Prediction-Based QoS Management for Real-Time Data Streams

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

Abstract

With the emergence of large wired and wireless sensor networks, many real-time applications need to operate on continuous unbounded data streams. At the same time, many of these systems have inherent timing constraints. Providing deadline guarantees for queries over dynamic data streams is a challenging problem due to bursty data stream arrival rates and time-varying stream contents. In this paper, we propose a prediction-based Quality-of-Service (QoS) management scheme for periodic queries over dynamic data streams. Our QoS management scheme features novel query workload estimators, which predict the query workload using execution time profiling and input data sampling, and adjusts the query QoS levels based on online query execution time prediction. We implement our QoS management algorithm on a real-time data stream query system prototype called RTStream. Our experimental evaluation of the scheme shows that our query workload estimator performs very well even with workload fluctuations and our QoS management scheme yields better overall system utility than the existing approaches for QoS management.

1 Introduction

Many applications need to operate on continuous unbounded data streams. The streaming data may come from sensor readings, internet router traffic trace, telephone call records or financial tickers. Many of these new applications have inherent timing constraints in their tasks. However, due to the dynamic nature of data streams, the queries on data streams may have unpredictable execution cost. First, the arrival rate of the data streams is unpredictable, which leads to variable input volumes to the queries. For example, in surveillance systems, the volume of sensor reading data streams varies dramatically as targets enter the monitored territory. Second, the content of the data streams may vary with time, which causes the *selectivity* (Sel) of the query operators to change over time. The selectivity of an operator is defined as:

$$Sel = size(output)/size(input)$$

The operator selectivity measures the fraction of data input that passes the current operator to the next operator. As operator selectivity varies, the size of the intermediate results changes, even when the input volume stays the same. Thus, the query execution cost could change dramatically as the data stream content changes. While it is possible to design the system based on the maximum system workloads, it is not practical nor efficient because in a lot of these applications, the transient system overload do not persist for long period of time.

To address these issues, we propose a QoS management algorithm, which uses online *profiling* and *sampling* to estimate the cost of the queries on dynamic streams. The profiling process is used to calculate the average cost (CPU time) for each operator to process one data tuple. As we show later in this paper, with all the input data and intermediate results in main memory, the execution time per data tuple tends to be predictable when certain conditions are met (e.g., low contention rates for the hash index). The sampling process is used to estimate the selectivity of the operators in a query. Our experiment data shows that for high-rate data streams, sampling is a viable and cost-effective way to estimate query selectivity. To our best knowledge, this is the first work that uses online sampling to estimate query cost and control query QoS. We implement our prediction-based QoS management algorithm on our real-time data stream query system and the experiment results show that our QoS management algorithm handles the workload fluctuations very well and yield better overall system utility than existing approaches.

The rest of the paper is organized as follows: Sections 2 gives an overview of the system model and our assumptions. Section 3 gives a motivating example, explaining why QoS management is necessary. Section 4 presents our prediction-based QoS management

scheme. The performance evaluation and experimental results are presented in section 5. Section 6 discusses the related work and section 7 concludes the paper.

2 System Model and Assumptions

A data stream is defined as a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamps) sequence of data items [11]. A Data stream management system (DSMS) is a system especially constructed to process persistent queries on dynamic data streams. DSMSs are different from traditional database management systems (DBMSs) in that traditional database management systems expect the data to be persistent in the system and the queries to be dynamic whereas DSMSs expect dynamic unbounded data streams and persistent queries. 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. Typically, the queries are executed on a *window* of data. A window on a data stream is a segment of data stream that is considered for the current query. Emerging applications, such as sensor networks, network traffic analysis, intelligent traffic management, and financial market analysis, have brought research related to data streams in focus. These applications inherently generate data streams and DSMSs are well suited for such applications.

2.1 Periodic Query Model

The research in DSMS so far has been mainly focused on a continuous query model ([8], [15], [5]). In a continuous query model, long-running continuous queries are present in the system and new data tuples trigger the query instances. These incoming data tuples then get processed and the corresponding query results are updated. One of the drawbacks of a continuous query model is that the application cannot control the frequency and the deadlines of the queries. The execution of queries is driven by the data rate of the streams. Hence, the workload of the system depends directly on the data rate of the incoming streams, which is quite unpredictable.

Recently, we proposed a periodic query model for data stream queries with timing constraints ([17]) and developed a DSMS prototype called RTStream based on the periodic query model. RTStream is an extension to STREAM [15], which is a general-purpose DSMS (based on a continuous query model) developed at the Stanford University. In the periodic query model, every query has an associated period. The query instances are generated periodically to process the incoming stream data.

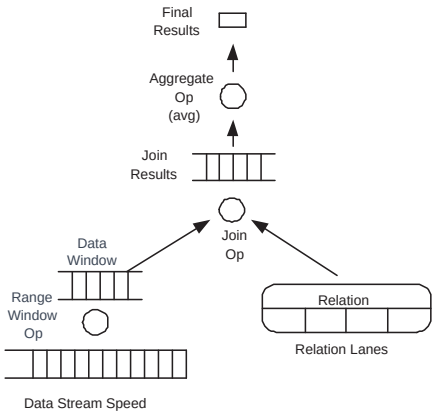

Figure 1. An Example Query Plan

2.2 Query Plan and Query Execution

In a DSMS, the system contains long-running and persistent queries. When a query arrives in the system, it is registered and it is triggered periodically based on its specified period. In our prototype system, all the queries are pre-registered in the system, and are converted to query plans (containing operators, queues and synopses) statically before the system starts. Queues in a query plan, store the incoming data streams and the intermediate results between the operators. A synopsis is associated with a specific operator in a query plan and it stores the accessary data structures needed for the evaluation of the operator. For example, a join operator may have a synopsis that contains a hash join index for each of its input. When the join operator is executed, these hash indices are probed to generate the join results. Example data stream and query specifications are given as follows:

$Stream : Speed(int\ lane, float\ value, char[8]\ type);$
$Relation : Lanes(int\ ID);$
$Query :\ Select\ avg(Speed.value)$
$From\ Speed[range1minute], Lanes$
$Where\ Speed.lane = Lanes.ID$
$and\ Speed.type = Truck;$
$Period\ 10\ seconds$
$Deadline\ 5\ seconds$

The query above operates on data streams generated by speed sensors and calculates the average speed of trucks in particular lanes during the last 1 minute. The query needs to be executed every 10 seconds and the deadline is 5 seconds after the release time of every periodic query *instance*. The query plan generated is shown in Figure 1. The query plan is made up of three query *operators* (range window operator, join operator and aggregate operator) and two queues (one for storing the range window output and one for storing the join output).

After the query plan is generated, the operators are sent to the scheduler to be executed. Depending on the

query model (e.g., continuous or periodic), a scheduling algorithm (e.g., round-robin or Earliest-Deadline-First) is used to meet the system requirements. In our system, the queries are scheduled using the Earliest-Deadline-First scheduler according to their deadlines.

2.3 Assumptions

In this paper, we assume real-time requirements of the system are soft, which means that a small number of queries missing their deadlines will not lead to system failure. This assumption is necessary as the system is dealing with unpredictable data streams and some queries may not meet their deadlines when the system is overloaded. We also assume the quality of the queries can be traded off for timeliness by dropping some of their inputs. One example application which has such requirements is the intelligent road traffic management system [14]. The system periodically computes the latest traffic statistics of different road segments and then sends them to a traffic simulator to calculate the road signal controls and the best routes for drivers. Obviously, late results are not acceptable in this case as the traffic simulator needs to have information from all the road segments to make accurate predictions. At the same time, if the results obtained are approximate (in this case, traffic statistics), the data is still useful for the traffic simulator as the system still gets some information about the traffic.

To handle the workload for high data stream query processing, all data structures used by the query need to be stored in physical memory for better performance. In this paper, we assume that the system has enough physical memory to store data tuples from the input data streams, intermediate results and accessary data structures (e.g., indices). This requirement is easy to satisfy given today's memory chip size. For example, with 512M physical memory, the system can maintain 100 queries on high rate data streams (1000 tuples/second per stream) with average window size of 10 seconds.

3 Need for Query QoS Management

Unpredictable data streams pose many challenges for data stream management systems that have timing constraints. As mentioned earlier, the DSMS may get overloaded as the arrival rates and contents of the incoming data streams change over time. The system must be able to handle these workload fluctuations, or some of the queries will miss their deadlines. The QoS management is performed at two levels. The *Inter-Query QoS Management* is used to allocate CPU time to different queries in the case of system overload so

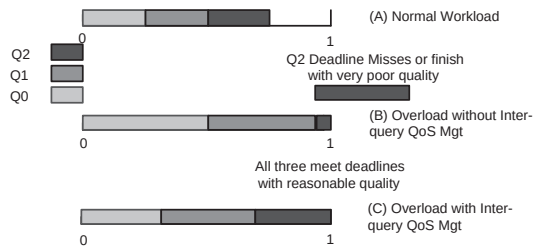

Figure 2. Need for Inter-Query QoS Management

that the overall system utility is maximized. The *Intra-Query QoS Management* is used to allocate available CPU time to different operators within the query plan so as to maximize the query quality.

It is necessary to manage data stream query QoS at both levels. Figure 2 gives an example to show why inter-query QoS management is necessary. As shown by Figure 2(A), at time 0, there are three queries, $Q0$, $Q1$ and $Q2$. All of them need to finish within one time unit. With normal system workload, all three queries will finish before their deadlines. When the system is overloaded, as shown in Figure 2(B), without inter-query QoS management, queries $Q0$ and $Q1$ and are able to finish before their deadlines but query $Q2$ is left with very little time to execute. Statically allocating CPU time slots does not solve the problem as query execution time varies depending on the data stream inputs and statically allocating CPU time results in less efficient utilization of CPU resources. Inter-Query QoS Management allocates CPU time to queries proportional to their estimated execution time. This ensures a fair distribution as shown in Figure 2(C). If we consider a single query plan, we use similar reasoning for intra-query QoS management. Substituting the queries in the above example with operators from one single query plan, we can see from Figure 2(B) that some operators may not get enough time to execute, depending on the policy used for scheduling the operators. Moreover, the output of one operator is the input of the other operator, i.e., the workloads of operators are correlated. In inter-query QoS management, the time remaining to execute the query is estimated periodically and if necessary, tuples are dropped to ensure that the query meets its deadline.

4 QoS Management Algorithm

As we can see from the previous section, when system is overloaded, the system needs to perform QoS management in order to meet query deadlines. For inter-query QoS management, the system needs to estimate the query execution time corresponding to dif-

ferent input data streams. For intra-query QoS management, the system needs to know the selectivity of different operators and their estimated execution time. Since our main objective is to ensure timeliness of query results, the query execution time estimation is the key to the QoS management.

In order to estimate the query execution time, we need three parameters for each query, namely, the input data stream volume, the operator selectivity and the execution time per data tuple for each operator. In this paper, we only consider queries that are ready to execute when performing the QoS management routine. Therefore, the input data volumes are known for these queries as the incoming data stream segments are already in the system. Note that it is beneficial to estimate the execution time of queries which do not have their complete input yet and use these estimations in the QoS management process, since the current ready-to-go queries may overlap with these future queries in their life spans. This is a challenging problem as it involves designing effective algorithms to monitor and predict the volume and content of future data streams. The complete study of this area is left for future work. Also note that we do not use the query execution time history to directly predict the execution time of the next query instance. The reason is that data streams in the system are dynamic and query execution time may vary significantly from one query instance to the next. Using execution history time directly may lead to large estimation errors. This approach may work for particular types of applications. Evaluating whether that works for different types of application workloads is beyond the scope of this paper.

4.1 Query Operator Overhead Estimation

Since the incoming data streams, intermediate results and accessory data structures are all stored in memory, the time taken for one operator to process a data tuple can be estimated effectively without considering any additional delays of fetching data from the disk. Table 1 shows the average execution time per tuple for different types of operators from our prototype system implementation. From Table 1, we can see that the execution cost of most operators are relatively small and fairly predictable with the exception of join operators. The cost of stream join, which joins two streams, varies from 3 microseconds to 100 microseconds because the number of the data tuples selected from data streams varies dramatically. Joining one data tuple with thousands of data tuples from another data stream results in the high stream join cost we see in the table. Based on our experiment results, the cost of each operator can be estimated as it closely follows the formula obtained from cost analyses. Here, we present our analyses for selection and join operators. The analyses for other operators are similar.

Operator	Avg Cost micro s/t micro s/t	Depends on Sel?	Depends on Syn Size?
Selection	0.16 - 0.3	Yes	No
Projection	0.16 - 0.2	No	No
Join	3 - 6	Yes	Yes
Stream Join	3 - 100	Yes	Yes
Distinct	0.16 - 0.3	Yes	Yes
Except	0.16 - 0.3	Yes	No
Group Aggr	0.16 - 0.6	Yes	Yes
Partition Win	0.16 - 0.3	No	No
Range Win	0.2 - 0.3	No	No

Table 1. Operator Cost and Dependency on Selectivity and Synopsis Size

4.1.1 Selection Operation Cost Analysis

The following notations are used for a selection operator O_{sel}:

- the input tuple volume, n
- the selectivity of the operator, s
- the execution time to evaluate the predicates, C_p
- the execution time to insert the output tuple to buffer, C_i

For the selection operator O_{sel}:

The number of output tuples $= n \times s$
The time for evaluating all input tuples $= n \times C_p$
The time for inserting output tuples $= n \times s \times C_i$
The total time $= n \times C_p + n \times s \times C_i$
The average cost per data tuple $= \frac{n \times C_p + n \times s \times C_i}{n}$
$= C_p + s \times C_i$

The costs C_p and C_i are expected to be constant for a particular set of predicates. As shown in Figure 3(a), when the selectivity is fixed to 0.2, the average cost for selection operator ranges between 0.17 to 0.18 microseconds when the average input size is larger than 400 tuples per second. When the average input size is lower than that, the average cost can be as high as 0.2 microsecond per tuple due to the overhead of operator context switch. When the selectivity of an operator varies (the input volume is kept constant), the average cost per tuple is expected to increase linearly with the selectivity and the slope of the line corresponds to C_i. As shown in Figure 3 (b), the cost curve from our experiments confirms our analysis.

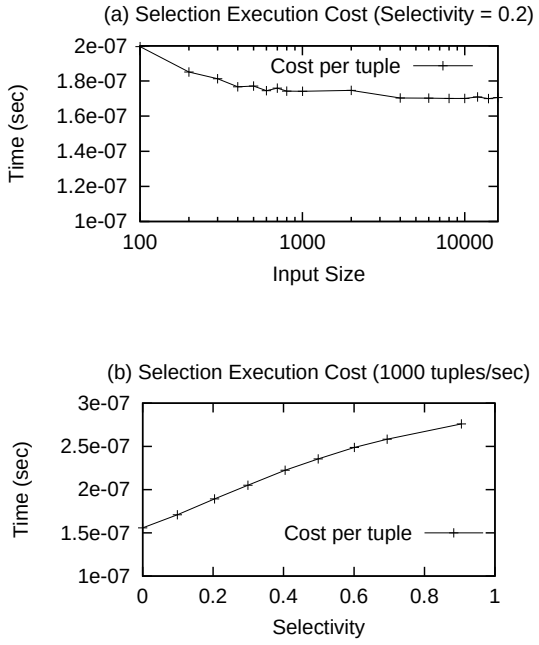

Figure 3. Selection Cost with Different Input Sizes and Selectivities

4.1.2 Join Operation Cost Analysis

For join operations, our system uses *Symmetric Hash Join* (SHJ) [18] [13]. SHJ works by keeping a hash table for each input in memory. When a tuple arrives, it is inserted in the hash table for its input and it is used to probe the hash table of the other input. This probing may generate join results which are then inserted in the output buffer. The following notations are used for a join operator O_{join}:

- the left and right input volume, n_L and n_R
- the selectivity of the operator, s
- the execution time to probe the left and right hash indices, C_{LProbe} and C_{RProbe}
- the execution time to hash left and right input, n_{LHash} and n_{RHash}
- the execution time to insert the output tuple to buffer, C_i

For the join operator O_{join}:
The number of output tuples $= n_L \times n_R \times s$
The time for processing left input tuples
$= n_L \times C_{RProbe} + n_L \times C_{LHash}$
The time for processing right input tuples
$= n_R \times C_{LProbe} + n_R \times C_{RHash}$
The time for inserting output tuples
$= n_L \times n_R \times s \times C_i$
The total time $= n_L \times (C_{RProbe} + C_{LHash})$
$+n_R \times (C_{RProbe} + C_{LHash}) + n_L \times n_R \times s \times C_i$

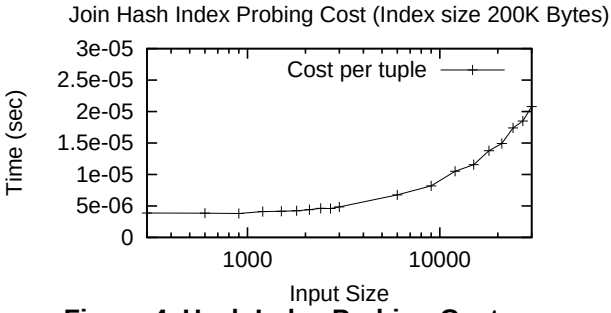

Figure 4. Hash Index Probing Cost

Of the three types of cost factors, hashing cost (C_{LHash} and C_{RHash}) and insertion cost (C_i) are much smaller than the probing cost (C_{LProbe} and C_{RProbe}) and generally remain constant. The probing cost, however, depends on the contention rate of the hash join index, which in turn depends on the input data volume and the allocated index size. As shown in Figure 4 where the hash index size is set to 200k bytes, the hash index probing cost is between 4 to 6 microseconds if the input size do not exceed 3000 tuples. When the input size exceeds 3000 tuples, the probing cost increases rapidly due to hash contention. The graph shows that if the hash index size is configured appropriately based on the expected number of data tuples in the index, the probing cost will remain within a close range.

4.1.3 Maintaining Cost Constant Using Profiling

From our analysis and experimental results for selection and join operators, it is clear that the system needs to know the precise values of the cost parameters (e.g., C_{Probe}) in the cost formula to estimate the execution time accurately. The solution is to keep track of the time used for each operator and compute these cost parameters periodically. The cost parameters are updated each time a query instance finishes. Suppose that after a query instance, the value for the cost parameter C is computed to be C_{new}, then C is updated using the single exponential smoothing formula:

$$C = C \times (1 - \alpha) + C_{new} \times \alpha \qquad 0 < \alpha < 1$$

We choose the exponential smoothing algorithm to give relatively higher weights on recent observations in forecasting than the older observations.

4.2 Selectivity Estimation Using Sampling

The next step is to estimate the selectivity of the query operators. Selectivity estimation has been stud-

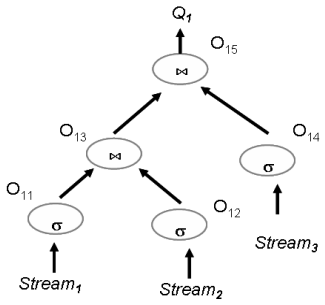

Figure 5. Query Plan with Joins

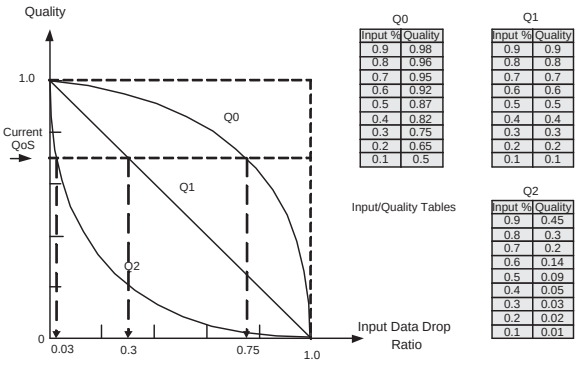

Figure 6. Inter-Query QoS Management with Query Quality Curves

ied in traditional database systems for query optimization [7]. Techniques like parametric methods, curve-fitting methods, sampling and various histogram methods have been proposed. In our system, we choose sampling as selectivity estimation algorithm because it is easy to implement and yields good estimation when handling high-rate data streams. Sampling also does not need to maintain static data structures for selectivity estimation, which has very high cost. Moreover, sampling is well-suited for a wide range of data types ([7]).

Our solution for selectivity estimation is to construct a *sampler* query plan for every query in the system. The query plans for the sampler queries are exactly the same as their corresponding real query plans. When a query instance is released to the scheduler, the sampler is executed first with sampled data tuples from the input. The data tuples are sampled from the real input according to preset sample ratio S_r. The sampling process selects a simple random sample without replacement. The results of the sampler query plans are used to estimate the selectivity and hence the execution time of the operators in the real query plans.

4.3 Inter-Query QoS Management

Inter-query refinement is performed to ensure that available CPU time is divided fairly among all active query instances. A *pseudo-deadline* is assigned to the queries which are ready to execute. This pseudo-deadline is based on the estimated execution time and the *input/quality table* for each query. The input/quality table is an application-specified table which maps the percentage of input tuples used in the query to the quality of the query results. First, we briefly describe how to estimate the execution time of a query plan.

Let $Q_1, Q_2, Q_3 \ldots$ be the query plans, such that Q_i is the query plan for the *ith* query and let O_{ij} be the *jth* operator in the *ith* query plan. We assume that the numbering scheme for the operators is the order in which the scheduler views the operators, i.e. the operator with a lower subscript is scheduled first. Let s_{ij} be the selectivity of operator O_{ij} and t_{ij} be the processing time of the operator per input tuple. Given the input n, the cost of the query Q_i (without join) is:

$$T_i = n \times t_{i1} + \sum_{j=2}^{N} n \times t_{ij} \times \left(\prod_{k=1}^{j-1} s_{ik} \right) \qquad (1)$$

For query plans involving joins, we should use equation 1 to calculate the cost for operators from the stream source to the join operator for both branch. Then add the cost of the two branches with the cost of join operator to get the total cost. We repeat this process until we reach the end of the query plan. For example, for the query plan in Figure 5, the estimated time of the query is:

$$T_1 = n_1 t_{11} + n_2 t_{12} + n_3 t_{14} + (n_1 s_{11})(n_2 s_{12}) t_{13} + ((n_1 s_{11})(n_2 s_{12}) s_{13})(n_3 s_{14}) t_{15}$$

Inter-query QoS management uses the estimated execution time to find an acceptable QoS level for all active queries. It is possible that for some queries, it is acceptable to drop a large amount of input data yet still be able to maintain good query quality. Whereas for some other queries, it is unacceptable to drop even a small amount input data. In order to solve this problem, the system allows the application to specify the relation between the query quality and the percentage of input data dropped using the Input/Quality table. As illustrated in Figure 6, three queries, $Q0$, $Q1$ and $Q2$, have different requirements in terms of maintaining query quality when system is overloaded. As shown by the input/quality table and the curve, the query quality of $Q0$ drops slowly with input dropping ratio. We define this type of the input/quality curve as *convex* QoS curve. In reality, $Q0$ can be a query to compute the average value of a data stream. It can still calculate the average value reasonably well when some input

```
// Pseudo Code for Inter-Query QoS Mgnt
// S_QI : Set of Active Query Instances
// QoS_Delta : QoS Negotiation Unit
// T_Avail : Available CPU time

double InterQueryQosMgnt (S_QI, double T_Avail ) {
   // Set Initial Target QoS Level at 100%
   double QoS_Target = 1.0;
   // Get exe time estimation for active queries
   double T_Est = exeTimeEst(S_QI , QoS_Target );
   while(T_Est < T_Avail ) {
      // Reduce Target QoS
      QoS_Target -= QoS_Delta ;
      T_Est = exeTimeEst(S_QI , QoS_Target );
   }
   return QoS_Target ;
}
```

Figure 7. Pseudo Code for Inter-Query QoS Managment

data tuples are dropped. The quality of $Q1$ drops linearly as input data dropping ratio. We call this type of the input/quality curve as *linear* QoS curve. On the other hand, $Q2$ is the opposite of $Q0$ as it can not tolerate dropping any input data tuples. We call this type of the input/quality curve as *concave* QoS curve. In reality, surveillance queries are of this type as they need to capture all anomalies in the system and they can not afford to miss any input data tuples. With the input/quality table, same output quality is translated into different input dropping ratios for different queries. For example, if the QoS of the target system is set to 70%, it translates into dropping 75%, 30% and 3% of the input data for $Q0$, $Q1$ and $Q2$ respectively. These ratios are called *drop ratios* and they denote the fraction of input tuples dropped. For query plans that operate on multiple stream inputs, the input/quality table specifies the optimal dropping ratios for each stream input for a given quality level.

As shown in Figure 7, the algorithm for inter-query QoS negotiation is a simple iterative process which keeps reducing the query QoS from 100% and calculates the cost of all active queries. The query QoS with which all the active query instances can finish before their deadlines is chosen and the corresponding pseudo-deadlines are calculated for the queries. The pseudo-deadlines are assigned proportional to the estimated time of the queries.

4.4 Intra-Query QoS Refinement

In inter-query QoS management, every query instance is assigned a pseudo-deadline, based on their estimated execution time. The query instances now perform intra-query refinement to meet their pseudo-deadlines instead of their actual deadlines. Before a query starts, it drops a fraction of the input data if the estimated execution time of the query exceeds the pseudo-deadline. We choose to drop the data tuple early in the query plan as it yields the best system utility [4] compared to dropping intermediate results later. Furthermore, the progress of the query is monitored periodically to ensure that the query meets its pseudo-deadline. If the query is running late, data tuples are dropped during execution to ensure that the query meets its deadline.

Suppose that the operator O_{ij} is scheduled currently and we are trying to estimate the processing time for the operator O_{ip}, where $p > j$.

The number of input tuples $= n \times \prod_{k=j}^{p-1} s_{ik}$
The estimated time for processing $= n \times t_{ip} \prod_{k=j}^{p-1} s_{ik}$
The estimated number of output tuples $= n \times \prod_{k=j}^{p} s_{ik}$

Hence, when operator O_{ij} is scheduled, the estimated time for query plan Q_i is given by

$$T_i = n \times t_{ij} + \sum_{p=j+1}^{N} n \times t_{ip} \times \left(\prod_{k=j}^{p-1} s_{ik} \right) \qquad (2)$$

where N is the total number of operators in the query plan Q_i. At time τ, this estimated time, T_i, needs to be compared to the remaining time for the deadline D_i, where $D_i = d_i - \tau$. If $T_i \leq D_i$, there is no need to drop tuples.

If $T_i > D_i$, we calculate ϵ, which denotes the fraction of input tuples that should be kept for further processing using equation 2. In a query plan with no join and hence no branches, dropping tuples at any operator O_{ij} affects the entire query plan. At operator O_{ij}, if $T_i > D_i$ and $\epsilon \times n$ is the number of input tuples after dropping tuples, for a query to finish before its deadline, the following must be true:

$$\epsilon \times n \times t_{ij} + \sum_{p=j+1}^{N} \epsilon \times n \times t_{ip} \times (\prod_{k=j}^{p-1} s_{ik}) < D_i$$

Taking the boundary case,

$$\epsilon \times (n \times t_{ij} + \sum_{p=j+1}^{N} n \times t_{ip} \times (\prod_{k=j}^{p-1} s_{ik})) = D_i$$

$$\epsilon \times T_i = D_i$$

For the query plans without joins, we calculate ϵ as follows:

$$\epsilon = \frac{D_i}{T_i}$$

For query plans involving joins, we should also make sure that tuples are not dropped from any stream unfairly. To distribute the drop amount fairly, we drop tuples according to the data dropping ratios given in the input/quality table. The table allows minimizing query workloads while still meeting certain QoS levels. During intra-query QoS management, if one branch of the join operator needs to drop more data tuples, the other branch also needs to drop a matching amount so that the dropping ratios between two inputs conform to the ratios specified in the input/quality table.

We use the process described above to determine the drop amounts from the incoming streams as well as during the execution of the query plan. The only difference is that in the former case, the estimated time T_i is the estimated execution time for the entire query plan when none of the operators has started. For each query we calculate the total number of tuples dropped during its execution. Then, using the input/quality table for the query, we can find the quality of the query results. This query quality is used as a measure of system performance. Ultimately, our objective is to maximize the average quality of the query results.

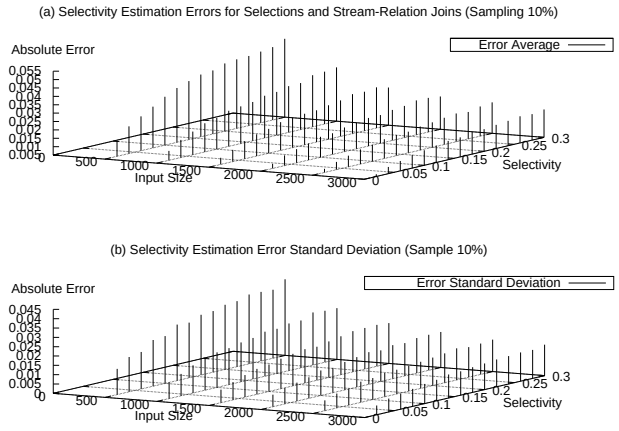

Figure 8. Selectivity Estimation Error and Error Standard Deviation

5 Performance Evaluation

We implement our prediction-based QoS management algorithms on a real-time data stream query prototype system called RTStream [17]. All experiments are carried out on a machine running Linux 2.6. The machine is equipped with a 2.8 Ghz Pentium 4 hyperthreading processor and 1 Gigabyte DDR 3200 SDRAM main memory.

5.1 Query Execution Cost Estimation

The first set of the experiments are used to evaluate how reliable random sampling can be in term of estimating the operator selectivity and query execution time. As shown in Figure 8, both the estimation error and its standard deviation decreases as the input size increases. When the input size is fixed, the selectivity estimation error increases with the actual value of selectivity.

Figure 9 shows the query execution time estimation error using the estimated selectivity values. Figure 9 (a) shows the execution cost estimation for the query plan shown in Figure 1. In the query plan, an aggregation operator is executed after a join operator (join $\rightarrow$ aggr). Surprisingly, the query execution time estimation is very good despite the bad selectivity estimations. The reason is that the cost of join operation is much higher than that of the aggregation operations (Table 1). As a result, when calculating the overall

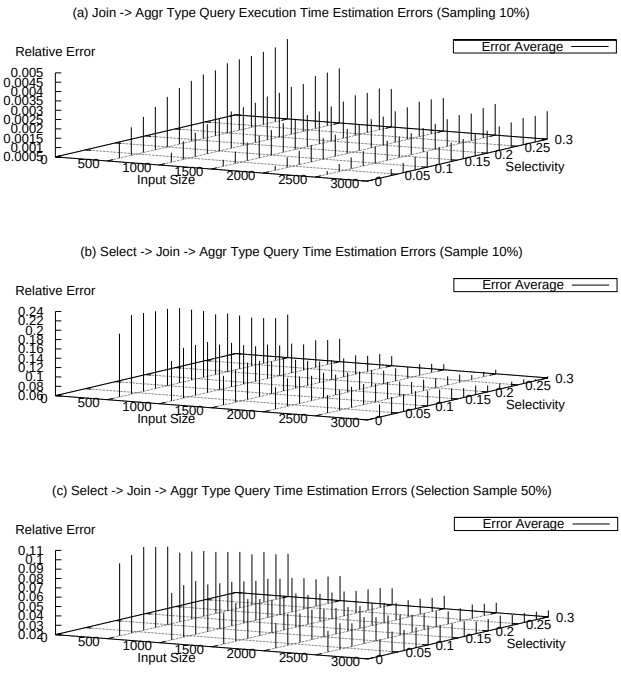

Figure 9. Query Execution Time Estimation using Sampled Selectivity Value

query cost, the cost of the aggregation operator can almost be neglected. Figure 9 (b) shows the execution time estimation of a typical select-join-aggregation query. As we can see from the graph, the estimation error ratio drops as the input size and the selectivity increases. When the input size is small and the selectivity is low, the query estimation time error can be as high as 22%. The reason is that if there is a large error in estimating the selectivity of the selection operation, the error will affect the estimated input volume of the join operator, which in turn affects the accuracy of the overall query execution time estimation. We address this problem by increasing the sampling ratio of the selection operator. Since selection is very cheap in terms of CPU overhead, we get very good estimation of its selectivity without sacrificing too much CPU time. The improved selectivity estimation will in turn improve the execution cost estimation of the whole query. As shown in Figure 9 (c), when we increase the sampling ratio for the selection operator, the estimation error ratio of the overall query cost is improved by more than 50%.

The above observations lead to an optimization technique called *dynamic sampling*. Essentially, if an operator is expensive to sample and its selectivity estimation is not critical in estimating the overall query cost (e.g., the join operator in the first case), its sampling ratio should be small; if an operator is cheap to sample but its selectivity estimation is important in estimating the overall query cost (e.g., the selection operator in the second case), its sampling ratio should be large to allow better selectivity estimation. The system can dynamically adjust those sampling ratios based on the execution time of different operators in the system.

5.2 Synthetic Workload Experiments

We conduct the performance evaluation with synthetic workloads to compare our approach with other approaches such as feedback-based workload control.

5.2.1 Workload Settings

The setting for the synthetic workload experiments is shown in table 2. The data stream management system is configured to use 512 megabytes of main memory space. Each queue is configured with 400K memory. The system is configured with such large memory space to eliminate the effects of memory constraints. The arrival of the data tuples conforms to Poisson distribution and the average data tuple arrival rate of the data streams is 1000 tuple/sec. In the experiments, we set the selection query selectivity to 0.2, the stream-to-stream join selectivity to 0.0001 and stream-to-relation join selectivity to 0.2. There are 10 queries for convex and linear QoS curve queries each and 4 concave QoS curve queries. The sampling ratio for the sampler query is set at 10%. The inter-query QoS management algorithm runs every 1 second and intra-query QoS management runs at the end of each operator or every 5 ms, whichever comes first. The total run time of one experiment is 300 seconds. The data streams and relations used in the experiments have the following schema:

Parameter	Value
Total Memory	512 M
Queue Size	400K
Stream #	8
Data Rate per Stream	1000 tuples/sec
Total Query #	24
Selection Sel.	0.2
Stream-2-Stream Join Sel.	0.0001
Stream-2-Rel Join Sel.	0.2
Query Period	1 - 4 sec
Query Deadline	1 sec
Convex QoS Curve Query #	10
Linear QoS Curve Query #	10
Concave QoS Curve Query #	4
Inter-Query QoS mgt period	1 sec
Intra-Query QoS mgt period	5 ms (or end of op)
Sampling Ratio	10%
Experiments Run Time	300 sec

Table 2. Synthetic Workloads Settings

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

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

For each data stream, there are three 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 period 2 second deadline 1 second;
2. select avg(S0.value), min (S0.value), max (S0.value) from S0 [range 2 second], R0 where S0.ID = R0.ID period 2 second deadline 1 second;
3. select S0.type, Count(*) from S0 [range 1 second] group by S0.type period 1 second deadline 1 second;

Query 1 is a stream-to-stream join query, which monitors two different data streams (S0 and S1) and

returns tuples that have the same sensor types and sensor IDs but different values. Query 2 is an aggregate query that maintains the statistics for the sensors specified in relation R0. Query 3 collects statistics about the incoming data stream. It maintains the number of tuples arriving for each type of sensor in the data stream.

5.2.2 Performance Metrics

The performance metric we use in comparing different algorithms is the average query quality. Let q_i be the final quality of a query Q_i, the average query quality is calculated as follows:

$$AvgQuality = \frac{\sum_{i=1}^{N} q_i}{N} \quad (3)$$

Where N is the total number of active query instances in the system. For queries that miss their deadlines, the query quality is equal to zero.

The query quality is computed by using the input/quality table for the queries. The percentage of the tuples retained for the query instance is used to find the corresponding query quality from this table. Computing the percentage of input tuples used for processing is straightforward. For example, consider a simple query plan Q_1 with only two operators, O_{11} and O_{12}, and one incoming data stream. If the first operator O_{11} drops 20% of its input data tuples and the second operator O_{12} drops 50% of its input data tuples, the percentage of input tuples retained = $80\% \times 50\% = 40\%$. In the case of query plan with joins, we chose the branch with higher drop amount compared to the dropping ratios in the input/quality curve. Thus, we find the worst case query quality for queries with joins.

5.2.3 Algorithms and Evaluations

We compared the performance of the following four algorithms:

- Best-Effort: The system does not drop any data tuples.
- Feedback: the system uses a feedback controller to control the data dropping ratio. The output query miss ratios are feeded back to control the incoming data streams. The feedback controller used is similar to the one in our previous paper [17].
- Fixed Sampling: The system fixes sampling ratio at 10%.
- Dynamic Sampling: The system use dynamic sampling for different queries.

The experiment results are shown in Figure 10. The system is saturated when the average arrival rate for the data stream is 1000 tuples per second. The average query quality is collected through at least 10 experiment runs (each lasts 300 seconds) and 90% confidence interval is less than 10% of the value shown. As shown by Figure 10 (a), as system workload increases, the average query quality of the fixed sampling algorithm drops around 2 percent due to the overhead of sampler queries. As system workload goes higher, the query quality of the best-effort algorithms begin to drop when the average data stream rate exceeds 1000 tuples per second. The query quality drops faster than the other three due to the fact that it does not drop data and more and more queries miss their deadlines. The feedback-control-based algorithm performs better since it begins to drop data tuples when queries miss their deadline. The sampling-based algorithms perform the best as they allocate CPU time to queries based on the queries' costs and their QoS curves. From the graph, we can see that dynamic sampling saves significant amount of sampling costs compared to the fixed sampling approach. It performs really well considering that it is working with over 50% more workload and it still manages to maintain average query quality close to 80%. The Figure 10 (b) shows the query quality standard deviations with variable system workload. As shown by the graph, the two sampling-based algorithms manage to keep the deviation less than 0.1 whereas the other two algorithms have their maximum deviation at 0.24 and 0.36 when the system workload is 150%. This means that our inter-query QoS management scheme works well and the system is fair to all queries in terms of query quality. The Figure 10 (c) compares the performance of dynamic sampling algorithm and feedback-control based algorithm under different workload fluctuation patterns. We configure the system workload to fluctuate around 120% according to different frequencies (or different periods). For example, if the workload fluctuation period is set at 2 seconds, the system data stream arrival rate is at 1400 tuple per second for one second and then changes to 1000 tuple per second. The workload maintains such a pattern during the course of an experiment. By comparing system performance under such workload pattern, we test system performance for queries on highly dynamic data streams. As shown by the results, our sampling-based approach handles workload fluctuations of different frequencies effectively because it does not rely on the workload history when making QoS management decisions. The feedback-control based approach, however, does not handle the high-frequency workload fluctuations very well since it relies on the history to decide whether to drop data tuples.

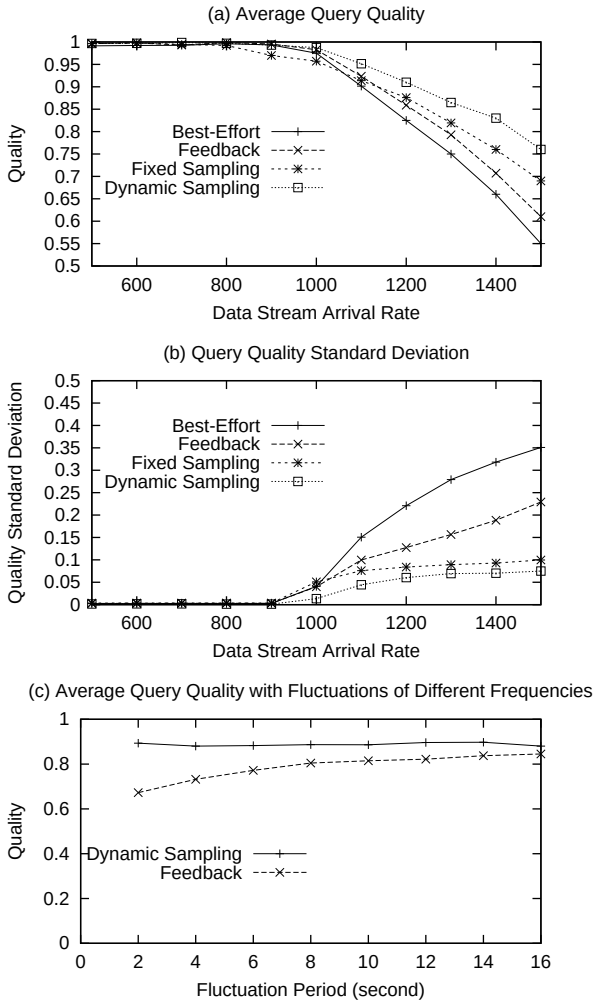

Figure 10. Query QoS with Synthetic Workload

The overheads of our prediction-based algorithms are already reflected in the results above as the experiments are carried out on a prototype system. From our measurements, the majority of the algorithm overheads come from the execution of the sampler query plans. With sampling ratio setting at 10%, the fixed sampling algorithm can take up to 10% of CPU time. While the dynamic sampling approach, depending on query workloads, costs 5% of total CPU time.

6 Related Work

In recent years, there has been a number of research projects and industrial efforts that focus on stream data management [5] [6]. The research in stream data management can be divided into categories such as query language [2], query processing [1], scheduling [9][3], memory management [3]. 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 [8] claims to provide real-time data stream processing capabilities. However, their real-time metric is the average latency of data tuples, while our system manages QoS of each periodic query instance according to its individual deadlines.

There has been work involving dropping tuples in DSMS to decrease the system load, mostly termed as "load shedding". Tatbul et. al. [16] propose a technique to dynamically insert or remove *drop* operators into query plans in order to handle the workload fluctuations. Babcock et. al. [4] propose load shedding techniques for a restricted class of stream queries. In both approaches, a query plan with load shedding operators is created statically. During execution, a query plan is chosen from the options available [16] and simply executed according to the run-time characteristics. Note that in the approach used in [4], each query plan has only one plan for load shedding associated with it, which is determined statically.

Our QoS management scheme is different from these approaches, as we execute sampler queries to estimate the selectivity of the actual query and use that estimation to negotiate the QoS between queries. Our system uses prediction and manages QoS *a priori*, i.e., before system actually gets overloaded while existing approaches adjust system workload after the system gets overloaded.

Sampling has long been used in tradition database systems for query optimization purposes [12] [7] [10]. Sampling gives a more accurate estimation than parametric and curve fitting methods used in traditional DBMS and provides a good estimation for a wide range of data types [7]. Furthermore, since no data structure is maintained in sampling-based approaches as opposed to histogram-based approaches, we do not need to worry about the overhead of constantly updating and maintaining the data structure. This is a very important point in the context of data streams as the input rate of the streams is constantly changing. Although sampling-based approaches for selectivity estimation have been studied in traditional database systems, we are not aware of any existing research work that uses sampling-based approaches to estimate data stream query workload and use those results to manage the query QoS.

7 Conclusions and Future Work

In this paper, we propose a prediction-based QoS management algorithm, which uses online *profiling* and

sampling to estimate the cost of the queries on dynamic data streams and use the estimations to adjust QoS among different queries. From our analysis and experimental data, we show that sampling is a viable solution for estimating the workload of queries on high rate, dynamic data streams. We incorporate the sampling approach to our QoS management framework and implement the algorithm on a prototype real-time data stream query system. The experiment results show that our algorithm handles the workload fluctuations well and provides good QoS to real-time queries in the system. Comparing to existing approaches, which adjust the system workloads after the system is overloaded, our prediction-based approach is unique in avoiding the system overload beforehand based on workload estimations. As a result, our prediction-base algorithm exhibits noticeable performance advantage over existing algorithms in handling queries on dynamic data streams.

For future work, we are considering schemes for predicting the future volume and selectivity of the data streams. With the prediction, the QoS management algorithm can consider the queries which are not currently active in the system, but whose instances will be invoked during the lifetime of current active queries. The QoS management will be able to take account of future query workloads and allocate CPU resources fairly among queries. The research problems include finding good data stream tracking algorithms and deciding how far to "look ahead" and still have high confidence in the predictions.

Acknowledgements

This work was supported, in part, by NSF grants IIS-0208758, CCR-0329609, and CNS-0614886.

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] 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.

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

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

[6] 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.

[7] D. Barbara, W. DuMouchel, C. Faloutsos, P. Hass, J. Hellerstein, Y. Ioannidis, H. Jagadish, T. Johnson, R. Ng, V. Poosala, K. Ross, and K. Sevcik. The new jersey data reduction report. Technical report, Bulletin of the Technical Committee on Data Engineering, 1997.

[8] 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.

[9] 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.

[10] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. In *ICDE*, pages 534–542, 2001.

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

[12] P. Haas, J. Naughton, and A. Swami. On the relative cost of sampling for join selectivity estimation. In *PODS '94: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 14–24, New York, NY, USA, 1994. ACM Press.

[13] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Distributed and Parallel Databases*, 1993.

[14] M. Mehta. Design and implementation of an interface for the integration of dynamit with the traffic management center. Master's thesis, MIT, 2001.

[15] 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.

[16] 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.

[17] Y. Wei, S. H. Son, and J. Stankovic. Rtstream: Real-time query for data streams. In *9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC)*, Apr. 2006.

[18] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-meory environment. In *PDIS*, 1991.