

# QoS Management of Real-Time Data Stream Queries in Distributed Environments

Yuan Wei, Vibha Prasad, Sang H. Son  
Department of Computer Science  
University of Virginia, Charlottesville 22904  
{yw3f, vibha, son}@cs.virginia.edu

## Abstract

*Many emerging applications operate on continuous unbounded data streams and need real-time data services. Providing deadline guarantees for queries over dynamic data streams is a challenging problem due to bursty stream rates and time-varying contents. This paper presents a prediction-based QoS management scheme for real-time data stream query processing in distributed environments. The prediction-based QoS management scheme features query workload estimators, which predict the query workload using execution time profiling and input data sampling. In this paper, we apply the prediction-based technique to select the proper propagation schemes for data streams and intermediate query results in distributed environments. The performance study demonstrates that the proposed solution tolerates dramatic workload fluctuations and saves significant amounts of CPU time and network bandwidth with little overhead.*

## 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 can be unpredictable, which leads to variable input volumes to the queries. 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:

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

It measures the fraction of data input that passes the current operator to the next. As operator selectivity varies, the

size of the intermediate results and final query results change, even when the input volume remains static. In distributed environments, the changing intermediate result size also affects the communication cost (CPU time and network bandwidth) associated with intermediate result propagation.

To address these issues, we proposed a prediction-based QoS management algorithm [19], 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. The sampling process is used to estimate the selectivity of the operators in a query. In this paper, we have used this prediction-based QoS management in a distributed environment to select the proper propagation schemes for data streams and intermediate query results. To our best knowledge, this is the first work that predicts query workload and used workload predictions to control query QoS in a distributed environment.

The rest of the paper is organized as follows: Section 2 gives an overview of the system model and our assumptions. Section 3 describes the prediction-based QoS management scheme. In section 4, we describe how this scheme can be used in a distributed environment. Section 5 presents the performance evaluation and experimental results. In section 6, we discuss 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 [8]. 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 a 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. Query Execution

In a DSMS, the system contains long-running and persistent queries. We have assumed a periodic query model [20] where every query has an associated period and the query instances are generated periodically to process the incoming stream data. In our prototype system, RTStream [20], 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 accessory 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 inputs. 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

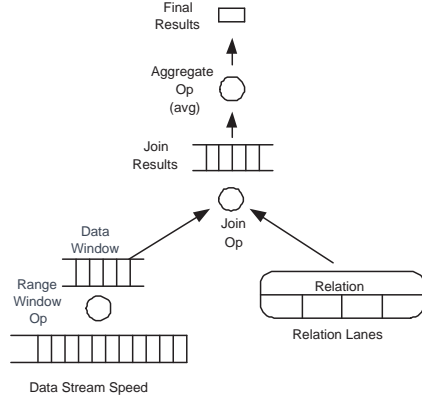


Figure 1. An Example Query Plan

system requirements. In our system, the queries are scheduled using the Earliest-Deadline-First scheduler according to their deadlines.

### 2.2. 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 might 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.

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 accessory data structures (e.g., indices). This requirement is easy to satisfy given today’s memory chip capacity. 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.

### 2.3. Example Application

An example application which matches the assumptions in this paper is the intelligent road traffic management system [13]. 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

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 Aggregation	0.16 - 0.6	Yes	Yes
Partition Window	0.16 - 0.3	No	No
Range Window	0.2 - 0.3	No	No

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

the traffic simulator as the system still gets some information about the traffic.

### 3. Prediction-based QoS Management

Three parameters are needed for each query to perform prediction-based QoS management, namely, the input data stream volume, the operator selectivity and the execution time per data tuple for each operator. Since our approach only considers queries that are ready to execute, the input data volumes are known.

#### 3.1. Query Execution Time Estimation

Table 1 shows the average execution time per tuple of the operators and their dependency on the current operator selectivity and synopsis (e.g. indices) size. The prediction-based QoS management scheme assumes that the incoming data streams, the intermediate results and the accessory data structures are all stored in memory. Hence the time taken for one operator to process a data tuple can be estimated effectively without considering any additional overhead of accessing data from the disk. Here, we briefly discuss the analysis for selection and join operators. The reader is referred to [19] for a detailed discussion.

##### 3.1.1 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  $O_{sel}$ :

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

The costs  $C_p$  and  $C_i$  are expected to be fairly constant for a particular set of predicates.

#### 3.1.2 Join Operation Cost Analysis

For join operations, *Symmetric Hash Join* (SHJ) [21, 10] is used. 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 for 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  $O_{join}$ :

$$\begin{aligned}
&\text{The number of output tuples} = n_L \times n_R \times s \\
&\text{The time for processing left input tuples} \\
&= n_L \times C_{RProbe} + n_L \times C_{LHash} \\
&\text{The time for processing right input tuples} \\
&= n_R \times C_{LProbe} + n_R \times C_{RHash} \\
&\text{The time for inserting output tuples} = n_L \times n_R \times s \times C_i \\
&\text{The total time} \\
&= n_L \times (C_{RProbe} + C_{LHash}) + n_R \times (C_{LProbe} \\
&+ C_{RHash}) + n_L \times n_R \times s \times C_i
\end{aligned}$$

Of the three type 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 for a particular system. The probing cost, however, depends on the contention rate of the hash index, which in turn depends on the input data volume and allocated index size.

### 3.1.3 Cost Constant Calculation Using Profiling

The exponential smoothing algorithm is chosen to give relatively higher weights on recent observations in forecasting than the older observations. Suppose that after a query instance is executed, the value for the cost parameter  $C$  is computed to be  $C_{new}$ , then  $C$  is updated using the single exponential smoothing formula:

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

### 3.2. Selectivity Estimation Using Sampling

In the prediction-based approach, a *sampler* query plan is constructed for every query in the system. The query plans for the sampler queries are exactly the same as their corresponding real queries' 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 execution of this sampler query plan is used to estimate the selectivity and hence the execution time of the operators in the query plan.

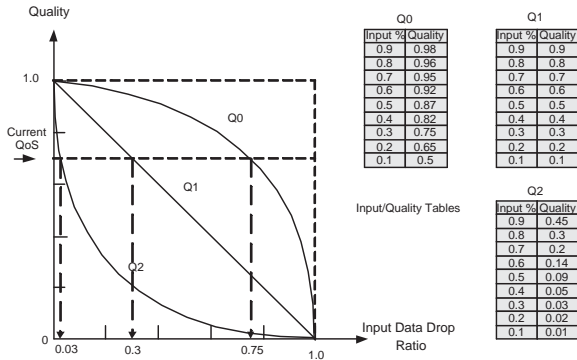


Figure 2. Inter-Query QoS Management with Query Quality Curves

### 3.3. Inter-Query QoS Management

Inter-query refinement is performed to ensure that all query instances get a fair chance to meet their deadline. 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 a user-defined table which maps the fraction of input tuples used to the quality of the query results.

As illustrated in Figure 2, three queries,  $Q_0$ ,  $Q_1$  and  $Q_2$ , have different requirements in terms of maintaining query quality when system is overloaded. As reflected by the input/quality table and the curve, the query quality of  $Q_0$  (an

example of convex QoS curve) drops slowly with input dropping ratio. A query with convex QoS curve can still calculate the average value reasonably well when a small percentage of input data tuples is dropped. The quality of  $Q_1$  (an example of linear QoS curve) drops linearly as input data dropping ratio. On the other hand,  $Q_2$  (an example of concave QoS curve) is the opposite of  $Q_0$  as it can not tolerate dropping any input data tuples. As an 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  $Q_0$ ,  $Q_1$  and  $Q_2$  respectively. These ratios are called *drop ratios* and denote the fraction of input tuples dropped.

The algorithm for inter-query QoS negotiation is an iterative process which keeps dropping the query QoS from 100% and calculates the cost of all active queries [19]. The query QoS for which all the queries finish their execution before their deadline 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.

### 3.4. Intra-Query QoS Refinement

In inter-query QoS management, every query instance is assigned a pseudo-deadline, based on the estimated execution time for the query. The query instances now perform intra-query refinement to fulfill their pseudo-deadlines instead of their actual deadlines. Before a query starts executing, it drops a fraction of the input data if the estimated execution time of the query exceeds the pseudo-deadline. The data tuples are dropped early in the query plan as doing so yields the best system utility [1]. The scheme used for determining the drop amounts is discussed in [19]. 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. Finally, for each query, the total number of tuples dropped during its execution are calculated. Then, the input/quality table for the query is used to find the quality of the query result. This query quality is used as a measure of system performance. The ultimate objective is to maximize the average quality of the query results.

## 4. QoS Management in a Distributed Environment

The prediction-based approach performs very well in maintaining query QoS in a centralized environment [19]. In this section, we discuss our dynamic data stream propagation algorithm in distributed DSMSs. In distributed DSMSs, one of the key problems is to reduce data stream propagation cost. Data stream propagation not only consumes network resources, but also consumes a fair amount of CPU time.

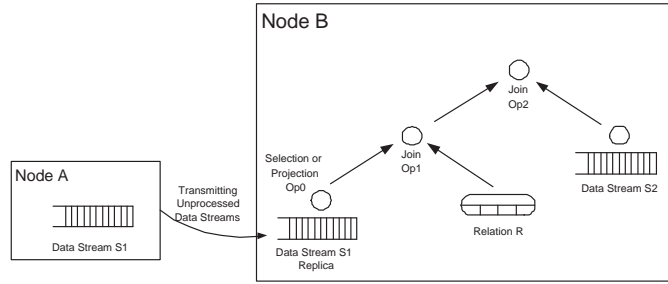


Figure 3. Transmitting Unprocessed Data Streams

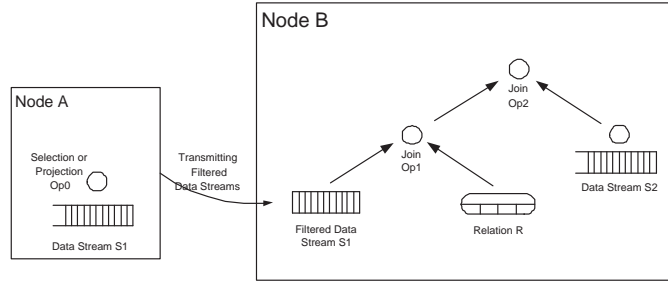


Figure 4. Transmitting Filtered Data Streams

From a previous study [11], the network operation costs as much as 35% of CPU time. However, this cost can be reduced if the remote node has the ability to choose whether to transmit unprocessed data streams or to process these data streams and transmit the intermediate results (which are generally smaller in size).

Suppose Node *B* contains a data stream query that operates on a local stream *S2*, a local relation *R* and a remote stream *S1* from node *A*. As shown in Figure 3, new data tuples in *S1* are sent to node *B* when they arrive at node *A* and the data stream *S1* is replicated at node *B*. One simple optimization is to move the obviously selective operators such as *Op0* to the remote node *A* and execute these operators at the remote node. As shown in Figure 4, the operator *Op0* can be moved to node *A* to reduce the network workloads. We call these obviously selective operators, *filters*, as their outputs are always smaller in size compared to their inputs. However, there are more aggressive approaches to save transmission cost. As shown in Figure 5, the system replicates relation *R* from node *B* to node *A* and executes the join operator *Op1* at node *A*. If the output size of the join operator *Op1* is smaller than its input size, the communication cost can be further reduced by transmitting the intermediate results from the join operator that are smaller in size.

However, unlike the filter operators, the join operators may produce outputs that are larger in size compared to their input. To address this problem, we use the prediction-based technique to predict the input/output size relation and determine whether to transmit unprocessed data streams or process intermediate results. The system creates a sampler oper-

ator at remote node for operators like the join operator *Op1*. The sampler operator samples a small number of data tuples from the incoming data streams and processes the join operation. If the output size is significantly smaller than the input size, the data stream source node should process the join operator locally and transmit the intermediate results to the remote node.

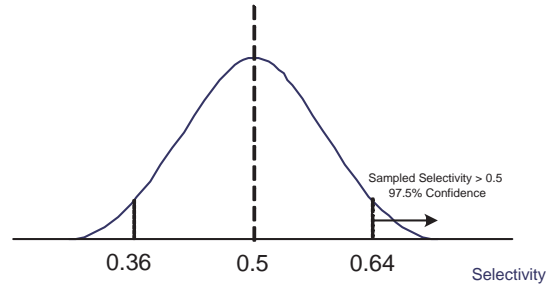


Figure 6. Sampling Result Selectivity Distribution

Once the sampling results are generated by the sampler, the system determines whether the current operator selectivity is small or large enough to switch data stream propagation strategy. A simple hypothesis testing can be used to decide whether the real output size is larger than the input data stream size with certain confidence value. For example, suppose that for the output size of operator *Op1* to be larger than the data stream input, the operator selectivity needs to be higher than 0.5. In the sampling process, we select 50 tuples from 1000 data tuples in stream *S1*. If the selectivity of the original 1000 data tuples is 0.5, the selec-

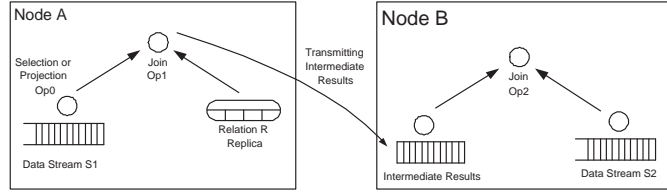


Figure 5. Transmitting Intermediate Results

tivity values from the random sampling follow the binomial distribution. A binomial distribution with sample size  $n$  and success probability  $p$  approximates normal distribution for large  $n$  and  $p$  not too close to 1 or 0 (statistics [14] recommends using this approximation only if  $np$  and  $n(1 - p)$  are both at least 5. Otherwise, a continuity correction should be applied). Since we sample 50 times out of the pool, the distribution of the sample selectivity mean approximates the normal distribution with mean 0.5 and standard deviation  $\delta = 0.5/\sqrt{50} = 0.0707$ . As shown in Figure 6, according to the normal distribution property, if the sampled selectivity is higher than  $0.5 + 0.0707 * 2 = 0.64$ , the current operator selectivity is higher than 0.5 with 97.5% probability. In terms of hypothesis testing, the hypothesis that the selectivity value is less than or equals to 0.5 is called the *null hypothesis* and the region where the selectivity value is higher than 0.64 is called the *rejection region*. This is because, if the sampled selectivity falls in that region, the null hypothesis is very unlikely and so it can be rejected. As the confidence value increases, the rejection region shrinks and vice versa.

Parameter	Value
Packet Sending Overhead	5 Microseconds
Packet Receiving Overhead	10 Microseconds
Tuples per packet	10
Sampler Operator Overhead	3 Microseconds per tuple
Data Stream Rate	1000 tuples/sec
Segment Size	1 second
Significance Level	15%

Table 2. Distributed DSMS Simulation Settings

proper transmission strategy is determined accordingly. As shown in Figure 7, at the receiving node, unprocessed data stream segments are processed first and then the results are assembled together with the received intermediate results. The data transmitted needs to be marked up properly so that it can be assembled correctly.

## 5. Performance Evaluations

We conducted a set of experiments to test the performance of the prediction-based algorithm in reducing the network workloads. In order to evaluate the performance of our approach, we developed a Java simulator based on hypothesis testing. The simulation settings are shown in Table 2. The overheads for sending and receiving a packet are set as 5 and 10 microseconds. According the Linux kernel research report [7], in Linux kernel 2.6.9, it takes 6 microseconds to send a packet and 17 microseconds to receive a packet using TCP/IP. The overheads are reduced in the simulation settings to reflect the technology advance. Each packet is set to contain 10 data tuples. As the experimental results shown in Table 1, the sampler operator overhead is set at 3 microseconds per tuple for join operator. The data stream arrival rate is set at 1000 tuples per second and each data stream sampling segment contains tuples within one second. The *significance level* for the hypothesis test used in the algorithm is set at 15%. A significance level of 15% means that the algorithm chooses to transmit intermediate results instead of unprocessed data streams only when the sampled results indicate that the intermediate results are smaller than the input with 85% or higher confidence. To evaluate the effectiveness of the proposed algorithm, we also show the performance of the ideal algorithm. The ideal algorithm is marked as the *Oracle* algorithm, which always uses the best propagation strategy but incurs no overhead. In this section, the results shown in the graph are based on at least 10 simulation runs and the

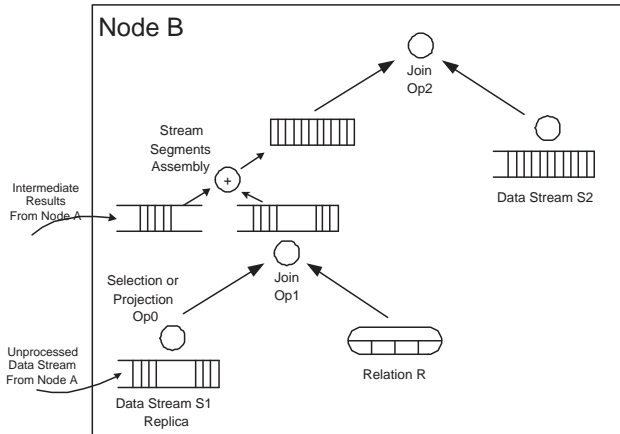


Figure 7. Assembly Data Stream Segments

To be able to switch between transmitting unprocessed data streams and transmitting processed intermediate results, the receiving node (in our example, node  $B$ ) has to be able to accept both types of data and use them to process stream query together. This is implemented by dividing the data stream to *segments* by either timestamps or sequence numbers of data tuples. Each segment is sampled separately and

95% confidence interval is within 5% of the value shown in the graph. The confidence intervals have been omitted in the figures to improve readability.

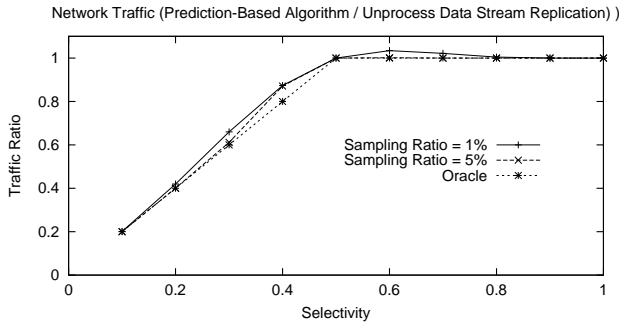


Figure 8. Network Traffic Reduction

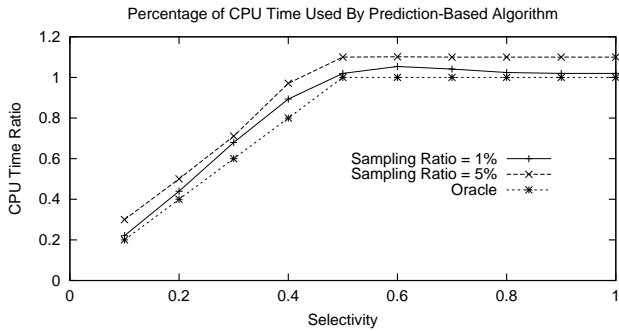


Figure 9. CPU Overhead Reduction

The network traffic results are shown in Figure 8. The ratio shown in the figure denotes the ratio between the network traffic volume generated by prediction-based algorithm to that generated by unprocessed data stream replication. We assume that the output tuples of the join operator are twice in size as compared to that of the data stream input. As a result, the network traffic volume can only be reduced when the join selectivity is less than 0.5. As shown in Figure 8, with sampling rate as low as 1% and 5%, the prediction-based algorithm can save significantly when the join selectivity is less than 0.5. In fact, in both cases, the algorithm performs very close to the ideal case shown by the Oracle algorithm. As expected, the algorithm performs better in terms of network traffic volume when the sampling ratio is higher at 5%. As the selectivity moves higher than 0.5, the algorithm generates slightly more network traffic. With sampling ratio at 1%, the algorithm produces 5% more network traffic when selectivity is 0.6. It is caused by the occasional mispredictions as a result of low sampling ratio. The problem disappears when the sampling ratio is set at 5%.

The total CPU time used by the algorithm is shown in Figure 9. The total CPU time shown in the figure contains both the network packet sending/receiving overhead and sampler operator execution overhead. The ratio shown is the CPU

time used by the prediction-based algorithm to the CPU time used to transmit unprocessed data streams. As we can see in Figure 9, when the join operator selectivity is less than 0.5, the prediction-base algorithm saves significant amount of CPU time. When the selectivity is higher than 0.5, the prediction-based algorithm costs more CPU time due to the sampling overhead and rare mispredictions. When sampling at 1%, the prediction-based algorithm works pretty well considering its CPU overhead is always within 10% of the Oracle algorithm.

## 6. Related Work

There have been significant research efforts devoted to the query optimization problem in the distributed data stream management systems [15, 5, 11, 12]. Adaptive filters [15] have been proposed to be used in distributed data stream management systems to regulate the data stream rate while still guaranteeing the adequate answer precisions. The D-CAPE paper [11] discusses the challenges for distributed data stream management systems and proposes *dynamic adaptation* techniques to alleviate the uneven workloads in the distributed environments. In this paper, we propose the idea of just-in-time sampling to estimate the output size of query operators and use the estimation results to control the intermediate query result propagation strategy. Compared to the algorithm for filter operators discussed in [15], our technique handles the join operators, which may have bigger output volume than their input volume.

Selectivity estimation has been one of the most studied problem in database community as the query optimizers use the estimation results to determine the most efficient query plans. Sampling [9], histogram [16, 17], index trees [3, 6], and discrete wavelet transform [18] are the most widely used selectivity estimation methods. Sampling has been used extensively in tradition database systems [9, 2, 4]. 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 [2]. 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. Prediction-based QoS management [19] is the first work to consider a sampling-based approach to estimate data stream query workload and use those results to manage the query QoS. To the best of our knowledge, this is the first work which uses this approach for prediction-based QoS management in distributed environments.

## 7. Conclusions and Future Work

In this paper, we describe the prediction-based QoS management algorithm for distributed environments where we apply dynamic sampling to determine the proper data stream propagation strategy. Our simulation results show that adjusting data transmission strategy using prediction results significantly reduces the communication cost with minimal amount of CPU overheads. There are a couple of ways to extend this work. First, more research is needed to provide solutions for the scenario where the selectivity of join operators is very small. It is a known problem that sampling yields high relative error rate when dealing with query operators with small selectivities [4]. The high relative estimation errors cause the sampling algorithm to select less optimal propagation strategies, thus compromising the performance of the proposed algorithm. Another way to extend the current work is to combine the operator history with the estimation. The selectivity history of the query operators provides valuable clues about the future operator selectivity. One of the ideas is to utilize the *volatility* of operator selectivity, i.e., using sampling only on those operators with volatile selectivities. In this way, sampling overhead on those less volatile operators can be saved. Lastly, evaluating this approach on a prototype system would be very valuable for the performance evaluation of the algorithm. As the performance study in this paper is via simulations, it is desirable to develop a prototype for a distributed DSMS and carry out detailed experiments to study the overhead associated with the prediction-based algorithm proposed in this paper.

## Acknowledgments

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

## References

- [1] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Intl. Conference on Data Engineering (ICDE)*, 2004.
- [2] 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.
- [3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Acta Informatica*, 1972.
- [4] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. In *ICDE*, pages 534–542, 2001.
- [5] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *the First Biennial Conference on Innovative Database Systems (CIDR)*, 2003.
- [6] D. Comer. The ubiquitous b-tree. In *Computing Surveys*, 1979.
- [7] J. Demter, C. Dickmann, H. Peters, N. Steinleitner, and X. Fu. Performance analysis of the tcp/ip stack of linux kernel 2.6.9. Technical report, University of Goettingen, 2005.
- [8] L. Golab and M. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2), 2003.
- [9] 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.
- [10] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Distributed and Parallel Databases*, 1993.
- [11] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *Very Large Data Bases (VLDB)*, 2005.
- [12] Y. Liu and B. Plale. Query optimization for distributed data streams. In *15th International Conference on Software Engineering and Data Engineering (SEDE'06)*, 2006.
- [13] M. Mehta. Design and implementation of an interface for the integration of DynaMIT with the traffic management center. Master's thesis, MIT, 2001.
- [14] S. Milton and J. Arnold. *Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences*. McGraw-Hill, 4th edition, 2003.
- [15] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *the ACM Intl Conf. on Management of Data (SIGMOD)*, 2003.
- [16] V. Poosala. *Histogram-based estimation techniques in databases*. PhD thesis, Univ. of Wisconsin-Madison, 1997.
- [17] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *23rd Int. Conf. on Very Large Databases*, Aug 1997.
- [18] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C, The Art of Scientific Computing*. Cambridge University Press, 1996.
- [19] Y. Wei, V. Prasad, S. H. Son, and J. A. Stankovic. Prediction-based QoS management for real-time data streams. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS 06)*, Dec. 2006.
- [20] Y. Wei, S. H. Son, and J. A. 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.
- [21] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-meory environment. In *PDIS*, 1991.