Dealing with Overload in Distributed Stream Processing Systems

Nesime Tatbul Stan Zdonik

Brown University, Department of Computer Science, Providence, RI USA
E-mail: {tatbul, sbz}@cs.brown.edu

Abstract

Overload management has been an important problem for large-scale dynamic systems. In this paper, we study this problem in the context of our Borealis distributed stream processing system. We show that server nodes must coordinate in their load shedding decisions to achieve global control on output quality. We describe a distributed load shedding approach which provides this coordination by upstream metadata aggregation and propagation. Metadata enables an upstream node to make fast local load shedding decisions which will influence its descendant nodes in the best possible way.

1. Introduction

Overload management has been an important challenge for large-scale dynamic systems where workload can unexpectedly increase while quality of service has to be maintained. These systems have developed various load management techniques from adaptive load distribution to admission control and load shedding. The choice of a specific technique depends on the characteristics of the workload, available protocols that define resource allocation policies as well as requirements of the application.

We consider the overload management problem in the context of distributed stream processing systems. In this environment, large numbers of continuous queries in the form of a collection of operator chains are distributed onto multiple servers. These queries are essentially dataflow diagrams that receive and process continuous streams of data from external push-based data sources. Real-time monitoring applications are especially well-suited to this kind of systems. In this domain, providing low-latency, high-throughput answers to queries is highly important.

Data streams can arrive in bursts, and provisioning the system resources for worst-case bursty load is in general not economically worthy. On the other hand, bursts in data rates may create bottlenecks at some points along the server chain. Bottlenecks may arise due to excessive demand on processing power at the servers or bandwidth shortage at the shared physical network that connects these servers. Bottlenecks slow down processing and dissemination, and cause delayed outputs. These bottlenecks have to be efficiently detected and quickly resolved to maintain quality of service.

Various load shedding techniques have been proposed for data stream processing systems (e.g., [4, 11, 15]). These techniques focus on single-server solutions. However, in distributed stream processing systems, each server node acts like a workload generator for its downstream nodes. Therefore, resource management decisions at a node will affect the characteristic of the workload received by its successor nodes.

Load shedding aims at dropping tuples at certain points along the server chain to reduce load. Unlike TCP congestion control, there are no retransmissions and dropped tuples are lost forever. This will have a negative effect on the quality of the results delivered at the query outputs. The main goal is to minimize the quality degradation. Because of the load dependency between nodes, a given node must figure out the effect of its load shedding actions on the load levels of its descendant nodes. Load shedding actions at all nodes along the chain will collectively determine the quality degradation at the outputs. This makes the problem more challenging than its centralized counterpart.

1.1. Motivating Example

Let us illustrate our point with a simple example. Consider the simple query network with two queries that are distributed onto two processing nodes A and B (Figure 1). Each small box represents a subquery with a certain cost and selectivity. Cost represents the CPU time it takes for one tuple to complete the subquery, and selectivity represents the ratio of the number of output tuples to the number of input tuples. Both inputs arrive at the rate of 1 tuple per second. Potentially each node can reduce load at its inputs by dropping tuples to avoid overload. Let's consider node A. Table 1 shows various ways that A can reduce its input rates and the consequences of this in terms of the load at both A and B, as well as the throughput observed at the query outputs (Note that we are assuming a fair scheduler

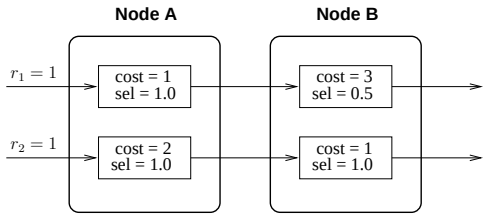

Figure 1. Example

Plan	Reduced rates at A	A.load	A.throughput	B.load	B.throughput	Result
0	1, 1	3	1/3, 1/3	4/3	1/4, 1/4	originally, both nodes are overloaded
1	1/3, 1/3	1	1/3, 1/3	4/3	1/4, 1/4	B is still overloaded
2	1, 0	1	1, 0	3	1/3, 0	optimal plan for A, but increases B.load
3	0, 1/2	1	0, 1/2	1/2	0, 1/2	both nodes ok, but not optimal
4	1/5, 2/5	1	1/5, 2/5	1	1/5, 2/5	optimal

Table 1. Alternate load shedding plans for node A

that allocates CPU cycles among the subqueries in a round-robin fashion). In all of these plans, A can reduce its load to the capacity limit. However, the effect of each plan on B can be very different. In plan 1, B stays at the same overload level. In plan 2, B's load increases more than twice its original load. In plan 3, B's overload problem is also resolved, but throughput is low. There is a better plan which removes overload from both A and B, while delivering the highest total throughput (plan 4). However, node A can only implement this plan if it knows about the load constraints of B. Because from A's point of view, the best local plan is plan 2. This simple example clearly shows that nodes must coordinate in their load shedding decisions to be able to achieve high-quality query results.

1.2. Design Goals

An effective load shedding solution for a distributed stream processing system should have the following properties:

Fast reactivity to load. The main goal of load shedding is to maintain low-latency query results even during load spikes. These situations need to be detected and addressed as soon as possible. To achieve this, the load shedding algorithm must be light-weight and the load shedding plans [1] must be efficiently deployed. In a distributed setting, having even a single overloaded node in the query pipeline is sufficient to boost latency at the outputs. Therefore, the solution must be equally attentive to each individual node's problem.

Global control on output quality. While maintaining low-latency results, the algorithm must also ensure that the quality of the query answers does not arbitrarily degrade. As evidenced by our example above, a load shedding plan that can deliver very high-quality results at a node locally, does not necessarily achieve similar quality at the global level. What eventually matters to the applications is the quality of the results that they observe at the query end-points. Therefore, regardless of where load is being shed in the node chain, the resulting effect at the query end-points must be kept under control.

Scalability. The load shedding algorithm should scale well with certain important performance factors. These include the number of server nodes, the number of input streams, and the amount of branching that may exist along the queries. Branching is important since it causes a stream to split and replicate its contents into multiple streams, creating more load in the system. Also, these replica streams may serve multiple different end-point applications. Therefore, shedding load before or after a split point makes a difference in output quality.

[1] A load shedding plan indicates where in the query plan to reduce load and the amount of the required reduction. Load can be reduced in various ways. In this paper, we focus on dropping tuples.

1.3. Solution Alternatives

We identify two general classes of solutions:

Centralized Approaches. In a centralized approach, a node can be designated as the coordinator. The coordinator node is informed about the complete query network topology. Additionally, it periodically collects performance statistics from each participating node. It produces a globally optimal plan based on this information. Each node receives its piece of the global plan and applies it. In this approach, nodes do not directly participate in planning and decision making; rather they passively implement the plan dictated by the coordinator. Therefore, this approach can respond slowly to local load spikes. However, high-quality outputs can possibly be achieved due to better global control.

Distributed Approaches. In a distributed approach, each node is responsible for detecting and handling its own overload locally. We expect this approach to react fast to load changes and thus maintain low latency. However, nodes need to communicate to come up with globally efficient solutions. If there is no global coordination between nodes, output quality can arbitrarily change. Hence, the challenge is to establish coordination with minimal overhead.

In the rest of this paper, we first describe our system model and the underlying assumptions that we make in Section 2. Then we present a sketch of our metadata-based solution approach in Section 3. In Section 4, we discuss important open issues that require further attention. We discuss related work in Section 5, and finally conclude in Section 6.

2. Models and Assumptions

We study the overload management problem for distributed stream processing systems in the context of our Borealis prototype system [1, 3]. Borealis accepts a collection of continuous queries, represents them as one large network of query operators (also known as a query diagram), and distributes the processing of these queries across multiple server nodes. Each participating server runs Aurora as its underlying query processing engine [2]. Borealis provides not only the core distributed operation functionality, but also various optimization and high availability techniques. The optimization techniques

include dynamic load distribution and balancing, parallel operator processing, and load shedding. Depending on the needs of the system, one or more of these optimizations can be applied at different levels of granularity, ranging from local to global. The Borealis statistics manager, deployed at each node, continuously collects data on various performance metrics such as operator costs, selectivities, tuple latencies, queue lengths, and so on. Statistical information is what enables most Borealis components to function effectively.

Data streams are modeled as append-only sequences of tuples. In addition to the regular data fields, tuples also carry a system-attached header with various fields such as arrival timestamp. These data streams are run through the queries which are composed of our well-defined set of operators, each of which performs operations such as filtering, windowed aggregation, merging, and correlation [2].

We mentioned earlier that bottlenecks in a distributed setting may arise both due to the lack of required processing power and also due to bandwidth limitations. In this paper, we limit our scope to the former problem.

There can be various quality metrics defined for the query outputs. For example, the goal can be to maximize the total query throughput, or the percentage of results delivered. There can also be priorities attached to individual queries. For the purposes of this paper, we assume that the quality metric to maximize is the total query throughput.

Finally, in this paper, we focus on tree-based server topologies.

3. A Metadata-based Approach

We propose a distributed load shedding approach which is based on informing upstream nodes about their children's constraints so that they can take load shedding actions that will influence their downstream nodes in the best possible way. This approach has two advantages:

- Nodes can make local load shedding decisions on their own, which provides fast reaction to load;
- Nodes are sensitive to their children's load constraints, which provides the load shedding node direct control on total quality loss at its downstream outputs.

In our approach, each node periodically sends metadata in the form of a Feasible Input Table (FIT) to its parent nodes. FIT is a summary of what a node expects in terms of its input rates and how that translates into a quality score at the downstream query end-points. When a node receives FITs from its children, it merges them into a single table. Furthermore, the parent maps the merged table from its outputs to its inputs and removes the entries that may be infeasible for itself. Finally, the parent propagates the resulting FIT to its own parents. This propagation continues until the input nodes receive the FIT for all their downstream nodes. Using its FIT, a node can shed load for itself and on behalf of its descendant nodes. Next we describe these steps in detail.

3.1. Feasible Input Table (FIT)

Each node maintains a Feasible Input Table (FIT). Given a node with m inputs, the FIT for this node is a table with $m+1$ columns. The first m columns represent the rates for the m inputs, and the last column represents the resulting output quality score. FIT represents the feasible input space for a node that will keep this node's and all of its descendants' CPU load below their available capacities. It is a way for a node to express its load expectations from its parent. The score column of a FIT is used to compare the estimated output quality for different entries. In some cases, the node may additionally need to keep a load shedding plan associated with a feasible input entry. As will be described below, this plan is completely local and transparent to the rest of the nodes.

A node with no descendants (i.e. a leaf node) generates its FIT from scratch; other nodes build on the FITs that they receive from their children.

3.2. Generating FIT

Leaf nodes generate their FIT based on the costs of the queries that they are assigned to execute. Consider such a node with a single query of cost c time units per tuple, which is fed by a single input with rate r tuples per time unit. The node is overloaded unless $r*c \leq 1$. Thus, it can handle an input rate of up to $1/c$ before it becomes overloaded. Any input whose rate is smaller than $1/c$ constitutes a feasible input for this node. Next consider a node with multiple query paths from its inputs to its outputs. Assume that there are m inputs, input i having an upper bound of R_i for its rate. This upper bound can be determined considering the total cost of the query paths fed by i in isolation from other paths. For each input i, we will select a set of values between 0 and R_i. Each different combination of the selected rate values for the input dimensions will constitute a row in FIT if this combination is a feasible one. Furthermore, each such combination will lead to a certain total query throughput. This total will be the score for that feasible input combination.

Determining Spread. Spread for an input dimension determines how far apart the rate values should be selected for that dimension. The smaller the spread, the closer we can get to the optimal feasible input. On the other hand, smaller spread requires that we consider a larger number of feasible points, which is more costly to store and process. We determine spread along each dimension in two alternative ways:

- *Based on a fixed maximum error from the optimal quality result:* Assume that the query path from input i has a total path selectivity of sel_i (e.g. if we send an input of rate r_i tuples per second along path i, the corresponding output rate will be $r_i * sel_i$). Given m inputs, the total throughput score will be $\sum_{i=1}^{m} r_i * sel_i$. When we divide a dimension into units of spread s_i, then the error in the total score along dimension i would be at most $s_i * sel_i$. Given a bound for maximum error in score denoted by ϵ_{max}, the

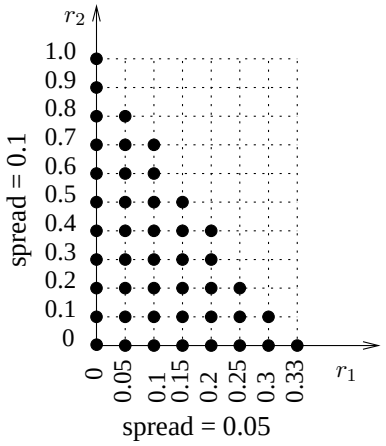

Figure 2. Feasible points

maximum error for each dimension must be at most $\frac{\epsilon_{max}}{m}$. Therefore, the spread along i must be: $s_i = \frac{\epsilon_{max}}{m * sel_i}$.

- *Based on a fixed table size for FIT:* Given a bound B for the number of points to be stored in FIT, we can determine the spread along each dimension as follows. The maximum error for each input dimension should be equal. Error on dimension i is $s_i * m * sel_i$. We know that R_i is the upper value bound for dimension i. Therefore, the number of points along i should be $B_i = \lceil \frac{R_i}{s_i} \rceil + 1$. Hence, $s_i = \frac{R_i}{B_i - 1}$. Therefore, $\frac{R_i}{B_i - 1} * m * sel_i$ must be equal along all dimensions. We also know that, $B = \prod_{i=1}^{m} B_i$. From these, we can easily compute B_i along one of the dimensions, using which we can compute the error for that dimension. Since errors should be equal on all dimensions, we can then compute s_i for all of the dimensions.

To illustrate, consider node B in Figure 1. The rate upper bounds for this node are $R_1 = 1/3$ and $R_2 = 1$. Assume that ϵ_{max} is given to be 0.1. Then, $s_1 = 0.1/(2 * 0.5) = 0.1$, and $s_2 = 0.1/(2 * 1.0) = 0.05$. The set of feasible points is shown in Figure 2. Thus, the FIT for this node has 46 entries. The entry (0, 1.0) provides the highest total throughput score of 1.0.

Handling Splits via Local Plans. The query network may include operators whose output may split to multiple paths. Each such path may have a different cost and a different path selectivity. Due to this difference, dropping from some branches may be more desirable. However, this is completely a local issue and need not be exposed to the parent. Instead, we allow a node to create FIT entries which are in fact not feasible and support these entries with additional local load shedding plans that drop tuples at split branches. By doing this, the node can achieve higher output scores.

We will illustrate this idea with a simple example. Consider the query network in Figure 3. The input splits into two branches. The top branch saves 2 units per dropped tuple at the expense of 1 output tuple, whereas the bottom branch saves 5 units per output tuple lost. Also, dropping from the input saves

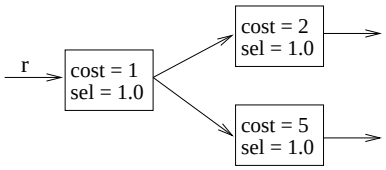

Figure 3. Splits

8 units while losing 2 output tuples. Dropping from the bottom branch is clearly the most beneficial. Therefore, the node should drop completely from the bottom branch before dropping from its inputs. If the input rate is r, then $5 * r$ out of the total load of $8 * r$ should be handled locally. In other words, the node will be able to handle an additional load of $5 * r$ beyond its capacity based on local drops. Hence, the node must make sure that $8 * r \leq 1 + 5 * r$. Therefore, $3 * r \leq 1$. Note that now the upper value bound for r must be $R = 1/3$. We will generate so-called *feasible* points based on R. Also, for each feasible point satisfying $3 * r \leq 1$, if it is also satisfying $8 * r \leq 1$, then no additional local drops are needed. Otherwise, we will create additional plans where some portion of the data on the bottom branch must be dropped locally. For example, given $r = 0.2$, it satisfies the extended load constraint ($3 * r \leq 1$), but it does not satisfy the original load constraint ($8 * r \leq 1$). We must additionally shed 60% of the load on the bottom branch. In this case, the total throughput score becomes 0.28. If we instead used the original constraint, then we would have to shed load to reduce to the nearest feasible point $r = 0.125$, which would give us the highest possible score of 0.25. In other words, by using the additional local plan, we can shed sufficient load and provide higher-quality output.

3.3. Propagating FIT Upstream

When a node A receives the FIT from its child node B, the feasible points in this table are expressed in terms of B's inputs. As we assume that the network bandwidth is sufficient and the transmission is reliable, these rates directly correspond to the rates at A's outputs. However, to be able to propagate the FIT further upstream, A has to express FIT in terms of its own inputs.

Each input i of A follows a query path to produce a certain output. Along this path, the rate of i changes by a factor determined by the product of the operator selectivies (say sel_i). Therefore, given an output rate r, the corresponding input rate for i is $\frac{r}{sel_i}$. To obtain A's FIT, we first apply this reverse-mapping to each row of B's FIT; the corresponding score for each row stays the same. Then, we eliminate from the resulting FIT the entries which may be violating A's load constraint.

If there is a split along the path from an input i to multiple outputs, and if all child branches of the split map to the same input rate value, then we just propagate that value as described above. Otherwise, we propagate the maximum of all input rates. The assumption here is that any additional reduction will be performed by applying tuple drops at remaining

branches of the split. The additional reduction is stored as a local plan associated with that particular FIT entry, and need not be propagated to the upstream nodes.

If node A has multiple child nodes, then the FITs of these children are combined by merging rows from each FIT with the rows from the other FITs. Any new entry violating A's load constraint has to be eliminated. The resulting score is the sum of the children's row scores.

3.4. FIT-based Load Shedding

Each node monitors its input rates. Based on these rates, it locates a row in its FIT and adopts a local load shedding plan that matches that row. More specifically, given input rates $r_1 \ldots r_m$, if FIT has at least one entry F where for all i, $F.r_i \geq r_i$ (with no local plan associated with it), then the node is not overloaded and no load needs to be shed. Otherwise, we find the FIT entry F with the highest score for which, for all i, $F.r_i \leq r_i$. Then we adjust input rates by dropping tuples such that for all i, $r_i == F.r_i$. In other words, r_i must be reduced by $1 - F.r_i/r_i$. Additionally, if there is a local plan associated with F, then that plan is also adopted.

To illustrate, if node B in Figure 1 observes input rates (0.2, 0.35), then it is not overloaded since (0.2, 0.4) is a feasible point (as shown in Figure 2). Therefore, no drops are necessary. However, if it observes input rates (0.5, 0.5) indicating an overload, then the best FIT entry is (0.15, 0.5), which reduces load to 0.95 and provides a total throughput score of 0.575. Node B should reduce r_1 by a factor of 0.7 by placing a drop at its top input. Note that once node B's FIT is properly propagated to node A, A will ensure, through its own load shedding mechanism, that node B never gets an input that would violate its FIT. Therefore, the second case we illustrated should not be observed.

It is important to note that FIT-based load shedding is very simple and efficient. Essentially, the table needs to be searched for the entry that matches the observed input rates. Since a FIT is ordered in descending order of the rates, the search can be performed efficiently. As a result, nodes can respond to overload very quickly.

4. Open Challenges

In this section, we briefly discuss some of the important open issues that we are planning to address as part of our future work.

Metadata Maintenance. FIT is constructed based on cost and selectivity statistics. As these statistics change, FIT has to be updated accordingly. If the FIT becomes stale, then parent nodes may lose effectiveness in shedding load in the best interest of their descendant nodes. As a result, we may end up with lower quality outputs than we can otherwise achieve. On the other hand, the size of a FIT can be large, and it can be inefficient to frequently update and propagate the whole table. Instead, an incremental maintenance approach is required.

Fairness and Priorities. Maximum total throughput is not a fair metric. It will penalize some of the costly query paths in exchange for much higher throughput that can be achieved from lower-cost ones. Some applications may care about fair quality degradation across queries. Additionally, priorities can be assigned on the basis of queries, sources, or tuples. An important point to note here is that as the quality model gets more complicated, it gets more difficult to capture and propagate the score from a downstream node to it parents. The hard part is figuring out the reverse mapping.

Handling Generalized Server Topologies. Metadata merging and propagation works well with tree-based server topologies where each server node has at most one parent node. In this case, metadata from children can be merged, revised, and forwarded to the parent in an incremental fashion. Therefore, it is enough for each node to communicate just with its immediate downstream and upstream neighbors. In more general topologies where a node can have multiple parent nodes, to achieve global coordination, nodes that are not immediate neighbors may also need to communicate. The reason is that two nodes may have a common descendant about which each one has only partial FIT information. In this case, the load shedding decisions made at one of these nodes would influence the decisions at the other one.

Addressing Bandwidth Bottlenecks. In bandwidth-limited environments, there are several additional issues to consider. First, metadata propagation also takes up bandwidth, and hence metadata exchange has to be kept to a minimum. Second, the feasible rate points in the FIT, which are defined in terms of the inputs of a node, do not directly map to the rates at the outputs of its parent, because rates may also slow down due to network delays. Lastly, we should also consider that under bandwidth limitations, shedding load at the earliest node in the server chain (even though that node itself may not be overloaded) is especially necessary. Bandwidth should not be wasted for tuples that will eventually be dropped down in the chain. Therefore, having parents shed load on behalf of their children is important. We are planning to tackle the CPU and the bandwidth problems under a common framework by treating the network slow-down as just another query operator with a certain time cost (selectivity = 1 since the network is reliable). Of course, this may not be so easy to achieve in shared networks due to frequent variations in cost.

Centralized vs. Distributed Tradeoffs. Lastly, we need to study the tradeoffs between the centralized and the distributed approaches. We speculated about how these approaches may behave in terms of meeting the design goals that we defined in Section 1. These hypotheses require experimental verification.

5. Related Work

The overload management problem in distributed stream processing systems has close relevance to the congestion control problem in computer networks [8]. Congestion in computer networks mainly arises when routers run out of buffer space, either because their processors can not keep up with the incoming input traffic, or because the outgoing link has a smaller bandwidth capacity than the incoming link [14]. Vari-

ous IP-layer architectures have been proposed to maintain Internet QoS including IntServ [6] and DiffServ [5]. More recently, Subramanian et al proposed OverQoS, an overlay-based QoS architecture, which does not require any support from the IP-layer [13]. OverQoS can prioritize packets that are more important to the application at the expense of the others.

Our upstream metadata propagation approach somewhat resembles the pushback mechanism developed for aggregate congestion control [9]. In this approach, a congested router can request its upstream routers to limit the rate of an aggregate (i.e. a certain collection of packets sharing a common property). The main purpose is to defend the network against DoS attacks. In our approach, nodes also specify their feasible input rates to their parents, but it is the parent that decides how to reduce rates on each stream.

The overload management problem has also been studied in the context of push-based data dissemination systems. For example, the Salamander data dissemination system provides a publish/subscribe substrate for push-based data delivery [10]. Salamander supports application-level QoS policies by allowing clients to plug in their data flow manipulation modules at any point in the data dissemination tree. These modules can prioritize, interleave, or discard certain data objects. This way the channel traffic can be adapted to the available client and network resources. Unlike our approach, Salamander does not try to coordinate the flow modification actions performed at different points of the data distribution tree. Another example is the data stream dissemination system studied by Shah et al [12]. This system selectively disseminates data updates between a network of data repositories to preserve user-defined coherency requirements.

There has been a great deal of recent work on data stream processing systems [7]. Several research prototypes have been built, each providing various techniques for load shedding. We give a few examples here. In our previous work, we have developed QoS-based load shedding algorithms within the context of the Aurora system, which achieved load reduction by inserting drop operators into running query plans [15]. We focused on where in the query plan to insert drops, how much load to shed at those points, and how to select the tuples to be dropped. The STREAM system has also developed several techniques, one of which is the statistical approximation approach for reducing load on aggregation queries [4]. The TelegraphCQ system's adaptive load shedding approach, called data triage, summarizes data instead of dropping it [11]. As we mentioned earlier in Section 1, these are all single-server approaches, and therefore they do not take the load dependencies between nodes into account.

6. Summary

In this paper, we have described the load shedding problem that arises in distributed stream processing systems. In order to address problems of scale, we presented a novel solution that allows nodes to make local decisions about how to shed load when the system is under stress. This approach requires nodes to periodically forward load information in the form of Feasible Input Tables (FITs) to their upstream neighbors. We have further described some of the challenges that remain in creating a solution that is widely deployable.

Acknowledgments. We would like to thank Uğur Çetintemel for his valuable input. This work has been supported in part by NSF under the grants IIS-0086057 and IIS-0325838, and by Army contract DAMD-17-02-2-0048.

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, Asilomar, CA, January 2005.

[2] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), August 2003.

[3] Y. Ahmad, B. Berg, U. Çetintemel, M. Humphrey, J. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik. Distributed Operation in the Borealis Stream Processing Engine (demo). In *ACM SIGMOD Conference*, Baltimore, MD, June 2005.

[4] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE Conference*, Boston, MA, March 2004.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. IETF RFC 2475, December 1998.

[6] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: An Overview. IETF RFC 1633, June 1994.

[7] L. Golab and M. T. Ozsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2), June 2003.

[8] V. Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4), August 1988.

[9] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM SIGCOMM Computer Communication Review*, 32(3), July 2002.

[10] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A Push-based Distribution Substrate for Internet Applications. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[11] F. Reiss and J. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *IEEE ICDE Conference*, Tokyo, Japan, April 2005.

[12] S. Shah, S. Dharmarajan, and K. Ramamritham. An Efficient and Resilient Approach to Filtering and Disseminating Streaming Data. In *VLDB Conference*, Berlin, Germany, September 2003.

[13] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[14] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[15] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.