
# ELF: Efficient Lightweight Fast Stream Processing at Scale

**Liting Hu, Karsten Schwan, Hrishikesh Amur, and Xin Chen,** ***Georgia Institute of Technology***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/hu

**This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.**

**June 19–20, 2014 • Philadelphia, PA**

978-1-931971-10-2

# ELF: Efficient Lightweight Fast Stream Processing at Scale

Liting Hu, Karsten Schwan, Hrishikesh Amur, Xin Chen
Georgia Institute of Technology
{foxting,amur,xchen384}@gatech.edu, schwan@cc.gatech.edu

## Abstract

Stream processing has become a key means for gaining rapid insights from webserver-captured data. Challenges include how to scale to numerous, concurrently running streaming jobs, to coordinate across those jobs to share insights, to make online changes to job functions to adapt to new requirements or data characteristics, and for each job, to efficiently operate over different time windows.

The ELF stream processing system addresses these new challenges. Implemented over a set of agents enriching the web tier of datacenter systems, ELF obtains scalability by using a decentralized "*many masters*" architecture where for each job, live data is extracted directly from webservers, and placed into memory-efficient *compressed buffer trees* (CBTs) for local parsing and temporary storage, followed by subsequent aggregation using *shared reducer trees* (SRTs) mapped to sets of worker processes. Job masters at the roots of SRTs can dynamically customize worker actions, obtain aggregated results for end user delivery and/or coordinate with other jobs.

An ELF prototype implemented and evaluated for a larger scale configuration demonstrates scalability, high per-node throughput, sub-second job latency, and sub-second ability to adjust the actions of jobs being run.

## 1 Introduction

Stream processing of live data is widely used for applications that include generating business-critical decisions from marketing streams, identifying spam campaigns for social networks, performing datacenter intrusion detection, etc. Such diversity engenders differences in how streaming jobs must be run, requiring synchronous batch processing, asynchronous stream processing, or combining both. Further, jobs may need to dynamically adjust their behavior to new data content and/or new user needs, and coordinate with other concurrently running jobs to share insights. Figure 1 exemplifies these requirements, where job inputs are user activity logs, e.g., *clicks*, *likes*, and *buys*, continuously generated from say, the *Video Games* directory in an e-commerce company.

In this figure, the `micro-promotion` application extracts user *clicks* per product for the past 300 *s*, and lists

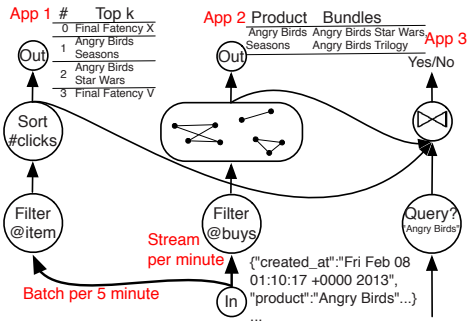



**Figure 1:** Examples of diverse concurrent applications.

the top-*k* products that have the most clicks. It can then dispatch coupons to those "popular" products so as to increase sales. A suitable model for this job is synchronous batch processing, as all log data for the past 300 *s* has to be on hand before grouping clicks per product and calculating the top-*k* products being viewed.

For the same set of inputs, a concurrent job performs `product-bundling`, by extracting user *likes* and *buys* from logs, and then creating 'edges' and 'vertices' linking those video games that are typically bought together. One purpose is to provide online recommendations for other users. For this job, since user activity logs are generated in realtime, we will want to update these connected components whenever possible, by fitting them into a graph and iteratively updating the graph over some sliding time window. For this usage, an asynchronous stream processing model is preferred to provide low latency updates.

The third `sale-prediction` job states a product name, e.g., *Angry Birds*, which is joined with the product-bundling application to find out what products are similar to *Angry Birds* (indicated by the 'typically bought together' set). The result is then joined with the micro-promotion application to determine whether *Angry Birds* and its peers are currently "popular". This final result can be used to predict the likely market success of launching a new product like *Angry Birds*, and obtaining it requires interacting with the first and second application.

Finally, all of these applications will run for some considerable amount of time, possibly for days. This makes it natural for the application creator to wish to update job functions or parameters during ongoing runs, e.g., to

change the batching intervals to adjust to high vs. low traffic periods, to flush sliding windows to 'reset' job results after some abnormal period (e.g., a flash mob), etc.

Distributed streaming systems are challenged by the requirements articulated above. First, concerning *flexibility*, existing systems typically employ some fixed execution model, e.g., Spark Streaming [28] and others [11, 12, 17, 22] treat streaming computations as a series of batch computations, whereas Storm [4] and others [7, 21, 24] structure streaming computations as a dataflow graph where vertices asynchronously process incoming records. Further, these systems are not designed to be naturally composable, so as to simultaneously provide both of their execution models, and they do not offer functionality to coordinate their execution. As a result, applications desiring the combined use of their execution models must use multiple platforms governed via external controls, at the expense of simplicity.

A second challenge is *scaling with job diversity*. Many existing systems inherit MapReduce's "*single master/many workers*" infrastructure, where the centralized master is responsible for all scheduling activities. How to scale to hundreds of parallel jobs, particularly for jobs with diverse execution logics (e.g., pipeline or cyclic dataflow), different batch sizes, differently sized sliding windows, etc? A single master governing different per-job scheduling methods and carrying out cross-job coordination will be complex, creating a potential bottleneck.

A third challenge is *obtaining high performance for incremental updates*. This is difficult for most current streaming systems, as they use an in-memory *hashtable*-like data structure to store and aggressively integrate past states with new state, incurring substantial memory consumption and limited throughput when operating across large-sized history windows.

The ELF (**E**fficient, **L**ightweight, **F**lexible) stream processing system presented in this paper implements novel functionality to meet the challenges listed above within a single framework: to efficiently run hundreds of concurrent and potentially interacting applications, with diverse per-application execution models, at levels of performance equal to those of less flexible systems.

As shown in Figure 2, each ELF node resides in each webserver. Logically, they are structured as a million-node overlay built with the Pastry DHT [25], where each ELF application has its own respective set of master and worker processes mapped to ELF nodes, self-constructed as a *shared reducer tree* (SRT) for data aggregation. The system operates as follows: (1) each line of logs received from webserver is parsed into a key-value pair and continuously inserted into ELF's local in-memory *compressed buffer tree* (CBT [9]) for pre-reducing; (2) the distributed key-value pairs from CBTs "roll up" along the SRT, which progressively reduces them until they reach the root to

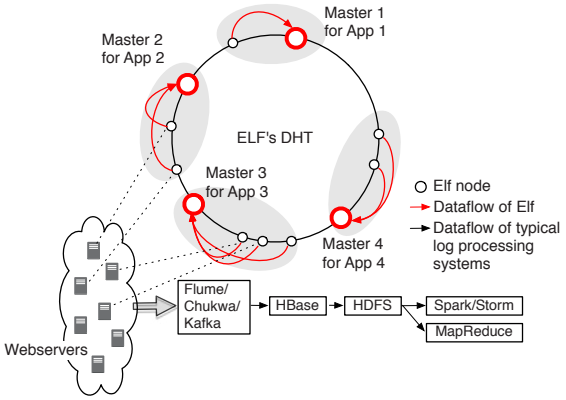



**Figure 2:** Dataflow of ELF vs. a typical realtime web log analysis system, composed of Flume, HBase, HDFS, Hadoop MapReduce and Spark/Storm.

output the final result. ELF's operation, therefore, entirely bypasses the storage-centric data path, to rapidly process live data. Intuitively, with a DHT, the masters of different applications will be mapped to different nodes, thus offering scalability by avoiding the potential bottleneck created by many masters running on the same node.

ELF is evaluated experimentally over 1000 logical webservers running on 50 server-class machines, using both batched and continuously streaming workloads. For batched workload, ELF can process millions of records per second, outperforming general batch processing systems. For a realistic social networking application, ELF can respond to queries with latencies of tens of millisecond, equaling the performance of state-of-the-art, asynchronous streaming systems. New functionality offered by ELF is its ability to dynamically change job functions at sub-second latency, while running hundreds of jobs subject to runtime coordination.

This paper makes the following technical contributions:

1. A decentralized 'many masters' architecture assigning each application its own master capable of individually controlling its workers. To the best of our knowledge, ELF is the first to use a decentralized architecture for scalable stream processing (Sec. 2).
2. A memory-efficient approach for incremental updates, using a *B*-tree-like in-memory data structure, to store and manage large stored states (Sec. 2.2).
3. Abstractions permitting cyclic dataflows via feedback loops, with additional uses of these abstractions including the ability to rapidly and dynamically change job behavior (Sec. 2.3).
4. Support for cross-job coordination, enabling interactive processing that can utilize and combine multiple jobs' intermediate results (Sec. 2.3).
5. An open-source implementation of ELF and a comprehensive evaluation of its performance and functionality on a large cluster using real-world webserver logs (Sec. 3, Sec. 4).

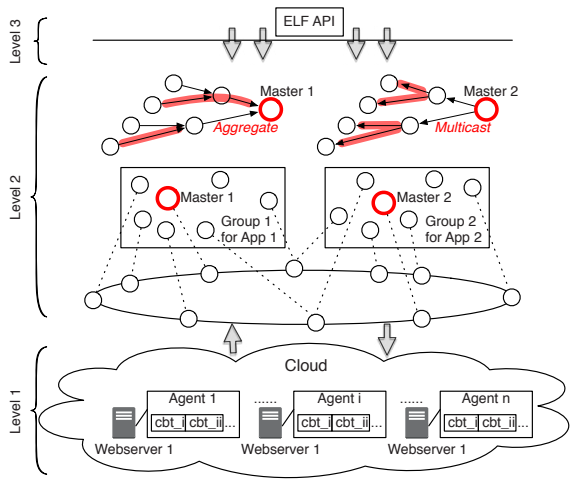



**Figure 3:** High-level overview of the ELF system.

# 2 Design

This section describes ELF's basic workflow, introduces ELF's components, shows how applications are implemented with its APIs, and explains the performance, scalability and flexibility benefits of using ELF.

## 2.1 Overview

As shown in Figure 3, the ELF streaming system runs across a set of agents structured as an overlay built using the Pastry DHT. There are three basic components. First, on each webserver producing data required by the application, there is an agent (see Figure 3 bottom) locally parsing live data logs into application-specific *key-value* pairs. For example, for the micro-promotion application, batches of logs are parsed as a map from product name (key) to the number of clicks (value), and groupby-aggregated like $\langle (a,1),(b,1),(a,1),(b,1),(b,1)\rangle \rightarrow \langle (a,2),(b,3)\rangle$, labelled with an integer-valued timestamp for each batch.

The second component is the middle-level, application-specific group (Figure 3 middle), composed of a master and a set of agents as workers that jointly implement (1) the *data plane*: a scalable aggregation tree that progressively 'rolls up' and reduces those local key-value pairs from distributed agents within the group, e.g., $\langle (a,2),(b,3)..\rangle, \langle (a,5)..\rangle, \langle (b,2),(c,2)..\rangle$ from tree leaves are reduced as $\langle (a,7),(b,5),(c,2)..\rangle$ to the root; (2) the *control plane*: a scalable multicast tree used by the master to control the application's execution, e.g., when necessary, the master can multicast to its workers within the group, to notify them to empty their sliding windows and/or synchronously start a new batch. Further, different applications' masters can exchange queries and results using the DHT's routing substrate, so that given any application's name as a key, queries or results can be efficiently routed to that application's master (within $O(logN)$ hops), without the need for coordination via some global entity. The resulting model supports the concurrent execution of diverse applications and flexible coordination between those applications.

The third component is the high-level ELF programming API (Figure 3 top) exposed to programmers for implementing a variety of diverse, complex jobs, e.g., streaming analysis, batch analysis, and interactive queries. We next describe these components in more detail.

## 2.2 CBT-based Agent

Existing streaming systems like Spark [28], Storm [4] typically consume data from distributed storage like HDFS or HBase, incurring cross-machine data movement. This means that data might be somewhat stale when it arrives at the streaming system. Further, for most realworld jobs, their '*map*' tasks could be '*pre-reduced*' locally on webservers with the most parallelism, and only the intermediate results need to be transmitted over the network for data shuffling, thus decreasing the process latency and most of unnecessary bandwidth overhead.

ELF adopts an 'in-situ' approach to data access in which incoming data is injected into the streaming system directly from its sources. ELF agents residing in each webserver consume live web logs to produce succinct *key-value* pairs, where a typical log event is a 4-*tuple* of $\langle timestamp, src\_ip, prority, body\rangle$: the *timestamp* is used to divide input event streams into batches of different epochs, and the *body* is the log entry body, formatted as a map from a string attribute name (key) to an arbitrary array of bytes (value).

Each agent exposes a simple HiveQL-like [26] query interface with which an application can define how to filter and parse live web logs. Figure 4 shows how the micro-promotion application uses ELF QL to define the top-$k$ function, which calculates the top-10 popular products that have the most clicks at each epoch (30 $s$), in the *Video Game* directory of the e-commerce site.

Each ELF agent is designed to be capable of holding a considerable number of '*past*' key-value pairs, by storing such data in compressed form, using a space-efficient, *in-memory* data structure, termed a *compressed buffer tree* (CBT) [9]. Its *in-memory* design uses an $(a,b)$-tree with each internal node augmented by a memory buffer. Inserts and deletes are not immediately performed, but buffered in successive levels in the tree allowing better

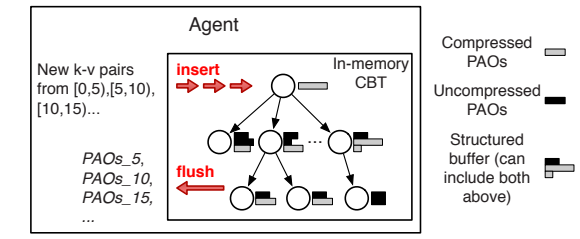
**Example log event**

```
{"created_at":"23:48:22 +0000 2013",
"id":299665941824950273,
"product":"Angry Birds Season",
"clicks_count":2,
"buys_count":0,
"user":{"id":343284040,
"name":"@Curry",
"location":"Ohio", ...} ...}
```

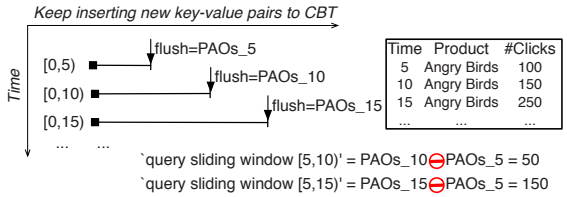
**ELF QL ->**

```
SELECT product,SUM(clicks_count)
FROM *
WHERE store == `video_games'
GROUP BY product
SORT BY SUM(clicks_count) DESC
LIMIT 10
WINDOWING 30 SECONDS;
```

**Figure 4:** Example of ELF QL query.




(a) Compressed buffer tree (CBT)




(b) Arbitrary sliding windows using CBT

**Figure 5:** Between intervals, new *k-v* pairs are inserted into the CBT; the root buffer is sorted and aggregated; the buffer is the split into fragments according to hash ranges of children, and each fragment is compressed and copied into the respective children node; at each interval, the CBT is flushed.

I/O performance.

As shown in Figure 5(a), first, each newly parsed key-value pair is represented as a *partial aggregation object* (PAO). Second, the PAO is serialized and the tuple $\langle hash, size, serializedPAO \rangle$ is appended to the root node's buffer, where *hash* is a hash of the key, and *size* is the size of the *serialized PAO*. Unlike a binary search tree in which inserting a value requires traversing the tree and then performing the insert to the right place, the CBT simply defers the insert. Then, when a buffer reaches some threshold (e.g., half the capacity), it is flushed into the buffers of nodes in the next level. To flush the buffer, the system sorts the tuples by hash value, decompresses the buffers of the nodes in the next level, and partitions the tuples into those receiving buffers based on the hash value. Such an insertion behaves like a *B*-tree: a full leaf is split into a new leaf and the new leaf is added to the parent of the leaf. More detail about the CBT and its properties appears in [9].

Key for ELF is that the CBT makes possible the rapid incremental updates over considerable time windows, i.e., extensive sets of historical records. Toward that end, the following APIs are exposed for controlling the CBT: (i) "*insert*" to fill, (ii) "*flush*" to fetch the tree's entire groupby-aggregate results, and (iii) "*empty*" to empty the tree's buffers, which is necessary when the application wants to start a new batch. By using a series of "*insert*", "*flush*", "*empty*" operations, ELF can implement many of standard operations in streaming systems, such as sliding windows, incremental processing, and synchronous batching.

For example, as shown in Figure 3(b), let the interval be 5 *s*, a sale-prediction application tracks the up-to-date *#clicks* for the product *Angry Birds*, by inserting new key-value pairs, and periodically flushing the CBT. The application obtains the local agent's results in intervals [0,5), [0,10), [0,15), etc. as $PAO_5$, $PAO_{10}$, $PAO_{15}$, etc. If the application needs a windowing value in [5,15), rather than repeatedly adding the counts in [5,10) with multiple operations, it can simply perform one single operation $PAO_{15} \ominus PAO_5$, where $\ominus$ is an "invertible reduce". In another example using synchronous batching, an application can start a new batch by erasing past records, e.g., tracking the promotion effect when a new advertisement is launched. In this case, all agents' CBTs coordinate to perform a simultaneous "empty" operation via a multicast protocol from the middle-level's DHT, as described in more detail in Sec.2.3.

**Why CBTs?** Our concern is performance. Consider using an *in-memory* binary search tree to maintain key-value pairs as the application's states, without buffering and compression. In this case, inserting an element into the tree requires traversing the tree and performing the insert — a read and a write operation per update, leading to poor performance. It is not necessary, however, to aggregate each new element in such an aggressive fashion: *integration can occur lazily*. Consider, for instance, an application that determines the top-10 most popular items, updated every 30 *s*, by monitoring streams of data from some e-commerce site. The incoming rate can be as high as millions per second, but CBTs need only be flushed every 30 *s* to obtain the up-to-date top-10 items. The key to efficiency lies in that "*flush*" is performed in relatively large chunks while amortizing the cost across a series of small "inserting new data" operations: decompression of the buffer is deferred until we have batched enough inserts in the buffer, thus enhancing the throughput.

## 2.3 DHT-based SRT

ELF's agents are analogous to stateful vertices in dataflow systems, constructed into a directed graph in which data passes along directed edges. Using the terms vertex and agent interchangeably, this subsection describes how we leverage DHTs to construct these dataflow graphs, thus obtaining unique benefits in flexibility and scalability. To restate our goal, we seek a design that meets the following criteria:

1. capability to host hundreds of concurrently running applications' dataflow graphs;
2. with each dataflow graph including minion vertices, as our vertices reside in distributed webservers; and
3. where each dataflow graph can flexibly interact with others for runtime coordination of its execution.

ELF leverages DHTs to create a novel '*many master*' decentralized infrastructure. As shown in Figure 6, all agents are structured into a P2P overlay with DHT-based

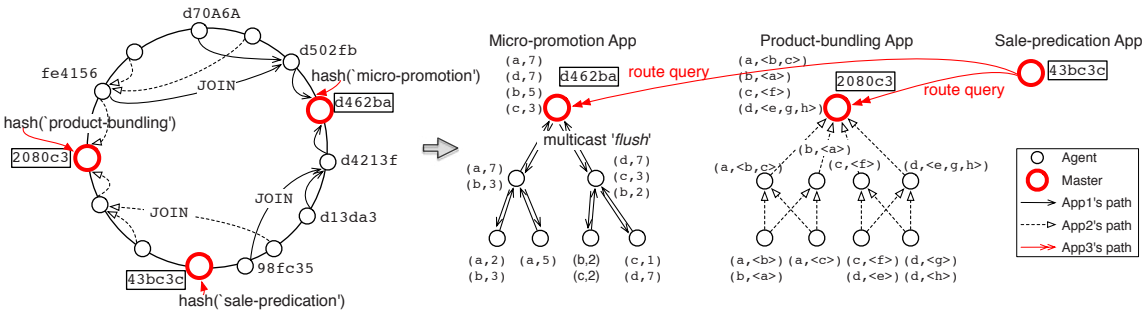



**Figure 6:** Shared Reducer Tree Construction for many jobs.

routings. Each agent has a unique, 128-*bit* nodeId in a circular nodeId space ranging from 0 to $2^{128}$-1. The set of nodeIds is uniformly distributed; this is achieved by basing the nodeId on a secure hash (SHA-1) of the node's IP address. Given a message and a key, it is guaranteed that the message is reliably routed to the node with the nodeId numerically closest to that key, within $\lceil log_{2^b} N \rceil$ hops, where $b$ is a base with typical value 4. SRTs for many applications (jobs) are constructed as follows.

The first step is to construct application-based groups of agents and ensure that these groups are well balanced over the network. For each job's group, this is done as depicted in Figure 6 left: the agent parsing the application's stream will route a JOIN message using appId as the key. The appId is the hash of the application's textual name concatenated with its creator's name. The hash is computed using the same collision resistant SHA-1 hash function, ensuring a uniform distribution of appIds. Since all agents belonging to the same application use the same key, their JOIN message will eventually arrive at a rendezvous node, with nodeId numerically close to appId. The rendezvous node is set as the job's master. The unions of all messages' paths are registered to construct the group, in which the internal node, as the forwarder, maintains a children table for the group containing an entry (IP address and appId) for each child. Note that the uniformly distributed appId ensures the even distribution of groups across all agents.

The second step is to "draw" a directed graph within each group to guide the dataflow computation. Like other streaming systems, an application specifies its dataflow graph as a logical graph of stages linked by connectors. Each connector could simply transfer data to the next stage, e.g., *filter* function, or shuffle the data using a portioning function between stages, e.g., *reduce* function. In this fashion, one can construct the pipeline structures used in most stream processing systems, but by using feedback, we can also create nested cycles in which a new epoch's input is based on the last epoch's feedback result, explained in more detail next.

**Pipeline structures.** We build aggregation trees using DHTs for pipeline dataflows, in which each level of the tree progressively '*aggregates*' the data until the result arrives at the root. For a non-partitioning function, the agent as a vertex simply processes the data stream locally using the CBT. For a partitioning function like `TopKCount` in which the key-value pairs are shuffled and gradually truncated, we build a single aggregation tree, e.g., Figure 6 middle shows how the *groupby*, *aggregate*, *sort* functions are applied for each level-$i$ subtree's root for the micro-promotion job. For partitioning functions like `WordCount`, we build $m$ aggregation trees to divide the keys into $m$ ranges, where each tree is responsible for the reduce function of one range, thus avoiding the root overload when aggregating a large key space. Figure 6 right shows how the '*fat-tree*'-like aggregation tree is built for the product-bundling job.

**Cycles.** Naiad [20] uses timestamp vectors to realize dataflow cycles, whereas ELF employs multicast services operating on a job's aggregation tree to create feedback loops in which the results obtained for a job's last epoch are re-injected into its sources. Each job's master has complete control over the contents of feedback messages and how often they are sent. Feedback messages, therefore, can be used to go beyond supporting cyclic jobs to also exert application-specific controls, e.g., set a new threshold, synchronize a new batch, install new job functionality for agents to use, etc.

**Why SRTs?** The use of DHTs affords the efficient construction of aggregation trees and multicast services, as their converging properties guarantee aggregation or multicast to be fulfilled within only $O(logN)$ hops. Further, a single overlay can support many different independent groups, so that the overheads of maintaining a proximity-aware overlay network can be amortized over all those group spanning trees. Finally, because all of these trees share the same set of underlying agents, each agent can be an input leaf, an internal node, the root, or any combination of the above, causing the computation load well balanced. This is why we term these structures "shared reducer trees" (SRTs).

Implementing feedback loops using DHT-based multicast benefits load and bandwidth usage: each message is replicated in the internal nodes of the tree, at each level,

so that only $m$ copies are sent to each internal node's $m$ children, rather than having the tree root broadcast $N$ copies to $N$ total nodes. Similarly, coordination across jobs via the DHT's routing methods is entirely decentralized, benefiting scalability and flexibility, the latter because concurrent ELF jobs use event-based methods to remain responsive to other jobs and/or to user interaction.

## 2.4 ELF API

| |
|---|
| **Subscribe(Id appid)**<br>Vertex sends JOIN message to construct SRT with the root's nodeid equals to appid. |
| **OnTimer()**<br>Callback. Invoked periodically. This handler has no return value. The master uses it for its periodic activities. |
| **SendTo(Id nodeid, PAO paos)**<br>Vertex sends the key-value pairs to the parent vertex with nodeid, resulting in a corresponding invocation of **OnRecv**. |
| **OnRecv(Id nodeid, PAO paos)**<br>Callback. Invoked when vertex receives serialized key-value pairs from the child vertex with nodeid. |
| **Multicast(Id appid, Message message)**<br>Application's master publishes control messages to vertices, e.g., synchronizing CBTs to be emptied; application's master publishes last epoch's result, encapsulated into a message, to all vertices for iterative loops; or application's master publishes new functions, encapsulated into a message, to all vertices for updating functions. |
| **OnMulticast(Id appid, Message message)**<br>Callback. Invoked when vertex receives the multicast message from application's master. |

**Table 1:** Data plane API

| |
|---|
| **Route(Id appid, Message message)**<br>Vertex or master sends a message to another application. The appid is the hash value of the target application's name concatenated with its creator's name. |
| **Deliver(Id appid, Message message)**<br>Callback. Invoked when the application's master receives an outsider message from another application with appid. This outsider message is usually a query for the application's status such as results. |

**Table 2:** Control plane API

Table 1 and Table 2 show the ELF's data and control plane APIs, respectively. The data plane APIs concern data processing within a single application. The control plane APIs are for coordination between different applications.

```
ArrayList<String> topk;
void OnTimer () {
if (this.isRoot()) {
   this.Multicast(hash("micro-promotion"), new topk(topk));
   this.Multicast(hash("micro-promotion"), new update()); }
   }
void OnMulticast(Id appid, Message message) {
if (message instanceof topk) {
   for(String product: message.topk) {
      if(this.hasProduct(product))
      //if it is an topk message, appear discount ... }
      }
//if it is an update message, start a new batch
else if (message instanceof update) {
   //if leaves, flush CBT and update to the parent vertex
   if (!this.containsChild(appid)) {
      PAO paos = cbt.get(appid).flush();
      this.SendTo (this.getParent(appid), paos);
      cbt.get(appid).empty(); }
      }
   }
```

**Figure 7:** ELF implementation of micro-promotion application

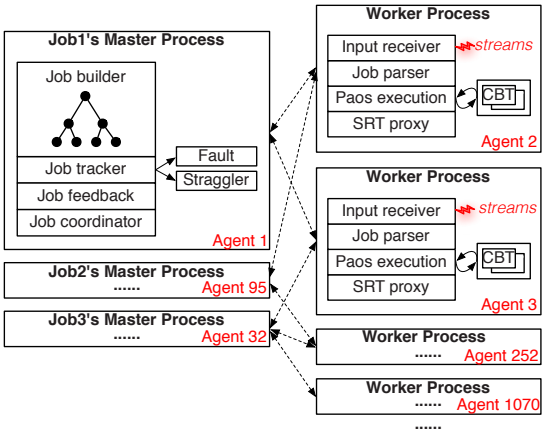



**Figure 8:** Components of ELF.

A sample use shown in Figure 7 contains partial code for the micro-promotion application. It multicasts update messages periodically to empty agents' CBTs for synchronous batch processing. It multicasts top-$k$ results periodically to agents. Upon receiving the results, each agent checks if it has the top-$k$ product, and if true, the extra discount will appear on the web page. To implement the product-bundling application, the agents subscribe to multiple SRTs to separately aggregate key-value pairs, and agents' associated CBTs are flushed only (without being synchronously emptied), to send a sliding window value to the parent vertices for asynchronously processing. To implement the sale-predication application, the master encapsulates its query and routes to the other two applications to get their intermediate results using `Route`.

# 3 Implementation

This section describes ELF's architecture and prototype implementation, including its methods for dealing with faults and with stragglers.

## 3.1 System Architecture

Figure 8 shows ELF's architecture. We see that unlike other streaming systems with static assignments of nodes to act as masters vs. workers, all ELF agents are treated equally. They are structured into a P2P overlay, in which each agent has a unique nodeId in the same flat circular 128-*bit* node identifier space. After an application is launched, agents that have target streams required by the application are automatically assembled into a shared reducer tree (SRT) via their routing substrates. It is only at that point that ELF assigns one or more of the following roles to each participating agent:

**Job master** is SRT's root, which tracks its own job's execution and coordinates with other jobs' masters. It has four components:

- *Job builder* constructs the SRT to roll up and aggregate the distributed PAOs snapshots processed by

local CBTs.

- *Job tracker* detects *key-value* errors, recovers from faults, and mitigates stragglers.
- *Job feedback* is continuously sent to agents for iterative loops, including last epoch's results to be iterated over, new job functions for agents to be updated on-the-fly, application-specific control messages like '*new discount*', etc.
- *Job coordinator* dynamically interacts with other jobs to carry out interactive queries.

**Job worker** uses a local CBT to implement some application-specific execution model, e.g., asynchronous stream processing with a sliding window, synchronous incremental batch processing with historical records, etc. For consistency, job workers are synchronized by the job master to 'roll up' the intermediate results to the SRT for global aggregation. Each worker has five components:

- *Input receiver* observes streams. Its current implementation assumes logs are collected with Flume [1], so it employs an interceptor to copy stream events, then parses each event into a job-specified *key-value* pair. A typical Flume event is a tuple with *timestamp*, *source IP*, and *event body* that can be split into columns based on different key-based attributes.
- *Job parser* converts a job's SQL description into a workflow of operator functions $\mathbf{f}$, e.g., aggregations, grouping, and filters.
- *PAOs execution*: each *key-value* pair is represented as a partial aggregation object (PAO) [9]. New PAOs are inserted into and accumulated in the CBT. When the CBT is "*flushed*", new and past PAOs are aggregated and returned, e.g., $\langle argu, 2, \mathbf{f} : count() \rangle$ merges with $\langle argu, 5, \mathbf{f} : count() \rangle$ to be a PAO $\langle agru, 7, \mathbf{f} : count() \rangle$.
- *CBT* resides in local agent's memory, but can be externalized to SSD or disk, if desired.
- *SRT proxy* is analogous to a socket, to join the P2P overlay and link with other SRT proxies to construct each job's SRT.

A key difference to other streaming systems is that ELF seeks to obtain scalability by changing the system architecture from $1:n$ to $m:n$, where each job has its own master and appropriate set of workers, all of which are mapped to a shared set of agents. With many jobs, therefore, an agent act as one job's master and another job's worker, or any combination thereof. Further, using DHTs, jobs' reducing paths are constructed with few overlaps, resulting in ELF's management being fully decentralized and load balanced. The outcome is straightforward scaling to large numbers of concurrently running jobs, with each master controls its own job's execution, including to react to failures, mitigate stragglers, alter a job as it is running, and coordinate with other jobs at runtime.

## 3.2 Consistency

The consistency of states across nodes is an issue in streaming systems that eagerly process incoming records. For instance, in a system counting page views from male users on one node and females on another, if one of the nodes is backlogged, the ratio of their counts will be wrong [28]. Some systems, like Borealis [6], synchronize nodes to avoid this problem, while others, like Storm [4], ignore it.

ELF's consistency semantics are straightforward, leveraging the fact that each CBT's intermediate results (PAOs snapshots) are uniquely named for different timestamped intervals. Like a software combining tree barrier, each leaf uploads the first interval's snapshot to its parent. If the parent discovers that it is the last one in its direct list of children to do so, it continues up the tree by aggregating the first interval's snapshots from all branches, else it blocks. Figure 9 shows an example in which $agent_1$ loses $snapshot_0$, and thus blocks $agent_5$ and $agent_7$. Proceeding in this fashion, a late-coming snapshot eventually blocks the entire upstream path to the root. All snapshots from distributed CBTs are thus sequentially aggregated.

## 3.3 Fault Recovery

ELF handles transmission faults and agent failures, transient or permanent. For the former, the current implementation uses a simple XOR protocol to detect the integrity of records transferred between each source and destination agent. Upon an XOR error, records will be resent. We deal with agent failures by leveraging CBTs and the robust nature of the P2P overlay. Upon an agent's failure, the dataset cached in the agent's CBT is re-issued, the SRT is re-constructed, and all PAOs are recomputed using a new SRT from the last checkpoint.

In an ongoing implementation, we also use hot replication to support live recovery. Here, each agent in the overlay maintains a routing table, a neighborhood set, and a leaf set. The neighborhood set contains the nodeIds hashed from the webservers' IP addresses that are closest to the local agent. The job master periodically checkpoints each agent's snapshots in the CBT, by asynchronously replicating them to the agent's neighbors.

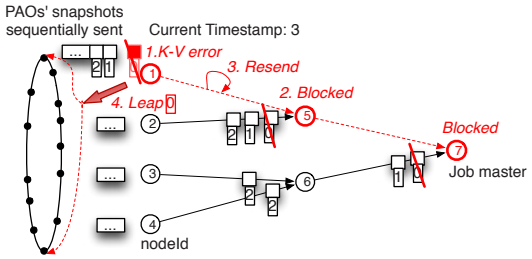



**Figure 9:** Example of the *leaping straggler* approach. $Agent_1$ notifies all members to discard $snapshot_0$.

ELF's approach to failure handling is similar to that of Spark and other streaming systems, but has potential advantages in data locality because the neighborhood set maintains geographically close (i.e., within the rack) agents, which in turn can reduce synchronization overheads and speed up the rebuild process, particularly in datacenter networks with limited bi-section bandwidths.

### 3.4 Straggler Mitigation

Straggler mitigation, including to deal with transient slowdown, is important for maintaining low end-to-end delays for time-critical streaming jobs. Users can instruct ELF jobs to exercise two possible mitigation options. First, as in other stream processing systems, speculative backup copies of slow tasks could be run in neighboring agents, termed the "*mirroring straggler*" option. The second option in actual current use by ELF is the "*leaping straggler*" approach, which skips the delayed snapshot and simply jumps to the next interval to continue the stream computation.

Straggler mitigation is enabled by the fact that each agent's CBT states are periodically checkpointed, with a timestamp at every interval. When a CBT's Paos snapshots are rolled up from leaves to root, the straggler will cause all of its all upstream agents to be blocked. In the example shown in Figure 9, $agent_1$ has a transient failure and fails to resend the first checkpoint's data for some short duration, blocking the computations in $agent_5$ and $agent_7$. Using a simple threshold to identify it as a straggler – whenever its parent determines it to have fallen two intervals behind its siblings – $agent_1$ is marked as a straggler. $Agent_5$, can use the *leaping straggler* approach: it invalidates the first interval's checkpoints on all agents via multicast, and then jumping to the second interval.

The *leaping straggler* approach leverages the streaming nature of ELF, maintaining timeliness at reduced levels of result accuracy. This is critical for streaming jobs operating on realtime data, as when reacting quickly to changes in web user behavior or when dealing with realtime sensor inputs, e.g., indicating time-critical business decisions or analyzing weather changes, stock ups and downs, etc.

## 4 Evaluation

ELF is evaluated with an online social network (OSN) monitoring application and with the well-known `WordCount` benchmark application. Experimental evaluations answer the following questions:

- What performance and functionality benefits does ELF provide for realistic streaming applications (Sec.4.1)?
- What is the throughput and processing latency seen for ELF jobs, and how does ELF scale with number of nodes and number of concurrent jobs (Sec.4.2)?
- What is the overheads of ELF in terms of CPU, memory, and network load (Sec.4.3)?

### 4.1 Testbed and Application Scenarios

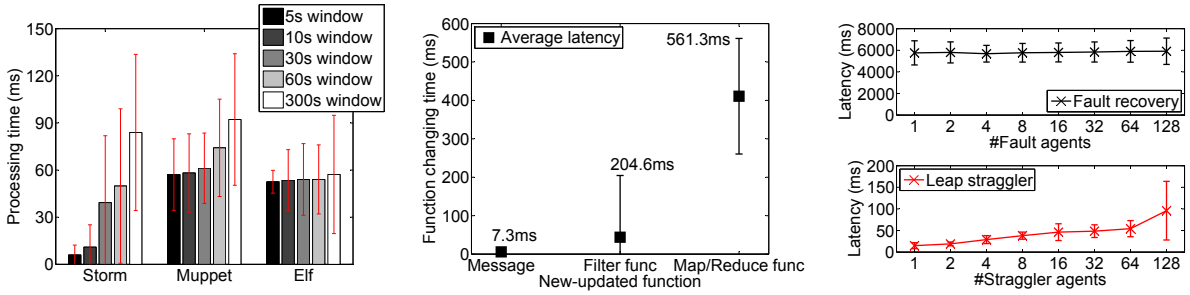
Experiments are conducted on a testbed of 1280 agents hosted by 60 server blades running Linux 2.6.32, all connected via Gigabit Ethernet. Each server has 12 cores (two 2.66GHz six-core Intel X5650 processors), 48GB of DDR3 RAM, and one 1TB SATA disk.

ELF's functionality is evaluated by running an actual application requiring both batch and stream processing. The application's purpose is to identify social spam campaigns, such as compromised or fake OSN accounts used by malicious entities to execute spam and spread malware [13]. We use the most straightforward approach to identify them – by clustering all spam containing the same label, such as an URL or account, into a campaign. The application consumes the events, all labeled as "sales", from the replay of a previously captured data stream from Twitter's public API [3], to determine the top-$k$ most frequently twittering users publishing "sales". After listing them as suspects, job functions are changed dynamically, to investigate further, by setting different filtering conditions and zooming in to different attributes, e.g., locations or number of followers.

The application is implemented to obtain online results from live data streams via ELF's agents, while in addition, also obtaining offline results via a Hadoop/HBase backend. Having both live and historical data is crucial for understanding the accuracy and relevance of online results, e.g., to debug or improve the online code. ELF makes it straightforward to mix online with offline processing, as it operates in ways that bypass the storage tier used by Hadoop/HBase.

Specifically, live data streams flow from webservers to ELF and to HBase. For the web tier, there are 1280 emulated webservers generating Twitter streams at a rate of 50 *events/s* each. Those streams are directly intercepted by ELF's 1280 agents, that filter tuples for processing, and concurrently, unchanged streams are gathered by Flume to be moved to the HBase store. The storage tier has 20 servers, in which the name node and job tracker run on a master server, and the data node and task trackers run on the remaining machines. Task trackers are configured to use two map and two reduce slots per worker node. HBase coprocessor, which is analogous to Google's BigTable coprocessor, is used for offline batch processing.

**Comparison with Muppet and Storm.** With ad-hoc queries sent from a shell via ZeroMQ [5], comparative results are obtained for ELF, Muppet, and Storm, with varying sizes of sliding windows. Figure 10a shows that ELF consistently outperforms Muppet. It achieves performance superior to Storm for large window sizes, e.g., 300 *s*, because the CBT's data structure stabilizes '*flush*'




(a) Comparison of processing times. (b) Latency for job function changes. (c) Fault and straggler recovery.

**Figure 10:** Processing times, latency of function changes, and recovery results of ELF on the Twitter application.

cost by organizing the compressed historical records in an $(a,b)$ tree, enabling fast merging with large numbers of past records.

**Job function changes.** Novel in ELF is the ability to change job functions at runtime. As Figure 10b shows, it takes less than 10 *ms* for the job master to notify all agents about some change, e.g., to publish discounts in the microsale example. To zoom in on different attributes, the job master can update the filter functions on all agents, which takes less than 210 *ms*, and it takes less than 600 *ms* to update user-specified map/reduce functions.

**Fault and Straggler Recovery.** Fault and straggler recovery are evaluated with methods that use human intervention. To cause permanent failures, we deliberately remove some working agents from the datacenter to evaluate how fast ELF can recover. The time cost includes recomputing the routing table entries, rebuilding the SRT links, synchronizing CBTs across the network, and resuming the computation. To cause stragglers via transient failures, we deliberately slow down some working agents, by collocating them with other CPU-intensive and bandwidth-aggressive applications. The leap ahead method for straggler mitigation is fast, as it only requires the job master to send a multicast message to notify everyone to drop the intervals in question.

Figure 10c reports recovery times with varying numbers of agents. The top curve shows that the delay for fault recovery is about 7 *s*, with a very small rate of increase with increasing numbers of agents. This is due to the DHT overlay's internally parallel nature of repairing the SRT and routing table. The bottom curve shows that the delay for ELF's leap ahead approach to dealing with stragglers is less than 100 *ms*, because multicast and subsequent skipping time costs are trivial compared to the cost of full recovery.

## 4.2 Performance

Data streams propagate from ELF's distributed CBTs as leaves, to the SRT for aggregation, until the job master at the SRT's root has the final results. Generated live streams are first consumed by CBTs, and the SRT only picks up truncated *key-value* pairs from CBTs for subsequent shuffling. Therefore, the CBT, as the starting point for parallel streaming computations, directly influences ELF's overall throughput. The SRT, as the tree structure for shuffling *key-value* pairs, directly influences total job processing latency, which is the time from when records are sent to the system to when results incorporating them appear at the root. We first report the per-node throughput of ELF in Figure 11, then report data shuffling times for different operators in Figure 12a 12b. The degree to which loads are balanced, an important ELF property when running a large number of concurrent streaming jobs, is reported in Figure 12c.

**Throughput.** ELF's high throughput for local aggregation, even with substantial amounts of local state, is based in part on the efficiency of the CBT data structure used for this purpose. Figure 11 compares the aggregation performance and memory consumption of the Compressed Buffer Tree (CBT) with a state-of-the-art concurrent hashtable implementation from Google's `sparsehash` [2]. The experiment uses a microbenchmark running the `WordCount` application on a set of input files containing varying numbers of unique keys. We measure the per-unique-key memory consumption and throughput of the two data structures. Results show that the CBT consumes significantly less memory per key, while yielding similar throughput compared to the hashtable. Tests are run with equal numbers of CPUs (12 cores), and hashtable

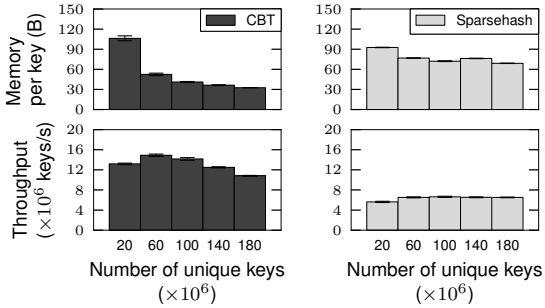
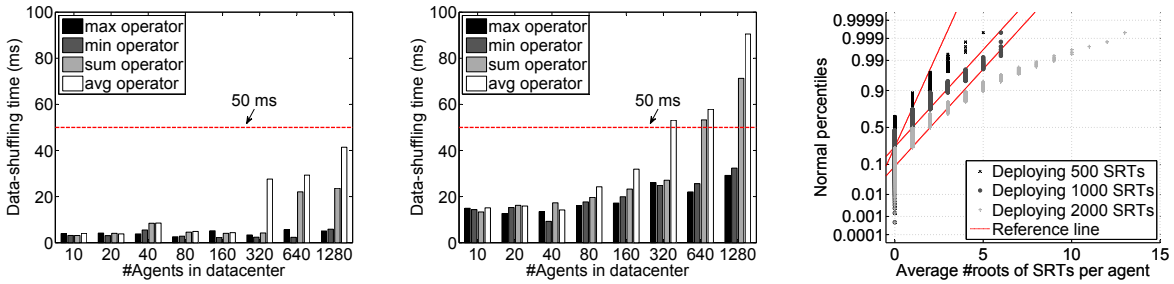



**Figure 11:** Comparison of CBT with Google Sparsehash.



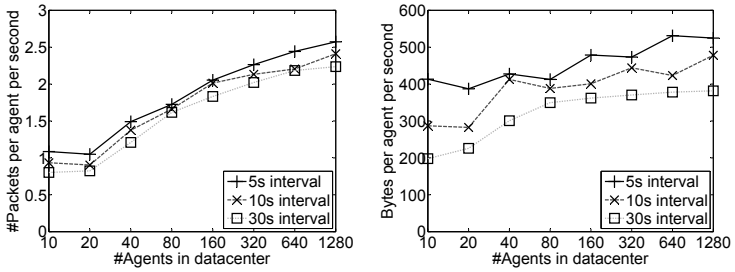

(a) Running operators separately. (b) Running operators simultaneously. (c) Normal probability plot of #roots.

**Figure 12:** Performance evaluation of ELF on data-shuffling time and load balance.

| SPS | CPU | Memory | I/O | C-switch |
|---|---|---|---|---|
| | %used | %used | wtps | cswsh/s |
| ELF | **2.96%** | **5.73%** | 3.39 | 780.44 |
| Flume | 0.14% | 5.48% | 2.84 | 259.23 |
| S-master | 0.06% | 9.63% | 2.96 | 652.22 |
| S-worker | 1.17% | 15.91% | 11.47 | 11198.96 |

*SPS*: stream processing system.
*wtps*: write transactions per second.
*cswsh/s*: context switches per second.

(a) Runtime overheads of ELF vs. others.




(b) Additional #packets overhead. (c) Additional bytes overhead.

**Figure 13:** Overheads evaluation of ELF on runtime cost and network cost.

performance scales linearly with the number of cores.

ELF's per-node throughput of over 1000,000 *keys/s* is in a range similar to Spark Streaming's reported best throughput (640,000 *records/s*) for `Grep`, `WordCount`, and `TopKCount` when running on 4-core nodes. It is also comparable to the speeds reported for commercial single-node streaming systems, e.g., Oracle CEP reports a throughput of 1 million *records/s* on a 16-core server and StreamBase reports 245,000 *records/s* on a 8-core server.

**Operators.** Figure 12a 12b reports ELF's data shuffling time of ELF when running four operators separately vs. simultaneously. By data shuffling time, we mean the time from when the SRT fetches a CBT's snapshot to the result incorporating it appears in the root. `max` sorts *key-value* pairs in a descending order of value, and `min` sorts in an ascending order. `sum` is similar to `WordCount`, and `avg` refers to the frequency of words divided by the occurrence of *key-value* pairs. As `sum` does not truncate *key-value* pairs like `max` or `min`, and `avg` is based on `sum`, naturally, `sum` and `avg` take more time than `max` and `min`.

Figure 12a 12b demonstrates low performance interference between concurrent operators, because both data shuffling times seen for separate operators and concurrent operators are less than 100 *ms*. Given the fact that concurrent jobs reuse operators if processing logic is duplicated, the interference between concurrent jobs is also low. Finally, these results also demonstrate that SRT scales well with the datacenter size, i.e., number of webservers, as the reduce times increase only linearly with exponential increase in the number of agents. This is because reduce times are strictly governed by an SRT's depth $O(log_{16}N)$, where $N$ is the number of agents in the datacenter.

**Balanced Load.** Figure 12c shows the normal probability plot for the expected number of roots per agent. These results illustrate a good load balance among participating agents when running a large number of concurrent jobs. Specifically, assuming the root is the agent with the highest load, results show that 99.5% of the agents are the roots of less than 3 trees when there are 500 SRTs total; 99.5% of the agents are the roots of less than 5 trees when there are 1000 SRTs total; and 95% of the agents are the roots of less than 5 trees when there are 2000 SRTs total. This is because of the independent nature of the trees' root IDs that are mapped to specific locations in the overlay.

## 4.3 Overheads

We evaluate ELF's basic runtime overheads, particularly those pertaining to its CBT and SRT abstractions, and compare them with Flume and Storm. The CBT requires additional memory for maintaining intermediate results, and the SRT generates additional network traffic to maintain the overlay and its tree structure. Table 13a and Figure 13b 13c present these costs, explained next.

**Runtime overheads.** Table 13a shows the per-node runtime overheads of ELF, Flume, Storm master, and Storm worker. Experiments are run on 60 nodes, each with 12 cores and 48GB RAM. As Table 13a shows, ELF's runtime overheads is small, comparable to Flume, and much less than that of Storm master and Storm worker. This

is because both ELF and Flume use a decentralized architecture that distributes the management load across the datacenter, which is not the case for Storm master. Compared to Flume, which only collects and aggregates streams, ELF offers the additional functionality of providing fast, general stream processing along with per-job managemen mechanisms.

**Network overheads.** Figure 13b 13c show the additional network traffic imposed by ELF with varying update intervals, when running the Twitter application. We see that the number of packets and number of bytes sent per agent increase only linearly, with an exponential increase in the number of agents, at a rate less than the increase in update frequency (from $1/30$ to $1/5$). This is because most packets are ping-pong messages used for overlay and SRT maintenance (initialization and keep alive), for which any agent pings to a limited set of neighboring agents. We estimate from Figure 13b that when scaling to millions of agents, the additional #package is still bounded to 10.

# 5 Related Work

**Streaming Databases.** Early systems for stream processing developed in the database community include Aurora [29], Borealis [6], and STREAM [10]. Here, a query is composed of fixed operators, and a global scheduler decides which tuples and which operators to prioritize in execution based on different policies, e.g., interesting tuple content, QoS values for tuples, etc. SPADE [14] provides a toolkit of built-in operators and a set of adapters, targeting the System S runtime. Unlike SPADE or STREAM that use SQL-style declarative query interfaces, Aurora allows query activity to be interspersed with message processing. Borealis inherits its core stream processing functionality from Aurora.

**MapReduce-style Systems.** Recent work extends the batch-oriented MapReduce model to support continuous stream processing, using techniques like pipelined parallelism, incremental processing for map and reduce, etc.

MapReduce Online [12] pipelines data between map and reduce operators, by calling reduce with partial data for early results. Nova [22] runs as a workflow manager on top of an unmodified Pig/Hadoop software stack, with data passes in a continuous fashion. Nova claims itself as a tool more suitable for large batched incremental processing than for small data increments. Incoop [11] applies memorization to the results of partial computations, so that subsequent computations can reuse previous results for unchanged inputs. One-Pass Analytics [17] optimizes MapReduce jobs by avoiding expensive I/O blocking operations such as reloading map output.

iMR [19] offers the MapReduce API for continuous log processing, and similar to ELF's agent, mines data locally first, so as to reduce the volume of data crossing the network. CBP [18] and Comet [15] run MapReduce jobs on new data every few minutes for "bulk incremental processing", with all states stored in on-disk filesystems, thus incurring latencies as high as tens of seconds. Spark Streaming [28] divides input data streams into batches and stores them in memory as RDDs [27]. By adopting a batch-computation model, it inherits powerful fault tolerance via parallel recovery, but any dataflow modification, e.g., from pipeline to cyclic, has to be done via the single master, thus introducing overheads avoided by ELF's decentralized approach. For example, it takes Spark Streaming seconds for iterating and performing incremental updates, but milliseconds for ELF.

All of the above systems inherit MapReduce's "single master" infrastructure, in which parallel jobs consist of hundreds of tasks, and each single task is a pipeline of map and reduce operators. The single master node places those tasks, launches those tasks, maybe synchronizes them, and keeps track of their status for fault recovery or straggler mitigation. The approach works well when the number of parallel jobs is small, but does not scale to hundreds of concurrent jobs, particularly when these jobs differ in their execution models and/or require customized management.

**Large-scale Streaming Systems.** Streaming systems like S4 [21], Storm [4], Flume [1], and Muppet [16] use a message passing model in which a stream computation is structured as a static dataflow graph, and vertices run stateful code to asynchronously process records as they traverse the graph. There are limited optimizations on how past states are stored and how new states are integrated with past data, thus incurring high overheads in memory usage and low throughput when operating over larger time windows. For example, Storm asks users to write codes to implement sliding windows for trend topics, e.g., using `Map<>`, `Hashmap<>` data structure. Muppet uses an in-memory hashtable-like data structure, termed a slate, to store past keys and their associated values. Each key-value entry has an update trigger that is run when new records arrive and aggressively inserts new values to the slate. This creates performance issues when the key space is large or when historical windowsize is large. ELF, instead, structures sets of key-value pairs as compressed buffer trees (CBTs) in memory, and uses lazy aggregation, so as to achieve high memory efficiency and throughput.

Systems using persistent storage to provide full fault-tolerance. MillWheel [7] writes all states contained in vertices to some distributed storage system like BigTable or Spanner. Percolator [23] structures a web indexing computation as triggers that run when new values are written into a distributed key-value store, but does not offer consistency guarantees across nodes. TimeStream [24] runs the continuous, stateful operators in Microsoft's StreamInsight [8] on a cluster, scaling with load swings through

repartitioning or reconfiguring sub-DAGs with more or less operators. ELF's CBT resides in a local agent's memory, but can be externalized to SSD or disk, if desired, to also fully support fault-tolerance.

ELF is most akin to Naiad [20], which uses vector timestamps to implement cyclic dataflows and also achieves tens of milliseconds for iterations and incremental updates. We differ from Naiad, which sends only data feedback, in that ELF's application-customized master can send feedback messages that can concern data, job control, and new job functions.

In contrast to all of the systems reviewed above, ELF obtains scalability in terms of the number of concurrent jobs run on incoming data via its fully decentralized "many masters" infrastructure. ELF's jobs can differ in their execution models, yet interact to coordinate their actions and/or build on each others' results.

# 6 Conclusion

ELF implements a novel decentralized model for stream processing that can simultaneously run hundreds of concurrent jobs, by departing from the common "*one master many workers*" architecture to instead, using a "*many masters many workers*" approach. ELF's innovations go beyond the consequent scalability improvements, to also providing powerful programming abstraction for iterative, batch, and streaming processing, and to offer new functionalities that include support for runtime job function change and for cross-job coordination.

Experimental evaluations demonstrate ELF's scalability to up to a thousand concurrent jobs, high per-node throughput, sub-second job latency, and sub-second ability to adjust the actions of jobs being run.

Future work on ELF will go beyond additional implementation steps, e.g., to enhance SRTs for fast aggregation of non-truncated *key-value* pairs, to further optimize performance and to add robustness by extending and experimenting with additional methods for fault recovery.

# References


[1] Flume. `http://flume.apache.org/`, 2013.

[2] Sparsehash. `http://code.google.com/p/sparsehash/`.

[3] Twitter streaming apis. `https://dev.twitter.com/docs/streaming-apis`, 2012.

[4] Storm. `https://github.com/nathanmarz/storm.git`.

[5] Zeromq. `http://zeromq.org/`, 2012.

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

[7] 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.

[8] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, and e. Tarnavski. Microsoft cep server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558–1561, Aug. 2009.

[9] H. Amur, W. Richter, D. G. Andersen, M. Kaminsky, K. Schwan, A. Balachandran, and E. Zawadzki. Memory-efficient groupby-aggregate using compressed buffer trees. In *SOCC*. 2013.

[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[11] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *SOCC*, pages 7:1–7:14. 2011.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*. 2010.

[13] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *IMC*. 2010.

[14] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD*, pages 1123–1134. 2008.

[15] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SOCC*, pages 63–74. 2010.

[16] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: Mapreduce-style processing of fast data. *Proc. VLDB Endow.*, 5(12):1814–1825, Aug. 2012.

[17] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, pages 985–996. 2011.

[18] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SOCC*, pages 51–62. 2010.

[19] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. In *USENIXATC*. 2011.

[20] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*. 2013.

[21] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177. 2010.

[22] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous pig/hadoop workflows. In *SIGMOD*, pages 1081–1090. 2011.

[23] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*. 2010.

[24] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: reliable stream computation in the cloud. In *Eurosys*, pages 1–14. 2013.

[25] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350. 2001.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. 2012.

[28] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*. ACM, Sept. 2013.

[29] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *Data Engineering*, 51:3, 2003.