Load Shedding for Aggregation Queries over Data Streams

Brian Babcock Mayur Datar Rajeev Motwani

Department of Computer Science
Stanford University, Stanford, CA 94305
{babcock, datar, rajeev}@cs.stanford.edu

Abstract

Systems for processing continuous monitoring queries over data streams must be adaptive because data streams are often bursty and data characteristics may vary over time. In this paper, we focus on one particular type of adaptivity: the ability to gracefully degrade performance via "load shedding" (dropping unprocessed tuples to reduce system load) when the demands placed on the system cannot be met in full given available resources. Focusing on aggregation queries, we present algorithms that determine at what points in a query plan should load shedding be performed and what amount of load should be shed at each point in order to minimize the degree of inaccuracy introduced into query answers. We report the results of experiments that validate our analytical conclusions.

1 Introduction

As information processing systems grow in complexity, they become increasingly difficult to administer and control. Consequently, an emphasis of much recent work (see Section 6 for examples) has been to create systems that are to some degree *self-regulating*, reducing the time, effort, and expertise required of system administrators. Adaptive, self-regulating systems are particularly appropriate for environments where data and query rates are dynamic or unpredictable, as is frequently the case for data stream processing systems. Many sources of streaming data are quite bursty [10, 19, 20], meaning data rates and system load are liable to be highly variable over the lifetime of a long-running continuous query. In this paper, we focus on one type of adaptivity, the ability to gracefully degrade performance when the demands placed on the system cannot be met given available resources, in the context of continuous monitoring queries over data streams.

Data streams arise naturally in a number of monitoring applications in domains such as networking (traffic engineering, intrusion detection, sensor networks) and financial services (arbitrage, financial monitoring). These data stream applications share two distinguishing characteristics that limit the applicability of standard relational database technology: (1) the volume of data is extremely high, and (2) on the basis of the data, decisions are arrived at and acted upon in close to real time. Traditional data processing approaches, where effectively data is loaded into static databases for offline querying, are impractical due to the combination of these two factors.

Many data stream sources (for example, web site access patterns, transactions in financial markets, and communication network traffic) are prone to dramatic spikes in volume (e.g., spikes in traffic at a corporate web following the announcement of a new product or the spikes in traffic experienced by news web sites and telephone networks on September 11, 2001). Because peak load during a spike can be orders of magnitude higher than typical loads, fully provisioning a data stream monitoring system to handle the peak load is generally impractical. However, in many monitoring scenarios, it is precisely during bursts of high load that the function performed by the monitoring application is most critical. Therefore, it is particularly important for systems processing continuous monitoring queries over data streams to be able to automatically adapt to unanticipated spikes in input data rates that exceed the capacity of the system. An overloaded system will be unable to process all of its input data and keep up with the rate of data arrival, so *load shedding*, i.e., discarding some fraction of the unprocessed data, becomes necessary in order for the system to continue to provide up-to-date query responses. The question we study is which tuples to drop, and where in the query plan to drop them, so that the degree of inaccuracy in the query answers introduced as a result of load shedding is minimized.

The use of load shedding as a technique to achieve graceful degradation in the face of unmanageable system load has been suggested in earlier work on data stream systems ([5, 21, 22]). While some heuristics for load shedding have been proposed earlier, a systematic approach to load shedding with the objective of maximizing query accuracy has

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

been lacking. The main contributions of this paper are:

1. We formalize the problem setting for load shedding as an optimization problem where the objective function is minimizing inaccuracy in query answers, subject to the constraint that system throughput must match or exceed the data input rate.

2. We describe a load shedding technique based on the introduction of random sampling operations into query plans, and we give an algorithm that finds the optimum placement of sampling operations for an important class of monitoring queries, viz., sliding window aggregate queries over data streams. Our algorithm takes into account the effects of operator sharing among queries having common sub-expressions.

We also present extensions to our techniques to handle set-valued queries and queries with differing quality-of-service requirements.

Overview of Approach We propose a technique involving the introduction of load shedding operators, or *load shedders*, at various points in the query plan. Each load shedder is parameterized by a sampling rate p. The load shedder flips a coin for each tuple that passes through it. With probability p, the tuple is passed on to the next operator, and with probability $1 - p$, the tuple is discarded. To compensate for the lost tuples caused by the introduction of load shedders, the aggregate values calculated by the system are scaled appropriately to produce unbiased approximate query answers.

The decisions about where to introduce load shedders and how to set the sampling rate for each load shedder are based on statistics about the data streams, including observed stream arrival rates and operator selectivities. We use statistical techniques similar to those used in approximate query processing systems to make these decisions in such a way as to achieve the best attainable accuracy given data input rates.

Road Map The rest of this paper is organized as follows: We begin in Section 2 by formally stating our model and assumptions. Section 3 presents our algorithms for optimal sampling for different scenarios, and some extensions to the basic algorithm are described in Section 4. Section 5 gives the results of an experimental evaluation of our techniques. We describe related work in Section 6, before ending with our conclusions in Section 7.

2 Problem Formulation

The class of continuous monitoring queries that we consider are *sliding window aggregate queries*, possibly including filters and foreign-key joins with stored relations, over continuous data streams. Monitoring queries involving joins between multiple streams or non-foreign-key joins between streams and stored relations are comparatively rare in practice and therefore not considered in this paper.[1] A *continuous data stream* S is a potentially unbounded sequence of tuples $\{s_1, s_2, s_3, \ldots\}$ that arrive over time and must be processed online as they arrive. A *sliding window aggregate* is an aggregation function applied over a sliding window of the most recently-arrived data stream tuples (for example, a moving average). The aggregation functions that we consider are SUM and COUNT, though our techniques can be generalized to other functions such as AVG and MEDIAN. Sliding windows may be either *time-based*, meaning that the window consists of all tuples that have arrived within some time interval w of the present, or *tuple-based*, meaning that the window consists of the N most recently arrived tuples. A *filter* is a local selection condition on tuples from a data stream.

We believe this class of queries is important and useful for many data stream monitoring applications, including network traffic engineering, which we will use as an example application domain throughout this paper. Network analysts often monitor sliding window aggregates covering multiple timescales over packet traces from routers, typically filtering based on the internet protocol used, source and destination port numbers, autonomous subnetwork of packet origin (identified via user-defined functions based on longest prefix matching of IP addresses), and similar considerations [17]. Foreign-key joins or semijoins with stored relations may be used in monitoring queries to perform filtering based on some auxiliary information that is not stored in the data stream itself (e.g., the industry grouping for a security in a financial monitoring application). For our purposes, such joins have the same structure and effects as an expensive selection predicate or a user-defined function.

Most data stream monitoring scenarios involve multiple continuous queries that must be evaluated as data streams arrive. Sharing of common sub-expressions among queries is desirable to improve the scalability of the monitoring system. For this reason, it is important that a load shedding policy take into account the structure of operator sharing among query plans rather than attempting to treat each query as an isolated unit.

The input to the load shedding problem consists of a set of queries $q_1, \ldots, q_n$ over data streams $S_1, \ldots, S_m$, a set of query operators $O_1, \ldots, O_k$, and some associated statistics that are described below. The operators are arranged into a *data flow diagram* (similar to [5]) consisting of a directed acyclic graph with m source nodes representing the

[1] Furthermore, hardness results [7] for sampling in the presence of joins apply equally to the load shedding problem, making load shedding for non-foreign-key joins a difficult problem.

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

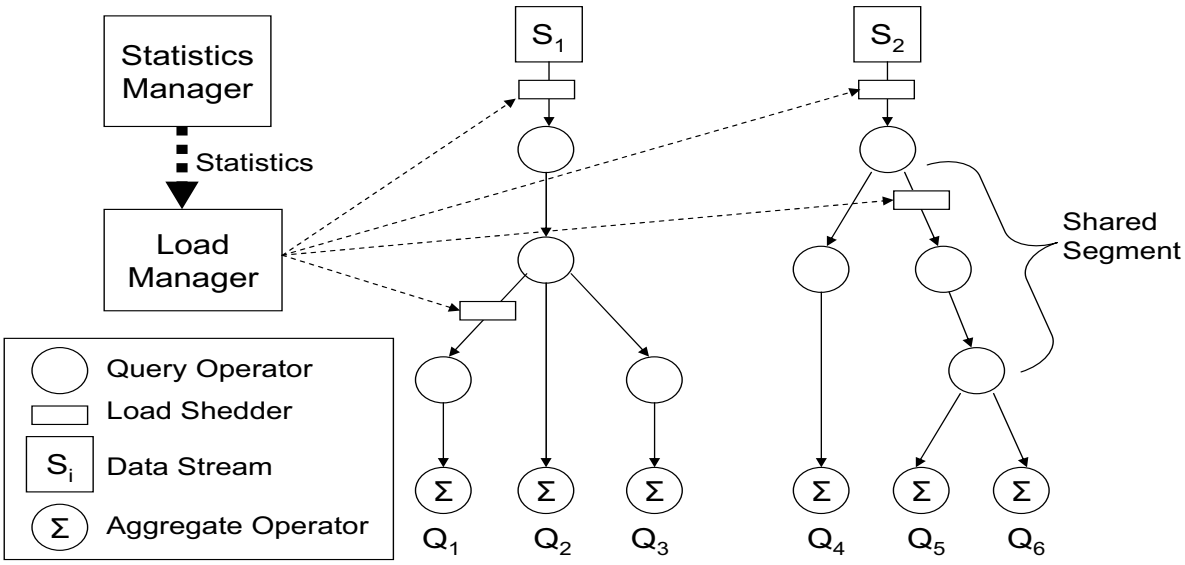

Figure 1. Data Flow Diagram

data streams, n sink nodes representing the queries, and k internal nodes representing the query operators. (Please refer to Figure 1.) The edges in the graph represent data flow between query operators. For each query q_i, there is a corresponding path in the data flow diagram from some data stream S_j though a set of query operators $O_{i_1}, O_{i_2}, \ldots, O_{i_p}$ to node q_i. This path represents the processing necessary to compute the answer to query q_i, and it is called the *query path* for query q_i. Because we do not consider joins between data streams, the data flow diagram can be thought of as being composed of a set of trees. The root node of each tree is a data stream S_j, and the leaf nodes of the tree are the queries that monitor stream S_j. Let $T(S_j)$ denote the tree of operators rooted at stream source S_j.

Every operator O_i in the data flow diagram is associated with two parameters: its selectivity s_i and its processing time per tuple t_i. The selectivity of an operator is defined as the ratio between the number of output tuples produced by the operator and the number of input tuples consumed by the operator. The processing time per tuple for an operator is defined as the average amount of time required by the operator to process each input tuple. The last operator along any query path is a windowed aggregate operator. The output of this operator is the final answer to the query and therefore not consumed by any other operator, so the selectivity of such an operator can be considered to be zero. Each SUM aggregate operator O_i is associated with two additional parameters, the mean μ_i and standard deviation σ_i of the values in the input tuples that are being aggregated. The final parameters to the load shedding problem are the rate parameters r_j, one for each data stream S_j. Rate parameter r_j represents the average rate of tuple arrival on stream S_j, measured in tuples per unit time.

Although we have described these input parameters (selectivity, stream rate, etc.) as known, fixed quantities, in reality their exact values will vary over time and cannot be known precisely in advance. In STREAM [21], our data stream management system, we have a Statistics Manager module that estimates the values of these parameters. Our query operators are instrumented to report statistics on the number of tuples processed and output by the operator and the total processor time devoted to the operator. Based on these statistics, we can estimate the selectivity and processing times of operators as well as the data stream rates. During times when statistics gathering is enabled, our SUM aggregation operator additionally maintains statistics on the sum and sum-of-squares of the aggregate values of tuples that it processes, allowing us to estimate the mean and standard deviation. As stream arrival rate and data characteristics change, the appropriate amount of load to shed and the right places to shed it may change as well. Therefore, in our system, estimates for the load shedding input parameters are periodically refreshed by the Statistics Manager, and load shedding decisions are periodically revisited.

2.1 Accuracy Metric

Let $A_1, A_2, \ldots, A_n$ be the answers to queries $q_1, q_2, \ldots, q_n$ at some point in time, and let $\widehat{A}_1, \widehat{A}_2, \ldots, \widehat{A}_n$ be the answers produced by the data stream monitoring system. If the input rates are high enough that load shedding becomes necessary, the data stream monitoring system

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

may not be able to produce the correct query answers, i.e., $\widehat{A}_i \neq A_i$ for some or all queries q_i. The quality of a load shedding policy can be measured in terms of the deviation of the estimated answers produced by the system from the actual answers. Since the relative error in a query answer is generally more important than the absolute magnitude of the error, the goal of our load shedding policy will be to minimize the relative error for each query, defined as $\epsilon_i = |A_i - \widehat{A}_i|/|A_i|$. Moreover, as there are multiple queries, we aim to minimize the maximum error across all queries, $\epsilon_{max} = \max_{1 \leq i \leq n} \epsilon_i$.

2.2 Load Constraint

The purpose of load shedding is to increase the throughput of the monitoring system so that the rate at which tuples are processed is at least as high as the rate at which new input tuples are arriving on the data streams. If this relationship does not hold, then the system will be unable to keep up with the arriving data streams, and input buffers and latency of responses will grow without bound. We capture this requirement in an equation, which we call the *load equation*, that acts as a constraint on load shedding decisions.

Before presenting the load equation, we will first introduce some additional notation. As mentioned earlier, each operator O_i is part of some tree of operators $T(S_j)$. Let U_i denote the set of operators "upstream" of O_i—that is, the operators that fall on the path from S_j to O_i in the data flow diagram. If some of the operators upstream of O_i are selective, the data input rate seen by operator O_i will be less than the data stream rate r_j at the stream source since some tuples are filtered out before reaching O_i. Furthermore, if load shedders are introduced upstream of O_i, they will also reduce the effective input rate seen by O_i. Let us define p_i as the sampling rate of the load shedder introduced immediately before operator O_i and let $p_i = 1$ when no such load shedder exists. Thus to measure the time spent in processing operator O_i, we are interested in the *effective input rate* for O_i, which we denote $r(O_i) = r_{src(i)} p_i \prod_{O_x \in U_i} s_x p_x$. (Here $src(i)$ denotes the index of the data stream source for operator O_i, i.e. $src(i) = j$ for $O_i \in T(S_j)$.) This leads to the load equation:

Equation 2.1 (Load Equation) *Any load shedding policy must select sampling rates p_i to ensure that:*

$$\sum_{1 \leq i \leq k} \left(t_i r_{src(i)} p_i \prod_{O_x \in U_i} s_x p_x \right) \leq 1$$

The left hand side of Equation 2.1 gives the total amount of time required for the system to process the tuples that arrive during one time unit, assuming that the overhead introduced by load shedding is negligible. Clearly, this processing time can be at most one time unit, or else the system will be unable to keep up with the arriving data streams. The assumption that the cost of load shedding is small relative to the cost of query operators, and can therefore be safely ignored, is borne out by experimental evidence, as discussed in Section 4.3. (In addition, we have extended our algorithm for the case when this assumption does not hold; see [4] for details.)

2.3 Problem Statement

The formal statement of the load shedding problem is as follows: *Given a data flow diagram, the parameters $s_i, t_i, \mu_i, \sigma_i$ for each operator O_i, and the rate parameters r_j for each data stream S_j, select load shedding sampling rates p_i to minimize the maximum relative error $\epsilon_{max} = \max_{1 \leq i \leq n} \epsilon_i$, subject to the constraint that the load equation, Equation 2.1, must be satisfied.*

3 Load Shedding Algorithm

In this section, we describe our algorithm for determining the locations at which load shedding should be performed and setting the sampling rate parameters p_i. The algorithm has two steps:

1. Determine the effective sampling rates for each query that will distribute error evenly among all queries.
2. Determine where in the data flow diagram load shedding should be performed to achieve the appropriate rates and satisfy the load equation.

These two steps are described in detail below.

3.1 Allocation of Work Among Queries

Recall that the error metric we use to measure the accuracy of a query response is the relative error. It is impossible to precisely predict the relative error in query answers that will arise from a particular choice of a load shedding policy, because the data values in the discarded tuples are unknown. However, if we assume some knowledge about the distribution of data values, for example based on previously-seen tuples from the data streams, then we can use probabilistic techniques to get good estimates of what the relative error will be. There is some variability in the relative error, even if the data distribution is known exactly, because the approximate answers produced by the system depend on the outcomes of the random coin flips made by the load shedders. Therefore, the approach that we take to compare alternative load shedding policies is as follows: for a fixed small constant δ (we use 0.01), we say that a load shedding policy achieves error ϵ if, for each query q_i, the relative error

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

resulting from using the policy to estimate the answer to q_i exceeds ϵ with probability at most δ.

3.1.1 Relating Sampling Rate and Error

Suppose the query path for a SUM query q_i consists of the sequence of operators $O_{i1}, O_{i2}, \ldots, O_{iz}$. Consider a load shedding policy that introduces load shedders along the query path with sampling rates $p_{i1}, p_{i2}, \ldots, p_{iz}$. Let τ be a tuple that would pass through all the query operators and contribute to the query answer in the absence of any load shedders. When load shedders are present, τ will contribute to the answer if and only if it passes through all the load shedders, which occurs with probability $P_i = p_{i1}p_{i2}\ldots p_{iz}$. We will refer to P_i as the *effective sampling rate* for query q_i.

Let $\mathcal{Q}_i$ denote the set of tuples from the current sliding window that would pass all selection conditions and contribute to the query answer in the absence of load shedders. Let N_i be the number of tuples in the set $\mathcal{Q}_i$. From the above discussion, it is clear that in the presence of load shedders, this aggregate query will be answered based on a sample of $\mathcal{Q}_i$ where each element gets included independently with probability P_i. For the tuples in the set $\mathcal{Q}_i$, let $v_1, v_2, \ldots, v_{N_i}$ denote the values of the attribute being summed, and let A_i be their sum. The approximate answer $\widehat{A}_i$ produced by the system will be the sum of v_i's for the tuples that get included in the sample, scaled by the inverse of the effective sampling rate $(1/P_i)$. The following proposition, which follows directly from a result due to Hoeffding (Theorem 2 in [15]), gives an upper bound on the probability that the relative error exceeds a certain threshold ϵ_i.

Proposition 3.1 *Let $X_1, X_2, \ldots, X_N$ be N random variables, such that each random variable X_j takes the value v_j/P with probability P and the value zero otherwise. Let $\widehat{A}_i$ be the sum of these random variables and let $A_i = \sum_{j=1}^{N} v_j$. If we denote by SS_i the sum $\sum_{j=1}^{N} v_j^2$, then*

$$Pr\{|\widehat{A}_i - A_i| \geq \epsilon |A_i|\} \leq 2\exp\left(-2P^2\epsilon^2 A_i^2/SS_i\right)$$

Thus, for a query q_i, to ensure that the probability that the relative error exceeds ϵ_i is at most δ, we must guarantee that $2\exp\left(-2P_i^2\epsilon_i^2 A_i^2/SS_i\right) \leq \delta$, which occurs when $P_i\epsilon_i \geq C_i$, where we define $C_i = \sqrt{\frac{SS_i}{2A_i^2}\log\frac{2}{\delta}}$. Letting the mean and variance of the tuples in $\mathcal{Q}_i$ be denoted by $\mu_i = \sum_{j=1}^{N_i} v_j/N_i$ and $\sigma_i^2 = \sum_{j=1}^{N_i}(v_j - \mu_i)^2/N_i$, respectively, the ratio SS_i/A_i^2 is equal to $(\sigma_i^2+\mu_i^2)/(N_i\mu_i^2)$. Thus the right-hand side of the preceding inequality reduces to $C_i = \sqrt{\frac{\sigma_i^2+\mu_i^2}{2N_i\mu_i^2}\log\frac{2}{\delta}}$.

If we want a load shedding policy to achieve relative error ϵ_i, we must guarantee that $P_i \geq C_i/\epsilon_i$. Thus, in order to set P_i correctly, we need to estimate C_i. Recall that we are given estimates for μ_i and σ_i (provided by the Statistics Manager) as inputs to the load shedding problem. The value of N_i can be calculated from the size of the sliding window, the estimated selectivities of the operators in the query path for q_i, and (in the case of time-based sliding windows) the estimated data stream rate r_j.

The larger the value of C_i, the larger the effective sampling rate P_i needs to be to achieve a fixed error ϵ_i with a fixed confidence bound δ. Clearly, C_i is larger for queries that are more selective, for queries over smaller sliding windows, and for queries where the distribution of values for the attribute being summed is more skewed. For a COUNT aggregate, $\mu_i = 1$ and $\sigma_i = 0$, so only the window size and predicate selectivity affect the effective sampling rate.

Since the values of the parameters that affect the effective sampling rate are known only approximately, and they are subject to change over time, using the estimated parameter values directly to calculate effective sampling rates may result in under-sampling for a particular query, causing higher relative error. For example, if the data characteristics change so that an attribute that has exhibited little skew suddenly becomes highly skewed, the relative error for a query which aggregates the attribute is likely to be higher than predicted. The impact of a mistake in estimating parameters will be more pronounced for a query whose P_i is low than for a query with higher P_i. Therefore, for applications where rapid changes in data characteristics are of concern, a more conservative policy for setting effective sampling rates could be implemented by adding a constant "fudge factor" to the estimates for C_i for each query. Such a modification would misallocate resources somewhat if the estimated parameters turn out to be correct, but it would be more forgiving in the case of significant errors in the estimates.

3.1.2 Choosing Target Errors for Queries

The objective that we seek to minimize is the maximum relative error ϵ_i across all queries q_i. It is easy to see that the optimal solution will achieve the same relative error ϵ for all queries.

Observation 3.2 *In the optimal solution, the relative error (ϵ_i) is equal for all queries.*

Proof: The proof is by contradiction. Suppose that $\epsilon_i < \epsilon_j$ for two queries q_i, q_j. Since $\epsilon_i = C_i/P_i < \epsilon_j$, we could reduce P_i to P_i' by introducing a load shedder before the final aggregation operator for q_i with effective sampling rate P_i'/P_i so that $\epsilon_i' = C_i/P_i' = \epsilon_j$. By doing so, we keep the maximum relative error unchanged but reduce the processing time, gaining some slack in the load equation. This slack can be distributed evenly across all queries by increasing all load shedder sampling rates slightly, reducing the relative error for all queries. ■

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

For an optimal solution, since the relative errors for all queries are the same, the effective sampling rate P_i for each query q_i will be proportional to the C_i value for that query, since $P_i = C_i/\epsilon_i = C_i/\epsilon_{max}$. Therefore, the problem of selecting the best load shedding policy reduces to determining the best achievable ϵ_{max} and inserting load shedders such that, for each query q_i, the effective sampling rate P_i, is equal to C_i/ϵ_{max}. In doing so we must guarantee that the modified query plan, after inserting load shedders, should satisfy the load equation (Equation 2.1).

3.2 Placement of Load Shedders

For now, assume that we have guessed the right value of ϵ_{max}, so that we know the exact effective sampling rate P_i for each query. (In fact, this assumption is unnecessary, as we will explain in Section 3.2.1.) Then our task is reduced to solving the following problem: *Given a data flow diagram along with a set of target effective sampling rates P_i for each query q_i, modify the diagram by inserting load shedding operators and set their sampling rates so that the effective sampling rate for each query q_i is equal to P_i and the total processing time is minimized.*

If there is no sharing of operators among queries, it is straightforward to see that the optimal solution is to introduce a load shedder with sampling rate $p_i = P_i$ before the first operator in the query path for each query q_i. Introducing a load shedder as early in the query path as possible reduces the effective input rate for all "downstream" operators and conforms to the general query optimization principle of pushing selection conditions down.

Introducing load shedders and setting their sampling rates is more complicated when there is sharing among query plans. Suppose that two queries q_1 and q_2 share the first portion of their query paths but have different effective sampling rate targets P_1 and P_2. Since a load shedder placed at the shared beginning of the query path will affect the effective sampling rates for both queries, it is not immediately clear how to simultaneously achieve both effective sampling rate targets in the most efficient manner, though clearly any solution will necessarily involve the introduction of load shedding at intermediate points in the query paths.

We will define a *shared segment* in the data flow diagram as follows: Suppose we label each operator with the set of all queries that contain the operator in their query paths. Then the set of all operators having the same label is a shared segment.

Observation 3.3 *In the optimal solution, load shedding is only performed at the start of shared segments.*

This observation (also made in [22]) is true for the same reason that load shedding should always be performed at the beginning of the query plan when no sharing is present. The effective sampling rates for all queries will be the same regardless of the position of the load shedder on the shared segment, but the total execution time will be smallest when the load shedding is performed as early as possible.

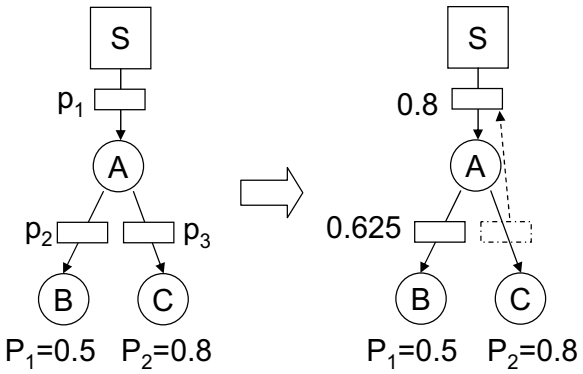

Figure 2. Illustration of Example 3.1

The preceding observation rules out some types of load shedding configurations, but it is not enough to determine exactly where load shedding should be performed. The following simple example will lead us to a further observation about the structure of the optimal solution:

Example 3.1 *Consider a simple data flow diagram with 3 operators as shown in Figure 2. Suppose the query nodes q_1 and q_2 must have effective sampling rates equal to 0.5 and 0.8 respectively. Each operator (A, B, and C) is in its own shared segment, so load shedding could potentially be performed before any operator. Imagine a solution that places load shedders before all three operators A, B, and C with sampling rates p_1, p_2, and p_3 respectively. Since $p_1p_2 = 0.5$ and $p_1p_3 = 0.8$, we know that the ratio $p_2/p_3 = 0.5/0.8 = 0.625$ in any solution. Consider the following modification to the solution: eliminate the load shedder before operator C and change the sampling rates for the other two load shedders to be $p_1' = p_1p_3 = 0.8$ and $p_2' = p_2/p_3 = 0.625$. This change does not affect the effective sampling rates, because $p_1'p_2' = p_1p_2 = 0.5$ and $p_1' = p_1p_3 = 0.8$, but the resulting plan has lower processing time per tuple. Effectively, we have pushed down the savings from load shedder p_3 to before operator A, thereby reducing the effective input rate to operator A while leaving all other effective input rates unchanged.*

Let us define a *branch point* in a data flow diagram as a point where one shared segment ends by splitting into $k > 1$ new shared segments. We will call the shared segment terminating at a branch point the *parent segment* and the k shared segments originating at the branch point *child segments*. We can generalize the preceding example as follows:

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

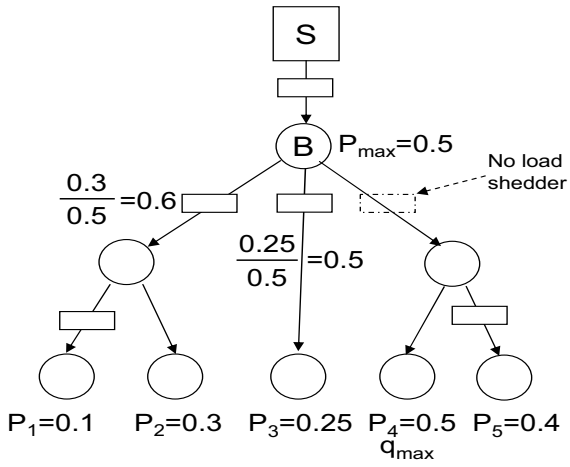

Figure 3. Illustration of Observation 3.4

Observation 3.4 *Let q_{max} be the query that has the highest effective sampling rate among all queries sharing the parent segment of a branch point B. In the optimal solution, the child segment of B that lies on the query path for q_{max} will not contain a load shedder. All other child segments of B will contain a load shedder with sampling rate P_{child}/P_{max}, where q_{child} is defined for each child segment as the query with the highest effective sampling rate among the queries sharing that child segment.*

Observation 3.4 is illustrated in Figure 3. The intuition underlying this observation is that, since all queries sharing the parent segment must shed at least a $(1 - P_{max})$-fraction of tuples, that portion of the load shedding should be performed as early as possible, no later than the beginning of the shared segment. The same intuition leads us to a final observation that completes our characterization of the optimal load shedding solution. Let us refer to a shared segment that originates at a data stream as an *initial segment*.

Observation 3.5 *Let q_{max} be the query that has the highest effective sampling rate among all queries sharing an initial segment S. In the optimal solution, S will contain a load shedder with sampling rate P_{max}.*

The combination of Observations 3.3, 3.4, and 3.5 completely specifies the optimal load shedding policy. This policy can be implemented using a simple top-down algorithm. If we collapse shared segments in the data flow diagram into single edges, the result is a set of trees where the root node for each tree is a data stream S_j, the internal nodes are branch points, and the leaf nodes are queries. For any internal node x, let P_x denote the maximum over all the effective sampling rates P_i corresponding to the leaves of the subtree rooted at this node. The pseudocode in Algorithm 1 operates over the trees thus defined to introduce load shedders and assign sampling rates starting with the call `SetSamplingRate`$(S_j, 1)$ for each data stream S_j.

Algorithm 1 Procedure $SetSamplingRate(x, R_x)$

```
if x is a leaf node then
  return
end if
Let x_1, x_2, ... x_k be the children of x
for i = 1 to k do
  if P_{x_i} < R_x then
    Shed load with p = P_{x_i}/R_x on edge (x, x_i)
  end if
  SetSamplingRate(x_i, P_{x_i})
end for
```

Theorem 3.6 *Among all possible choices for the placement of load shedders and their sampling rates which result in a given set of effective sampling rates for the queries, the solution generated by procedure* `SetSamplingRate` *has the lowest processing time per tuple.*

Proof: For any node x in the tree, we will refer to the product of the sampling rates of all the load shedders on the path from the root to node x as the *prefix path probability* of x. (The prefix path probability of x is defined to be 1 if there are no load shedders between the root and node x.) Note that in every recursive invocation of the `SetSamplingRate`(x, R_x) procedure, the second parameter R_x is equal to the prefix path probability of node x. To prove the theorem, we first prove the claim that for each node x other than the root, the prefix path probability of x is equal to P_x.

The proof of the claim is by induction on the height of the tree. It is trivially true for the root node. Consider any node b in the tree which is the child of some other node a. Assume that the claim holds for node a. When `SetSamplingRate` is called with a as an argument, it places a load shedder with sampling rate P_b/P_a at the beginning of edge (a, b). Thus, by the inductive hypothesis, the product of sampling rates of load shedders from the root to node b equals $P_a \times \frac{P_b}{P_a} = P_b$. This proves the claim.

Thus we guarantee that the prefix path probability of any node is equal to the highest effective sampling rate of any query which includes that node in its query path. No solution could set a prefix path probability less than this value since it would otherwise violate the effective sampling rates for that query. Thus the effective input rate of each operator is the minimum that can be achieved subject to the constraint that prefix path probabilities at the leaf nodes should equal the specified effective sampling rates. This proves the optimality of the algorithm. ■

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

3.2.1 Determining the Value of ϵ_{max}

An important point to note about the algorithm is that except for the first load shedder that is introduced just after the root node, the sampling rates for all others depend only on the ratios between effective sampling rates (each sampling rate is equal to $P_i/P_j = C_i/C_j$ for some i, j) and not on the actual P_i values themselves. As a consequence, it is not actually necessary for us to know the value of ϵ_{max} in advance. Instead, we can express each effective sampling rate P_i as $C_i\lambda$, where $\lambda = 1/\epsilon_{max}$ is an unknown multiplier. On each query path, there is at most one load shedder whose sampling rate depends on λ, and therefore the load equation becomes a linear function of λ. After running Algorithm 1, we can easily solve Equation 2.1 for the resulting configuration to obtain the correct value of λ that makes the inequality in Equation 2.1 tight.

Another consequence of the fact that only load shedders on initial segments depend on the actual P_i values is that the load shedding structure remains stable as the data stream arrival rates r_j change. The effective sampling rate P_i for each query q_i over a given data stream S_j depends on the rate r_j in the same way. Therefore, changing r_j does not affect the ratio between the P_i values for these queries. The only impact that a small change to r_j will have is to modify the sampling rates for the load shedders on the initial segments.

When determining ϵ_{max} in situations when the system load is only slightly above system capacity, an additional consideration sometimes needs to be taken into account: when no load shedding is performed along the query path for a given query, the error on that query drops to zero. By contrast, for each query, there is a minimum error threshold (C_i) below which no error guarantees based on Proposition 3.1 can be given as long as *any* load shedding is performed along the query path. As the effective sampling rate P_i increases, the relative error ϵ_i decreases continuously while $P_i < 1$ then makes a discontinuous jump at $P_i = 1$. Our algorithm can be easily modified to incorporate this discontinuity; the details are omitted here but can be found in the full version of this paper [4].

4 Extensions

We briefly discuss how to extend our techniques to incorporate quality of services guarantees and a more general class of queries.

4.1 Quality of Service

By taking as our objective the minimization of the maximum relative error across all queries, we have made the implicit assumption that all queries are equally important. In reality, in many monitoring applications some queries can be identified as being more critical than others. When the system is heavily loaded and load shedding becomes necessary, it would be preferable to shed load from the less important queries before affecting the more important queries. Our techniques can easily be adapted to incorporate varying quality of service requirements for different queries, either through the introduction of query weights, or query priorities, or both.

One modification to our technique would be to allow users to associate a weight or importance w_i with each query q_i. With weighted queries, the goal of the system becomes to minimize the maximum *weighted* relative error. This is easily accomplished by a straightforward change: when computing the effective sampling rate target for the queries, instead of ensuring that C_i/ϵ_{max} is equal for all queries q_i, we ensure that $C_i/(w_i\epsilon_{max})$ is equal. In other words, instead of $P_i \propto C_i$ we have $P_i \propto C_i w_i$. Thus, the weights can easily be assimilated into the C_i's.

An alternative way of specifying query importance is to assign a discrete priority level (e.g., high, medium, or low) to each query. When query priorities are specified, the goal of the system becomes to minimize the maximum relative error across all queries of the highest priority level. If all highest-priority queries can be answered exactly, then the system attempts to minimize the maximum relative error across queries with the second-highest priority level, and so on. Queries with lower priorities are ignored. Again, the modifications to our techniques to handle prioritized queries are straightforward.

4.2 More General Query Classes

We have discussed the load shedding problem in the context of a particular class of data stream monitoring queries, aggregation queries over sliding windows. However, the same techniques that we have developed can be applied to other classes of queries as well. One example is monitoring queries that have the same structure as the ones we have studied, except that they have set-valued answers instead of ending with an aggregation operator. This type of monitoring query could be useful when the goal of monitoring is to identify individual tuples that are unusual in some way, e.g., tuples with unusually large values for some attribute. In the case of set-valued queries, an approximate answer consists of a random sample of the tuples in the output set. The metric of relative error is not applicable to set-valued queries. Instead, we can measure error as the percentage of tuples from the query answer that are missing in the approximate answer. The goal of the system then becomes to minimize the maximum value of this quantity across all queries, optionally with query weights or priorities. Our algorithm can be made to optimize for this objective by simply setting C_i

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

for each query equal to 1 (or alternatively, equal to the query weight).

Another class of queries that might arise in data stream monitoring applications is aggregation queries with "group-bys". One way to adapt our techniques to incorporate group-by queries is to consider a group-by query to be multiple queries, one query for each group. However, all these groups (queries) share the entire query path and as a result the effective sampling rate will be the same for all of them. Consequently, the group that will have the maximum relative error will be the one with the maximum C_i value. Since our error metric is the maximum relative error among all groups across queries, within each group-by query, the group with maximum C_i value will be the only group that will count in the design of our solution. Thus, we can treat the group with maximum C_i value as the representative group for that query.

4.3 Incorporating Load Shedding Overhead

The results in Section 3 are based on the assumption that the cost (in terms of processing time) to perform load shedding is small relative to the the cost of query operators. In an actual system implementation, even simple query operators like basic selections generally have considerable overhead associated with them. A load shedder, on the other hand, involves little more than a single call to a random number generator and thus can be very efficiently implemented. In empirical tests using our STREAM system, we found that the processing time per tuple for a load shedding operator was only a small fraction of the total processing time per tuple even for a very simple query.

In some applications, however, the relative cost of load shedding may be larger, to the point where ignoring the overhead of load shedding when deciding on the placement of load shedders leads to inefficiencies. This could occur, for example, if the techniques described in this paper were to be applied in a specialized, high-performance stream processing application where the processing time per tuple was extremely low. Our same basic approach could be applied in such a context by associating a processing cost per tuple with load shedding operators and including their cost in the load equation. We have developed a modified algorithm based on dynamic programming to find the best placement of load shedders given a load equation modified in this fashion. The details are presented in [4].

5 Experiments

We have implemented the load shedding algorithm described in this paper in our data stream management system called *STREAM*, for *STanford stREam dataA Manager*. STREAM is a general-purpose stream processing system, suitable for a variety of data stream monitoring applications, though one of our focuses has been the network monitoring domain. The queries and data we used for our experiments are drawn from network monitoring. The STREAM system architecture is described in [21].

Using the STREAM system, we sought to answer two empirical questions about our algorithm:

1. Do the load shedding decisions made by our algorithm, which are based on the probability bounds from Proposition 3.1, produce good approximate answers on real-world data streams?

2. Does our algorithm provide an appreciable benefit over a basic load shedding scheme that simply discards a fraction of arriving tuples when stream rates exceed system capacity?

Experiments measuring the actual accuracy achieved by our algorithm on real data (the first question) are useful to help understand the impact of two factors that are difficult to gauge without empirical evidence. First, our load shedding algorithm is based on parameters that are estimated from past data in order to make predictions about future data. To the degree that the future does not resemble the past, this could be a cause of inaccuracy. Second, our algorithm is based on inequalities that place upper bounds on the probability of an error of a given size. Because these bounds are not tight—a tight bound is not mathematically tractable—the actual probability of error may be less than predicted. This factor might introduce inefficiency in our allocation of resources between queries. However, as evidenced by the results presented in Section 5.2, in practice the effects of these two factors are not too large.

Although our load shedding algorithm is fairly simple, it does require the presence of some monitoring infrastructure that tracks stream rates, operator selectivities, etc, in order to function effectively. A comparison of the degree of accuracy obtained by our algorithm with what can be achieved using a simpler policy is useful in evaluating whether the increased system complexity necessary for our technique is justified by improvements in query accuracy. The experimental results show that the relative error achieved by a baseline load-shedding scheme, which sheds load by dropping a fraction of newly-arriving tuples as they arrive, is about two to four times as high on average than the error achieved by our algorithm. The gap between the worst-case performance of the two techniques is even wider.

5.1 Experimental Setup

We ran our experiments on a single-processor Linux server with 512 MB of main memory. The dataset we used for our experiments contains one hour of wide-area network

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

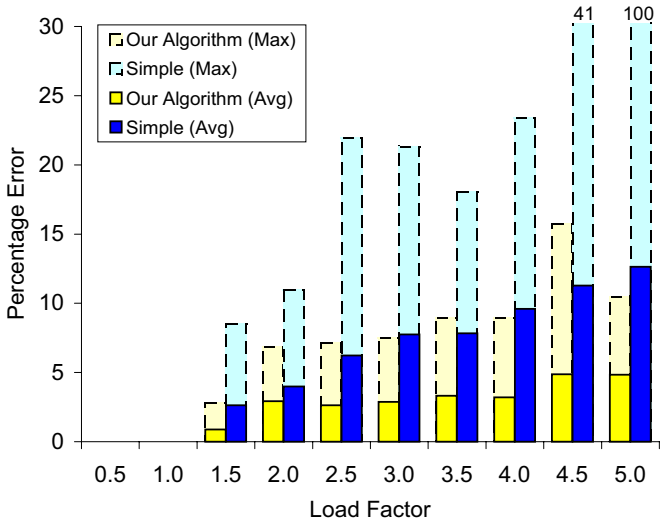

Figure 4. Accuracy at various levels of load

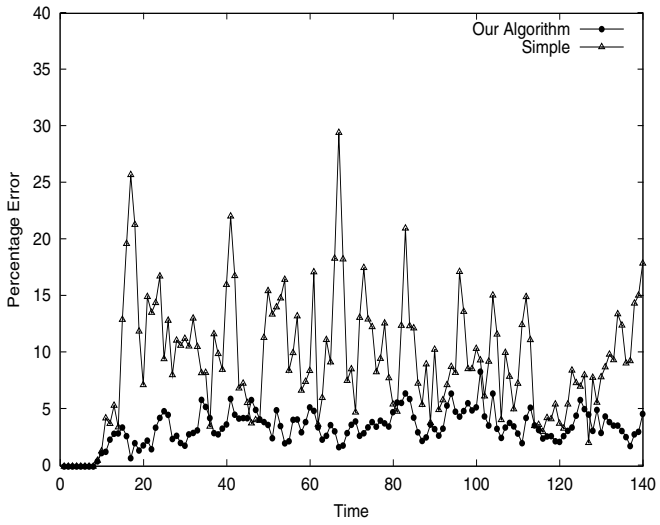

Figure 5. Accuracy of load shedding

traffic between Digital Equipment Corporation and the rest of the world. DEC has made the dataset freely available for download from the Internet Traffic Archive [16]. We reduced the data set, which was collected in ASCII tcpdump format, to a binary format suitable for efficient processing by our system. The reduced format that we used has 24-byte records, one record per IP packet, consisting of six 4-byte columns. There is one timestamp column and five integer-valued columns, *srcHost*, *destHost*, *srcPort*, *destPort*, and *size*, representing the source and destination hosts and ports for the packet and the packet size. Packet size was only known for TCP packets, so the size field is set to -1 to identify UDP packets.

We constructed a 7-query workload consisting of typical monitoring queries that might be posed by a network analyst. The queries applied various filters based on transport protocol (TCP vs. UDP), port of origin (e.g. packets originating from ports 80 and 20 generally represent HTTP and FTP traffic, respectively), packet size, and host of origin or destination. The queries computed either counts of packets or sums of the total bytes in packet payloads, measured over sliding windows of various durations. There were several shared selection conditions among the queries. All seven monitoring queries were executed simultaneously, with operator sharing where applicable.

The sliding windows in the queries were based on the timestamps present in the data (as opposed to the clock times when tuples arrived in the system during a particular test run). In this way, we ensured that the correct query answers were the same regardless of the rate at which the dataset was streamed to the system.

We compared the performance of two different load shedding strategies: (1) the algorithm described in this paper and (2) a baseline strategy that drops tuples as they enter the system, but not at any intermediate points in the query plan. The only difference between the strategies was in the algorithm used to place load shedding operators. Both strategies used statistics about operator execution times, operator selectivities, and stream rates that were monitored by the system during query execution and updated at one-second intervals to determine the degree of load shedding that was necessary. We measured the performance of each strategy using a wide range of different input stream rates.

5.2 Experimental Results

Figure 4 shows the average and worst case approximation error of the two load shedding strategies at various levels of system load. System load is expressed as a multiple of system capacity. (For example, a system load of 2 corresponds to an input stream rate that is twice as high as the system throughput when no load shedding is present.)

When conducting the experiments, we first constructed a trace containing the correct query answers by disabling load shedding, streaming the dataset to the system, and recording the query responses, tagged with timestamps from the dataset. To measure the error between an approximate answer trace produced by load shedding and the exact answers, we discretized time into one-second intervals and computed the average relative error for each query in each interval. Then, for each one-second interval, we used the maximum of the seven error values computed for the seven queries as the overall measure of performance for that interval. The average approximation error depicted in Figure 4

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

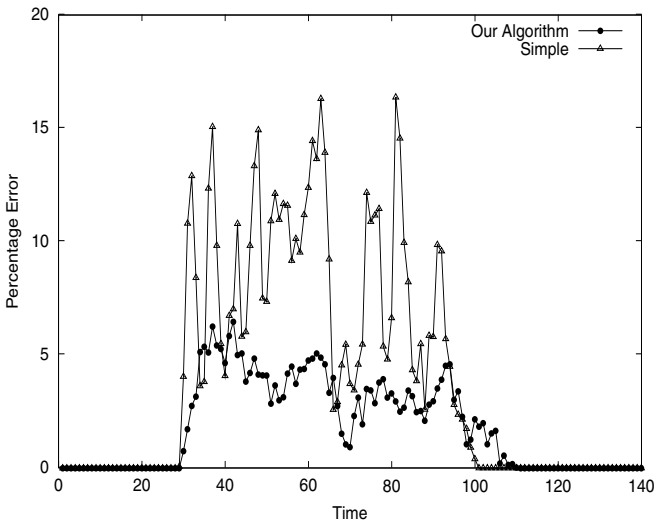

Figure 6. Adjusting to variable stream rates

is computed by calculating the approximation error for each one-second time interval and then averaging across all time intervals. The maximum approximation error is similarly computed by taking the maximum across all time intervals. We observe that the error of the baseline strategy is, on an average, more than twice as high as compared to our strategy. The difference is amplified if we consider the worst-case error. In general we observe that, even when the system load is five times the system capacity, our strategy gives low average relative error (less than 5%) on real data sets for the network traffic monitoring application.

Figure 5 provides a more detailed view of how answer quality varies over time. The data for the figure was generated using a system load factor of 3, i.e., the input stream rate was three times greater than the system capacity. As can be seen from the figure, the relative error from our algorithm was generally in the 2%-5% range, while the relative error for the simple strategy was around 5%-20%. Both methods exhibited some fluctuation in relative error over time, with the magnitude of the fluctuations being somewhat greater for the simple strategy than for our load shedding algorithm.

To test the ability of our system to adapt to changes in system load over time, we conducted an experiment where the input stream rate was varied over time. We initially sent data to the system at a rate just below the system capacity. After some time, we suddenly increased the rate to three times the system capacity. Finally, after some more time had passed, we restored the input rate to its original level. The system responded as expected, as shown in Figure 6. Initially, no load shedding was performed and exact answers were returned by the system. After the stream rate increased, load shedding became necessary, and query answers became approximate. However, the quality of the approximation was better when the placement of load shedding operators was determined using our algorithm as compared to the simple strategy. When the system rate returned to a lower level, the system detected that fact and reverted to a configuration where no shedding was performed.

6 Related Work

There has been a great deal of recent research on data stream query processing, both from the algorithms and systems perspectives. The survey paper [3] contains a good overview of work in the field. Some recent systems that incorporate stream processing include Aurora [5], Niagara [8], STREAM [21], and Telegraph [12], among many others.

There has been prior work in the general area of adaptivity in data stream systems, notably [14], which uses adaptive tuple routing algorithms to dynamically modify query plan structure in response to changing data characteristics. The most closely related work of which we are aware is the load shedding performed by the Aurora system [22]. The approach taken in [22] is different than ours in that it relies on quality of service functions that specify the utility of processing each tuple from a data stream. A different quality of service function is provided for each query. The Aurora approach is best suited for applications where the value of processing each individual tuple is independent of the other tuples that are processed, an assumption that does not hold for aggregation queries, the query class that is our focus in this paper.

The papers [9, 18] also consider the problem of how to most effectively shed load in an overloaded data stream system. The class of queries considered is windowed joins between data streams, and the metric being optimized is the number of tuples produced in the approximate answer (in the case of [9]) or the output rate of tuples (in the case of [18]). For this reason, the techniques proposed in [9, 18] are not applicable to our problem setting, where there are no joins between streams and the metric being optimized is the accuracy of approximate answers to aggregate queries.

Some other approaches to dealing with data stream rates that temporarily exceed system capacity focus on limiting the memory utilization of the system. Improving the efficiency of the allocation of memory among query operators, as proposed in [18] and [21], or among inter-operator queues, as in [2], are alternatives to load shedding that can improve system performance when limited main memory is the primary resource bottleneck.

The analytical techniques used in this paper are similar to those used in approximate query processing systems such as AQUA [1]. Our work is similar to [6] in that both take

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

as their objective the minimization of relative error across a given set of aggregate queries. However, the techniques in [6] are different from ours and are designed for standard stored data sets rather than data streams. Research in online aggregation, where progressively refined answers are given to an aggregation query over a stored data set based on the tuples processed so far, is more similar to data stream query processing, but prior work in online aggregation, such as [11, 13], has concentrated on a single-query scenario rather than optimizing across multiple continuous queries.

7 Summary

It is important for computer systems to be able to adapt to changes in their operating environments. This is particularly true of systems for monitoring continuous data streams, which are often prone to unpredictable changes in data arrival rates and data characteristics. We have described a framework for one type of adaptive data stream processing, namely graceful performance degradation via load shedding in response to excessive system loads. In the context of data stream aggregation queries, we formalized load shedding as an optimization problem with the objective of minimizing query inaccuracy within the limits imposed by resource constraints. Our solution to the load shedding problem uses probabilistic bounds to determine the sensitivity of different queries to load shedding in order to perform load shedding where it will have minimum adverse impact on the accuracy of query answers.

8 Acknowledgements

This work was supported in part by NSF Grants IIS-0118173 and EIA-0137761, with additional support from a Rambus Corporation Stanford Graduate Fellowship (Babcock), a Siebel Scholarship (Datar), and an Okawa Foundation Research Grant, an SNRC grant, and grants from Microsoft and Veritas (Motwani). A preliminary version of this work was presented in the MPDS 2003 workshop.

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proc. 1999 ACM SIGMOD Conf.*, pages 275–286, June 1999.

[2] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in stream systems. In *Proc. 2003 ACM SIGMOD Conf.*, June 2003.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 2002 ACM Symp. on Principles of Database Systems*, June 2002.

[4] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams (full version). In preparation.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams–a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, Aug. 2002.

[6] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *Proc. 2001 ACM SIGMOD Conf.*, pages 295–306, May 2001.

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc. 1999 ACM SIGMOD Conf.*, pages 263–274, June 1999.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. 2000 ACM SIGMOD Conf.*, pages 379–390, May 2000.

[9] A. Das, J. Gehrke, and M. Riedwald. Approximate join processing over data streams. In *Proc. 2003 ACM SIGMOD Conf.*, pages 40–51, 2003.

[10] S. Floyd and V. Paxson. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[11] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. 1999 ACM SIGMOD Conf.*, pages 287–298. ACM Press, 1999.

[12] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[13] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *Proc. 1997 ACM SIGMOD Conf.*, pages 171–182, May 1997.

[14] J. Hellerstein, S. Madden, V. Raman, and M. Shah. Continuously adaptive continuous queries over streams. In *Proc. 2002 ACM SIGMOD Conf.*, June 2002.

[15] W. Hoeffding. Probability inequalities for sums of bounded random variables. In *Journal of the American Statistical Association*, volume 58, pages 13–30, Mar. 1963.

[16] Internet Traffic Archive, trace DEC-PKT-4. http://www.acm.org/sigcomm/ITA/.

[17] T. Johnson. Personal communication.

[18] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. 2003 Intl. Conf. on Data Engineering*, Mar. 2003.

[19] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. 2002 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, Aug. 2002.

[20] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1):1–15, Feb. 1994.

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

[22] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. 29th Intl. Conf. on Very Large Data Bases*, Sept. 2003.

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE