Load Shedding Techniques for Data Stream Systems

Brian Babcock* Mayur Datar† Rajeev Motwani‡

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

1 Introduction

Many data stream sources (communication network traffic, HTTP requests, etc.) are prone to dramatic spikes in volume. 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. Therefore, it is important for systems processing continuous monitoring queries over data streams to be able to 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.

While some heuristics for load shedding have been proposed earlier ([C+02, M+03]), a systematic approach to load shedding with the objective of maximizing query accuracy has been lacking. The main contributions of our work 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 throughput must match or exceed the data input rate.
2. We describe a load shedding technique based on the introduction of random sampling operators into query plans, and we give an algorithm that finds the optimum placement of sampling operators for an important class of monitoring queries, sliding window aggregate queries over data streams.

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. We do not consider queries that involve joins between multiple streams, as such queries are comparatively rare in practice. A *sliding window aggregate* is a aggregation function (we consider SUM and COUNT) applied over a sliding window of the most recently-arrived data stream tuples

The inputs to the load shedding problem consist of a set of queries $q_1, \ldots, q_n$ over data streams $S_1, \ldots, S_m$ and a set of query operators $O_1, \ldots, O_k$ that are arranged in a data flow diagram consisting of a collection of trees. The root of each tree is a data stream, the leaves are the queries that reference that data stream, the internal nodes are query operators, and edges represent data flow. Every operator O_i is associated with two parameters: its selectivity s_i and its processing time per tuple t_i. 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, measuring average rate of tuple arrival. We assume that, using historical values, we can get reasonable estimates for the various parameters.

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. Moreover, as there are multiple queries, we aim to minimize the maximum error across all queries.

The technique that we propose for shedding load is to introduce random sampling 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.

*Supported in part by NSF Grant IIS-0118173 and a Rampus Corporation Stanford Graduate Fellowship.

†Supported in part by NSF Grant IIS-0118173 and a Siebel Scholarship.

‡Supported in part by NSF Grant IIS-0118173, an Okawa Foundation Research Grant, an SNRC grant, and grants from Microsoft and Veritas.

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. Let $src(i)$ denote the index of the data stream source feeding operator O_i, and let U_i denote the set of operators "upstream" of O_i—that is, the operators that fall on the path from $src(i)$ to O_i. The constraint that tuples must be processed at a rate at least as fast as their arrival rate can be expressed by the following load equation:

Equation 1 (Load Equation) *Any load shedding policy must select sampling rates p_i to ensure that:*

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

The equation assumes that the sampling operators themselves have zero processing time per tuple. The problem thus becomes to select sampling rates p_i to minimize the maximum relative error across queries while satisfying the load equation. Because the relative error depends on the outcome of the random coin tosses made by the load shedders, the relative error cannot be predicted exactly. Therefore, we say that a load shedding policy achieves error ϵ if, for each query q_j, the relative error resulting from using the policy to estimate the answer to q_j exceeds ϵ with probability at most δ, where δ is some small constant (e.g. 0.01).

3 Load Shedding Algorithm

The *effective sampling rate* P_j corresponding to each query q_j is equal to the product of the sampling rates of the different load shedders introduced on the path from $src(j)$ to q_j. The introduction of load shedders is equivalent to answering this aggregation query using a uniform random sample of the input tuples where each tuple is included independently with probability P_j. Our algorithm for determining where load shedding should be performed and setting the sampling rate parameters p_i has two steps:

1. Determine effective sampling rates for the queries that will distribute error evenly among all queries.
2. Find p_i values that achieve the desired effective sampling rates and satisfy the load equation.

3.1 Allocation of Work Among Queries

Since we aim to minimize the maximum relative error, it is easy to see that an optimal solution will have the same relative error for every query. Let us assume that we have correctly guessed the maximum relative error ϵ_{max}[1]. Assuming the estimated parameters (selectivities, stream rates, etc.) to be accurate, we can use Hoeffding bounds [Hoe63] to find an effective sampling rate P_j for every query q_j so that the relative error on q_j will be no more than ϵ_{max}, with high probability. The rate P_j may be different for each query.

Equation 2 (Effective Sampling Rate) *The target effective sampling rate P_j for query q_j is equal to:*

$$P_j = \frac{1}{\epsilon_{max}} \sqrt{\frac{\sigma_j^2 + \mu_j^2}{2 N_j \mu_j^2} \log \frac{2}{\delta}}$$

In Equation 2, N_j is the number of tuples in each sliding window that contribute to the answer for query q_j (i.e., the tuples that pass all filters). The value of N_j can be calculated from the size of the sliding window, the estimated selectivities of the operators in the query path for q_j, and (in the case of time-based sliding windows) the estimated data stream rate r_j. Recall that we are given estimates for the mean μ_j and standard deviation σ_j for each aggregation operator as inputs to the load shedding problem.

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_j for each query q_j, modify the diagram by inserting load shedding operators and setting their sampling rates so that the effective sampling rate for each query q_j is equal to P_j and the total processing time is minimized.*

3.2 Placement of Load Shedders

If there is no sharing of operators among queries, the optimal solution is straightforward: introduce a load shedder with sampling rate $p_i = P_j$ before the first operator in the query path for each query q_j. However, when some operators are shared by multiple queries, the situation becomes more complicated, since some queries require higher effective sampling rates than others to achieve the same relative error. There is a tradeoff between placing load shedders early in the query plan (i.e. close to the root of the data flow diagram), where they are most effective at decreasing processing time but negatively impact the accuracy of many queries, and placing load shedders late in the query plan (i.e. on a branch of the data flow diagram that leads to only a single query), where the reduction in processing time is less but the accuracy of fewer queries is affected. We have developed an algorithm that selects the optimal p_i values so that the product of p_i's along each query path equals the desired effective sampling rate for that query path while minimizing the processing time (the left-hand side of Equation 1).

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. It is easy to see that in any optimal solution, load shedding is

[1] In fact, we don't need to guess this value, but instead can compute it accurately. Details are in [BDM03].

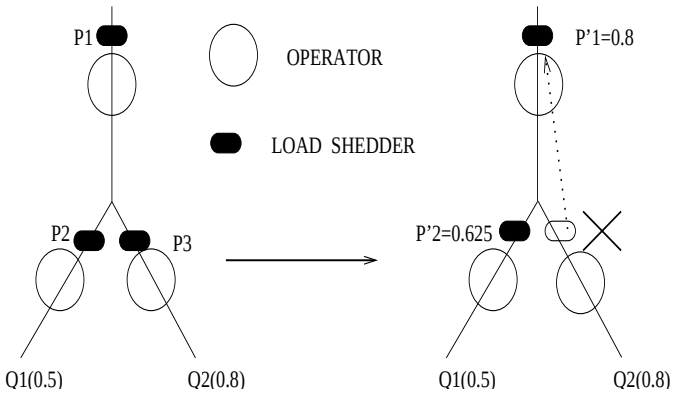

Figure 1: Illustration of Example 1

only performed at the start of shared segments. This 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 demonstrates the main idea behind the algorithm that constructs the optimal solution:

Example 1 *Consider a simple data flow diagram with 3 operators as shown in Figure 1. 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 $p'_1p'_2 = p_1p_2 = 0.5$ and $p'_1 = p_1p_3 = 0.8$, but the resulting plan has lower processing time per tuple. Effectively, we have pushed "down" ("up" as per the diagram) 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.*

Repeated application of the idea illustrated in Example 1 leads to our algorithm for placing load shedders. 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_k, 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_j 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`$(S_k, 1)$ for each data stream S_k. The algorithm is efficient, requiring only a single bottom-up traversal of the data flow diagram.

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 1 *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.*

4 Conclusion

We formalize the task of load shedding in data stream systems as an optimization problem and provide a brief sketch of our solution to this problem. The solution has two parts: first choose target sampling rates for each query, then place load shedders to realize the targets in the most efficient manner possible.

References

[BDM03] B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems (full version). 2003. Draft in preparation.

[C+02] D. Carney et al. Monitoring streams–a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, August 2002.

[Hoe63] W. Hoeffding. Probability inequalities for sums of bounded random variables. In *Journal of the American Statistical Association*, volume 58, pages 13–30, March 1963.

[M+03] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.