Load Shedding on Data Streams

Nesime Tatbul
Brown University
tatbul@cs.brown.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

Michael Stonebraker
M.I.T.
stonebraker@lcs.mit.edu

1 Introduction

A data stream is a continuous and typically rapid feed of data items from a variety of sources like sensors, geo-positioning devices, or computer programs. Many new applications call for real-time monitoring on data streams. Data Stream Management Systems have emerged to enable efficient processing of data streams to serve the needs of such applications.

One of the major challenges in data stream management is to support real-time processing with limitations on system resources like CPU, memory, or bandwidth. With a large number of data streams and continuous queries, it is possible to experience shortage in resources as data arrival rates unpredictably increase. This will cause latency in processing. To avoid late results, the system must shed some of the load in a controlled fashion. Thus, *load shedding* is the process of dropping excess load from the system when the demand on resources is above the system capacity. Load shedding reduces resource requirements by dropping data, thereby sacrificing the accuracy of the query answers. The main goal is to minimize this degradation in accuracy.

2 Load Shedding in Aurora

Aurora Data Stream Management System [2] manages the processing of data streams by a *query network* – a collection of continuous queries, each of which consists of a sequence of operators. We model load shedding as the automatic insertion of *drop* operators into a running network [9]. A drop operator transmits fewer output tuples than it gets as input. We consider two fundamental types of drop operators. *Random drop* discards a specified fraction of its inputs randomly, whereas *semantic drop* filters out certain input values based on a predicate.

The load shedding process consists of three fundamental decisions: (1) when, (2) where in the query network, and (3) how much load to shed. In addition to various system statistics such as operator costs and selectivities, we exploit application-specific Quality of Service (QoS) information to make these decisions. We model QoS as a set of functions that relate a parameter of the output to its utility. The two main functions we consider in load shedding are (1) value-based QoS that shows the importance of the values in the output space, and (2) loss-tolerance QoS that maps the fraction of data delivered to its utility.

Load shedding is an optimization problem and can be formally stated as follows. We are given a query network N, a set of input streams I with certain data arrival rates, and a processing capacity C for the system that runs N. Let $N(I)$ indicate the network N operating on inputs I, and $\mathit{Load}(N(I))$ represent the load as a fraction of the total capacity C that network $N(I)$ presents. Load shedding is typically invoked when $\mathit{Load}(N(I)) > C$. The problem is to find a new network N' that is derived from network N by inserting drops along existing arcs of N such that $\mathit{Load}(N'(I)) < C$ and $U_{accuracy}(N(I)) - U_{accuracy}(N'(I))$ is minimized. $U_{accuracy}$ is the aggregate utility that is measured from the loss-tolerance QoS graphs of the application set. $U_{accuracy}(N(I))$ represents the measured utility when there is no load shedding (i.e., there are no inserted drops). Thus, $U_{accuracy}(N(I)) - U_{accuracy}(N'(I))$ is the loss of utility introduced by load shedding. It is this quantity that we want to minimize.

One important property of our load shedding technique is that it is designed to be as general to work with any reasonable scheduling algorithm. The basic assumption is that any cycles that are recovered as a result of load shedding are used sensibly by the scheduler.

We will now briefly summarize our approach to each of the major decisions in load shedding.

1. Determining when to shed load. We continuously evaluate the current processing load of the query network. If the load is above the capacity, the excess needs to be shed. Otherwise, unnecessary drops, if there are any, must be removed. Using operator costs and selectivities, we compute a *load coefficient* for each input stream. This coefficient represents the number of processor cycles required to push a single input tuple through the network. At run-time, these coefficients are instantiated by the input rates to compute the actual load of the network.

2. Determining where to shed load. Tuples can be dropped at any point in the processing network. There are two properties that make a point more desirable than the others: (1) maximal load gain, and (2) minimal aggregate utility loss. Dropping tuples earlier in the network avoids wasting work and saves more processing cycles. Dropping them from arcs shared among multiple applications may result in more utility loss. Our technique first identifies potential drop locations in the query network. We compute *loss/gain ratios* for each of these locations. In order to guarantee minimal loss per maximal gain, drops should be inserted to these locations in the order of increasing ratios. Our technique for deriving loss-tolerance QoS from value-based QoS enables a unified treatment for random and semantic drops in determining where and how much load to shed.

3. Determining how much load to shed. Once we have determined where to insert a drop operator, we must decide the magnitude of that drop. In the case of a random drop, this involves deciding on the percentage of tuples to drop. In the case of a semantic drop, we must also decide which tuples to discard (i.e., the form of the predicate). We need to shed as much load as needed to recover the cycles that exceed the processing capacity. The decision of how much load to shed is made interleaved with the decision of where to shed load. Each drop location can save cycles up to a certain amount. We greedily choose drop locations in an order to minimize utility loss while maximizing cycles saved until the excess load is removed. For semantic drops, after the amount of drop has been decided, the filtering predicate is determined from value-based QoS and output value histograms such that the lowest utility value intervals are dropped first.

The run-time overhead can be reduced when operator costs, selectivities, histograms for output values and an estimation about proportions among input data rates are known in advance. In this case, our approach statically builds a data structure called the Load Shedding Road Map (LSRM). LSRM materializes a sequence of drop insertion plans along with the load savings each provides. At run-time, when an overload is detected, we simply search the LSRM to find a plan for recovering the required number of processor cycles. As statistics change, the order of loss/gain ratios of the drop locations may change. In this case, the drop insertion plans in the LSRM may start to deviate from optimal, i.e., they may not guarantee minimal utility loss any more for some levels of overload. Thus, there is a tradeoff between an optimal LSRM and the overhead of maintaining it. Our approach can be tuned to adjust the level of tolerance for non-optimality in exchange for better run-time performance.

Load shedding with random drops and semantic load shedding are alternatives for each other. If value-based QoS and output value histograms are available, semantic load shedding should be used as it causes less value utility loss [9].

3 Related Work

Load shedding is not a new idea. It has been previously studied in the context of networking and multimedia streaming. To control congestion in computer networks, localized algorithms are applied at individual network nodes, mostly based on queue sizes, timestamps, or sender-specified priority bits [10]. In contrast, our algorithm is based on global knowledge about a query network and provides an end-to-end solution. This solution takes applications' QoS specifications as well as actual message contents into account. In multimedia streaming, the focus has been on adjusting the amount of data transmission for effective network bandwidth usage. Both network and application layer techniques have been proposed [3].

Load shedding is essentially an approximate query answering technique. Various techniques for producing approximate answers in exchange for faster execution have been studied in the database literature before [1]. However, in the context of data streams, approximation has to be applied as data continues to arrive. The process is also more dynamic in that the degree of approximation has to be adjusted as the difference between supply and demand on resources changes. More recent work have explored approximate query processing techniques on data streams both for aggregation queries [5, 6] and sliding window joins [4, 7]. We not only consider individual operations, but also complete query networks. These networks may be composed of a variety of operators and may serve multiple applications with shared operations. Also, our approach covers the complete process from detection of the overload to its resolution. We originally proposed to do semantic load shedding by filtering data that has lower utility to the applications [2]. Das et.al. have a different view of semantic load shedding, concentrating on join processing and the semantic distance of the approximate answer [4]. Dropping tuples when input rate exceeds the service rate has also been discussed in rate-based evaluation of window joins [7]. In this work, the focus has been on random drops rather than semantic ones. The STREAM system uses several approximation techniques on stream queries [8]. Synopses are used to reduce memory requirements of operators in a query plan; random sampling is used as a means of load shedding. We not only provide techniques for sampling using random drops but also provide semantic load shedding based on tuple values.

4 Future Directions

Our agenda for future work includes generalizing our techniques to query networks with more complex operators. We will also extend our load shedding algorithms for the management of other resources like memory.

References

[1] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4):3–45, 1997.

[2] 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 *VLDB Conference*, pages 215–226, Hong Kong, China, August 2002.

[3] S. Cen, C. Pu, and J. Walpole. Flow and Congestion Control for Internet Streaming Applications. In *Multimedia Computing and Networking (MMCN98)*, 1998.

[4] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. In *ACM SIGMOD Conference*, San Diego, CA, June 2003 (to appear).

[5] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams Databases. In *ACM SIGMOD Conference*, pages 13–24, Santa Barbara, CA, May 2001.

[6] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-pass Summaries for Approximate Aggregate Queries. In *VLDB Conference*, pages 79–88, Roma, Italy, September 2001.

[7] J. Kang, J. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *IEEE ICDE Conference*, Bangalore, India, March 2003.

[8] 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 *CIDR Conference*, pages 245–256, Asilomar, CA, January 2003.

[9] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003 (to appear).

[10] C. Yang and A. V. S. Reddy. A Taxonomy for Congestion Control Algorithms in Packet Switching Networks. *IEEE Network*, 9(5):34–44, 1995.