# Towards Efficient Stream Processing in the Wide Area

Matvey Arye, Siddhartha Sen, Ariel Rabkin, Michael J. Freedman
Princeton University
{arye,sssix,asrabkin,mfreed}@cs.princeton.edu

Stream processing—the processing of large volumes of live data in real time—has been studied for decades [5]. However, recent years have seen an explosion in the number and variety of devices producing data streams, from software logs to handheld phones to aerial cameras, with an estimated 2.5 quintillion bytes of data created each day [1]. It is now commonplace for related data streams to arise from diverse resources in disparate locations, such as user posts in a social media service like Twitter, or server logs in a content distribution network (CDN) like Akamai. Traditional stream processing systems are unsuitable in this context, since they are designed either for single datacenter scenarios with a small number of colocated nodes (*e.g.*, [2,3]), or for sensor network scenarios with highly constrained, homogeneous nodes with minimal per-node resources (*e.g.*, [7]).

In contrast, we focus on stream processing in a highly distributed, heterogeneous environment. The stream processing system we are building, JetStream, is designed for Internet services that operate at a global scale, spanning multiple datacenters and involving nodes with diverse capacities and connectivities. In such scenarios, planning and executing a query over streaming data is challenging, because resources like link bandwidth vary dramatically between and across devices and datacenters. Whereas prior work on wide-area stream processing often focused on tolerating faults through replication (*e.g.*, [6]), we believe novel techniques for query optimization and placement are required for this new, wide-area setting.

In our vision, users will have to give only a high-level description of the query, perhaps as a dataflow graph of *operators*. JetStream will track available network resources and optimize the query plan to network conditions, assigning operators from this new plan to specific nodes for execution. This is similar to how a relational database compiles SQL queries into concrete execution plans.

Whereas a conventional database picks a query plan in advance, we expect query plans to adapt over time, depending on available computational and network resources. Further, we intend to explore hierarchical control. Some high-level decisions about placement may be made by a centralized entity (*e.g.*, which datacenter will be responsible for computing a particular aggregate) while other decisions should be delegated to increasingly localized controllers (*e.g.*, which servers within a datacenter are assigned which tasks). We expect hierarchical system of control to work better in the wide-area context, where a centralized controller might have a stale view of conditions.

A wide range of techniques have been proposed for efficient stream processing. Much of this work was in a single-node or single-datacenter environment, where interconnect bandwidth is plentiful. In the wide area, we expect networks to be the major bottleneck and source of unreliability, and therefore we think placement will be a dominating concern. Below, we outline three techniques which we expect to be especially valuable.

First, JetStream uses the semantic properties of operators to slide and split operators within the user's query in a manner than soundly preserves its results, in order to reduce bandwidth consumption, shift processing load, or load balance computation. A simple example is pushing a filter operator behind a monotonic operator: an operator ADD-$c$ that adds some constant $c$ to each input datum, followed by a filter LT-$t$ that only allows items less than some threshold $t$ to pass through, is equivalent to LT-$t$ followed by ADD-$c$ followed by LT-$t$. Such "pushback" transformations move computation towards the stream sources, reducing the bandwidth required. While such transformations have appeared in previous work, we have also developed novel, composable transformations that exploit the distributive property of aggregate operators to split queries over multiple nodes and datacenters, allowing for more even load distribution and fan-in across the wide area.

Second, we can customize tree aggregation for our domain. Aggregation trees have been explored for sensor networks (*e.g.*, [7]), but this work typically assumes that nodes have minimal storage, operate in a broadcast medium, and need to minimize the number of transmissions to preserve power. These assumptions may not hold in the heterogeneous environments we consider. Altering these assumptions allows us to explore new tree formation, scheduling, and fault recovery approaches.

Third, transformations that yield approximate query results may be necessary, particularly in cases for which semantic transformations and partial aggregation do not sufficiently reduce the resource costs of a query. Such transformations include sampling the data stream or using an approximate or "synopsis" data structure [4]. Ideally, the system should allow the user to specify the desired tradeoff between query error, resource consumption, and liveness, automatically adjusting the query plan accordingly.

An approximation query might rely on an approximation for old data as well as new data. Many streaming applications compare or "join" real-time data against historical data or quantities. For example, a user watching a plot of per-minute web traffic might wish to compare it with past records, perhaps on a different time scale. To support these operations efficiently, we are examining the in-network use of approximation and summarization data structures (such as aggregate-based, OLAP-style hypercubes). These would be first-class parts of the system, visible to the query planner.

Approximation not only reduces bandwidth consumption, it helps us handle faults, which are a fact of life in the wide

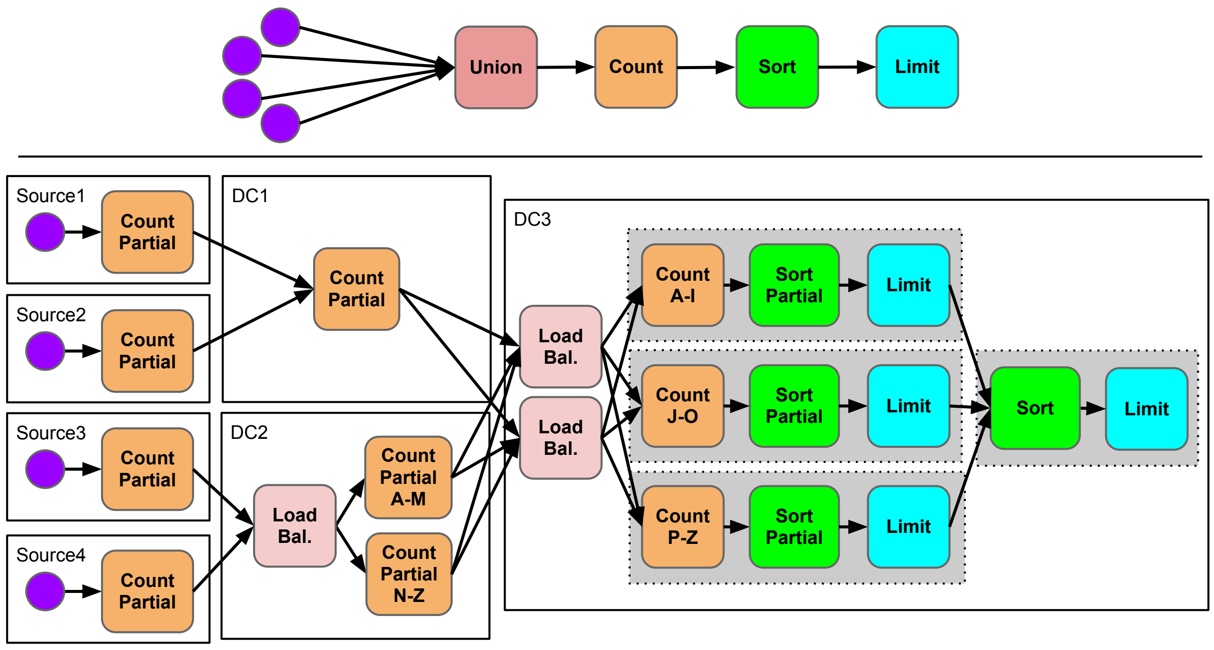

**Figure 1: A Top-k query example. The original query is shown above the optimized version of that query.**

area. Our fault-tolerance philosophy is to avoid replicating operators (and duplicating traffic) when possible, preferring instead to leverage in-network storage (*e.g.*, to temporarily backup data) or return approximate results.

We conclude with an example. Figure 1 shows some of these transformations in action for a query $Q$ that computes the $k$ most popular domains being requested from a CDN's geographically-distributed servers. (The query operates on *windows* of streaming data, but we mostly ignore this issue for simplicity.) The input representation of $Q$ is simple and flexible: after taking the union of the streams, the frequency of each domain is counted and the results are sorted and limited to the $k$ most popular.

JetStream can exploit the flexibility of the query's operators to optimize for the given physical topology. In this example, a source of streaming data may be an individual CDN server or a logging server deployed at each PoP in which the CDN operates. After performing local aggregation on the frequency of its domains over a given window, each source is directed to the nearest of two datacenters (DC1 or DC2) to performs partial aggregation across multiple sources. Once the streams enter DC1, they are again combined using a partial count operator that simply performs a sum. This merging further reduces the output bandwidth, which is desirable since the output needs to traverse a wide-area link.

In this example, DC2 has lower-capacity nodes than DC1, so it distributes the processing over two machines using count's group-by clause. Since the counts are grouped by domain, we simply separate the streams alphabetically. We distribute computation similarly in DC3, where distribution also serves to prevent large fan-in to any single node.

Finally, DC3 partially sorts and limits the domains' frequencies on three separate servers (shown in dotted boxes) before sending them to a fourth server to compute the final top $k$ results; this reduces the stream bandwidth entering this ultimate server. This "pushback" of the limit operator is possible because the operator that combines two sorted and disjoint lists is monotonic in rank: items never move up when the lists are merged.

We believe that the management and analysis of disparate streaming data sources are ripe for innovation. Much as database management systems did for business data and MapReduce is doing for stored unstructured data, we seek to allow users to analyze globally distributed data streams, without hand optimizing the computation on that data.

## 1. REFERENCES

[1] Bringing big data to the enterprise. `http://www-01.ibm.com/software/data/bigdata/`.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[3] D. J. 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*, 12(2):120–139, 2003.

[4] C. C. Aggarwal and P. S. Yu. A survey of synopsis construction in data streams. In *Data Streams – Models and Algorithms*, pages 169–207. 2007.

[5] L. Golab and M. T. Özsu, editors. *Data Stream Management*. Morgan & Claypool Publishers, 2010.

[6] J.-H. Hwang, U. Çetintemel, and S. B. Zdonik. Fast and reliable stream processing over wide area networks. In *ICDE Workshops*, pages 604–613, 2007.

[7] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny AGgregation service for ad-hoc sensor networks. In *OSDI*, pages 131–146, 2002.