Dormant_Query_List();
  Initialize_New_Query_Instances ();

  Inspect_Active_Query_List();
  Remove_Tardy_Query_Instances();

  Run_Next_Query ();

  If (NextQuery.finished()) {
    Insert_to_Dorm_Query_List();
  }
  else {
    Insert_to_Active_Query_List();
  }
}
```

Figure 4. EDF Scheduler in Pseudo Code

list. After inspecting two linked lists, the scheduler then executes the next active query. If the query finishes in the time slot, the query is inserted to the dormant query list, waiting to be activated; otherwise, the query is inserted back to active query list. The current implementation requires that the relative deadline of a periodic query to be less than or equal to its period so that at any given time, there is at most one active query instance for each periodic query.

5.3 Concurrency Control

The system allows append-only operations on data streams, which matches the nature of data streams. The system also supports updatable relations. The system uses a timestamp-based concurrency control algorithm. It requires that the updates to the relations are marked explicitly by timestamps that are in non-decreasing order. Relation updates that arrive out of order are sorted before they are applied to the relations. If an update has been applied and an older update arrives, the older update is dropped. The updates to data streams and relations are handled by special system operators called *stream source* operator and *relation source* operator, which run with a priority higher than that of the application queries.

5.4 Data Admission Controller

To implement the data admission controller described in section 4, the query miss ratio statistics need to be maintained. The long term query miss ratio used by the data admission controller is the average miss ratio of the last 200 query instances. The short term query miss ratio is the average miss ratio of the last 20 query instances. The query miss ratio threshold is set as 5% in the current system implementation. The data admission controller is invoked periodically by the scheduler to adjust the data admission ratios. If queries are divided into multiple service classes, the system maintains

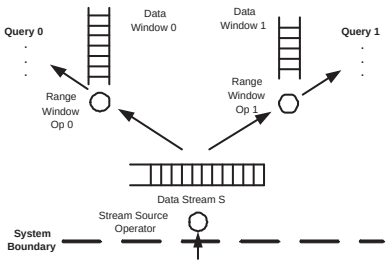

Figure 5. Data Admission Control Implementation

short-term and long-term miss ratio statistics for each service class and each service class has its own data admission controller.

The data admission control need to be implemented at the query level so that different queries can have different data admission ratios even if they share the same data stream source. As shown by Figure 5, two queries share the same data stream input S. The incoming data tuples of data stream S are first processed by the *stream source* operator. After the stream source operator, the data stream tuples are in the system. The data admission process for the queries is carried out at *stream data window* operators. In the example shown in Figure 5, *range window* operators Op 0 and Op 1 perform the data admission for query 0 and query 1 separately. With this design, different queries can use different data admission ratios.

Operators perform data admission using *random* dropping. Though not covered in this paper, *semantic dropping* can be used to improve system performance if query semantics are considered. The system also allows the data sources to mark the important data tuples to make sure they get processed by the system. The importance flag is marked by setting the highest bit of the data tuple timestamp. The data tuples with the flags are admitted to the system regardless of the current data admission ratios. However, operators maintain the target data admission percentage by dropping more unmarked data tuples.

6 Performance Evaluation

We conduct the performance evaluation with both synthetic workloads and real workloads from network monitoring applications. For synthetic workload experiments, we test the system performance with both short and long workload bursts. The real workload experiments are performed to show that the current system with our overload protection mechanism can handle the workload fluctuations of a real application. All experiments are carried out on a single machine running Redhat Linux 8.0. The machine is equipped with a 2.8 Ghz Pentium 4 hyperthreading processor and 1 Gigabyte DDR 3200 SDRAM main memory.

Parameter	Value
Total Memory	256 M
Page Size	4 K
Page # per Queue	50
Stream #	12
Data Rate per Stream	200 tuples/sec
Query per Stream	4
Total Query #	48
Selection Sel.	0.1
Stream-2-Stream Join Sel.	0.01
Stream-2-Rel Join Sel.	0.1
Query Period	1 - 4 sec
Query Deadline	1 - 4 sec
P_{MR}	0.2
I_{MR}	0.1
Data AC Period	1 sec
Experiments Run Time	300 sec

Table 1. Synthetic Workloads Settings

6.1 Synthetic Workloads

The settings for the synthetic workloads experiments are shown in Table 1. The data stream management system is configured to use 256 megabytes of main memory space. The system page size is set to be 4 kilobytes and each temporal result queue between two adjacent operators has 50 pages. The system is configured with such large memory space to eliminate the effects of memory constraints. There are 12 data streams registered in the system and for each registered data stream, there are 4 periodic queries associated with it. The data streams used in the experiments are variable-rate data streams and the average data tuple arrival rate of one data stream is 200 tuple/sec. The arrival of the data tuples conforms to Poisson distribution and the data rate shown is the average arrival rate. The data tuples in the streams are assigned with special values so that the selectivity values are configurable. In the experiments, we set the selection query selectivity to 0.1, the stream-to-stream join selectivity to 0.01 and stream-to-relation join selectivity to 0.1. The sampling period of the data admission controller is set to 1 second to give fast response to workload fluctuations. The total run time of one experiment is 300 seconds.

The data streams and relations used in the experiments have the same schema as in the example shown in Figure 1.

$$Stream\ S : (ID : integer, value : float, type : char(8))$$

$$Rlation\ R : (ID : integer)$$

For each data stream, there are four queries associated with it. For example, the periodic queries corresponding to data stream $S0$ are given below:

1. select * from S0 [range 4 second], S1 [range 4 second] where S0.type = S1.type and S0.ID = S1.ID and S0.value <> S1.value rtspec stime 0 second etime 301 second period 2 second deadline 2 second importance 0;

2. select * from S0 [range 6 second], R0 where S0.ID = R0.ID rtspec stime 0 second etime 301 second period 3 second deadline 3 second importance 0;

3. select avg(S0.value), min (S0.value), max (S0.value) from S0 [range 2 second], R0 where S0.ID = R0.ID rtspec stime 0 second etime 301 second period 2 second deadline 1 second importance 0;

4. select S0.type, Count(*) from S0 [range 1 second] group by S0.type rtspec stime 0 second etime 301 second period 1 second deadline 1 second importance 0;

Query 1 is a stream-to-stream join query, which monitors two different data streams (S0 and S1) and returns those tuples that have same sensor types and same sensor IDs but different values. Query 2 joins a stream (S0) with a relation (R0). It selects all the data tuples sent by those sensors identified by the IDs in the relation R0. Query 3 is an aggregate query that maintains the statistics for those sensors specified in relation R0. Query 4 collects statistics about the incoming data stream. It maintains the report count for each type of sensor in the data stream.

There are 12 streams registered in the system and the system has total 48 queries. The same set of queries without real-time query specifications are executed on the original STREAM prototype to compare the system performance. We conduct experiments on the following four systems:

- Stream: The original STREAM system which uses a round-robin scheduler and supports continuous queries.

- RTStream: The RTSTREAM system which runs with an EDF scheduler and supports the periodic query model. Data admission control is not used in this system.

- RTStream-DAC-S: The RTSTREAM system with single data admission controller for the whole system.

- RTStream-DAC-M: The RTSTREAM system with multiple data admission controllers (one for each service class).

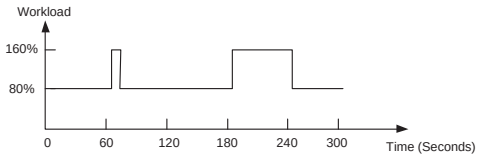

Figure 6. Workload Illustration

We are not aware of other existing real-time data stream query systems or algorithms. Therefore, no other baselines are available for performance comparisons.

To evaluate the system, we study the system performance with normal workloads and extreme workloads. As shown in Figure 6, the system workloads are configured in such way that with normal system workloads, the system CPU utilization is 80%. To evaluate the system performance under heavy workloads, we create the two workload bursts during the experiment. The workloads during the burst are twice the normal system workloads, which would need 160% of the CPU processing capacity. As shown in Figure 6, the first workload burst is a short burst. It begins at 60th second and lasts for only 10 seconds. The second workload burst is a long burst. It begins at 180th second and ends at 240th second. We choose such heavy workload fluctuations to test the system performance under extreme overload situations. All the experiment results shown are based on more than 10 runs and the 90% confidence intervals are less than 10% of the corresponding data values.

6.1.1 Single Service Class

The first set of experiments evaluate the system performance with queries of one service class. The experiment results are shown in Figure 7. As shown in Figure 7 (a), when the system is overloaded, the latencies of continuous queries increase monotonically with time. The maximum latency reaches as high as 23 seconds. The latency of the system continues to grow even after the burst ends at the 240th second. The reason is that a large number of data tuples are accumulated in the queues and the system needs to process those data tuples before it can process new incoming data tuples. Even though the length of the queue is decreasing after the workload burst, the latency still increases due to the fact that there are more data tuples per second during the workload burst and it takes more than one second to process those data tuples accumulated in one second during the burst.

In the periodic query model, the latencies of the queries are bounded by the specified query deadlines. As shown in the Figure 7 (b), similar to the latency of the continuous queries, the short-term miss ratio of the periodic query continues to increase until the burst goes away at the 240th second. After that, the short-term

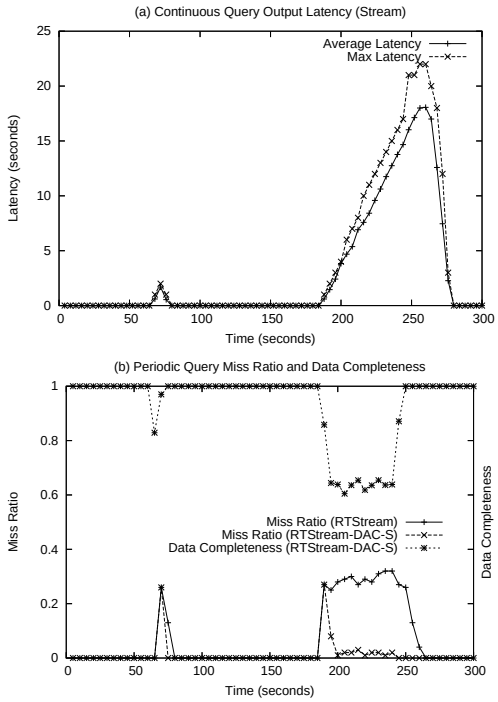

Figure 7. Single Service Class: Latency and Miss Ratios

miss ratio begins to drop. When data admission control is introduced, the system can respond to the workload fluctuations very faster. In the experiment, the data admission process returns the system query miss ratio to below 5% within 10 seconds. The process is fairly fast given the magnitude of the workload fluctuations. The data completeness of the system is maintained around 63% during the long workload burst. After the burst, the data completeness is restored to 100 % within 10 seconds.

The overhead of applying our data admission algorithm is almost negligible. From our measurements in the experiments, the overhead of running the scheduler and data admission controller costs less than 2% of the CPU time. Due to the space limitations, we will not show the overhead graph here.

6.1.2 Differentiated Services

Differentiated services are required by many applications. In case of overload, the system has to guarantee that the most important set of queries get processed. The second set of our experiments test the service differentiation capability of the system. In this set of experiments, 12 streams and the associated queries are divided in to three service classes, classes 0, 1 and 2, with the class 0 queries being the most important. Each service class has 4 data streams and 16 queries.

The experiment results are shown in Figure 8. The

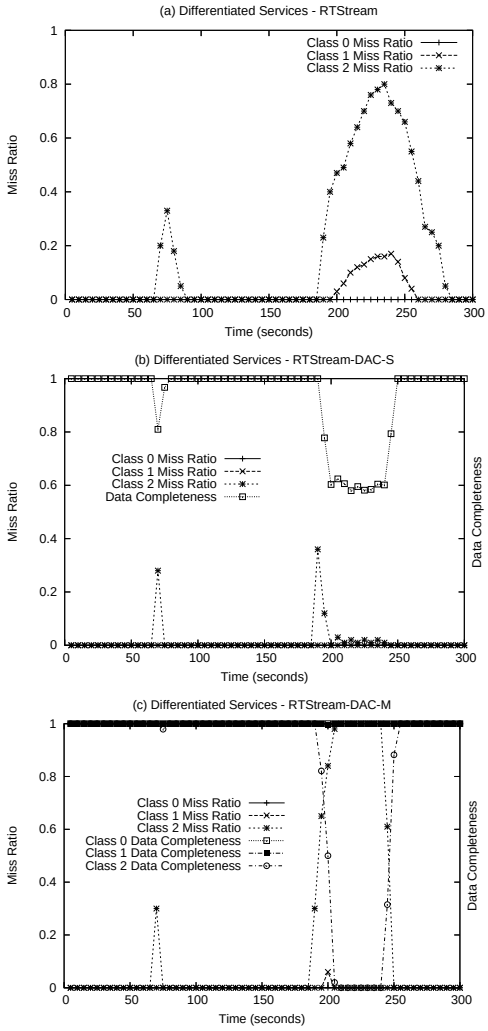

Figure 8. Differentiated Services: Miss Ratios and Data Completeness

system performance without data admission is shown in graph (a). Without data admission, the miss ratios of class 1 and class 2 queries keep increasing until the workload burst disappears at 240th second. Graphs (b) and (c) show the miss ratio and data completeness of our two data admission schemes. The first one is called RTStream-DAC-S, which stands for data admission control with single controller. In this scheme, there is only one data admission controller and all queries in the system share the same data admission controller. The other one is called RTStream-DAC-M, which has multiple data admission controllers, each corresponding to one service class. The results of single controller system is shown in graph (b). As shown by the graph, the system handles the workload fluctuations very well. The miss ratios of class 0 and class 1 queries remain 0 throughout the experiment and the miss ratio of class 2 queries is restored to below 5% within 10 seconds. In multiple controllers scheme shown in graph (c), the data completeness of class 0 and class 1 queries remains very close to 100% and the miss ratio of class 0 queries remains at 0. The miss ratio of class 1 queries has some fluctuations around 200th second. It is restored to zero within 5 seconds. However, this comes with at cost of class 2 queries. During the long workload burst, almost all incoming data for class 2 queries are dropped. Comparing graphs (b) with (c), we can see clearly the QoS tradeoffs in the system. If the application is willing to tolerate low data completeness ratios, the query miss ratios of different service classes can be substantially improved; if the miss ratios of the lower service class queries can be sacrificed, the service quality of the higher class queries can be preserved.

6.2 Real Network Monitoring Workloads

We also test our system prototype using workloads that simulate a network traffic monitoring system, which monitors the network traffic packet headers and network performance measurements. The results of the queries are used to optimize the network or counteract the malicious attacks (e.g., DDOS attacks). The timing requirements of the system are obvious since the performance of the network and essentially financial gains depend on them. The system monitors streams of network packet headers, which have the following format:

$Packet(srcIP,\ srcPort,\ destIP, destPort, len,$
$flags, timestamp)$;

The workloads used in the experiments are made of the following types of queries:

- Network Traffic Statistics Queries: Collecting statistics about the ongoing network traffic.
- Source Destination Traffic Monitoring: Monitoring the network traffic associated with particular source or destination addresses.
- Application Traffic Monitoring: Monitoring the network traffic associated with particular types of applications.

Parameter	Value
Total Memory	256 M
Page Size	4 K
Page # per Queue	400
Stream #	2
Data Rate per Stream	25000 tuples/sec
Query per Stream	8
Total Query #	16
Query Period	1 - 2 sec
Query Deadline	1 - 2 sec
P_{MR}	0.5
I_{MR}	0.2
Data AC Period	1 sec
Experiments Run Time	90 sec

Table 2. Network Monitoring Settings

Due to space limitation, we can not provide the whole list of queries used in the experiments. The network trace used in this experiment is trace #UFL-1099267976-1 from Passive Measurement and Analysis (PMA) project [17]. The experiment settings are shown in Table 2. The system is configured to monitor two network traces from two different router interfaces. The average rate is 25000 packets per second. In order to handle the high stream data arrival rate, the queue size is increased to 400 pages. There are total 16 queries registered in the system. The experiment lasts for 90 seconds, which is the length of the network traffic trace.

The network traffic workloads and experiment results are shown in Figure 9. The network traffic has an average data arrival rate of 25000 tuples per second. The rate variations are about 2000 tuples per second. Despite the small data arrival rate variations, the system still has large query workload fluctuations due to the query selectivity variations. The variations could be caused by a sudden increase of certain type of traffic workloads. For example, the selectivity of a query that monitors http traffic may change radically if the http packet volume suddenly increases in the traffic. As shown in the second graph, the system experiences a large workload burst starting at the 10th second. Our system responds to the workload fluctuations within 6 seconds and returns the query miss ratio back to 0. We can see that the workload fluctuations in network monitoring application are quite large as the system needs to reduce the data completeness to as low as 20% in order to handle the workload burst. After the workload burst, the system was restored to stable state (100% data completeness) in 7 seconds. We also show the out-

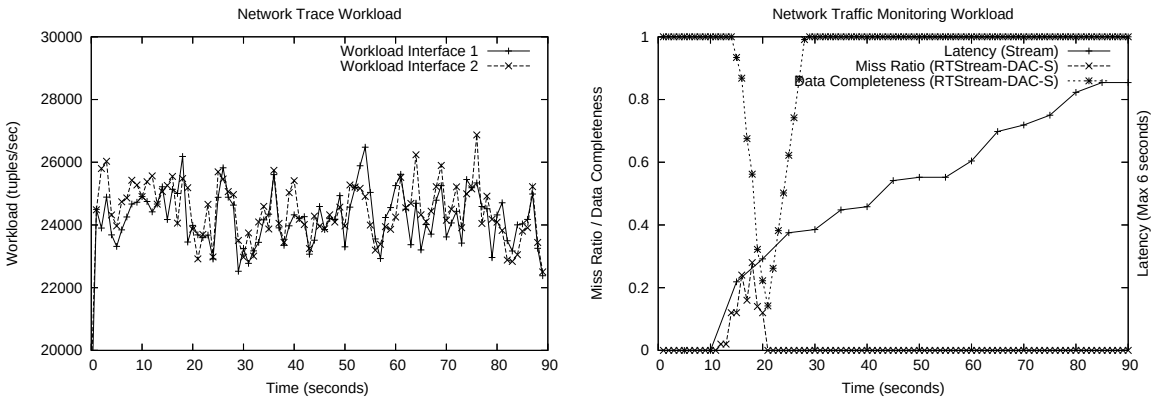

Figure 9. Network Traffic Monitoring Workloads and Results

put latency of continuous queries. The workload burst puts so much query workload into the system that without dropping data, the system takes very long time to return to stable state (as shown by the increasing latencies). The results show that our PQuery model with data admission control can handle the workload fluctuations reasonably well in this type of applications.

7 Related Work

In recent years, there has been a number of research projects and industrial efforts that focus on stream data management, including STREAM [7], Aurora [8], TelegraphCQ [11], NiagaraCQ [12], Gigascope [13], etc. The research in stream data management can be divided into categories such as query languages [2], query processing [1], scheduling [10][4], memory management [4], statistics maintenance [3], stream management in distributed environments [6] and clustering [15]. The real-time data stream query processing systems are different from real-time multimedia streaming systems in the sense that the stream query systems focus on executing complicated SQL-like queries on data streams instead of encoding/decoding/transferring multimedia streams. The Aurora project [9] claims to provide real-time data stream processing capabilities. However, their real-time metric is the average latency of data tuples, while our system uses the deadline miss ratios of periodic queries. The query models are different.

There are several papers that discuss the QoS and load shedding issues in stream data processing. Tatbul et. al. [19] propose a technique to dynamically insert or remove *drop* operators into query plans in order to handle the workload fluctuations. Babcock et. al. [5] propose load shedding techniques for a restricted class of stream queries. Like their approaches, the data admission approach proposed in this paper also reduces system workload by dropping data tuples. One difference is that we treat each query as a black box and use a PI controller to control the data admission process. Our solution drops data tuples before they are processed by the queries, which is more efficient compared to dropping intermediate results. Another difference is that our system considers the query time requirements and data admission process aims to minimize the query deadline miss ratios. Our system also supports query service differentiation and allows different service classes to have different data admission ratios.

There have been research results for scheduling query operators. An operator scheduler focusing on streaming data management is presented in [10]. The authors argue that a fine-grained scheduling approach in combination with various optimization techniques (such as batch operations) can significantly improve the system performance. They also discuss application-aware extensions that make scheduling decisions according to per-application QoS specifications. The proposed scheduling algorithms focus on optimizing throughput, latency and memory consumptions. However, the only timing requirements discussed in the paper is the output latency. Providing deadline guarantees for queries is not the main focus of the paper. Babcock et. al. [4] propose a scheduling algorithm called *chain scheduling*. The authors claim that the algorithm is almost optimal in minimizing run-time memory usage for single-stream queries involving selections, projections and foreign-key joins with stored relations. The paper mainly focuses on the minimizing the memory requirements of the system and the proposed algorithm does not deal with query timing requirements.

8 Summary and Future Work

In this paper, we present the design and implementation of our real-time data stream query processing system prototype named RTSTREAM. We propose the periodic query model as an alternative to the continuous query model for real-time applications. We discuss the semantic difference between these two models and their advantages and disadvantages. We introduce our real-

time periodic query QoS metrics and the query language extensions to support real-time periodic query specifications. We also design and implement data admission as an overload protection mechanism to reduce the system workloads in case of overload. In performance evaluation, we use synthetic workloads to test the system performance under extreme workload fluctuations and use real workloads from network traffic monitoring applications to show that the current system and data admission control mechanism can handle the workload fluctuations of a real application.

For future work, we are currently looking into several directions. One direction is to build the mathematical model for data stream queries so that the controller tuning process can be automated and the controller performance can be guaranteed. One direction is to address the memory constraints. How to adjust query quality and operator scheduling orders to deal with memory constraints at run time is an interesting research problem. Another direction is to extend the current system to distributed environments. With data stream sources and DSMS systems spread in distributed environments, the data stream dissemination, intermediate results generation and propagation, query QoS adaptation are all important and challenging research problems.

References

[1] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 2004.

[2] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.

[3] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *the 23rd ACM Symp. on Principles of Database Systems (PODS)*, 2004.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *VLDB Journal Special Issue on Data Stream Processing*, 2004.

[5] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Intl. Conference on Data Engineering (ICDE)*, 2004.

[6] B. Babcock and C. Olston. Distributed top-k monitoring. In *the ACM Intl Conf. on Management of Data (SIGMOD)*, 2003.

[7] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 2001.

[8] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on aurora. *VLDB Journal Special Issue on Data Stream Processing*, 2004.

[9] D. Carney, U. Centintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tabul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *28th VLDB Conference*, 2002.

[10] D. Carney, U. Centintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *the 29th International Conference on Very Large Data Bases (VLDB)*, 2003.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *2003 Conf. on Innovative Data Systems Research (CIDR)*, 2003.

[12] J. Chen, D. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[13] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD (Industrial Track)*, 2003.

[14] L. Golab and M. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2), 2003.

[15] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15, 2003.

[16] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *2003 Conf. on Innovative Data Systems Research (CIDR)*, 2003.

[17] Passive Measurement and Analysis. National laboratory for applied network research (nlanr). http://pma.nlanr.net/.

[18] M. Sullivan. Tribeca: A stream database manager for network traffic analysic. In *the 22nd International Conference on Very Large Data Base (VLDB)*, 1996.

[19] N. Tatbul, U. Centintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *the 29th International Conference on Very Large Data Bases (VLDB)*, 2003.