Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area

Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman, *Princeton University*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/rabkin

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

Aggregation and Degradation in JetStream: Streaming analytics in the wide area

Ariel Rabkin, Matvey Arye, Siddhartha Sen*, Vivek S. Pai, and Michael J. Freedman
Princeton University

Abstract

We present JetStream, a system that allows real-time analysis of large, widely-distributed changing data sets. Traditional approaches to distributed analytics require users to specify in advance which data is to be backhauled to a central location for analysis. This is a poor match for domains where available bandwidth is scarce and it is infeasible to collect all potentially useful data.

JetStream addresses bandwidth limits in two ways, both of which are explicit in the programming model. The system incorporates structured storage in the form of OLAP data cubes, so data can be stored for analysis near where it is generated. Using cubes, queries can aggregate data in ways and locations of their choosing. The system also includes adaptive filtering and other transformations that adjusts data quality to match available bandwidth. Many bandwidth-saving transformations are possible; we discuss which are appropriate for which data and how they can best be combined.

We implemented a range of analytic queries on web request logs and image data. Queries could be expressed in a few lines of code. Using structured storage on source nodes conserved network bandwidth by allowing data to be collected only when needed to fulfill queries. Our adaptive control mechanisms are responsive enough to keep end-to-end latency within a few seconds, even when available bandwidth drops by a factor of two, and are flexible enough to express practical policies.

1 Introduction

This paper addresses the problem of analyzing data that is continuously created across wide-area networks. Queries on such data often have real-time requirements that need the latency between data generation and query response to be bounded. Existing stream processing systems, such as Borealis, System-S, Storm, or Spark Streaming [2, 6, 29, 32], address latency in the context of a single datacenter where data streams are processed inside a high-bandwidth network. These systems are not designed to perform well in the wide area, where limited bandwidth availability makes it impractical to backhaul all potentially useful data to a central location. Instead, a system for wide-area analytics must prioritize which data to transfer in the face of time-varying bandwidth constraints.

We have designed and built such a system, called JetStream, that extends today's dataflow streaming programming model in two ways. We incorporate structured storage that facilitates **aggregation**, combining related data together into succinct summaries. We also incorporate **degradation** that allows trading data size against fidelity. Just as MapReduce helps developers by handling fault tolerance and worker placement, JetStream aims to drastically reduce the burden of sensing and responding to bandwidth constraints.

Wide-area data analysis is a problem in a variety of contexts. Logs from content distribution networks and other computing infrastructure are created on nodes spread out across the globe. Smart electric grids, highways, and other infrastructure also generate large data volumes. Much of this data is generated "near the edge," with limited network connectivity over cellular or wireless links. Unlike traditional sensor network deployments, many of these infrastructure sensors are not energy limited, and can have substantial co-located computation and storage.

Wide-area analysis also applies to data that does not resemble traditional logs. Networks of video cameras are used for a wide variety of applications. These include not only urban surveillance but also highway traffic monitoring and wildlife observation. The cost of electronics, including sensors, storage, and processors, is currently falling faster than the cost of wireless bandwidth or of installing new wired connectivity. As a result, bandwidth is already becoming the limiting constraint in such systems [9] and we expect the gap between sensing capacity and bandwidth to increase in the coming years.

In the examples above, a large amount of data is stored at edge locations that have adequate compute and storage capacity, but there is limited or unpredictable bandwidth available to access the data. Today's analytics pipelines lack visibility into network conditions, and do not adapt dynamically if available bandwidth changes. As a result, the developer must specify in advance which data to store or collect based on pessimistic assumptions about available bandwidth. The consequence is that bandwidth is typically over-provisioned compared to average usage, and so capacity is not used efficiently.

*Current affiliation: Microsoft Research

JetStream's goal is to enable real-time analysis in this scenario, by reducing the volume of data being transferred. Storing and aggregating data where it is generated helps, but does not always reduce data volumes sufficiently. Thus, JetStream also includes degradations. These are data transformations analogous to "lossy" compression: they reduce data size at the expense of accuracy. Examples include computing per-minute aggregates for queries requesting per-second data, or dropping some fraction of the data via sampling. Since degradations impose a (tunable) accuracy penalty, JetStream is designed to monitor available bandwidth and use the minimal degree of degradation required to keep latency bounded.

Integrating aggregation and degradation into a streaming system required us to address three main challenges:

(1) Incorporating storage into the system while supporting real-time aggregation. Aggregation for queries with real-time requirements is particularly challenging in an environment where data sources have varying bandwidth capacities and may become disconnected. JetStream integrates structured storage as a first-class abstraction in its programming model, allowing aggregation to be specified in a flexible but unified way.

(2) Dynamically adjusting data volumes to the available bandwidth using degradation mechanisms. Such adaptation must be performed on a timescale of seconds to keep latency low.

(3) Allowing users to formulate policies for collecting data that maximize data value and that can be implemented effectively by the system. The policy framework must be expressive enough to meet the data quality needs of diverse queries. In particular, it should support combining multiple degradation mechanisms.

In meeting these challenges, we created the first wide-area analysis system that adjusts data quality to bound the latency of streaming queries in bandwidth-constrained environments. Our architecture decouples bandwidth sensing from the policy specifying how to aggregate and degrade the data to guarantee a timely response. The interfaces defined by our architecture support a wide-range of sensing methods and response techniques—for example, we implemented a diverse set of degradations, including those using complex data structures and multi-round protocols. We consider our architecture and its associated interfaces to be the key contribution of this paper.

By ignoring bandwidth limitations, previous systems force users to make an unappealing choice: they can be optimistic about available bandwidth and backhaul too much data, leading to buyer's remorse if bandwidth is costly; or they can be pessimistic about bandwidth and backhaul only limited data, leading to analyst's remorse if a desired query cannot be answered. By integrating durable storage into the dataflow and supporting dynamic adjustments to data quality, JetStream allows a user to focus on fundamentally different trade-offs: deciding which query results are needed in real-time and which inaccuracies are acceptable to maintain real-time performance.

2 Design Overview

JetStream is designed for near-real-time analysis of changing data, such as log data or audiovisual data. The system integrates data storage at the edge to allow users to collect data that may be useful for future analysis without necessarily transferring it to a central location. Users can define ad-hoc queries ("give me the video from camera #129 between 6am and 7am last night") as well as standing queries ("send a down-sampled copy of the video data from every camera back to a control center" or "tell me the top-10 domains by number of requests over 10 seconds"). Standing queries can be useful in and of themselves or they can be used to create centralized data structures to optimize the performance of common queries.

Standing queries are considerably more challenging than ad-hoc queries, and therefore are the focus of this paper. A standing query has a hard real-time requirement: if the query cannot handle the incoming data rate, then queues and queuing delays will grow, resulting in stale results. Giving users fresh results means that the system must keep latency bounded. This bound must be maintained even as the incoming data volume and the available bandwidth fluctuate.

Since JetStream aims to provide low-latency results on standing queries, it borrows the basic computation model of many of today's stream-processing systems. A worker process runs on each participating compute node. A query is implemented by a set of linked dataflow operators that operate on streams of tuples. Each operator is placed on a particular host and performs some transformations on the data. The system routes tuples between operators, whether on the same host or connected by the network.

2.1 Integrating structured storage

In a departure from previous stream processing systems, we integrate structured storage inside the operator graph. This storage lets us keep data where it is generated until it is needed. In our vision, nodes at the edge of the network can store hours or days worth of data that the user does not need to immediately analyze, but which may (or may not) be required later.

Because edge storage can involve large data volumes, we use structured storage to reduce query times. Past streaming systems incorporated storage in the form of a durable buffer of input tuples [2]. This would perform poorly for ad-hoc queries, since it would require re-scanning all stored data on every query. We instead adopt the data cube abstraction previously used in OLAP databases [15], which supports queries efficiently. We discuss our use of cubes in detail in Section 3.

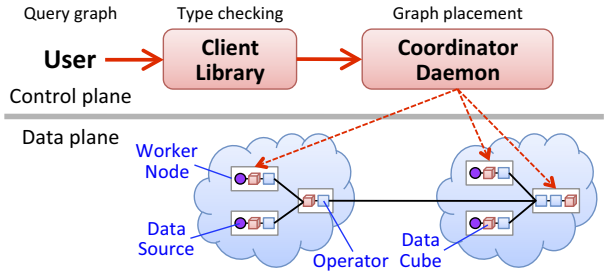

Figure 1: JetStream's high-level architecture. Users define query graphs with operators and cubes. A coordinator deploys the graph to worker nodes.

Integrating structured storage allows us to simplify other operators, enabling more graceful handling of missing or delayed data. Previous streaming systems defined a large set of stateful operators, including Sort, Aggregate, Join, and Resample [2]. When made stateful, each of these operators requires complex semantics to cope with missing inputs. In comparison, in our design, cubes are the only element with durable state and the only place where streams are merged, and so subsume the functionality of these operators. They are responsible not only for storage, but also for aggregation inside queries.

We opt for stream equi-joins over general joins, since general joins across streams would impose a global synchronization barrier, causing excessive latency when links are congested or disabled. We have not found the lack of a general join to be a serious limitation. Cubes can handle stream equi-joins from an arbitrary number of streams and this has been enough for us in practice.

2.2 Reducing data volumes

Operators can apply lossy transformations that reduce data volumes. For example, they can randomly sample tuples from a stream, or drop all tuples with values below a threshold. To maximize data quality, it is desirable to apply data reduction only when necessary to conserve bandwidth. JetStream therefore includes specialized *tunable degradation* operators. The system automatically adjusts these operators to match available bandwidth, thus minimizing the impact on data quality. These mechanisms are described in Section 4.

Because the system dynamically changes degradation levels, it needs a way to indicate the fidelity of the data it is sending downstream. Therefore, we added the ability for operators to send metadata messages both upstream and downstream. Metadata messages are ordered with respect to data messages and act as punctuations [30]. This mechanism is broadly useful in the system: we use it to signal congestion and also to implement multi-round protocols within the operator graph (see Section 5).

Class	Operators
I/O	FileReader, Echo, FileWriter, UnixCmd
Parser	CSVParser, RegexParser
Filter	Grep, LessThan, Equals
Map	ExtendTupleWithConst, Project, AddTimestamp, URL2Domain, Histogram2Quantile
Degradation	VariableSubscriber, DegradeHistogram, Sampling

Table 1: Examples of the operators currently provided by JetStream's client library.

2.3 Programming model

Operator graphs are constructed using a client library. The library's programming interface makes it convenient to configure operators and data cubes and to link them together into dataflow graphs. The system includes a library of pre-defined operators, and users can also define their own operators using a library of base classes. These base classes streamline development for common operator functionality such as map and filter, which both require implementing a single virtual function. We list some examples in Table 1 to give the flavor of the tasks operators perform. We discuss the details of the system architecture and implementation in Section 6.

Figure 2 gives a simple example of our programming model. Suppose there is a set of N nodes, each with a directory full of images. Over time, a camera adds data to this directory. The system will scan the directory and copy new images across the network to a destination, tagging each with the time and hostname at which it was created. In this example, placement is explicit: the semantics of the application requires that the readers are on the source nodes and that the result is produced at the union node. If there had been intermediate processing, the programmer could have let the system handle the placement of this computation. Later in the paper, we will extend this example to cope with insufficient bandwidth; we offer it here to give a sense of the programming model.

JetStream is primarily an execution engine, more like the Dryad or MapReduce engines than like the DryadLINQ or Pig languages [10, 20, 24, 31]. If in the future a widely-accepted declarative programming language for stream processing emerges, we expect that JetStream should be able to support it.

3 Aggregation

Integrating structured storage into a streaming operator graph solves several problems. Cubes, unlike key-value or general relational models, have unambiguous semantics for inserting and aggregating data. This lets JetStream handle several different forms of aggregation in a unified way. Windowed aggregation combines data residing on the same node across time. For example, all web request

```
g = QueryGraph()
dest = Operators.StoreImages(g, IMAGE_OUT_DIR)
dest.instantiate_on(union_node)

for node in source_nodes:
  reader = Operators.FileReader(g, options.dirname)
  reader.instantiate_on(node)
  add_time = Operators.Timestamp()
  add_host = Operators.Extend("$HOSTNAME")
  g.chain([reader, add_time, add_host, dest])
```

Figure 2: Example code for a query. Each source node has an image reader connected by a chain of operators to a common destination operator.

records for the same second can be merged together to produce a per-minute request count. Tree aggregation combines data residing on different nodes for the same period. This can be applied recursively: data can be grouped within a datacenter or point of presence, and then data from multiple locations is combined at a central ("union") point. Both these forms of aggregation are handled in JetStream by inserting data into structured storage and then extracting it by appropriate queries.

While JetStream borrows the cube interface from OLAP data warehouses, we substitute a different implementation. Cubes in data warehouses typically involve heavy precomputation. Many roll-ups or indexes are constructed, incurring high ingest times to support fast data drill-downs. In contrast, JetStream uses cubes for edge storage and aggregation and only maintains a primary-key index. This reduces ingest overhead and allows cubes to be effectively used inside a streaming computation.

Integrating storage with distributed streaming requires new interfaces and abstractions. Cubes can have multiple sources inserting data and multiple unrelated queries reading results. In a single-node or datacenter system it is clear when all the data has arrived since synchronization is cheap. In a wide-area system, however, synchronization is costly, particularly if nodes are temporarily unavailable. In this setting, queries need a flexible way to decide when a cube has received enough data to send updates downstream. Our context also requires cubes to deal with late tuples that arrive after results have been emitted. Before explaining how we solve these problems, we describe our storage abstraction in more detail.

3.1 Data Cubes and Their API

A data cube is a multi-dimensional array that can encapsulate numerical properties and relationships between fields in structured input data, similar to a database relation. It is defined by a set of dimensions, which specify the coordinates (the key) of an array cell, and a set of aggregates, which specify the statistics (values) stored in a cell.

Suppose for example that we are collecting statistics about traffic to a website. We might define a cube with dimensions for URLs and time periods. The cube would

```
g = QueryGraph()
dest = Cube(g, "stored_images")
dest.add_dimension(TIME, "timestamp")
dest.add_dimension(HOSTNAME, "timestamp")
dest.add_aggregate(BLOB, "img_data")
dest.instantiate_on(union_node)

for node in source_nodes:
  reader = Operators.FileReader(g, options.dirname)
  reader.instantiate_on(node)
  add_time = Operators.Timestamp()
  add_host = Operators.Extend("$HOSTNAME")
  g.chain([reader, add_time, add_host, dest])
```

Figure 3: Running example but with the destination now a cube.

then map each unique pair of URL and time period to a cell with a set of aggregates, such as the total number of requests and the maximum request latency. Each web request, when added to the cube, updates the cell corresponding to its URL and time period.

A query can ***slice*** a cube to yield a subset of its values, such as "all URLs starting with foo.com ordered by total requests." A query can ***roll up*** a slice, aggregating together the values along some dimension of the cube, such as asking for the total request count, summed across all URLs in the slice. Roll-ups use the same aggregation function as insertion. Whereas insertion potentially aggregates data at write time, a roll-up performs aggregation at query time. Aggregate functions must be deterministic and order-independent (i.e., `max` and `average`, but not `last-k tuples`); this means that the system need not worry about the ordering between inserts.

Aggregates can be more complex than simple integers. A cube cell can include a histogram or sketch, describing a whole statistical distribution. (This relies on the fact that the underlying sketches or histograms can be combined straightforwardly, without loss of accuracy.) One might, for instance, build a cube not merely of request counts over time, but directly representing the distribution of latencies over time. This allows a query to do powerful statistical processing, such as finding quantiles over arbitrary subsets of events. Histograms and sketches are fixed-size, regardless of the underlying data size; they are an especially compact form of aggregation.

The cube abstraction is powerful enough to directly express all the aggregation we need in JetStream. Windowed operations (such as moving averages) are repeated roll-ups over a time-varying slice of the cube. Both sliding windows and tumbling windows fit into this model.

Even when data is not aggregated together, a data cube is a useful storage abstraction that allows queries on multiple attributes. For example, Figure 3 shows how a cube can be integrated into our running example. Each image frame is stored as an aggregate, with the timestamp and source hostname as dimensions. This allows queries based on any combination of time and source. (Modern

video encoding does not store each frame separately. A more complex example might store short segments of video data in cube cells. Alternatively, a more complex implementation of the cube API might offer programmers the abstraction of a sequence of frames, while using a differential coding scheme underneath.)

Many implementations of the cube abstraction are possible; ours uses MySQL as the underlying data store. This allows us to leverage MySQL's optimizations for handling large data volumes as well as its support for transactions. Transactions greatly simplify failure recovery, as we explain in Section 6.

3.2 Integrating Cubes with Streaming

While the semantics of inserting data into the cube is straightforward ("apply the aggregation function"), extracting data from cubes is not. A query needs to make a policy decision of when and how often to read data from a cube. This decision affects the latency, bandwidth consumption, and completeness of results. In JetStream, these policies are encapsulated in specialized operators called subscribers.

Aggregation trades latency for bandwidth. The longer a query waits before reading the aggregate, the more data potentially can be aggregated into a fixed-size summary, reducing data volumes at the price of latency. There is also a trade-off between latency and completeness. It may not be worth waiting for stragglers before emitting a result. (As discussed below, the system can sometimes correct an inaccurate result later.) In the local area, stragglers can be masked by speculative execution or retries [10]. In the wide area, this strategy is unable to compensate for late data caused by limited bandwidth or connectivity.

We allow users to tune these trade-offs by giving subscriber operators fine-grained control over when to emit the results of aggregation. Subscribers have a richer API than other operators. They are notified whenever a tuple is inserted into the cube. They can also query the cube for slices and rollups. This allows fairly complex policies. A subscriber might repeatedly query for the last 10 seconds of data, relative to the system clock. Or it might track the highest timestamp from each source feeding into the cube, and only query up to the point which all the sources have reached. The default policy in JetStream combines these two policies: if all sources have contributed data, the result is emitted immediately. Otherwise, the subscriber has a fixed timeout before emitting results. Like all operators, the parameters of a subscriber can be adjusted by the user.

If an update arrives at a cube that would modify a result that has already been emitted by a subscriber, that update is termed *backfill*. Because cubes are the location where data is joined, the general problem of late updates only appears in JetStream as backfill at cubes.

We handle backfill using similar techniques to prior stream-processing work [2]. A backfill update results in a subscriber emitting a *delta record* that contains the old and new values. Delta records propagate in the same manner as new data. They can cause tuples to be revoked, e.g., if an operator filters an update that was previously allowed. The effect of a delta update on an aggregate depends on the aggregation function. Some aggregates, such as `average`, can retract an item that was previously added. For other items retraction can be an expensive operation. For example, `max` requires keeping a full list of its inputs to enable updates that reduce the value of the item with the largest value. Like Naiad [23], we only allow such functions if backfill input is impossible or the source data is available locally.

Subscribers are free to query the cube multiple times and are part of the metadata flow. They can therefore take part in nontrivial iterative protocols before emitting data. We exploited the flexibility of this interface when implementing a specialized subscriber that carries out a multi-round filtering protocol for finding the global top-k elements (by a user-determined ranking) over distributed data, without transferring all data. This is discussed further in Section 5.

3.3 Aggregation is Sometimes Insufficient

Not all queries aggregate well. For example, our running example of streams of image data is a case where aggregation is of limited value. There is no straightforward way to combine images taken at different times or from different cameras pointing at different scenes.

For data that can be aggregated in principle, the underlying data distribution may make aggregation ineffective at saving bandwidth. Data where the distribution of aggregate groups has a long tail will not aggregate well. This can depend on the coarseness of aggregation. For example, aggregating web requests by URL is ineffective because the popularity of URLs is long-tailed [16]. Aggregating the same data by domain can be much more effective, since domain popularity is less long-tailed. Figure 4 illustrates the point using data from the Coral content distribution network [13].

4 Adaptive Degradation

Even with partial aggregation at sources, some queries will require more bandwidth than is available. If there is insufficient bandwidth, the query will fall ever farther behind, as new data arrives faster than it can be processed. To keep latency low, JetStream allows queries to specify a graceful degradation plan that trades a little data quality for reduced bandwidth. For example, audiovisual data can be degraded by downsampling or reducing the frame rate. Similarly, quantitative data can be degraded by increasing the coarseness of the aggregation or using sampling. We

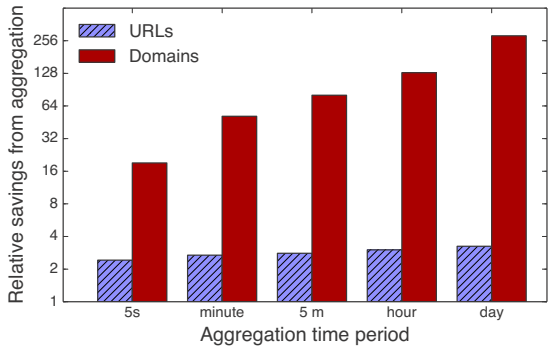

Figure 4: For CoralCDN logs, domains aggregate effectively over time and URLs do not.

discuss techniques applicable to quantitative data in more detail in Section 5.

Since degradations impose an accuracy penalty, they should only be used to the extent necessary. JetStream achieves this by using explicit feedback control [21]. Since wide-area networks can have substantial buffering beyond our visibility or control, waiting for backpressure to fill up queues incurs large delays and provides incomplete congestion information. By using explicit feedback, JetStream can detect congestion before queues fill up, enabling it to respond in a timely manner. Moreover, this feedback reflects the degree of congestion, allowing JetStream to tune the degradation to the right level.

JetStream's congestion response is decentralized: nodes react independently to their inferred bandwidth limits. This avoids expensive synchronization over the wide area, but it also means that sources may send data at different degradation levels, based on their bandwidth conditions at the time. Queries that aggregate data over sources or time must therefore handle varying degradation levels. We discuss how this is done for quantitative data in §5.2.

JetStream achieves adaptive congestion control via three components: (i) degradation operators that apply data transformations to the data stream; (ii) congestion monitors that measure the available bandwidth; and (iii) policies that specify how the system should adjust the level of degradation to the available bandwidth. Figure 5 illustrates the interaction between these three components; we discuss each component in the following sections.

4.1 Degrading data with operators

Degradation operators can be either standard operators that operate on a tuple-by-tuple basis, or cube subscribers that produce tuples by querying cubes. A degradation operator is associated with a set of degradation levels, which defines its behavior on cubes or data streams. For example, our variable subscriber offers the ability to roll-up data across different time intervals (e.g., sending output every 1, 5 or 10 seconds) and characterizes this degradation in terms of the estimated bandwidth use relative to the operator's maximum fidelity (in this example, $[1.0, 0.2, 0.1]$). This interface gives flexibility to operators. Some operators have fine-grained response levels, while others are widely spaced—for example, an audiovisual codec might only support a fixed set of widely spaced bitrates.

As we discuss in §5.1, many useful degradations have a data-dependent bandwidth savings. The interface we adopted gives operators flexibility in how they estimate the bandwidth savings. Levels (and their step size) can be (i) dynamically changed by the operator, e.g., based on profiling, (ii) statically defined by the implementation, or (iii) configurable at runtime (as with our currently implemented operators).

4.2 Monitoring available bandwidth

JetStream uses congestion monitors to estimate the relative available capacity of the system, or the ratio between the maximum possible data rate and the current rate. A ratio greater than one indicates spare capacity, while less than one indicates congestion. A congestion monitor is attached to each queue in the system and allows the associated policy to determine whether the data rate can be increased, or must be decreased, and by how much. Congestion monitoring can be done in many ways; we use two techniques in our prototype.

For detecting network congestion, we track the time required to process data. Sources insert periodic metadata markers in their output, specifying that the data since the last marker was generated over k seconds. When a network receiver processes this marker (which occurs after all prior data tuples are processed), it sends an upstream acknowledgment. The congestion monitor records the time t between seeing the last marker and receiving this acknowledgment, and uses $\frac{k}{t}$ as an estimate of the available capacity. (A similar approach is used to adapt bitrates in HTTP streaming [1].) The advantage of this approach is that it gives a meaningful estimate of how much spare capacity there is when the system is not yet congested.

For detecting bottlenecks in our storage implementation, we have a congestion monitor that uses differences in queue lengths over time to extrapolate ingestion rate. We cannot use the data window measurement to monitor data cubes because cubes can batch writes. With batching, performance is not linear with respect to the data volume in each window. Queue monitoring can detect congestion quickly, and the rate of queue growth indicates how congested the system is. However, it does not indicate how much spare capacity there is if the system is not overloaded and the queue is empty.

Congestion monitors report their capacity ratio upstream, both to other congestion monitors on the same host and across the network using a metadata message.

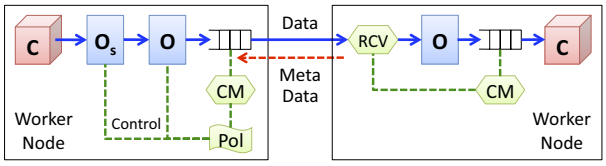

Figure 5: JetStream's mechanisms for detecting and adapting to congestion. Along with cubes and operators, JetStream employs explicit application queues, congestion monitors (CM), policy managers (Pol), and network receivers (RCV) to control when adaptations should be performed.

```
g = QueryGraph()
dest = Cube(g, "stored_images")
dest.add_dimension(TIME, "timestamp")
dest.add_dimension(HOSTNAME, "timestamp")
dest.add_aggregate(BLOB, "img_data")
dest.instantiate_on(union_node)

for node in source_nodes:
  reader = Operators.FileReader(g, options.dirname)
  reader.instantiate_on(node)
  add_time = Operators.Timestamp()
  add_host = Operators.Extend("$HOSTNAME")
  drop_frames = Operators.LowerFramerate()
  downsample = Operators.Downsample(min=0.5)
  g.chain([reader, add_time, add_host, drop_frames,
      downsample, dest])
  g.congest_policy([downsample, drop_frames])
```

Figure 6: Running example with a degradation policy added: downsampling will be applied first, then frame-rate lowering.

When a monitor is queried, it returns the minimum of its own and the downstream ratio. In this way, congestion signals propagate to sources in a multi-step pipeline.

4.3 Congestion response policies

Congestion response policies tune degradation operators based on the available bandwidth reported by congestion monitors. To effectively control data volumes while minimizing errors, policies need to be able to tune multiple degradation operators.

Oftentimes, a given degradation technique is only useful up to a certain level of degradation. Rolling up request logs from a 10-second level to a 30-second level may impose an acceptable delay for a data analysis pipeline, but waiting for longer periods of time may not. If the bandwidth required at the largest-acceptable roll-up coarseness is still too large, it is necessary to use some other technique, such as dropping statistics about unpopular items.

Consider the following policy: "By default, send all images at maximum fidelity from CCTV cameras to a central repository. If bandwidth is insufficient, switch to sending images at 75% fidelity, then 50% if there still isn't enough bandwidth. Beyond that point, reduce the frame rate, but keep the images at 50% fidelity." This policy involves two degradation operators: one to decrease image fidelity (F), and one to drop frames (D). As the system encounters congestion, F should respond first, followed by D once F is fully degraded. However, if capacity becomes available, D should respond first, followed by F. We say that F has higher priority than D.

Figure 6 shows our running example, modified with this policy. The policy is represented in two parts. Each degradation operator is configured with its maximum degradation level. Because this is part of the operator configuration, it can use operator-specific notions, such as frame rates. A separate policy statement specifies the priority of the operators. This policy statement is agnostic to the semantics of the operators in question.

At runtime, each policy is encapsulated in a policy object. Each policy object is attached to a particular congestion monitor and the set of local degradation operators that it manages. The operators periodically query the policy with their available and current degradation levels (per §4.1). The policy returns the degradation level the operator should use, based on the current congestion ratio and the state of the other operators.

Importantly, an operator's priority is unrelated to the operator's position in the dataflow graph. In this example, the dataflow structure does per-frame degradation after frame dropping, to avoid wasted computation on frames that will later be dropped. This is irrespective of which operator has higher priority in the policy.

Our framework can be extended to cope with still-more-complex policies. One might, for instance, desire policies that apply two different degradations simultaneously, such as interleaving time-roll-up steps with some other degradation operation.

5 Degrading Quantitative Data

Above, we discussed the abstractions that JetStream provides for data degradation. We now discuss how these abstractions can be used to degrade quantitative analytics data. While the techniques we describe are mostly well-known, we evaluate them in the new context of wide-area streaming analytics.

A wide range of degradations are possible for quantitative data. Here are some that JetStream supports:

- **Dimension coarsening:** A subscriber that performs roll-ups of data cube dimensions. To reduce output size, the subscriber emits progressively coarser data. For example, rolling up per-second data to output per-minute data or rolling up URLs to the domain level.
- **Local value threshold:** A filter that only forwards elements whose value is above a (tunable) threshold on a particular node. For example, only passing Apache request log entries where the latency of the request exceeded 1 second.

- **Global value threshold:** A filter that only forwards elements whose total value, across all nodes, is above a threshold. This is implemented using a multi-round distributed protocol [7]. For example, one can create a filter that only keeps the top 1000 URLs requested across a CDN in a 10-second window.
- **Consistent sampling:** Drops a fraction of inputs based on the hash of some dimension. Two filters with the same selectivity will pass the same elements.
- **Synopsis approximation:** Replaces a histogram, sketch, or other statistical synopsis with a less accurate but smaller synopsis.

5.1 Which degradations to use?

The best degradation for a given application depends not only on the statistics of the data, but also on the set of queries that may be applied to the data. As a result, the system cannot choose the best degradation without knowing this intended use. We leave for future work the challenge of automatically synthesizing a degradation strategy based on the data and a set of potential queries or other downstream uses of the data. Here, we offer some guidelines for data analysts on which degradations to use for different circumstances.

- For data with synopses that can be degraded, such as histograms or sketches, degrading the synopsis will have predictable bandwidth savings and (for some synopses) predictable increases in error terms.
- If the dimension values for the data have a natural hierarchy, then aggregation can effectively reduce the data volume for many distributions. This is particularly applicable to time-series data. As we discussed in §3.3, coarsening is ineffective on long-tailed distributions, however.
- A long-tailed distribution has a small number of highly ranked "head" items, followed by a long tail of low-ranked items. Many queries, such as "top k items" only concern the head. In these cases, a filter can remove the large-but-irrelevant tail. (Note that this depends on the user's subsequent plans for the data, and not just the statistics of the data itself.) Either a local or global value threshold is applicable here.
- A global value threshold gives exact answers for every item whose total value is above a threshold; a local threshold will have worse error bounds, but can do well in practice in many cases. (We demonstrate this empirically in Sec. 7.3.)
- Consistent samples help to analyze relationships between multiple aggregates. For example, to analyze the correlation between hit count and maximum response latency in a CDN, a query can sample from the space of URLs. This yields exact values for the aggregates of all the URLs in the sample.

One would like degradation operators to have a predictable bandwidth savings, so that JetStream can pick the appropriate degradation level. One would also like degradations to have predictable effects on data quality, so that users can reason about the errors introduced. Most degradation operators have only one or the other of these properties, however. Because no one degradation is optimal, we took pains to allow compound policies. As a result, users can specify an initial degradation with good error bounds, and a fall-back degradation to constrain bandwidth regardless.

5.2 Merging heterogeneous data

As we discussed above, degradation levels will vary over time, and will vary across different nodes feeding into a cube at the same time. It is therefore desirable to be able to combine data of different fidelities without paying an additional penalty. We call this property *mergeability*. Formally, we define mergeability as the ability to combine data of different fidelities so that the merged result has the same error bounds as the input with the lowest fidelity.[1]

Mergeability constrains the ways in which data can be approximated and analyzed. Suppose that two sources are sending data every five seconds to a central point. One source then switches to sending data every six seconds. To represent the merged answer accurately, the data must be further coarsened, to every 30 seconds. This implies that the system sent 5.5x more data than it needed to if both sources had simply sent every 30 seconds.

Mergeability guided several design choices in JetStream. Dimension coarsening is only mergeable if the coarsening steps are consistent and strictly hierarchical. We therefore define an explicit hierarchy for time dimensions. Cubes can store (and subscribers can roll up) time only in fixed intervals. (The first layers of the hierarchy are 1, 5, 10, 30, and 60 seconds.) We also require slices to start at a timestamp that is a multiple of the query's step size. Taken together, these requirements make time coarsening mergeable.

Mergeability constrains other degradations besides dimension coarsening. Histograms are an effective approximation for computing quantiles. Making our histogram implementation mergeable required careful programming. In our implementation, every division between buckets in an n-bucket histogram is also a division in an n+1 bucket histogram, and consequently histograms are mergeable regardless of the number of buckets.

Some degradations are intrinsically not mergeable. As an example, consider web request data generated at two servers. Lets analyze the top-k request counts by URL

[1] This definition is stronger than that typically used in the theory streaming literature, which covers merging data of the same fidelity [3].

at *both* servers, so that we sum the counts for any URLs common to both. For such a query, the top-k lists from each server cannot be merged together.[2] Similarly, for per-minute top-k lists, there is in principle no way to compute a daily top-k. For the same reasons, sets with a value cutoff ("elements with value above x") cannot always be merged to give a correct set with a new value cutoff. This is tolerable in practice for applications that do not need to perform roll-ups, or where the data distribution is such that the error bounds are small. A global filtering protocol, such as the multi-round protocol supported by JetStream, can give correct results when data is spread across many sources.

6 Architecture and Implementation

JetStream's architecture has three main components: worker daemons access and process data on a distributed set of nodes, a coordinator daemon distributes computation across available workers, and a client library defines the computation to be performed and provides an interface to the running system.

The life-cycle of a query: A query starts when a client program creates a dataflow graph and submits it for execution. The library checks the dataflow graph for type and structural errors, such as integer operators being applied to string fields or filter operators with no inputs. The graph is then sent to the coordinator, which chooses the assignment of operators to worker nodes. The placement algorithm attempts to minimize the data traversing wide-area networks, placing operators on the send-side of the link whenever possible.

After determining operator placement, the coordinator sends the relevant subset of the graph to each node in the system. The nodes then create any necessary network connections amongst themselves, and start the operators. The query will keep running until it is stopped via the coordinator, or until all the sources send a `stop` marker indicating that there will be no more data. As discussed above, degradation is handled in a decentralized fashion, and the coordinator's involvement is not required to maintain a running query.

Implementation: The JetStream coordinator is implemented in about 2000 lines of Python, and is purely memory resident. The worker is implemented in a single process, similar to a database management system. The worker is about 15,000 lines of C++, including 3000 for system-defined operators. Operators are implemented as C++ classes, and can be dynamically loaded. Within a node, each chain of operators from a source operator or network link is executed sequentially, without queuing, by a single thread at a time. All tuples are processed sequentially in the order received.

Failure Recovery: In a streaming system, failure recovery requires two things: Each failed piece of the computation (e.g., each query operator on a failed node) must be restarted and reattached to the graph. Additionally, for stateful pieces of the computation, their state must be restored to what it would have been absent the failure. JetStream currently only does the former. The latter can be implemented within our model but is a lower priority for us than for datacenter streaming systems.

The coordinator has a complete view of which operators should be on each node, and therefore can restart them when a node fails and recovers. This is implemented in our existing prototype. Sources periodically try to reconnect to their destinations and therefore will automatically recover when the failed node restarts. The coordinator's state can be made durable using group-consensus tools like Zookeeper [18]. (This is not currently implemented.)

Unlike past streaming systems, we do not attempt to make JetStream failure-oblivious. Datacenter-based streaming systems rely on the presence of an underlying reliable data store (such as a reliable message queue, HDFS or BigTable) that is assumed to be always available [5, 29, 32]. Using this data store, these systems can hide the existence of failures from computations, restarting work immediately and carefully avoiding duplicated or dropped data.

In wide-area analytics, failures cannot be hidden, since data will be inaccessible if the network is partitioned. For many analytic uses, users prefer queries that promptly supply approximate results based on the available data, and revise these results as late data arrives [4]. (We discussed how to incorporate these backfill updates in §3.2.) Thus, the results of a computation will necessarily be affected by failures. Retroactively changing the results to recover completely from the failure is a low priority for us, since enough time might have passed that the temporary results have already been acted upon.

That said, it is possible to do precise failure recovery in the JetStream model, as we sketch below. In the datacenter context, a number of failure recovery techniques have been described. All these schemes have two basic ingredients: (i) The system must keep metadata to track which tuples have been processed by each operator. This metadata must be recorded atomically with the update to ensure process-once semantics. (ii) There must be a mechanism to retransmit the tuple until there is an acknowledgment that it was fully processed by the chain.

Our system meets both these underlying requirements. Since our underlying data store is a database, we already have sufficient transaction support to atomically commit updates along with the appropriate metadata. We could add sequence numbers to tuples and use the fact that

[2] Fagin *et al* [11] prove that it is not generally possible to find the top k_1 elements and their exact values using only the top k_2 from each of the subsets, for any fixed k_2 (see their example 4.4).

	Name	Output	Operators	Cubes	LoC	BW ratio
1	Big requests	Requests above 95th percentile of size, with percentiles computed for past minute.	3n + 8	3n + 3	97	22
2	Slow requests	All requests with throughput less than 10 kbytes/sec.	4n + 2	1	5	24
3	Requests-by-URL	Counts for each URL-response code pair.	3n + 2	n + 1	5	351
4	Success by domain	The fraction of success responses for each domain.	6n + 4	n + 2	30	445
5	Quantiles	95th percentiles of response time and size, for each HTTP response code.	4n + 5	n +1	25	715
6	Top-k domains	Top-10 domains every five seconds.	n + 3	n + 1	40	2300
7	Bad referrers	The 10 domains most responsible for referrals that led to a 404 response, every five seconds.	8n + 2	n + 1	16	18600
8	Bandwidth by node	Overall bandwidth used by each node, over time.	4n + 2	n + 1	15	49800

Table 2: Complexity and efficiency of example queries. For operator and cube counts, *n* is the number of data source nodes in the system. LoC is the number of non-comment lines of code needed to write the query, not counting shared library code. Bandwidth ratio is the ratio between source input size and the data volume transferred over the network (e.g., 22 means a factor-of-22 savings from aggregation and filtering).

tuples are not reordered between data cubes to make sure that upstream cubes retransmit data to downstream cubes until appropriate acknowledgments are received.

7 Evaluation

In this section, we evaluate four main questions about JetStream's design.

- §7.1 Does JetStream make it easy to write distributed analytic queries?
- §7.1 How effectively do hierarchical aggregation and static filtering reduce bandwidth consumption?
- §7.2 What latency can JetStream maintain, in the presence of changing bandwidth limits?
- §7.3 How well does JetStream's adaptive degradation work with complex policies?

Given JetStream's focus on adapting bandwidth consumption, we omit extensive performance benchmarking and comparisons to existing streaming systems. The throughput of JetStream is largely limited by the underlying database and serialization code, neither of which is relevant to our technical contributions.

7.1 Expressivity and Efficiency

To evaluate the usability of JetStream's programming framework, we use it to ask a number of analytical questions about a dataset of CoralCDN logs. Table 2 summarizes the eight queries we evaluate, drawn from our experience observing and managing CoralCDN. The queries include summary counts, histograms, filtered raw logs, top-k queries, and outlier detection.

Our aim is to understand (i) if these questions can be expressed succinctly in JetStream's programming model, and (ii) if many of our queries, even before applying data degradation techniques, experience significant bandwidth savings by storing data near where it is generated.

To test the queries, we select logs from 50 nodes in CoralCDN for January 20, 2013, and transfer each log to a node on the VICCI testbed [26]. To emulate wide-area clusters of CDN nodes, we select 25 VICCI servers from each of the MPI-SWS and Georgia Tech sites (in Saarbruecken, Germany, and Atlanta, GA respectively). These nodes serve as the sources of data; the queries produce their output at a node in Princeton, NJ. The total size of these logs is 51 GB, or 140 million HTTP requests. Since the logs are drawn from actual operational nodes, they are not equal in size. The largest is approximately 2 GB, while the smallest is 0.4 GB. This sort of size skew is a real factor in operational deployments, not an experimental artifact.

We observe large savings compared to backhauling the raw logs, ranging from a factor of 22x to more than four orders of magnitude. The large gains are primarily due to the partial aggregation present in all these examples. Thousands of requests can be tallied together in source cubes to produce a single tuple to be copied across the wide area for merging at the union cube. This demonstrates that edge storage and partial aggregation are valuable design choices in wide-area analytics.

Code size is small but varies across queries. We wrote about 150 lines of shared code for processing command line arguments, parsing and storing CoralCDN logs, and printing results. These are shared lines of code and are therefore not included in the unique LoC measures above. For simple aggregation tree queries, only a few lines are needed, specifying what is to be aggregated. Queries with more complex topologies require more code. In our experience, the code size and complexity is comparable to that of MapReduce programs. As with MapReduce, higher-

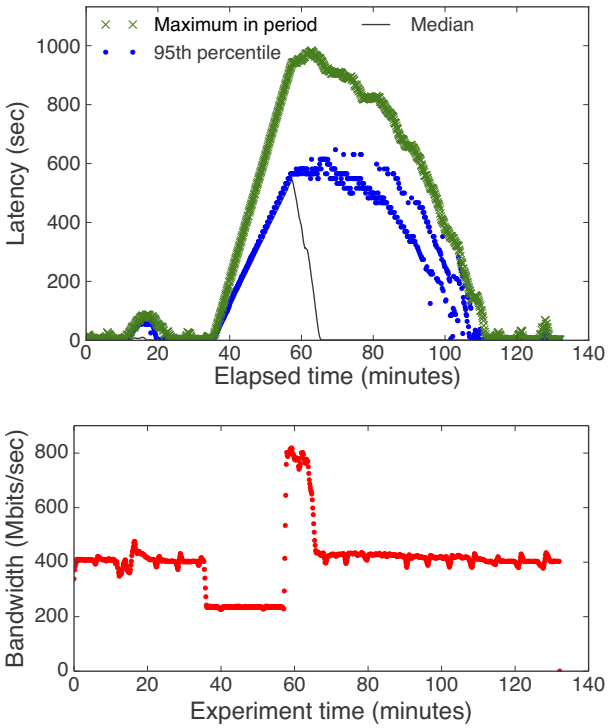

Figure 7: (Top) Latencies, with percentiles computed over each 8-second window. (Bottom) Data rate at receiver, showing extent and duration of traffic shaping. Without degradation, latency grows without bound, and takes a long time to recover from disruption.

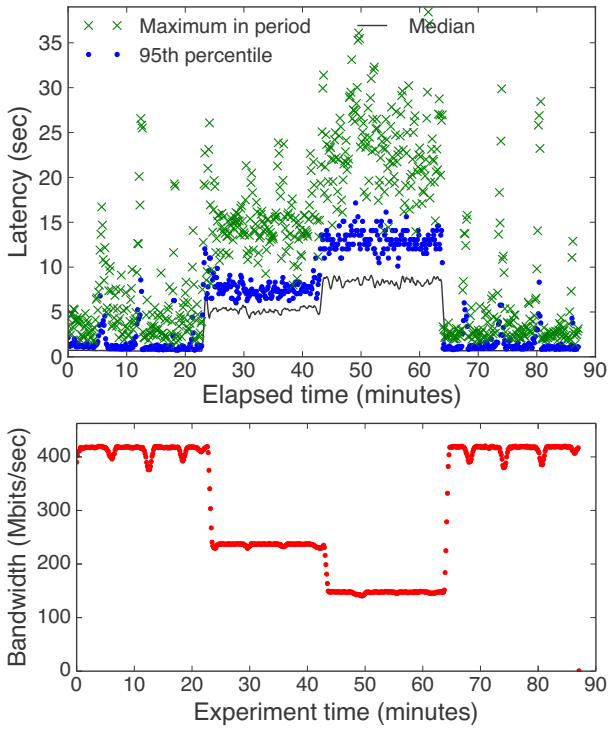

Figure 8: The same as Figure 7, but with degradation enabled. The system rapidly adjusts to available bandwidth, keeping latency low and bounded.

level frameworks could further reduce the code size and complexity. Most programming mistakes were caught by the client-side type checker, reducing the difficulty and time cost of development.

Only two user-defined functions (UDFs) are required for these sample queries; one to convert URLs to domains and a second to compute the success ratio in Query #4. The parsing is performed with a generic configurable operator for parsing field-separated strings. This suggests that the cube API plus our existing operators are sufficient to express a wide range of tasks.

We have preliminary evidence that JetStream's programming model can be learned by non-experts and does not require knowledge of the system internals. Query #3 was written by an undergraduate who did not participate in JetStream's development, and received only limited assistance from the core development team.

7.2 System Throughput and Latency

We benchmark the system's overall throughput and latency characteristics using a relatively simple processing pipeline under several different network configurations. This experiment used 80 source nodes running on the VICCI infrastructure, divided between MPI-SWS (Germany), Georgia Tech and the University of Washington (United States). The source nodes send image data to a single union node at Princeton. All images are the same size (approximately 26 kilobytes). Nodes are configured to send a maximum of 25 images per second, a rate that the network can support without degradation. This is a 400 mb/sec data rate, so we are using nearly half our institution's gigabit WAN link during the experiment. The configured degradation policy is to reduce the frame rate if bandwidth is insufficient.

Figure 7 shows the behavior of the system with degradation disabled. Generally, latency is low: However, around minute 15, a slight drop in available bandwidth resulted in some nodes experiencing uncontrolled queue growth, leading to significant latency. (This is visible as a small bump in the latency plot.) At minute 35, we impose a 400 kbit/sec bandwidth cap on each source node using the Linux kernel's traffic shaping options. The latency of all the nodes starts rising sharply and continuously. Around minute 60, we disable bandwidth shaping and latency starts to drop. Notice that the 95th percentile and maximum latency recovers much more slowly than median latency. Some nodes are able to drain their queues quickly, while other nodes are starving for bandwidth. As a result, it takes roughly 45 minutes for the system to resume its previous behavior.

Figure 8 shows that our degradation mechanisms prevent these unwanted behaviors. We repeated a similar

experiment, but with degradation enabled. Here, after we apply bandwidth shaping, the degradation mechanisms activate, and successfully keep queue size and latency bounded at a few seconds. Approximately 40 minutes into the experiment, we apply more bandwidth shaping, to 250 kb/sec per node, and again latency stays bounded. Notice the absence of a latency spike at each of the bandwidth transitions; the system reacts promptly enough that such spikes are not even visible on the graph. At the 60 minute mark we disable traffic shaping, and the system again reacts promptly, returning to its original state.

7.3 Complex Degradation Policies

In our final experiment, we demonstrate that JetStream's degradation mechanisms, operating on a realistic workload, maintain responsiveness in situations in which full-fidelity data would exceed available resources. For this experiment, use the Requests-by-Url query from Table 2, executing under the same setup as in §7.1. We compress a full day's logs (and thus the diurnal variation) into five minutes of wall-clock time. We impose a bandwidth cap of 80 kb/second per source node, which is artificially low, but serves to emphasize system behavior. The limit is low enough to force the configured degradation policies to activate as data rates shift.

We compare the effect of four degradation policies. Each policy starts by sending data every second, if possible, and performing roll-ups to five-second windows when bandwidth is scarce. One policy (*max window 5*) does no further degradation. The *max window 10* policy will further degrade to 10-second windows. The remaining two policies employ either consistent-sampling (based on a hash of the URL) or a local value threshold (dropping tuples with the lowest counts).

Figure 9 shows the bandwidth and degradation from each of the four policies and the bandwidth used by the null policy (no degradation). As the load increases, most of the source nodes hit the bandwidth cap and switch to 5-second windows. As the load keeps rising, the more heavily-used nodes again reach their cap. Both the thresholding and sampling policies can keep bandwidth usage under the cap.

We noted earlier that many CoralCDN URLs are unique, and therefore do not aggregate well. This is visible in the results here. Time coarsening by a factor of 5 and 10 reduces bandwidth, but by factors much less than 5 or 10. Much bandwidth is used reporting statistics about low-traffic URLs—the tail of the distribution. Local value thresholding effectively truncates this tail. As can be seen on the graph, this has a minuscule effect on the relative error, while reducing bandwidth by a factor of two. (In this context, relative error is the maximum error in the request count for any one URL as a fraction of the total number of requests in that time period).

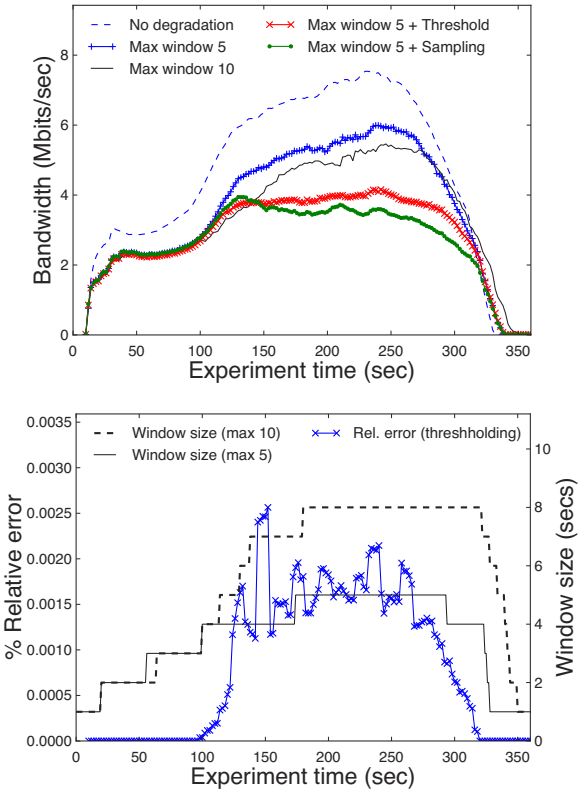

Figure 9: (Top) Output bandwidth for five different degradation choices. (Bottom) Window size and thresholding error over time. For this query, window sizes larger than 5 seconds have limited benefit, while thresholding has minimal accuracy penalty. Degradation policies should depend on the data.

This experiment shows that JetStream is able to effectively combine different degradation techniques. The limits of time coarsening on this workload illustrates why compound policies are useful. The fact that different degradations activate at the same point in the experiment shows that the control loop works effectively with a variety of degradation operators.

8 Related Work

Our work has a large debt to the stream processing community. The original work on streaming [2, 6, 8] addressed the question of how to process incoming updates with minimal latency. In contrast, JetStream targets dispersed and changing (stored) data sets, in the presence of dynamic bandwidth constraints.

Spark-Streaming, MillWheel, and Storm [5, 29, 32] are systems for large-scale stream processing within datacenters. All these systems rely on an underlying fault-tolerant storage system, respectively HDFS, BigTable, or a reliable message queue. Such systems or their implementation techniques, such as Spark's memory-backed resilient storage, could help scale JetStream within datacenters,

but are orthogonal to our concerns of efficiency and low latency across the wide area.

Some research on streaming systems has considered the wide area [19, 27]. One focus of this work was on using redundant paths for performance and fault tolerance. In contrast, we use degradation to cope with insufficient bandwidth. Hourglass [27] places query operators to minimize network usage in an ad-hoc topology, but since it does not include storage, users must choose which data to collect (and thus the set of supported queries) beforehand.

Previous wide-area streaming work assumed that computation resources were scattered in an ad-hoc manner, e.g., on PlanetLab nodes. As a result, sophisticated algorithms were needed for placement. This assumption is overly pessimistic in our context. Due to the rise of centralized datacenters, we expect there to be only two or three options for operator or cube placement: the site where the data is generated, the nearest point-of-presence, or else a centralized datacenter.

Some single-node stream-processing systems, such as TelegraphCQ [8], included relational storage, and others have advocated tighter integration between stream processing and relational databases [12]. The StreamCube system evaluated which layers of a cube hierarchy to materialize in the context of stream processing [17]. These uses of storage do not pose the latency-completeness tradeoffs we address with our subscriber interface, nor do they facilitate bandwidth reduction in distributed contexts.

Tree aggregation has been studied in the sensor network community as a method of reducing bandwidth and power consumption, notably in the Tiny Aggregation Service [22]. Much subsequent sensor network research used mesh topologies to compensate for unreliable connections and faulty nodes. In contrast, our hardware is not power-constrained and we assume that conventional IP networking will deliver suitable routes. Protocols such as RCRT [25], the Rate-Controlled Reliable Transport Protocol for sensor networks, estimate available bandwidth explicitly and convey rate allocation decisions to data sources. They could serve as an alternative implementation of the congestion monitor in JetStream. However, these works do not address how the application reacts to the congestion signals, which in JetStream is specified by the degradation operators, the policy manager, and the interface between them.

Tree aggregation and local storage are also used in the Ganglia [14] monitoring system. Ganglia supports a limited set of queries and is oblivious to bandwidth conditions.

Our work seeks to reduce data volumes while minimizing the reduction in accuracy. Similarly, BlinkDB [4] deploys sampling-based approximations on top of MapReduce and Hive to reduce latency. In BlinkDB, the data is carefully pre-sampled with specific statistical goals; small probing jobs are used to estimate query run-time. In contrast, streaming wide-area analytics systems such as ours have to measure and adapt to available bandwidth, without the benefit of a prior data-import step. We also support a range of degradation techniques, not just sampling.

Our previous workshop publication [28] argued for decentralized wide-area analysis of the form conducted by JetStream. It gave a high-level description of our ideas and discussed use cases at length, but lacked this paper's detailed design, implementation, or evaluation.

9 Conclusions

This paper has presented a system, JetStream, for wide-area data analysis in environments where bandwidth is scarce. Our storage abstraction allows data to be stored where it was generated and efficiently queried when needed. It simplifies aggregation of data both across time and across sources. Degradation techniques supplement aggregation when available bandwidth is insufficient for error-free results.

No single degradation technique is always best; a combination of techniques can perform better than any individual technique. Thus, our system supports combining multiple techniques in a modular and reusable way using policies. Our separation between congestion monitors, degradation operators, and policies creates a powerful, extensible framework for streaming wide-area analysis.

Acknowledgments

The authors appreciate the helpful advice and comments of Jinyang Li, Jennifer Rexford, Erik Nordström, Rob Kiefer, our shepherd Ramesh Govindan, and the anonymous reviewers. This work was funded under NSF awards IIS-1250990 (BIGDATA) and CNS-1217782, as well as the DARPA CSSG program.

References

[1] 3GPP Technical Specification 26.234. Transparent end-to-end packet switched streaming service (PSS); Protocols and codecs, 2013.

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

[3] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *PODS*, 2012.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.

[5] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[6] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *DM-SSP*, 2006.

[7] P. Cao. Efficient top-k query calculation in distributed networks. In *PODC*, 2004.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, 2003.

[9] Y. M. Chen, L. Dong, and J.-S. Oh. Real-time video relay for uav traffic surveillance systems through available communication networks. In *IEEE WCNC*, 2007.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[12] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.

[13] M. J. Freedman. Experiences with CoralCDN: A five-year operational view. In *NSDI*, 2010.

[14] Ganglia monitoring system. `http://ganglia.sourceforge.net/`, 2013.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[16] L. Guo, E. Tan, S. Chen, Z. Xiao, and X. Zhang. The stretched exponential distribution of internet media access patterns. In *PODC*, 2008.

[17] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2), 2005.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[19] J.-H. Hwang, U. Cetintemel, and S. B. Zdonik. Fast and reliable stream processing over wide area networks. In *Data Eng. Workshop*, 2007.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

[21] D. Katabi, M. Handley, and C. E. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, 2002.

[22] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[23] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[25] J. Paek and R. Govindan. RCRT : Rate-Controlled Reliable Transport Protocol for Wireless Sensor Networks. *ACM Trans. Sensor Networks (TOSN)*, 7(3), 2010.

[26] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton Univ., Dept. Comp. Sci., 2011.

[27] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[28] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. J. Freedman. Making every bit count in wide-area analytics. In *HotOS*, May 2013.

[29] Storm. `https://github.com/nathanmarz/storm/`, 2012.

[30] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowledge and Data Eng.*, 15 (3):555–568, 2003.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[32] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.