

# **ELF: EFFICIENT LIGHTWEIGHT FAST STREAM PROCESSING AT SCALE**

A Thesis  
Presented to  
The Academic Faculty

by

Liting Hu

In Partial Fulfillment  
of the Requirements for the Degree  
Doctor of Philosophy in the  
College of Computing, School of Computer Science

Georgia Institute of Technology  
August 2016

Copyright © 2016 by Liting Hu

# ELF: EFFICIENT LIGHTWEIGHT FAST STREAM PROCESSING AT SCALE

Approved by:

Dr. Matthew Wolf, Committee Chair  
College of Computing, School of  
Computer Science  
*Georgia Institute of Technology*

Professor Karsten Schwan, Advisor  
College of Computing, School of  
Computer Science  
*Georgia Institute of Technology*

Professor Dilma Da Silva  
Department of Computer Science and  
Engineering  
*Texas A&M University*

Professor Ling Liu  
College of Computing, School of  
Computer Science  
*Georgia Institute of Technology*

Professor Umakishore Ramachandran  
College of Computing, School of  
Computer Science  
*Georgia Institute of Technology*

Professor Douglas M. Blough  
College of Computing, School of  
Electrical and Computer Engineering  
*Georgia Institute of Technology*

Dr. Greg Eisenhauer  
College of Computing, School of  
Computer Science  
*Georgia Institute of Technology*

Date Approved: 6 May 2016

*To my family*

## ACKNOWLEDGEMENTS

I am thankful to my advisor, Dr.Karsten Schwan and Dr.Matthew Wolf, who guided me tirelessly throughout my PhD study. Karsten introduced me to the research field and taught me to do research from the very beginning, while giving me lots of freedom to explore the topic that I am interested in. He is very supportive, patient and insightful. Many random ideas from him have been proved to be invaluable in the future. I still remember when my paper got rejected, he drove me directly to the business department to meet professors to exchange ideas, which means much more to me than just beautiful words.

Dr.Matthew Wolf is a brilliant researcher who is always able to push ideas one level further. Matt is very accessible. He has put significant amounts of time in my work and also gave me tons of advices on my job talk and career seeking. I am fortunate to work with him and benefit from his perspectives.

I would like to express my deep gratitude to Dr.Dilma Da Silva, Dr.Ling Liu, Dr.Douglas M.Blough, Dr.Umakishore Ramachandran and Dr.Greg Eisenhauer for serving on my thesis committee. Dr.Dilma Da Silva was my manager at IBM Research. She is always nice, encouraging and believes that I can do good work. Dr.Ling Liu is a sharp, intelligent professor. She offered me quite useful helps on my job seeking and offer negotiation. Dr.Douglas M.Blough helped me greatly on my new work. Their insightful feedback on my thesis has significantly improved the quality of the thesis.

Dr.Ajay Gulati is my mentor during my first summer internship at VMware. Ajay is the most talented people that I have ever met. He always encourages me to think and work creatively. He is the top man in our field but he is very nice, humble and

never arrogant. I am grateful for his help and I appreciate the time that he spent on teaching me to program gracefully, debug, present slides and many other things.

Dr. Michael Kozuch is my mentor during my summer internship at Intel labs, CMU. He is a nice, patient and caring person. Michael has a sense of humor. His humor influences me to look at all tough things in an optimistic way, and then we can keep moving forward even when there are a lot of difficulties existing.

I also own a debt of gratitude of the mentors who introduce me to the industry. They are Zhenyu Guo from Microsoft Research Asia, Hui Kang from IBM research, Michael V Le from IBM research, Kyung D Ryu from IBM research. They grant me hands-on research and engineering experiences and long-time friendship.

I would like to express my special thanks to Dr. Ada Gavrilovska. She helped me through the lows of my life and always encouraged me. I thank my labmates, Junjie Zhang, Fang Zhen, Chengwei Wang, Qingyan Wang, Wei Meng, Hrishikesh Amur, Vishal Gupta, Sudarsun Kannan, Mukil Kesavan, Mahendra Kutare, Min Lee, Alexander Merritt, Yanwei Zhang, Priyanka Tembey, Hobin Yoon, Dipanjan Sengupta, Ketan Bhardwaj, Minsung Jang, Yuzhe Tang and Jian Huang. Thanks Mrs. Susie McClain for helping me about paper works in the past seven years.

I have made great friends in Atlanta. Thanks for Feifei Qian for sharing much happy time together. Thanks for De Wang's wife Jing Chen for cooking me delicious food and offering me her house to sleep when I have the gap for renting. They made my stay at Atlanta lots of fun.

I have a great mother, Hong Zhou. Her unconditional love is always my source of strength. She sacrificed a lot for building a good life for me. My grandma Xingchun Wei raises me up. She is the wisest woman in the world. My father Caihan Hu is a funny man with warm. Finally, I want to thank my husband, Xin Chen. The moment that I met him, I know my life is going to be different. He gives me the best accompany no matter when I am happy or sad. He teaches me many life truths and

influences the way that I look at the world and people. He is the great husband and father. This thesis is dedicated to them.

# TABLE OF CONTENTS

<b>DEDICATION</b>	<b>iii</b>
<b>ACKNOWLEDGEMENTS</b>	<b>iv</b>
<b>LIST OF TABLES</b>	<b>x</b>
<b>LIST OF FIGURES</b>	<b>xi</b>
<b>SUMMARY</b>	<b>xiii</b>
<b>I INTRODUCTION</b>	<b>1</b>
1.1 A Motivating Example	4
1.2 ELF’s Approach	6
1.3 Contributions	8
1.4 Dissertation Structure	8
<b>II STREAMING PROCESSING OVERVIEW</b>	<b>10</b>
2.1 Examples of Real world Streaming Applications	10
2.2 General-purpose Streaming Processing Model	12
2.3 ELF Streaming Processing Model	16
<b>III ELF DESIGN</b>	<b>19</b>
3.1 Overview	19
3.2 Compressed Buffer Trees (CBTs)	21
3.2.1 Log events	21
3.2.2 HiveQL-like query	22
3.2.3 CBT abstraction	23
3.2.4 CBT benefits	25
3.3 Shared Reducer Trees (SRTs)	26
3.3.1 SRT abstraction	26
3.3.2 Pipeline structures	29
3.3.3 Cycles	29

3.3.4	SRT benefits . . . . .	29
3.4	Evaluation . . . . .	30
3.4.1	CBT throughput . . . . .	30
3.4.2	SRT data shuffling time . . . . .	32
3.4.3	SRT load balance . . . . .	33
3.5	Limitations and Extensions . . . . .	34
3.5.1	Latency . . . . .	35
3.5.2	Throughput . . . . .	36
3.5.3	Communication patterns . . . . .	36
3.6	Discussion . . . . .	37
<b>IV</b>	<b>ELF IMPLEMENTATION . . . . .</b>	<b>38</b>
4.1	System Architecture . . . . .	38
4.1.1	Job master . . . . .	38
4.1.2	Job worker . . . . .	39
4.1.3	Workflow . . . . .	41
4.2	CBT-based Agent APIs . . . . .	42
4.3	DHT-based SRT APIs . . . . .	44
4.3.1	Multicast . . . . .	44
4.3.2	Aggregation . . . . .	45
4.3.3	Anycast . . . . .	46
4.4	User Query APIs . . . . .	47
4.4.1	Data plane APIs and control plane APIs . . . . .	47
4.4.2	Micro-promotion application example . . . . .	47
4.5	Feedback and Coordination . . . . .	49
4.5.1	Feedback action for MicroSale . . . . .	49
4.5.2	Coordination action for ProductBundling . . . . .	50
4.6	Evaluation . . . . .	51
4.6.1	Testbed and application scenarios . . . . .	51



4.6.2	Application performance . . . . .	53
4.6.3	Overheads . . . . .	54
4.7	Discussion . . . . .	56
<b>V</b>	<b>PROCESSING ELF AT SCALE . . . . .</b>	<b>58</b>
5.1	Failure Model . . . . .	59
5.2	Synchronization Assumptions . . . . .	60
5.3	Challenges of Fault and Straggler Tolerance . . . . .	62
5.4	Fault Recovery . . . . .	63
5.4.1	Detecting key-value errors . . . . .	63
5.4.2	State checkpointing and replay . . . . .	64
5.5	Straggler Mitigation . . . . .	65
5.6	Evaluation . . . . .	67
5.6.1	Error detecting time . . . . .	67
5.6.2	Checkpointing and replay load balancing . . . . .	68
5.6.3	Fault and straggler recovery . . . . .	70
5.7	Discussion . . . . .	72
<b>VI</b>	<b>RELATED WORK . . . . .</b>	<b>73</b>
6.1	Streaming Databases . . . . .	73
6.2	MapReduce-style Systems . . . . .	73
6.3	Large-scale Streaming Systems . . . . .	76
6.4	Data aggregation systems . . . . .	78
<b>VII</b>	<b>CONCLUSION . . . . .</b>	<b>80</b>
7.1	Lessons Learned . . . . .	80
7.2	Future Work . . . . .	82
	<b>REFERENCES . . . . .</b>	<b>86</b>
	<b>VITA . . . . .</b>	<b>93</b>

## LIST OF TABLES

1	Data plane APIs. . . . .	48
2	Control plane APIs. . . . .	48
3	Runtime overheads of ELF vs. others. . . . .	54
4	Current MapReduce projects and related software. . . . .	74
5	Current large-scale streaming systems and related software. . . . .	76

## LIST OF FIGURES

1	Examples of diverse concurrent applications. . . . .	4
2	Dataflow of ELF vs. a typical realtime web log analysis system, composed of Flume, HBase, HDFS, Hadoop MapReduce and Spark/Storm. . . . .	7
3	Declarative code that defines the pipeline of a sample microsale application. Stream events (line 1) are first extracted from a log file (line 2) using a Json extractor (line 3) and filtered based on certain conditions (line 4). Next, the input tuples are <i>reduced</i> with the user-defined function <b>ScoreReducer</b> (line 7) to produce a list of <i>key-value</i> pairs (line 6). Finally, the user-defined functions <b>MicroSale</b> (line 9) and <b>ProductBundling</b> (line 11) are executed. . . . .	14
4	General streaming processing model. Distributed streams from web-servers are first moved to data storage, e.g., HDFS. Then, batch jobs are pipelined for offline, long-term data analysis. Streaming jobs are submitted to a computation graph or topology for online, continuous analysis. . . . .	15
5	ELF streaming processing model. The stream generated by each web-server is locally parsed, and stored in agent's memory as immutable datasets for all intervals. Per-interval intermediate results are continuously reduced via a global DHT-based aggregation tree with embedded reduce functions. . . . .	17
6	High-level overview of the ELF system. . . . .	19
7	Example of a twitter event parsed into different key-value pairs and queued to be sent to CBTs for local pre-reducing. . . . .	22
8	Example of ELF QL query. . . . .	22
9	Between intervals, new <i>key-value</i> 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. . . . .	24
10	Shared Reducer Tree Construction for many jobs. . . . .	27
11	Comparison of CBT with Google Sparsehash. . . . .	31
12	Performance evaluation of ELF on data-shuffling when running operators separately. . . . .	32
13	Performance evaluation of ELF on data-shuffling when running operators simultaneously. . . . .	33

14	Performance evaluation of ELF on load balance. . . . .	34
15	Components of ELF. . . . .	39
16	Workflow of ELF. . . . .	41
17	Between intervals, new <i>key-value</i> 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. . . . .	43
18	(a) d46a1c and 98fc35 join the tree by routing JOIN requests towards the root d462ba. The requests are received at d462ba and d4213f without going any further, adding the branches of the aggregation tree. (b) SRT's key-value pairs are reduced towards the root. . . . .	46
19	ELF implementation of micro-promotion application. . . . .	49
20	Feedback abstraction for <b>MicroSale</b> . . . . .	50
21	Coordination abstraction for <b>ProductBundling</b> . . . . .	50
22	Comparison of processing times of ELF on the Twitter application. .	53
23	Additional #packets overhead of ELF with different update intervals.	55
24	Additional bytes overhead of ELF with different update intervals. . .	55
25	Example of the <i>consistency</i> semantics. <i>Agent</i> <sub>1</sub> triggers blocks of <i>Agent</i> <sub>5</sub> and <i>Agent</i> <sub>7</sub> . . . . .	61
26	Example of using XOR to detect key-value transmission errors. . . . .	63
27	<i>R</i> 's datasets are replicated to <i>C</i> , <i>D</i> and <i>E</i> at each checkpoint. If <i>R</i> fails, <i>C</i> , <i>D</i> , <i>E</i> rebuild <i>R</i> 's datasets from the last checkpoint. <i>C</i> takes over <i>R</i> 's tasks. . . . .	65
28	Example of the <i>leaping straggler</i> approach. <i>Agent</i> <sub>1</sub> notifies all members to discard <i>snapshot</i> <sub>0</sub> . . . . .	66
29	Error detecting time. . . . .	68
30	Checkpointing and replay load balancing for 5 SRTs. . . . .	69
31	Checkpointing and replay load balancing for 10 SRTs. . . . .	69
32	Checkpointing and replay load balancing for 15 SRTs. . . . .	70
33	Fault recovery. . . . .	71
34	Straggler recovery. . . . .	71

## SUMMARY

Large Internet companies like Facebook, Amazon, and Twitter are increasingly recognizing the value of stream data processing, using tools like Flume, Muppet, or Storm to continuously collect and process incoming data in real time to help govern company activities. Applications include monitoring marketing streams for business-critical decisions, identifying spam campaigns from social network streams, datacenter’s intrusion detection and troubleshooting, and others.

Technical challenges for stream processing include the following: 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. In contrast to batch jobs pipelined to run sequentially, streaming jobs are more likely to run concurrently. Stream processing platforms, therefore, must not only offer throughput and low latency, but also scale with the ever-larger concurrent jobs.

This dissertation presents a new stream processing model, termed ELF, which addresses these new challenges. ELF proposes a novel decentralized “many masters many workers” architecture implemented over a set of agents enriching the web tier of datacenter systems. ELF uses a DHT protocol to assign the jobs respective sets of master/workers mapping to the agents of the web tier, where for each job, the live data streams generated by web servers are first divided into mini-batches, then inserted and aggregated as space-efficient *compressed buffer trees* (CBTs) in local agents’ memories. Second, per-batch results are ‘flushed’ from CBTs, to be rolled up

and aggregated via *shared reducer trees* (SRTs), in ways that naturally balance SRT-induced load, reduce processing latencies, and allow online job changes along with cross-job coordination. An ELF prototype implemented and evaluated for a larger scale configuration demonstrates scalability, high per-node throughput, sub-second job latency, and subsecond ability to adjust the actions of jobs being run.

# CHAPTER I

## INTRODUCTION

The advent of big data mirrors our technological evolution as a society: we have the ability to easily and cheaply capture and store massive amounts of data in a way that was simply impossible before. Google took the 50 million most common search terms to identify areas infected by the flu virus. Oren Etzioni predicts if the price of plane ticket is increasing or decreasing in the future, to help customer to determine when to buy the ticket. Large Internet companies like Facebook, Amazon, and Twitter are increasingly recognizing the value of stream data processing, using tools like Flume [12], Muppet [50], or Storm [6] to continuously collect and process incoming data in real time to help govern company activities. Applications include monitoring marketing streams for business-critical decisions, identifying spam campaigns from social network streams, datacenter’s intrusion detection and troubleshooting, and others.

Given the scope, it is unsurprising that the computer systems and infrastructure propelling these changes are facing unprecedented challenges including the following:

**1. Efficient integration with past data:** a challenging task for streaming jobs is to integrate “present” with “past” data: it demands space-efficient in-memory storage of considerable data state and time-efficient operation over variably sized windows of stored such state. Existing hash-based aggregators do not offer the space-efficiency required for operating across potentially large-sized history windows. Such functionality is needed for a broad range of applications, an example being transactional fraud identification, which involves gathering the usual activity patterns for some **longer-term** duration, summarizing them as a “signature”, which is then compared

in realtime to current data arrivals.

**2. Rapid job initiation, termination, and update:** low delay is critical when initiating activities like fraud detection and/or when changing the signatures used by such jobs. Large delays in stopping or restarting jobs, or in changing job functions, would lead to increased risk and potentially substantial monetary loss.

**3. Scaling with insight:** it should be straightforward for multiple streaming jobs to be driven by the same source of data, and in addition, to coordinate with each other to share insights. For Twitter’s 400 million tweets per day, for instance, simultaneously running jobs may use that data to detect trending conversation topics, find popular twitter stars, identify new spam campaigns, etc. Further, such jobs must interact to analyze inter-job correlations, e.g., to determine leadders of trending conversations. Co-running many such jobs suggests the need for a decentralized “*many master/workers*” architecture rather than the centralized approach used by existing streaming systems governed by Hadoop’s “*single master/workers*” paradigm, like Muppet [50], Storm [6], Spark Streaming [87] and MapReduce Online [30].

**4. Transparent interoperation:** stream processing should enhance rather than replace the batch operations routinely used for offline analyses generating daily or monthly reports or learning about historical trends. Technically, this implies the need for stream processing to interoperate with said batch functionality in ways that avoid undue data copying or transformation, and without compromising the operation of batch processing systems. Current streaming systems [6] [39] do not address this need, nor does earlier work like Aurora [89] and Borealis [14].

This dissertation addresses the challenges articulated above with new abstractions for stream data processing, realized in the ELF decentralized streaming system:

For the first challenge, ELF leverages the memory-efficient *compressed buffer tree*

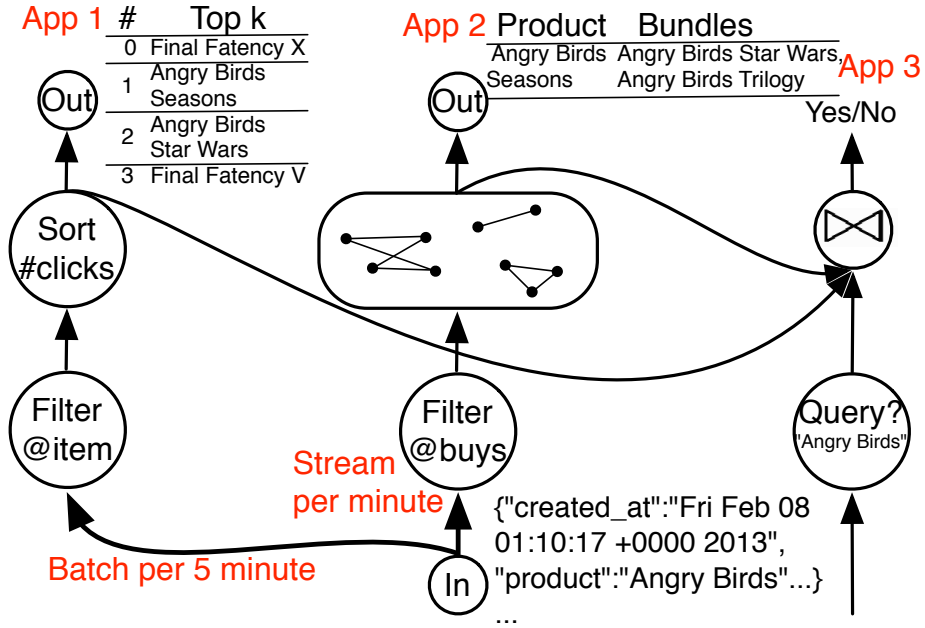


(CBT) [17]. With CBTs, “past” aggregated content is stored in compressed form as buffers in memory, thus enlarging the capacity of ‘past data’ windows. High throughput is obtained by organizing these compressed buffers as a  $B$ -tree that amortizes the cost of repeated “integrating new data” operations via lazy integration methods in which the merging of new *key-value* pairs with stored data is deferred until data is queried.

For the second and third challenges, ELF presents a decentralized data structure, termed *shared reducer tree* (SRT), to automatically assign each job a respective set of master and worker processes, through the DHT-based peer-to-peer overlay and its routing protocols. Thus, each job has its own master process to track its executions, adjust its functions and coordinate with other jobs on-the-fly. Due to the use of DHT, intuitively, diverse jobs’ aggregation paths, along with SRT-induced load, are well balanced, being able to scale to large number of concurrent jobs.

For the fourth challenge, ELF acts as a transparent layer running on top of the realtime web log processing infrastructure typically used by large Internet companies like Facebook, Amazon, or Twitter. ELF extracts and copies select webserver log data for stream processing, while such live data is simultaneously processed by the offline batch processing systems constructed with log collection infrastructures like Flume [12], Facebook’s Scribe [8], Yahoo’s Chukwa [70], or LinkedIn’s Kafka [49], backed by storage systems like HBase and HDFS maintaining data for subsequent use by batch data processing systems like Hadoop.

We implemented the ELF architecture in a stack of open source systems, including Apache Flume, Pastry, Scribe, Past, ZeroMQ, with Protocol Buffer as the common foundation. ELF is evaluated experimentally over 1000 logical webserver 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



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

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.

In the remainder of this chapter, we exemplify ELF's motivation with an realworld e-commerce example, then highlight ELF's approach.

### 1.1 A Motivating Example

We can best define big data by thinking of three **V**s, **V**olume (terabytes, petabytes amount), **V**elocity (realtime or near-realtime processing) and **V**ariety (social networks, blog posts, logs, sensors data), and the motivation to process big data are revolving around these three **V**s.

We exemplify ELF's approach with an e-commerce example shown as in Figure 1. When users interact with *amazon.com*, their user activities such as *clicks*, *likes*, *buys*,

from say, the *Video Games* department, will be continuously logged to the back-end databases, Amazon’s applications will run on these streaming data to generate business-critical decisions.

In this figure, the left pipeline represents the first application, the **micro-promotion** app, which extracts user *clicks* per product for the past 300s, and lists the top- $k$  popular products that have the most clicks. It can then dispatch coupons to those “popular” products so as to increase sales. To implement this app, first, the system needs to handle the **Volume** part like Hadoop system, because the realtime log streams are coming in at a rate of millions per seconds. Second, the system also needs to handle the **Velocity** part. The system needs to update the result in milliseconds, and hence the traditional database model where data is first stored and indexed and then subsequently processed in disk is slow and not a fit. The streaming data has to be in memory. The question is: ***How to store, organize and aggregate the substantial size of data in a limited size of memory while still achieving comparable processing throughput as disk?***

The middle pipeline represents the second application, the **product-bundling** app, which extracts 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 (60s). To implement this app, the main challenge is to guarantee the clean consistency of states across nodes, because the data may be stale due to asynchrony in the distributed system. For instance, consider each line of log event is sent to a different node responsible for updating the graph, if the node responsible for *Angry Birds* series falls behind the node for *Final Fantasy* series, e.g., due to load, then a snapshot of their states would be inconsistent: *Angry Birds*

graph would reflect an older timestamp of the stream than *Final Fantasy* graph, and would generate confusing results. The question is: Different applications have different levels of consistency requirements. ***How to provide various relaxed consistency guarantees to them, in one framework?***

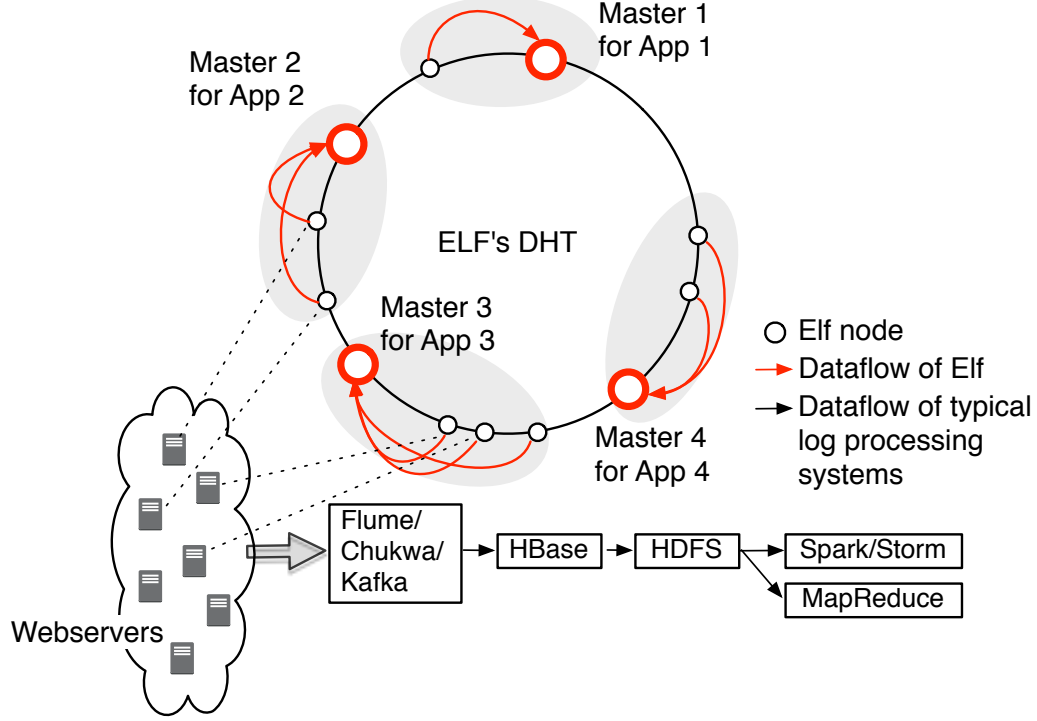
The right pipeline represents the third application, the `sale-prediction` app. Given a product name, e.g., *Angry Birds*, the `sale-prediction` app communicates information with the `product-bundling` app to find out what products are similar to *Angry Birds* (indicated by the ‘typically bought together’ set). The result is then joined with the `micro-promotion` app 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. Since it involves user interactive queries which frequently changes query predicates for other applications, the main challenge is ***how to guarantee the ‘milliseconds’ query latencies for communicating with various collocated applications?***

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 on-going 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. But, ***how to dynamically changing job functions on the fly without interfering the normal running of jobs?***

## ***1.2 ELF’s Approach***

The ELF (**E**fficient, **L**ightweight, **F**lexible) stream processing system presented in this dissertation implements novel functionality to meet the questions 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.



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

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 [72], 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 a webserver is parsed into a key-value pair and inserted into ELF’s local in-memory *compressed buffer tree* (CBT [17]) 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 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.

### **1.3 Contributions**

ELF has several key 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.
2. A memory-efficient approach for incremental updates, using a  $B$ -tree-like in-memory data structure, to store and manage large stored states.
3. Abstractions permitting cyclic dataflows via feedback loops, with additional uses of these abstractions including the ability to rapidly and dynamically change job behavior.
4. Support for cross-job coordination, enabling interactive processing that can utilize and combine multiple jobs’ intermediate results.
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.

### **1.4 Dissertation Structure**

The rest of this dissertation is organized as follows. Chapter 2 outlines the technical limitations of general streaming processing systems, then introduces ELF’s clues. Chapter 3 presents ELF’s detailed design including the compressed buffer tree (CBT) component and shared reducer tree (SRT) component. Chapter 4 describes the implementation of ELF, and Chapter 5 describes the state management of ELF including

consistency, straggler mitigation and fault tolerance. Chapter 6 discusses related work, and Chapter 7 concludes.

## CHAPTER II

### STREAMING PROCESSING OVERVIEW

This chapter discusses several real world streaming processing applications, and then presents an example to illustrate how a general-purpose stream processing system can execute the sample application, together with a broader discussion regarding the optimizing opportunities found in the processing of the application. Following the example is a comparison of how ELF’s processing model executes the same application, and ELF’s goals, to help the reader follow the design proposed in subsequent chapters.

#### *2.1 Examples of Real world Streaming Applications*

A concrete scenario is a “micro-sale” promotion driven by online logging of popular items being viewed (e.g., if over 3000 customers have been viewing an item in the last 3 seconds, this item is tagged as popular). To raise total sales, a consequent micro-sale offering, e.g., an additional 20% discount will be placed on those popular items. In order for such promotions to be effective, however, “clicks” must be continuously logged analyzed, as well as “buys”. A failed promotion, for instance, is one in which the “clicks” rate increases significantly, but the “buys” rate improvement is too small.

Continuing this scenario, several streaming applications will be designed and executed, and they might have different processing requirements. Next, this section discusses four examples of streaming applications with specific processing requirements.

**Micro-sales.** Consider an application monitoring an Internet merchant’s click stream, to determine the current top- $k$  hot items in say, the PC-Games directory, for purposes



of running an online promotion. Each incoming stream log related to PC-Games is first intercepted and parsed into a key-value pair, in which the key is the item name and the value is the count of “clicks”. For ease of exposition, we use the following heuristic. After some pre-specified time interval (e.g., 5 secs.), the application counts the number of clicks per item and sorts them in descending order, based on the count value. The output is a list of  $\langle item, count \rangle$  pairs that reports the top- $k$  hot items in each interval. Such online analytics make it possible to initiate micro-promotions for the most popular items, say, those with at least 1,000 clicks in the current time interval. The key to carrying out the actions described above are (i) low delay in determining the top- $k$  items, since users will not likely linger on a web page for too long, and (ii) high throughput to capture a sufficiently large number of users in the given time window.

**Online tracking.** When running online applications like micro-sales, it is important to track their effectiveness. For example, when a 20% discount appears on a web page to increase sales of the product “*SimCity*”, the effectiveness of the promotion can be determined by tracking the “buys” and “profits” for *SimCity*, as the promotion is ongoing. The output is two continuously updated curves showing these variations. If “clicks” increases, but the “buys” curve’s positive slope is too small, the promotion is not effective. If both “clicks” and “buys” increase, but the “profits” curve’s slope is small, profit is too low due to an excessively discounted unit price. This application’s requirement is to dynamically deploy new queries like those required for online tracking, at any time deemed necessary by end users.

**Dynamic query adjustment.** Consider a typical log event with multiple attributes. For example, when a user clicks *SimCity* on the merchant’s web page, the log event records the timestamp, the user’s profile (e.g., Id, name, and age), the item’s profile (e.g., its Id, directory, and color). For such events, dynamic adjustment

zooms in on interesting attributes, where the definition of “interesting” (the corresponding filter predicate) varies with current context. For instance, to investigate an ineffective promotion, it may be useful to look at extended attributes of web requests, like those indicating customer ages, e.g., if 90% of the potential customers are over 30 with likely stable incomes, small discounts may not be effective, prompting alternative strategies. This application’s requirement is to dynamically change current queries and processing logic at runtime with low delay.

**Learning from the past.** It is important to be able to analyze and understand long term trends in the datasets being captured. For example, at the beginning of the *iPhone 4*’s release, customer reviews were negative in response to unresolved technical issues. However, as time progressed, overall comments showed the *iPhone 4* to be the most popular product in the iPhone series. This application’s requirement is to maintain considerable history information for the queries being run, with a query determining the top- $k$  hot items that learns from past experiences, by incrementally updating the historical results obtained from previous such computations. Gaining such time-dependent insights requires the application to be able to cache and update previous computational results, in a substantially sized set of key-value pairs.

## ***2.2 General-purpose Streaming Processing Model***

This section illustrates the general-purpose streaming processing model with a sample application code, and then explores the optimization opportunities found in the previous model.

**Application description.** Many stream processing activities can be run on the data generated in online e-commerce, including (i) *click* streams from users browsing the sites, (ii) *sales* streams from the checkout experience, and (iii) additional streams

concerning user activities like reviews and comments, coupled with (iv) streams providing information about internal activities like machine usage logs, real-time bot or fraud checking, and others. Processing results can be key to company operation: e.g., from (i), one can compute item popularity scores based on click rates and from (ii), item sales packages can be determined, to generate immediate “customers who bought this also bought that” recommendations. Stated more precisely, the following are some of the stream processing jobs supporting e-commerce activities:

1. Calculate items’ popularity scores based on users’ clicks, and continuously report a list of “hot” items with scores above some pre-specified threshold, for the past hour at 5-second intervals. A possible reaction is an extra 20% discount appearing on the web pages holding the hot items so as to further increase sales.
2. Compute the overlap between two streaming jobs’ datasets, to display “customers who visited this item ultimately bought these items together”, for purposes of product bundling, which provides customers with cost and time savings from a single purchase of complementary offerings, and also increase sales.

As shown in Figure 3 with declarative SQL-like code to describe those applications’ high-level pipeline structures, as also done in systems like Hive [78], Pig [65], and DryadLINQ [84].

Figure 4 depicts the alternative execution pipelines when running them in a Hadoop-like batch processing system and/or in a Storm-like stream processing system.

The sample program has three stages. The *computation stage* ( $s_1$ ) runs in parallel on a group of machines, e.g., MapReduce’s map or Storm’s spout. The *data-shuffling stage* ( $s_2$ ) aggregates  $s_1$ ’s outputs by transmitting requisite data between machines, e.g., MapReduce’s reduce or Storm’s bolt. Finally, the *feedback-coordination stage* ( $s_3$ ) receives  $s_2$ ’s outputs, back to the computational nodes to run user-specified functions,

```

1  s1 = EXTRACT item:string,clicks:long,price:long,...
2      FROM "/users/foo/stream_278293145"
3      USING DefaultJsonExtractor(event)
4      HAVING IsValidEvent(event) AND clicks != 0;
5  s2 = REDUCE s1 ON item
6      PRODUCE item, score
7      USING ScoreReducer("clicks");
8  s3 = PROCESS s2.above(threshold)
9      USING MicroSale
10     FEEDBACK job1
11     USING ProductBundling
12     COORDINATE job1, job2;

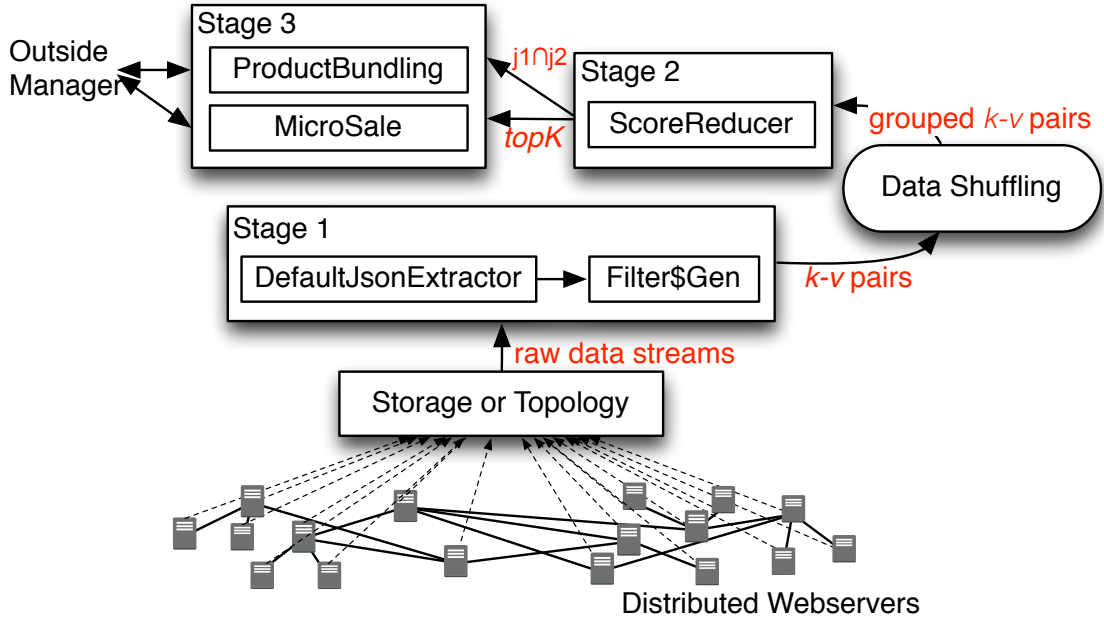
```

**Figure 3:** Declarative code that defines the pipeline of a sample microsale application. Stream events (line 1) are first extracted from a log file (line 2) using a Json extractor (line 3) and filtered based on certain conditions (line 4). Next, the input tuples are *reduced* with the user-defined function `ScoreReducer` (line 7) to produce a list of *key-value* pairs (line 6). Finally, the user-defined functions `MicroSale` (line 9) and `ProductBundling` (line 11) are executed.

and coordinates with other jobs. The pipeline is straightforward, but requires several optimizations to obtain low delay for pipeline output coupled with high pipeline throughput, described next.

First, `ScoreReducer` on line 7 of Figure 3 is actually a *groupby-aggregate* function that partitions users’ clicks into groups according to item name, then aggregates the clicks on each group. However, clicks can be reduced early by pushing the function `ScoreReducer` directly to the webserver. This decreases unnecessary network communications, because only the intermediate results must be transmitted during the *data shuffling stage*. ELF exploits this “locality” property via its direct linkage to the web server tier – the data collection infrastructure – so as to capture and mine live streams locally, rather than first moving data to remote machines for processing.

Second, `ScoreReducer` is an iterative function based on recent records, in the past hour in this case. Other streaming systems use in-memory data structures, e.g., Spark’s RDD [87] and Muppet’s slates [50], to hold historical records to avoid re-computation, but their choice of data structures limits the volumes of historical



**Figure 4:** General streaming processing model. Distributed streams from webservers are first moved to data storage, e.g., HDFS. Then, batch jobs are pipelined for offline, long-term data analysis. Streaming jobs are submitted to a computation graph or topology for online, continuous analysis.

records available to processing nodes. ELF uses a highly memory-efficient data structure – the *compressed buffer tree* – to store considerable volumes of historical data, thereby creating opportunities for rapidly and in realtime integrating “present” with “longer-term past” data.

Third, there is a need to change job functions at runtime, e.g., based on the current job’s outputs or on inputs from users. Further, there should be coordination across jobs to share insights. For example, **MicroSale** on line 9 of Figure 3 is actually a feedback function dispatching discounts to web pages that hold chosen items, based on continuous 5-second interval outputs from  $s_2$ . Similarly, **ProductBundling** on line 11 of Figure 3 requires message exchanges and dissemination between  $job_1$  and  $job_2$  at runtime. Other systems rely on outside managers to perform these functions. Since ELF already has set up topologies for per job “forward” stream processing, we simply reuse the same topologies to move new codes or messages “back” to wherever they

are needed, thereby dynamically controlling the streaming jobs being run.

Fourth, with numerous, possibly dynamically changing stream processing jobs running simultaneously, a single master tracking all  $n$  jobs' workers can be a bottleneck. ELF uses a decentralized architecture to avoid this problem.

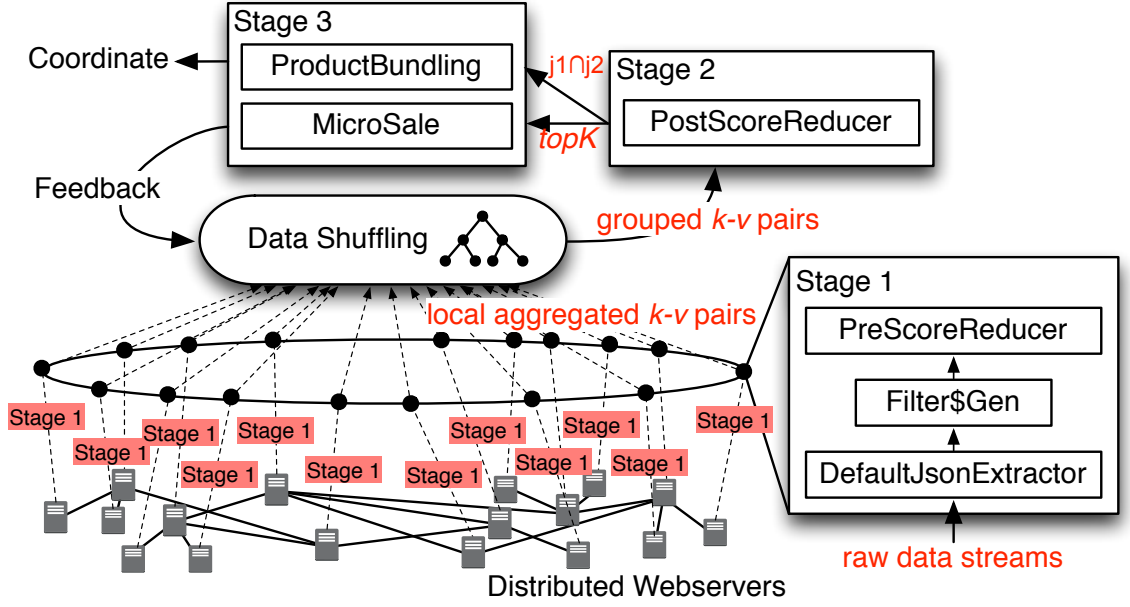
### ***2.3 ELF Streaming Processing Model***

We reiterate ELF's goals:

1. memory-efficient store and integrate with past data;
2. scalability to thousands of nodes and concurrent jobs, at second-scale latency for per stream processing;
3. support for feedback-coordination; and
4. interoperate transparently with batch and monitoring systems.

Figure 5 shows a high level sketch of ELF's computation model. Concerning 1., each webserver is associated with an agent to observe its live streams. The agent divides the input stream events into mini-batches with timestamps, parsing each event into a  $\langle item, click \rangle$  pair. Those batched  $\langle item, click \rangle$  pairs are stored and organized in the agent's memory as a memory-efficient compressed buffer tree (CBT) structure [17], which consumes  $2\times$  less memory than Google's sparsehash [4]. The CBT implements an  $(a, b)$  tree in which each level compresses part of the sorted historical data, enabling rapid merging with historical states when the tree is flushed.

Concerning 2., to obtain scalability to many stream processing jobs, agents are part of a distributed shared overlay. When a new job is submitted by the client, a DHT-based aggregation tree specified for that job, termed a shared reducer tree (SRT), is automatically built on the overlay. The SRT's root, also the job master, is the agent whose ID's hash value is closest to the jobID's hash value. Because different



**Figure 5:** ELF streaming processing model. The stream generated by each webserver is locally parsed, and stored in agent’s memory as immutable datasets for all intervals. Per-interval intermediate results are continuously reduced via a global DHT-based aggregation tree with embedded reduce functions.

jobIDs map to different agents, this also results in a balanced distribution of masters over agents. Low latency SRT operation proceeds as follows: the root publishes a multicast message to its tree members to synchronize all CBTs’ states to “flush”; this results in the stored datasets of CBTs to be repeatedly accessed and gradually reduced from the SRT tree’s leaves to its root. The current SRT implementation uses checksumming to detect errors in data transmission, giving rise to recovery actions that either leap ahead or recover the lost *key-value* pairs.

Concerning 3., a feedback control module makes it possible to change a job’s processing logic on-the-fly, and/or to publish application-specific messages like **MicroSale** in response to continuous outputs. Coordination APIs allow the job to communicate with other running jobs for cross-job coordination, e.g., for **ProductBundling**.

Finally, for 4., because ELF bypasses the log collection and storage tier, it can smoothly interoperate with batch processing systems, providing users with one-shot,

fast and instant answers about current stream state.



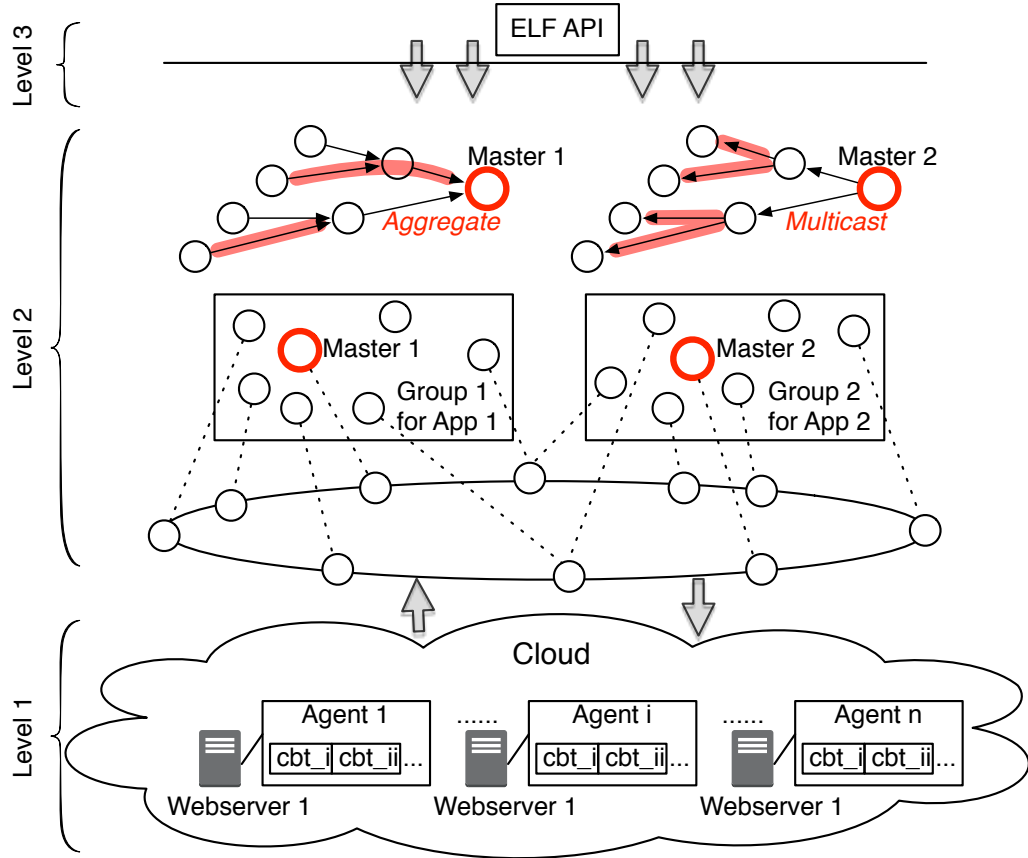
## CHAPTER III

### ELF DESIGN

In this chapter, we present the high-level overview of ELF and introduce ELF's components, with an illustrative example showing how ELF works and experimental results.

#### 3.1 Overview

As shown in Figure 6, 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



**Figure 6:** High-level overview of the ELF system.

each webserver producing data required by the application, there is an agent (see Figure 6 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 6 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  $O(\log N)$  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 6 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.

### 3.2 Compressed Buffer Trees (CBTs)

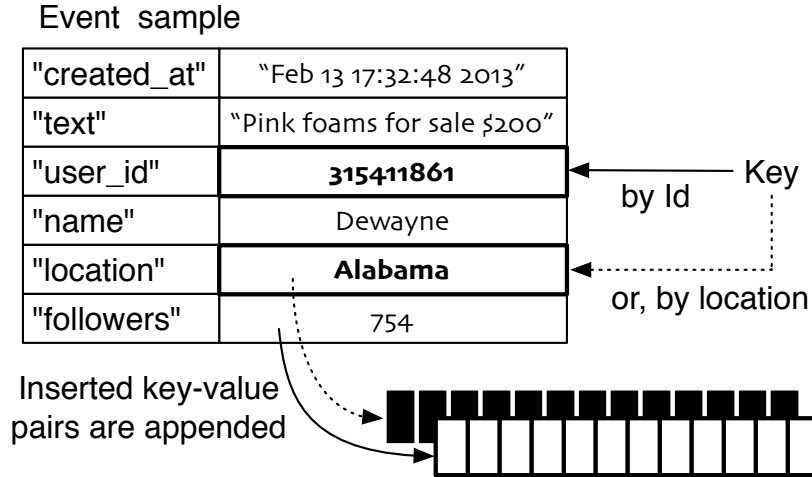
Existing streaming systems like Spark [87], Storm [6] typically consume data from distributed storage like HDFS or HBase, incurring cross-machine data movement. This means that data might be somewhat stale when it arrives at the streaming system. Further, for most realworld jobs, their ‘*map*’ tasks could be ‘*pre-reduced*’ locally on webserver 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 are injected into the streaming system directly from its sources, and ‘*pre-reduced*’ by ELF’s in-memory compressed buffer tree (CBT) data structure. This section discusses the incoming log events, ELF HiveQL-like query and the CBT abstraction.

#### 3.2.1 Log events

ELF agents residing in each webserver consume streaming log events to produce succinct *key-value* pairs. Streaming log events can be Twitter tweets, Facebook updates, Foursquare checkins, and others. They are continuously generated by web servers, processed by agents, and forwarded to final storage. For example, a typical Flume [12] event is a tuple  $\langle ts, src, pri, body \rangle$ , where:

- *ts* is the timestamp from the source machine;
- *src* is the IP address of the source machine;
- *pri* denotes event priority, such as trace, debug, info or error, which are often provided by logging systems like log4j;
- *body* is the log entry body, formatted as a map from a string attribute name (key) to an arbitrary array of bytes (value), where keys are used to group events, as done in MapReduce.



**Figure 7:** Example of a twitter event parsed into different key-value pairs and queued to be sent to CBTs for local pre-reducing.

Each ELF application has a unique key space. For example, as shown in Figure 7, if the event is a tweet, and the application is one that determines the user with the most followers, then the key is user.Id and the value is the followers' count. If the application is one that determines the location where users tweet most, then the key is the location and the value is the count of tweets from that location.

### 3.2.2 HiveQL-like query

Each agent exposes a simple HiveQL-like [78] query interface with which an application can define how to filter and parse live web logs. Figure 8 shows how the

#### 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 8:** Example of ELF QL query.

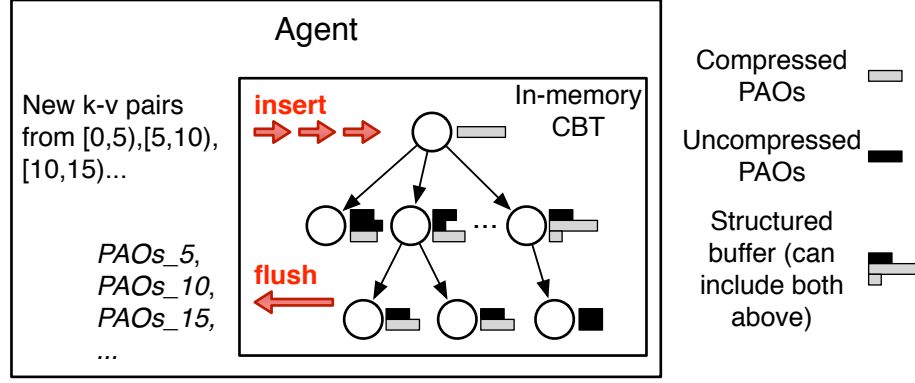
micro-promotion application uses ELF QL to define the top- $k$  function, which calculates the top-10 popular products that have the most clicks at each epoch (30  $s$ ), in the *Video Game* directory of the e-commerce site.

Every 30 seconds, the newly aggregated key-value pairs are stored in the *compressed buffer tree* (CBT)’s “cache” and combined with the previous window-size’s results. All of these distributed intermediate results are then aggregated globally by the SRT’s aggregation tree. Next, we describe the CBT abstraction.

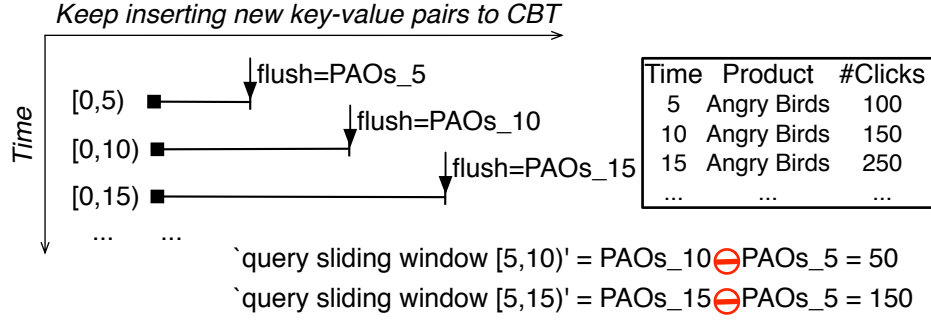
### 3.2.3 CBT abstraction

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) [17]. 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 I/O performance.

As shown in Figure 9(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 [17].



(a) Compressed buffer tree (CBT)



(b) Arbitrary sliding windows using CBT

**Figure 9:** Between intervals, new *key-value* pairs are inserted into the CBT; the root buffer is sorted and aggregated; the buffer is 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.

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 9(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 the next section.

#### 3.2.4 CBT benefits

**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.

### 3.3 *Shared Reducer Trees (SRTs)*

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, 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.

This section first describes the shared reducer tree (SRT)’s abstraction, then discusses how the SRT builds the pipeline computation model and cycles.

#### 3.3.1 SRT abstraction

The centralized infrastructure that traditional streaming systems use does not fit in well because: first, the central master is strictly coupled to the underlying physical node, which could easily be the performance bottleneck and valuable to failures; second, regardless of the performance issue, to guide the master flexibly switch between lots of applications, even the experienced programmer needs to pay a lot of efforts to deal with the mess like, how to fairly and correctly deliver records from concurrent applications and how to prioritizing or sequencing the message exchanges between them.

ELF leverages DHTs to create a novel ‘*many master*’ decentralized infrastructure. As shown in Figure 10, all agents are structured into a P2P overlay with DHT-based routings. Each agent has a unique, 128-bit `nodeId` in a circular `nodeId` space ranging



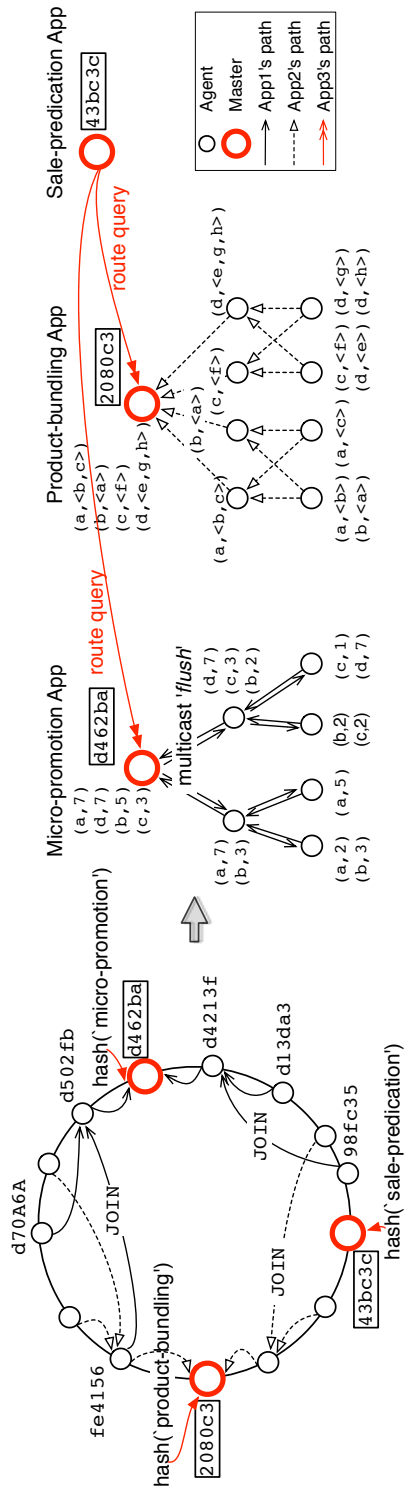


Figure 10: Shared Reducer Tree Construction for many jobs.

from 0 to  $2^{128}-1$ . The set of **nodeId**s 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 10 left: the agent parsing the application’s stream will route a JOIN message using **appld** as the key. The **appld** 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 **applds**. 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 **appld**. 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 **appld**) for each child. Note that the uniformly distributed **appld** 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.

### 3.3.2 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 10 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 10 right shows how the ‘*fat-tree*’-like aggregation tree is built for the product-bundling job.

### 3.3.3 Cycles

Naiad [62] 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.

### 3.3.4 SRT benefits

**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  $O(\log N)$  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.

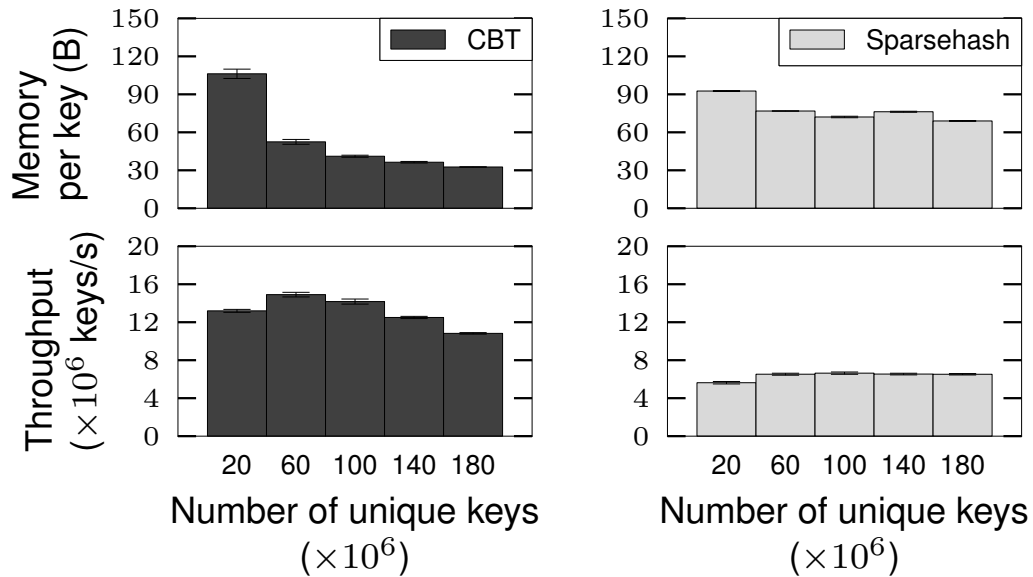
### 3.4 *Evaluation*

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. Experimental evaluations answer the following questions:

- What is the throughput and data shuffling time seen for ELF jobs?
- How does ELF scale with number of nodes and number of concurrent jobs?

#### 3.4.1 CBT throughput

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. We first report the per-node throughput of ELF in Figure 11.



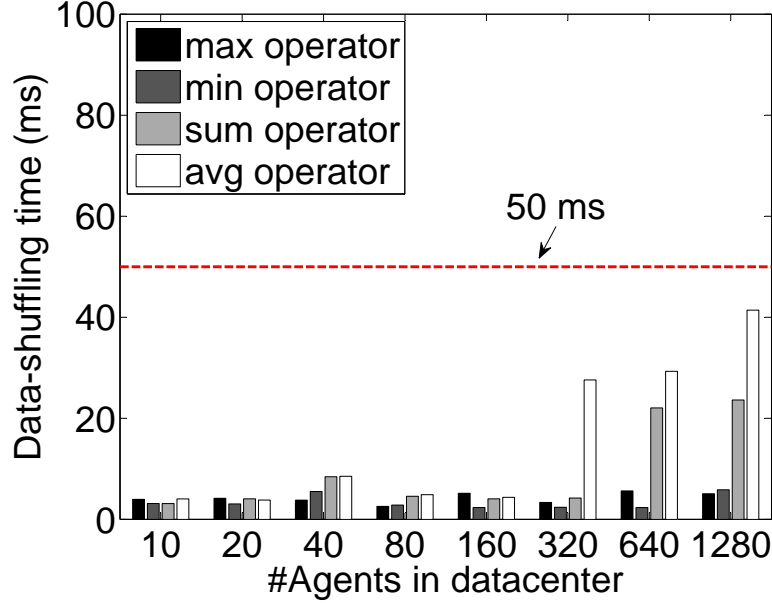
**Figure 11:** Comparison of CBT with Google Sparsehash.

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` [4]. 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 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.

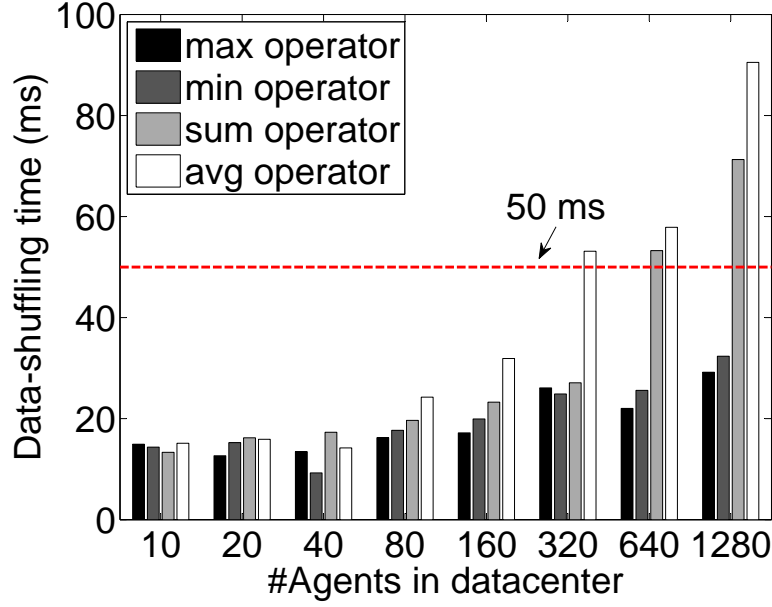
### 3.4.2 SRT data shuffling time



**Figure 12:** Performance evaluation of ELF on data-shuffling when running operators separately.

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 then report data shuffling times for different operators in Figure 12 and Figure 13.

Figure 12 and Figure 13 report 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



**Figure 13:** Performance evaluation of ELF on data-shuffling when running operators simultaneously.

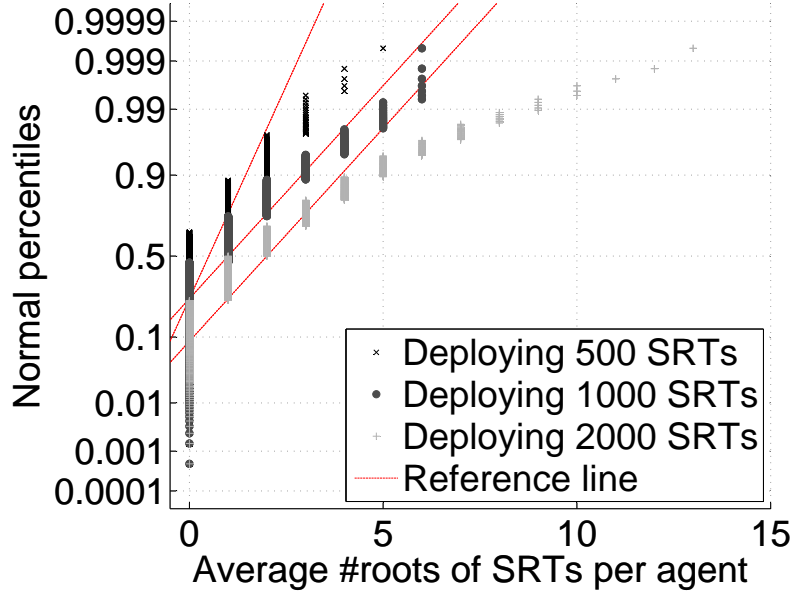
`avg` take more time than `max` and `min`.

Figure 12 and Figure 13 demonstrate 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 webserver, 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.

### 3.4.3 SRT load balance

The degree to which loads are balanced, an important ELF property when running a large number of concurrent streaming jobs, is reported in Figure 14.

Figure 14 shows the normal probability plot for the expected number of roots



**Figure 14:** Performance evaluation of ELF on load balance.

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.

### 3.5 *Limitations and Extensions*

While the previous section characterized scenarios in which ELF can support batch, stream, and/or query processing models efficiently, we also need to highlight cases that are beyond ELF’s design space. We now survey some of the key limitations for the current design and discuss several possible extensions to the model.



### 3.5.1 Latency

As explained before, the main performance metric in which ELF streaming processing system can fall behind the real-time processing system is latency. Here ‘latency’ refers to the time between the generation of a batch of events and the complete processing of that same batch of events.

In particular, there are two core issues for dealing with latency. CBTs by their nature buffer incoming events, and when a synchronization query causes a flush to occur, there will be some variation in flush times across all of the participating nodes, since their individual tree layout and buffer locations will depend on exactly what contents they have received. Additionally, because the SRTs are built using the randomized network layout from the DHT, there is no guarantee of low latency routing of the resulting messages. So for any “timestep”-driven use case, there is a potential for substantial jitter, or variation in latency, between different time steps due to the uncontrolled load balancing. The user can manage this effect at the application level by adjusting the synchronization interval so that it can account for reasonable amounts of jitter. However, it would be useful to extend the model to allow for better stream-lining of the CBT flush and the construction of optimized SRT networks for those applications that might depend on tighter synchronization.

Thus, ELF is not efficient when a user performs frequent fine-grained synchronizations on it, as the cost of repeated updates for each small interval may be high. For example, compared to Storm, ELF would not be a good fit for a time window of 1 second scale, because the CPU overhead of flushing CBTs every second, the traffic overhead of multicasting synchronization command through SRT branches every second, and the time cost of waiting and synchronizing every node’s state is not negligible. It would be possible for extending an API for users to bypass CBT and perform stateless computation with fine-grained updates. There are also ways for users to manually batch updates and synchronize them deterministically to achieve

better wide-scale performance.

### **3.5.2 Throughput**

As discussed earlier in this chapter, CBTs are designed to reside in memory and to make efficient use of memory by compressing groupby-aggregated key-value pairs. In other work, it has been shown that a CBT is capable of processing up to 600 million key-value pairs per second. This is a high water mark that we did not reach in our current system, and there would be additional research needed in order to construct ELF so that it could scale to that size. Additionally, some current MapReduce systems like Hadoop already handle processing rates above that, although not from raw streaming data sources. However, for most of the motivating enterprise application scenarios, you would not have a stream rate larger than the current CBT maximum on a per node basis, and the distributed scalability of ELF would allow one to deploy the system with a dynamic event router so that the stream could be handled by multiple spawned agents.

### **3.5.3 Communication patterns**

ELF has the potential to implement various execution flows, such as pipelines, cycles, point-to-point or any combination of the above. While these communication patterns lie in the application layer, real network layer may already have other more efficient primitives, like IP broadcast or in-network aggregation. These primitives are usually costly to emulate with just application layer's event message passing, especially for a large scale cluster, so extending ELF with direct support for network layer would help, much like Spark already contains an efficient "broadcast" operation that is implemented upon a variant of BitTorrent.

### 3.6 Discussion

We have presented ELF’s design including the *compressed buffer tree* (CBT) and the *shared reducer tree* (SRT). CBT is a *B*-tree-like data structure that enables high-throughput local aggregation of large in-memory append-and-aggregation datasets. SRT efficiently aggregates distributed data using tree-like data structures embedded into a peer-to-peer overlay shared by multiple queries, with load balancing between queries and also offering fast deployment for new queries being posed. Because of these two novel functionalities, ELF applications provide rapid answers to time-critical queries for distributed data arriving at high rate.

ELF targets at the aggregations in which the *key-value* pairs are gradually truncated while rolling up from CBTs, as leaves, to SRT’s root, e.g., **Grep**, **TopKCount**. For running applications like **WordCount** on a large dataset, if with few key repeats, ELF’s performance may experience bottlenecks, as ultimately all groups go to the same agent (root). It happens in practice and slows down Hadoop. A natural question is: ***How does ELF solve the similar bottleneck?***

We offer the following solutions:

1. **Fat SRT:** for each level of SRT, we automatically tune the number of internal agents of that level, so as to distribute keys to new added agents to share the reducing load and avoid agent-level bottlenecks. The SRT are therefore, transformed from a regular tree to a “**fat tree**”.
2. **Hierarchical SRT:** the large key space are divided and hashed to many SRTs, so each of them is only responsible for partial keys’ aggregation. The root of these SRTs are then constructed into an upper-layer aggregation tree for the final reduce, with the additional time cost for building this hierarchical tree is as low as milliseconds.

## CHAPTER IV

### ELF IMPLEMENTATION

ELF operates in three stages: (1) *data-preparation* uses the CBT for local aggregation; (2) *data-shuffling* uses the SRT for global aggregation; (3) *feedback-coordination* reverses the use of the SRT for online job changes and coordination.

This chapter describes ELF’s architecture and prototype implementation, including the implementation of CBT component and SRT component. Next, we discuss how to combine CBT with SRT to implement realworld streaming job’s feedback and coordination.

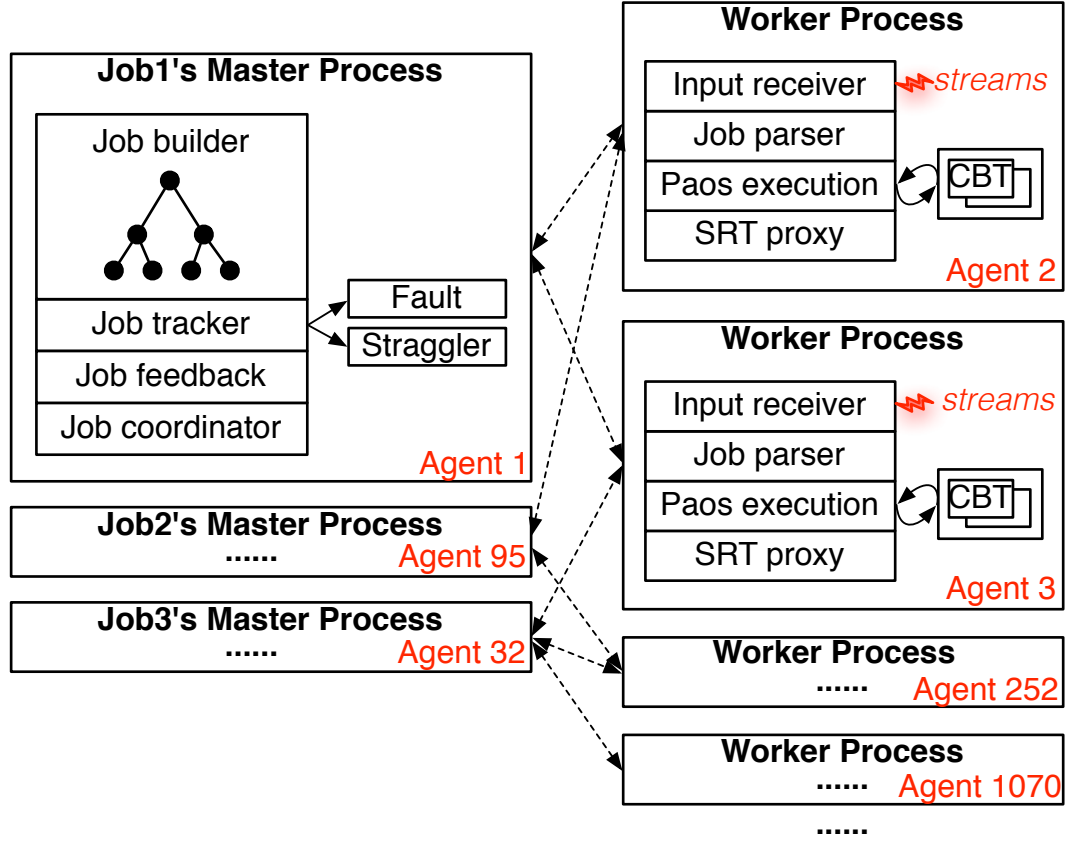
#### 4.1 *System Architecture*

Figure 15 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*, *job worker*.

##### 4.1.1 Job master

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.



**Figure 15:** Components of ELF.

- *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.

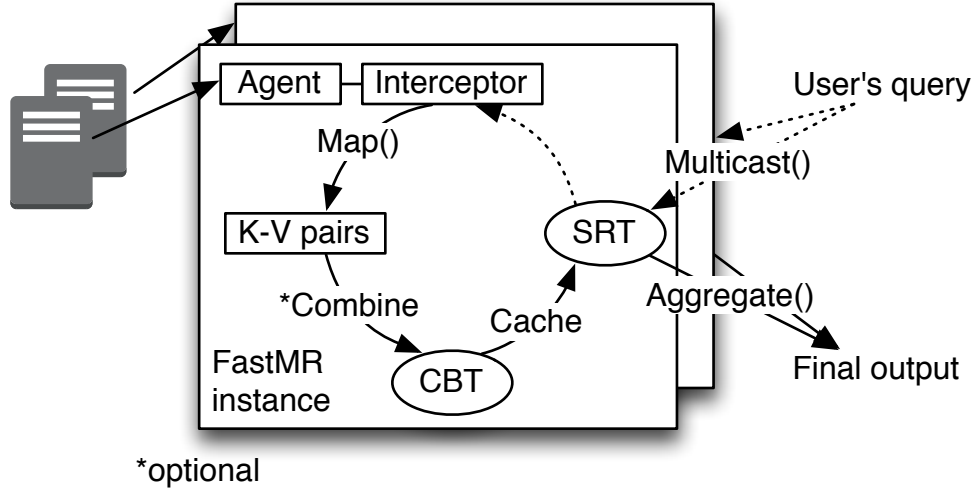
#### 4.1.2 Job worker

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 [12], 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 *f*, e.g., aggregations, grouping, and filters.
- *PAOs execution*: each *key-value* pair is represented as a partial aggregation object (PAO) [17]. 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.,  $\langle argu, 2, f : count() \rangle$  merges with  $\langle argu, 5, f : count() \rangle$  to be a PAO  $\langle agru, 7, f : count() \rangle$ .
- *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.

From an architectural point of view, the main difference between ELF and 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,



**Figure 16:** Workflow of ELF.

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.

#### 4.1.3 Workflow

The workflow of ELF is shown in Figure 16, which illustrates how ELF’s components work together.

As shown in the figure, local aggregation is performed by the CBT, which accepts key-value pairs from agents, with aggregated results stored in agent’s memory and made available to the SRT. These distributed intermediate results are globally aggregated via the SRT. The results of SRT-level distributed aggregation are available at the SRT’s root, but can be multicast to whoever needs it. The SRTs multicast service is also used to disseminate new queries to all members and adjust current queries, which is accomplished the feedback-coordination stage.

## 4.2 *CBT-based Agent APIs*

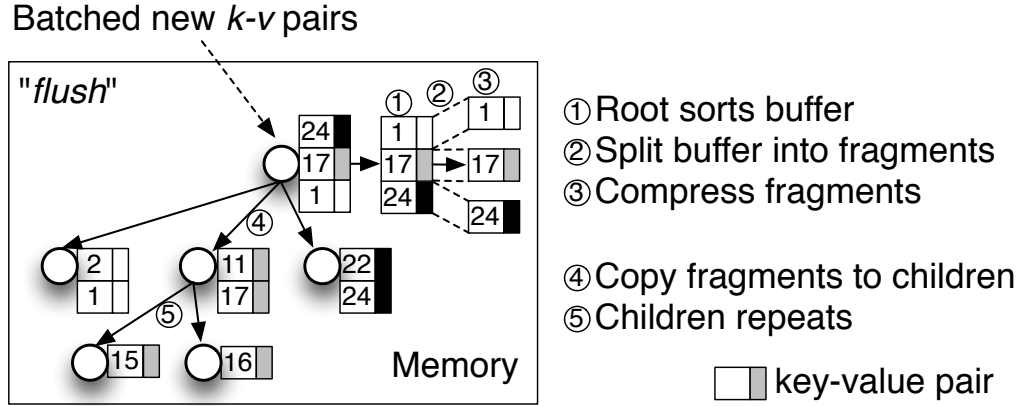
Reduce functions in MapReduce take as inputs some key  $k$  and the list  $L$  of all values associated with that key, then “reduce”  $L$  to emit new key-value pairs. CBT plays an analogous role, but operates only on local streams and over some time window. These restrictions are due to the fact that stream data arrives continuously and CBT memory is bounded (CBT can be externalized to disk as well, but our experimentation uses only in-memory CBT).

A common approach for implementing a group-by-aggregate operation is to use a hash-based aggregator: a hashtable stores one “accumulator” for each key, and intermediate key-value pairs are then hashed by key and accumulated. The aggregated key-value pairs are read iteratively from the hashtable and transferred to the reducers. CBTs reduce certain overheads of hash-based aggregation: (i) high memory overhead per hash table entry (i.e., the pointers to the key and the accumulator add 16B per entry; the accumulators add another 8B); and (ii) allocator overheads: the intermediate key, value, and accumulator objects are allocated from the heap, with each allocation involving the user-space memory allocator. CBTs avoid these overheads through effective use of buffering and compression. They operate as described next.

Consider using a binary search tree to maintain key-value pairs, 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. The CBT, therefore, first buffers key-value pairs and then adds entire buffers to each node in the search tree. Memory efficiency is obtained via compression as follows.

The entire CBT resides in memory. The root is uncompressed, but the buffers of all other nodes are compressed. When a buffer of a non-leaf node reaches some threshold (e.g., half its capacity), it is emptied into the buffers of nodes in the next level. To empty the buffer, the system decompresses the buffer (if not already decompressed),





**Figure 17:** Between intervals, new *key-value* pairs are inserted into the CBT; the root buffer is sorted and aggregated; the buffer is 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.

sorts the tuples by hash value, performs a linear aggregation pass to aggregate PAOs with the same key and partitions the tuples into those receiving buffers based on the hash value. When a leaf node is full it splits to form a new leaf node that is added as a new child to the parent. If the number of children of the parent exceeds the maximum allowed, the parent also splits possibly propagating splits up to the root (as in the case of a B-Tree[19]). This process is illustrated in Figure 17.

The highlights of this solution include the following. First, compression is effective as it operates across larger buffers. Second, buffers are uncompressed only when they are merged (and then re-compressed), again operating over larger data sets. Third, merging uses efficient merge-sort methods, since each buffer is maintained in sorted form. Fourth, when extending CBTs to operate on disk, buffer-based operations transform the original multiple, random I/Os of small updates into fewer, larger I/O operations.

As a result, the CBT offers fast, memory-efficient aggregation of intermediate key-value pairs, at levels of throughput comparable to competitive data structures like hashtables, but with reduced total memory requirements. Additional detail about

CBTs appears in [17].

### 4.3 *DHT-based SRT APIs*

ELF’s methods for aggregating large distributed datasets across potentially thousands of servers benefit from the self-organizing, resilient nature of its peer-to-peer based implementation. In this implementation, each SRT node is assigned as its address a random `nodeId` from a circular 128-bit node identifier space. Each node’s constant-sized routing table is updated based on new node joins and node removals. The overlay’s `route(nodeId, msg)` operation routes a message `msg` from the current node to the one closest to `nodeId` in the overall `nodeId` space. The routing mechanism ensures the delivery of a packet within  $O(\log_{2^b} N)$  hops for an  $N$ -node network, where  $b$  is typically 4. The routing table also enables the overlay to exploit network locality in the underlying network, i.e., the total delay by routing a message relative to the delay between source and destination in the underlying network, is below two [26]. Common examples of P2P overlay software are Chord [76], Pastry [72], Tapestry [90], and Bamboo [71]. They are all self-organizing, self-repairing, and resilient to failures, with many services (e.g., multicast, anycast, lookup) built on top of their functionalities. ELF exploits these functionalities, i.e., their aggregation, multicast, and anycast methods, to implement cluster-wide data aggregation, query deployment, and updates, explained in more detail next.

#### 4.3.1 Multicast

ELF uses Scribe [27], built on top of Pastry, to create the multicast tree for each streaming application. It also uses this functionality to disseminate a new query or update an existing one. Specifically, Scribe’s `subscribe(topicId)` and `publish(topicId, msg)` primitives (i) allow a node to join a multicast tree whose `groupId` equals to `topicId`, and (ii) make it possible to send a message `msg` to a group of nodes in this multicast tree. `topicId` is a number in the same 128-bit `nodeId` space; it can be

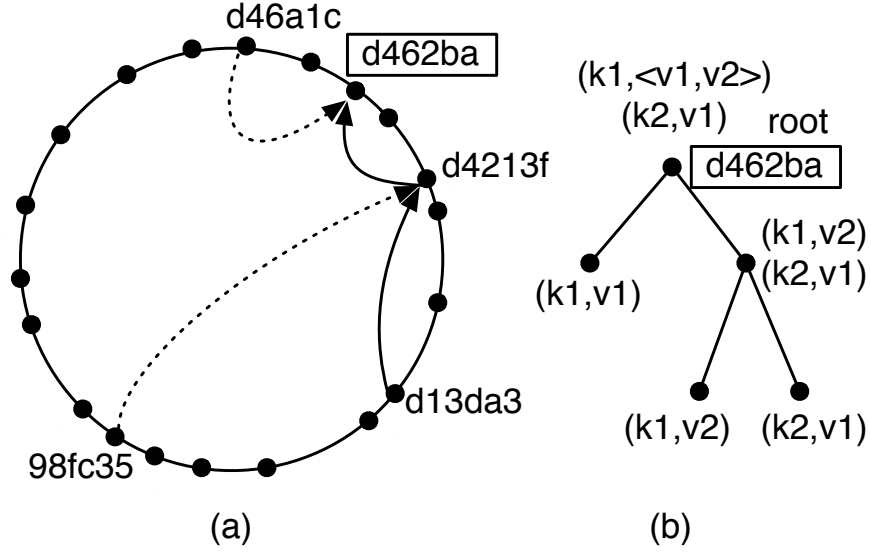
computed by using a hash function on the topic’s name: `hash(topicName)`.

Each ELF web data stream application has a unique `appName`: `appId`. Initially, all participating agents belonging to certain stream application subscribe to a common tree, by using `subscribe(hash(appName))`. The agent whose `nodeId` is numerically closest to `hash(appName)` becomes the root. When deploying a new query, the root encapsulates the query code into messages disseminated to all tree members. Additional multicast messages are used to start and stop queries. Using a multicast tree affords ELF deployments and updates to have small delays, as they follow a limited number of hops, benefiting from the overlay’s locality properties.

### 4.3.2 Aggregation

ELF’s implementation enhances Scribe’s multicast trees to also support aggregation functions. Specifically, when agents periodically send their updates for map results towards the root, all intermediate nodes in the path aggregate the datasets collected from their children, applying the aggregation functions along the entire path. Aggregation, therefore, occurs in  $O(\log_{2b} N)$  hops. This also means that aggregation functions have to be composable, such as `sum`, `maximum`, `minimum`, etc., which satisfy the hierarchical computation property [57]. For example, note that operations like “`average`” do not satisfy this property, so its intermediate results are stored as tuples  $\langle sum, count \rangle$ .

A single physical SRT can support a large number of groups with a wide range of group sizes, and a high rate of membership turnover. As shown in Figure 18, the key to such efficiency lies in the following: (1) The tree is the union of the paths from the group members to the root. When a member joins a group, it merely routes a join request towards the root using the overlay, adding overlay links to the tree as needed. If the join request reaches a node that is already a member of the tree, the request is not propagated any further [27]. (2) All stream applications and their logical



**Figure 18:** (a) d46a1c and 98fc35 join the tree by routing JOIN requests towards the root d462ba. The requests are received at d462ba and d4213f without going any further, adding the branches of the aggregation tree. (b) SRT's key-value pairs are reduced towards the root.

trees, therefore, share the same peer-to-peer overlay, resulting in the advantage that the overhead of maintaining that overlay are amortized over many group spanning trees. Such sharing permits multiple concurrent stream applications to run with low overhead.

#### 4.3.3 Anycast

The anycast service allows an outside node to send a message to a nearby member of another tree. Implemented using a distributed depth-first search of the group tree, the benefits derived from its implementation are that it completes after visiting some small number of nodes  $O(\log_{2^b} N)$ , and so that it has natural load balancing because anycasts from different senders emanate from different nodes.

ELF uses anycast to link different stream applications. An example concerning micro-sales is that when tracking hot products, one application may decide to offer discounts on *Apple MacBook Pros*, the other on the *Apple MacBook Air*. However,

since these are products from the same company and of similar style, such concurrent promotions likely compete for the same set of customers. Anycast methods linking the two different micro-sales and their SRTs can be used to correct issues like these. For instance, if the output from the “buys” tree of *Apple Mac Pro* shows profits to be even less than without the promotions (because customers are purchasing the *Apple Mac Air*), then Anycast methods linking these two trees can be used to automatically adjust their different discount rates, to pursue joint maximum profit. Note that the same principles apply to product bundling.

## 4.4 *User Query APIs*

This section describes ELF’s control plane APIs and data plane APIs, and illustrates how to implement a sample micro-promotion application using these APIs.

### 4.4.1 Data plane APIs and control plane APIs

Table 1 and Table 2 show the ELF’s data plane APIs 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. Programmers can use them to implement a variety of interesting applications for many advanced use cases.

### 4.4.2 Micro-promotion application example

A sample use case shown in Figure 19 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

**Table 1:** Data plane APIs.

<b>Subscribe(Id appid)</b>
Vertex sends JOIN message to construct SRT with the root's nodeid equals to appid.
<b>OnTimer()</b>
Callback. Invoked periodically. This handler has no return value. The master uses it for its periodic activities.
<b>SendTo(Id nodeid, PAO paos)</b>
Vertex sends the key-value pairs to the parent vertex with nodeid, resulting in a corresponding invocation of <b>OnRecv</b> .
<b>OnRecv(Id nodeid, PAO paos)</b>
Callback. Invoked when vertex receives serialized key-value pairs from the child vertex with nodeid.
<b>Multicast(Id appid, Message message)</b>
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.
<b>OnMulticast(Id appid, Message message)</b>
Callback. Invoked when vertex receives the multicast message from application's master.

**Table 2:** Control plane APIs.

<b>Route(Id appid, Message message)</b>
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.
<b>Deliver(Id appid, Message message)</b>
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.

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**.

```

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 19:** ELF implementation of micro-promotion application.

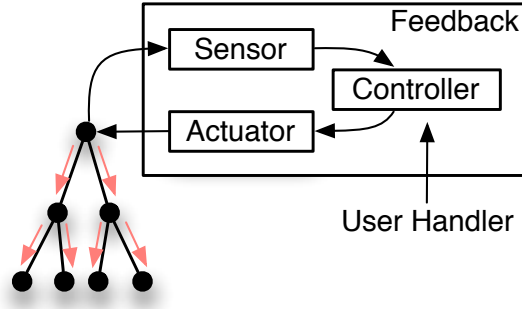
## 4.5 *Feedback and Coordination*

For streaming jobs that proceed over evolving streams for extended amounts of time, it is natural for users to wish to change some functions during ongoing runs and coordinate with other jobs.

### 4.5.1 Feedback action for MicroSale

ELF supports feedback in a fashion similar to what is done in “closed loop” feedback control, by reusing the reverse path of the SRT to dispatch messages and new job-specified functions.

This is supported with three abstractions, as shown in Figure 20. The “sensor” interprets the job master’s output and gives it to the “controller”. The “controller” prepares for commands via a handler registered by the user, or uses some user-predefined set points, e.g., the microsale job in Sec.2.2 prepares different discounts for different



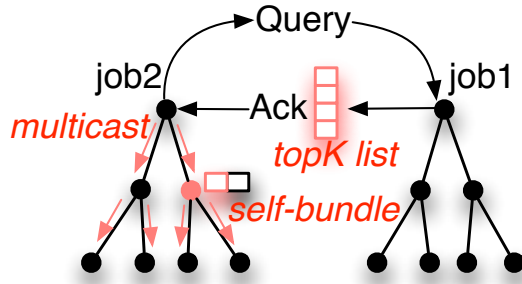
#### MicroSale:

```

sensor(jobID, key, value)
//fetch k-v pairs from root of SRT
controller(jobID, handler)
//register user-specified handler
actuator(jobID)
//generate discounts to nodes

```

Figure 20: Feedback abstraction for **MicroSale**.



#### ProductBundling:

```

job2.anycast(job1, topK)
//job2 query job1's topK list
job2.multicast(job1, list)
//multicast job1's topK list to members
job2.selfbundle(item)
//agent decides by itself which to bundle

```

Figure 21: Coordination abstraction for **ProductBundling**.

popularity scores. The “actuator” generates new functions or messages for distribution to webservers (via **multicast**).

#### 4.5.2 Coordination action for ProductBundling

An additional **anycast** abstraction is provided for a job to query other jobs’ internal states. By using anycast, queries from a single sender of one job can be routed to the topologically nearest agent in another job of potential receivers. The feedback control module, together with anycast that exchanges shared insights between jobs, can implement many complex cross-job algorithms.

Consider the microsale job in Sec.2.2 as an example. The application wants to group, “the hot products that most customers visited were ultimately bought together with ...”, as a package deal and sell the bundle at a discount. For example, you might



buy a hot laptop and get a bundle deal with the monitor, printer, cables and antivirus software. However, product bundling cannot simply bundle “the frequently browsed together items” or “the frequently sold together items” due to the single dataset’s inaccuracy. Moreover, as customers’ interests are subject to frequent change, it is not appropriate to continue to use the same bundle for some long time duration.

To implement such functionality, other systems rely on some outside manager to centrally calculate the best match while retaining all information from all sites’ servers. ELF’s simpler method, shown in Figure 21, has  $job_2$  from the *Sales* stream **anycast** a query towards  $job_1$  from the *Click* stream for the hot top- $k$  item list. Upon receiving the *ack* message together with the list,  $job_2$  publishes the list to all of its servers. Each server can then independently check whether its product package overlaps with the listed items. If *true*, the webserver notifies  $job_2$ ’s master for subsequent bundling.

## 4.6 Evaluation

The ELF design is evaluated with an online social network (OSN) monitoring application. Experimental evaluations answer the following questions:

- What performance and functionality benefits does ELF provide for realistic streaming applications?
- What is the overheads of ELF in terms of CPU, memory, and network load?

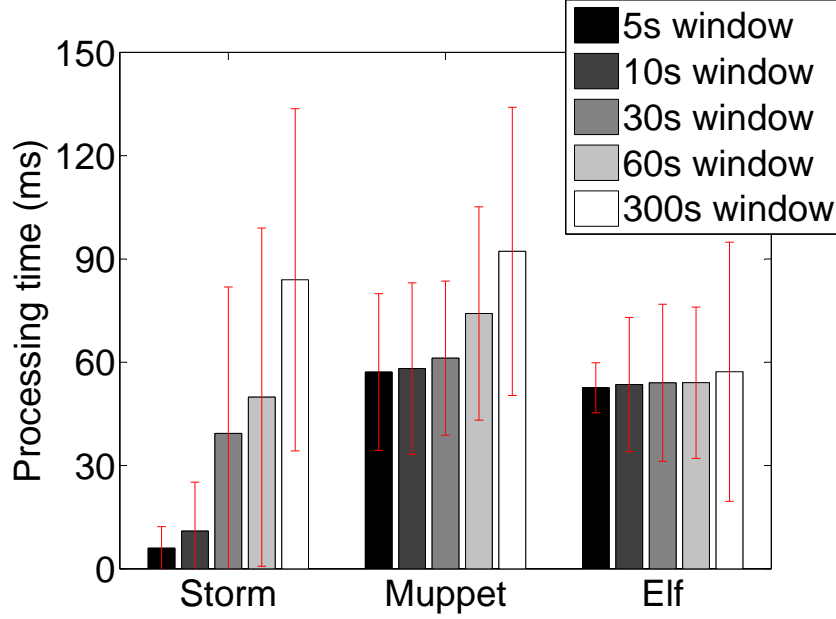
### 4.6.1 Testbed and application scenarios

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 [38]. 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 [10], 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 webserver to ELF and to HBase. For the web tier, there are 1280 emulated webserver 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.



**Figure 22:** Comparison of processing times of ELF on the Twitter application.

#### 4.6.2 Application performance

With ad-hoc queries sent from a shell via ZeroMQ [11], Figure 22 shows the time required to deliver updated results for ELF in comparison with MapReduce, Muppet, and Storm, for increasing observation window sizes, in a 1280 agent configuration. Each agent generates 50 streaming events per second, which means that 64,000 new lines of streaming log data are written into HBase per second, from which updated results are pushed every 5 seconds. Results show that ELF consistently outperforms Muppet, and ELF achieves performance superior to Storm when window sizes are enough large, e.g., 300 s.

These results are not surprising, as ELF is specifically designed for stream applications, by using CBTs for high throughput local aggregation and avoiding unnecessary computations by caching previous computation results. ELF outperforms Muppet and Storm, which target at the same class of applications. For Muppet, results have to be pulled from slates via a web front end, whereas ELF’s results directly appear at SRT’s tree roots. If we instead, measure when results appear in Muppet slates,

the difference in performance between ELF and Muppet is negligible. Compared to Storm, the CBT’s data structure stabilizes ‘*flush*’ cost by organizing the compressed historical records in an  $(a, b)$  tree, enabling fast merging with large numbers of past records.

### 4.6.3 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 3 and Figure 23 24 present these costs, explained next.

**Table 3:** Runtime overheads of ELF vs. others.

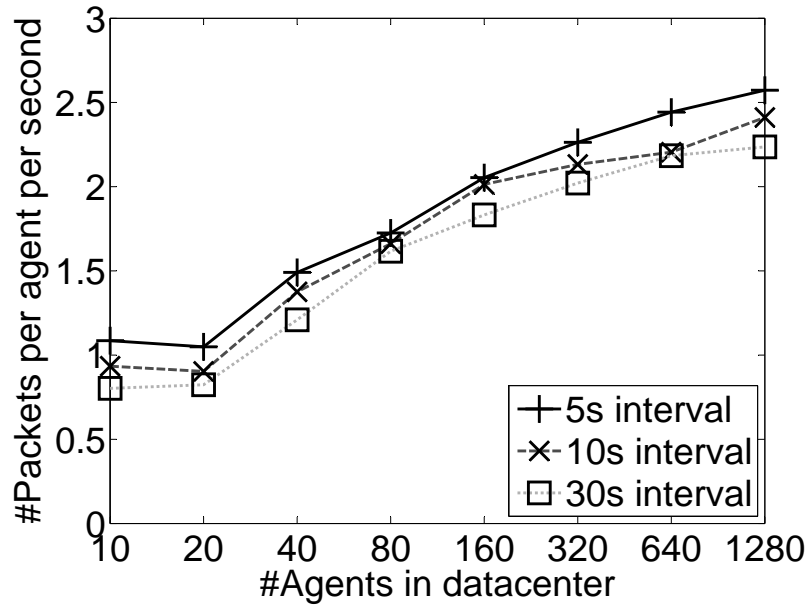
SPS	CPU	Memory	I/O	C-switch
	%used	%used	wtps	cswsh/s
ELF	<b>2.96%</b>	<b>5.73%</b>	3.39	780.44
Flume	0.14%	5.48%	2.84	259.23
S-master	0.06%	9.63%	2.96	652.22
S-worker	1.17%	15.91%	11.47	11198.96

*SPS*: stream processing system.

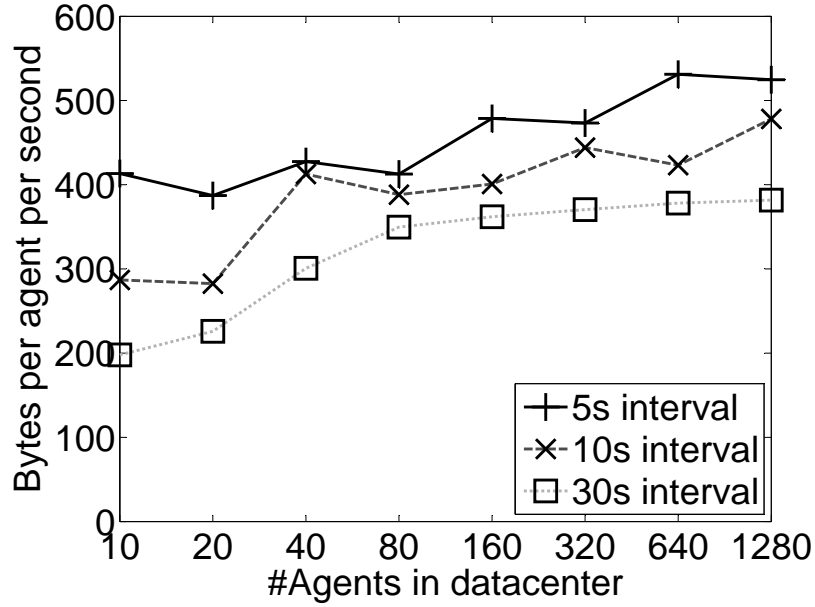
*wtps*: write transactions per second.

*cswsh/s*: context switches per second.

**Runtime overheads.** Table 3 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 3 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.



**Figure 23:** Additional #packets overhead of ELF with different update intervals.



**Figure 24:** Additional bytes overhead of ELF with different update intervals.

Compared to Flume, which only collects and aggregates streams, ELF offers the additional functionality of providing fast, general stream processing along with per-job management mechanisms.

**Network overheads.** Figure 23 and Figure 24 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 23 that when scaling to millions of agents, the additional #package is still bounded to 10.

## 4.7 Discussion

The majority of the streaming processing systems have three architecture components: shuffle, execution model and caching. The shuffle component is responsible for exchanging intermediate data between two computation stages. For MapReduce, there are two stages: map and reduce. For Spark, there may be many stages built at shuffle dependencies. The execution model component determines how user defined functions are translated into a physical execution plan. The caching component allows reuse of intermediate data across multiple stages.

ELF’s CBT is related to the caching component. CBT outperforms in (1) *locality* by pushing the batch computation down to the end webserver; (2) *memory efficiency* by batching data into small timestamps and saves as in local memory; (3) *better write performance* by adopting the buffer tree because lazy aggregation requires fast writes and ordering, but not low-latency reads.

ELF’s SRT is related to the shuffle component. The shuffle component is often the bottleneck of the scalability of a streaming processing framework. For example, in MapReduce, big data is shuffled between the map stage and the reduce stage for bulk synchronization. The sort operation may be executed during the shuffle stage,

which is usually required to handle very large data that does not fit in main memory, same as the aggregation and combine operation. SRT outperforms in (1) *scalability* by assigning each job a separate master, to improve the reliability of streaming systems, particularly when there are numerous concurrent jobs; (2) *load balancing* by decomposing one master’s load into  $n$  masters for  $n$  jobs running across the network.

Performance evaluations at the scale of thousands of agents demonstrate CBT’s superior throughput for local pre-reduce, by up to 1.5x Spark Streaming’s reported best throughput (640,000 records/s) for **Grep**, **WordCount**, and **TopKCount** when running on 4-core nodes. We also demonstrate low performance interference between concurrent SRTs’ operators and over 95% probability for load balancing for 2000 concurrent jobs.

## CHAPTER V

### PROCESSING ELF AT SCALE

Much of “big data” is generated in real time and is most valuable at its time of generation. For example, a social network may want to identify trending conversation topics within minutes, an ad provider may want to train a predictor model for user ad clicks, or a service operator may want to mine log files to detect failures within seconds [88]. To enable these low-latency stream processing applications, it is important for their streaming computation models to be able to scale transparently to large clusters, so as to handle the volumes of data and computation they involve.

However, processing streaming computation models at large scale is challenging. Distributed computing failures may happen due to processing node failures, network disconnections, software bugs, and resource limitations. Even though the individual components have a relatively small probability of failure, when large set of such components are working together the probability of one component failing at a given time is non-negligible and in practice failure is the norm rather than the exception.

At this scale, there are two major problems, **faults** and **stragglers** (slow nodes), which are inevitable in large clusters running “big data” applications. The previous chapters covered the ELF design and implementation including the CBT component and the SRT component, and several applications of ELF to execute specialized streaming computation models such as pipeline and feedback cycles. Nonetheless, the question remains: *How to handle transmission faults and agent failures, transient or permanent, and recover from fault while maintaining the states? How to mitigate straggler, including to deal with transient slowdown?*



In this chapter, we study these questions by exploring ELF at scale. First, we describe the failure model of ELF and our assumptions. Second, we show that ELF can *effectively* checkpoint and replay each CBT agent’s state, and recover transient or permanent failures from transmission faults and agent failures by making the use of the shortcut between SRT’s branch parent and children. Third, we show that ELF can mitigate straggler and deal with transient slowdown by deliberately leaping stragglers, so as to maintain low end-to-end latency for time-critical streaming jobs.

## 5.1 *Failure Model*

ELF has two types of operators, CBT-based ‘map’ operator and SRT-based ‘reduce’ operator. CBT-based ‘map’ operator performs local computations, such as filter, groupby-aggregate, sort, on raw data events as they are generated from associated data sources, and saved them as intermediate results in CBT in memory. SRT-based ‘reduce’ operator performs global computations, such as aggregate, join. We consider the following fundamental operators:

1. **Filter:** filters each input tuple against a predicate.
2. **Transform:** transforms each input tuple into another output tuple.
3. **Groupby-aggregate:** computes aggregate functions over windows of tuples that slide with time (possibly grouping the data first and sort).
4. **Join:** joins tuples on streams when these tuples fall within some time window of each other.

An operator  $O_i$  in the operator network, can be parallelized to several instances  $o_j^i$  where  $j$  denotes the operator instance id. Formally, an operator  $O_i$ , is modelled as

$$O_i = \{o_1^i, o_2^i, \dots, o_n^i\}, \text{ where } n \in N^+$$

Where  $n$  describes the degree of parallelization of the operator  $O_i$ . In our system,  $n$  is the number of leaves of ELF’s SRT. A cluster has a set of nodes  $N = \{n_1, \dots, n_n\}$ . Every node  $n$  processes a non-overlapping subset of data sources.

Our approach handles stragglers (slower  $O_i$ ), transmission error and fail-stop failures (e.g., software crashes) of processing operators (failure  $O_i$ ), network failures, and network partitions where any subset of nodes lose connectivity to one another. We consider long delays as network failures.

## 5.2 *Synchronization Assumptions*

Operators perform their computations over windows of tuples. For example, an aggregate operator may compute the average datacenter utilization *every hour*. Some operators, such as *Join*, still block when some of their input streams are missing. 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 [87]. Therefore, the failure of a data source prevents the system from processing the remaining streams.

We assume that data sources have loosely synchronized time clocks, and each tuple is associated with a timestamp when they are pushed into the system. When two or more streams are joined, or otherwise combined by SRT, ELF delays tuples until timestamps match on all incoming streams. Therefore, we make the following assumption: The clocks at data sources must therefore be sufficiently synchronized to ensure these buffering delays are smaller than the maximum incremental processing latency,  $X$ , specified by the application [22].

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



### 5.3 *Challenges of Fault and Straggler Tolerance*

The previous work for fault and straggler tolerance can be divided into three categories: upstream backup, replication, gap recovery.

**Upstream backup.** The upstream node retains a copy of the data events that it sent since last checkpoint. If a node fails, a standby node immediately takes over its role, and then the upstream node replays the message to the standby node to rebuild the state. When the data event is sent out safely, the upstream node will be notified and then the upstream node can delete the data events. Many modern large-scale stream processing systems based on message queuing use this approach. The challenge of this approach is that it incurs high recovery latency and they typically rely on the user’s code to manage the recovery of state. For example, Storm’s Trident layer automatically keeps state in a replicated database instead, committing updates in batches. While this simplifies the programming interface, it increases the base processing cost, by requiring updates to be replicated transactionally across the network [87].

**Replication.** In replication, there are two copies of the processing components, that is, the primary node and secondary node, and these two nodes run in parallel. Input records are sent to both. The primary node’s output is connected to the downstream nodes and the backup node’s output is not connected. Once a failure happens, the secondary node takes over the operation and connects to the downstream nodes. The replication approach incurs low recovery latency. However, the biggest challenge is the synchronization between the primary node and secondary node. The primary node and the secondary node both process data on its own speeds. Simply replicating the nodes is not enough; the system also needs to run a synchronization protocol to ensure that the primary node and the secondary node see the data events in the same

order. Replication is thus costly, though it recovers quickly from failures.

**Gap recovery.** In gap recovery, the loss of data event is expected, and it provides the weakest guaranteed processing. When a node fails, another node is chosen to take over the operations of the failed task. The new task starts from the last checkpoint, or an empty state, and starts processing the inputs directly to it by the upstream processing task. Therefore, the challenge is that certain tuples can be lost during the recovery phase because the new task starts from an empty state.

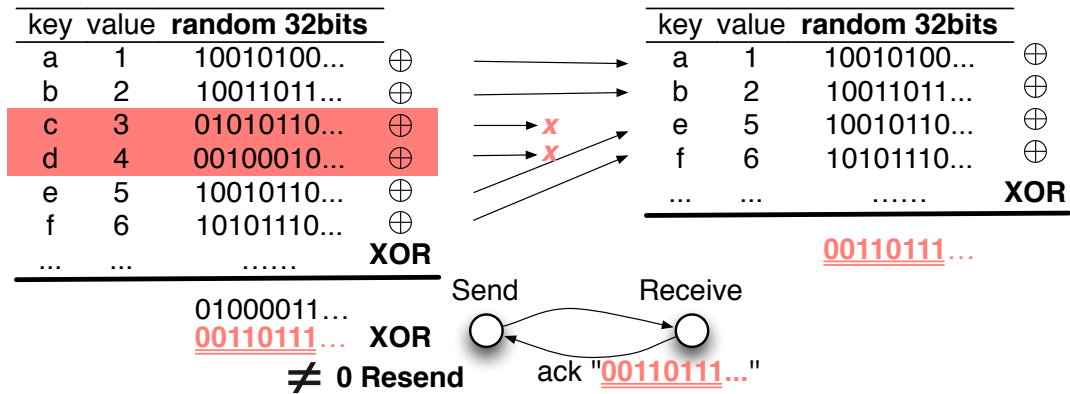
## 5.4 Fault Recovery

ELF handles transmission faults and agent failures, transient or permanent. ELF's fault recovery consists of detecting point to point transmission error and CBT state checkpointing and replay.

### 5.4.1 Detecting key-value errors

Millions of records are shuffled per second in ELF. Because traffic noise and other interference is inevitable in large-scale systems, it is important to guarantee that each record emitted by an agent is correctly transferred to the receiving agent.

ELF's implementation ensures record correctness using a fixed 8 bytes per agent,



**Figure 26:** Example of using XOR to detect key-value transmission errors.

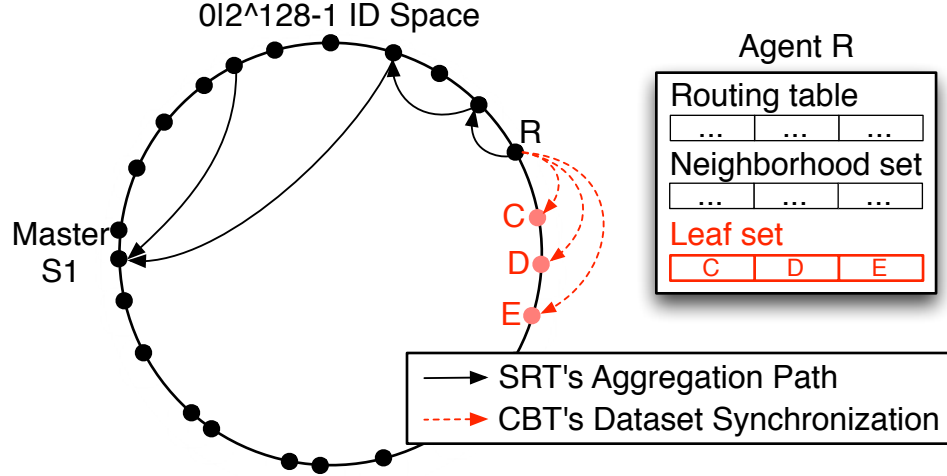
with one *ack* per interval, to detect any *key-value* error. Specifically, when a *key-value* pair is sent, it is given a random 64 bit id. At each interval, the sending agent “xors” all *key-value* pairs successfully sent and keeps the result, e.g., 01000011... The receiving agent also “xors” all *key-value* pairs that has been received, and then *acks* the final result, e.g., it should be the same as 01000011... If the sent value “xor” *ack* value equals to 0, then it means all *key-value* pairs have been correctly transferred, otherwise not. Figure 26 illustrates the above process. We note that Storm uses a similar approach to track an entire topology, whereas ELF uses it for point to point reliability. The approach ensures chances of an “ack val” accidentally becoming 0 is extremely small, e.g., at  $10K$  *acks* per second, it will take 50,000,000 years until a mistake is made.

#### 5.4.2 State checkpointing and replay

A streaming processing system is expected to be able to “checkpointing-replay” parts of a stream, reproducing the results of a user application. It is important for several reasons, most notably for recovering from failures, which start to happen frequently when scaling systems to large clusters. For infinite streams, the system needs to retain a certain history of the stream and coordinate what to retain, for how long, and what to replay.

In ELF, When an agent fails, in circumstance of permanent failure or longer than the job can suffer, the data that were on the node and all tasks the agent were currently running, would be regarded as lost. ELF can be extended to allow the dataset cached in the agent’s CBT, and all tasks to be recomputed in parallel on other agents.

As shown in Figure 27, each agent in the overlay maintains a routing table, a neighborhood set, and a leaf set [72]. The leaf set contains the agentIds hashed from the webservers’ IP addresses that are children of the local agent. The job master

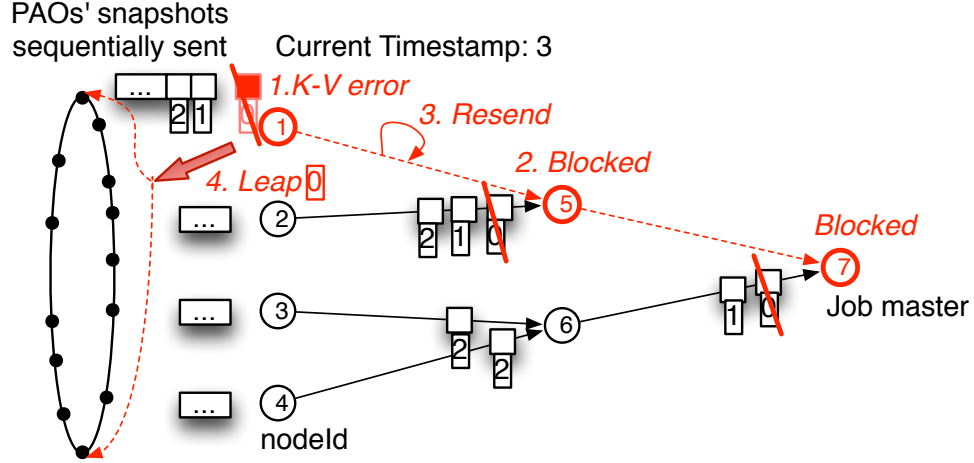


**Figure 27:**  $R$ 's datasets are replicated to  $C$ ,  $D$  and  $E$  at each checkpoint. If  $R$  fails,  $C$ ,  $D$ ,  $E$  rebuild  $R$ 's datasets from the last checkpoint.  $C$  takes over  $R$ 's tasks.

periodically checkpoints each agent's snapshots in the CBT, by asynchronously replicating them to the agent's children. For example, for a job computing a running click of items, the job master  $S_1$  could choose to checkpoint clicks every 5 minutes. Agent  $R$  partitions the CBT's snapshots into three parts and replicates to her children  $C$ ,  $D$  and  $E$  every 5 minutes. Keep-alive messages exchanged between children agents and parent provide the basis of early detection, with unresponsive agents presumed failed. Upon failure detection by any of the agent's children, recomputation of the data of the missing CBT uses one of the CBT's mirrors (e.g.,  $C$  takes over  $R$ 's task after  $R$ 's failure); the job master is notified about the failure and recomputes the task from the last checkpoint by multicasting a synchronization message to all workers.

## 5.5 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,



**Figure 28:** Example of the *leaping straggler* approach.  $Agent_1$  notifies all members to discard  $snapshot_0$ .

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 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 28,  $agent_1$  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 a multicast message indicating ‘empty’ command, 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.

## 5.6 *Evaluation*

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. The 1280 agents emulated web servers to generate Twitter streams at a rate of 50 *events/s* each. Those streams are directly intercepted, filtered, and groupby-aggregated by ELF agents. Meanwhile, unchanged data stream logs are gathered by Flume to be moved to the HBase store. Having both live and historical data helps us understand and check the accuracy of ELF’s fault recovery strategy.

Experimental evaluations answer the following questions:

- How fast can ELF recover from faults?
- How fast can ELF recover from stragglers?

### 5.6.1 Error detecting time

The time required for completing fault recovery or straggler leaping is directly affected by the detecting time of error node, and internally by the horizontal scale of the system being run, represented by ELF SRTs. Figure 29 shows how error detecting times change with horizontal scaling – for different numbers of participating agents, where the right Y-axis represents the depth of the shared reducer tree (SRT) spanning the datacenter. As evident from the figure, SRT scales well with the size of datacenter, as detecting times increase linearly with the depth of SRT, for exponentially increasing numbers of agents. This is because time increases are strictly governed by the SRT’s depth –  $O(\log_{24} N)$  as it needs to traverse the tree to inform the master about faults, where  $N$  is the number of agents in the datacenter. Note, however, that these are measurements taken on only 60 server blades total, so there

will likely be additional networking delays observed in datacenter configurations with larger numbers of servers (and less cores per server node).

### 5.6.2 Checkpointing and replay load balancing

In conventional fault tolerance systems, a relatively small number of backup nodes carry all the load of checkpointing and replay. This poses a problem when there are many concurrent running streaming jobs since the small set of backup nodes may not have the extra capacity and availability for many jobs concurrently.

ELF overcomes the inherently unbalanced checkpointing and replay load in conventional fault tolerance systems. ELF's randomization properties ensure that the tree is well balanced and that the forwarding/aggregating interior nodes that perform checkpointing and replay operations are evenly balanced over the nodes.

We create 5 SRTs, 10 SRTs and 15 SRTs in a 256 nodes simulated data center, respectively. Figure 30, Figure 31, Figure 32 list the accumulative number of being

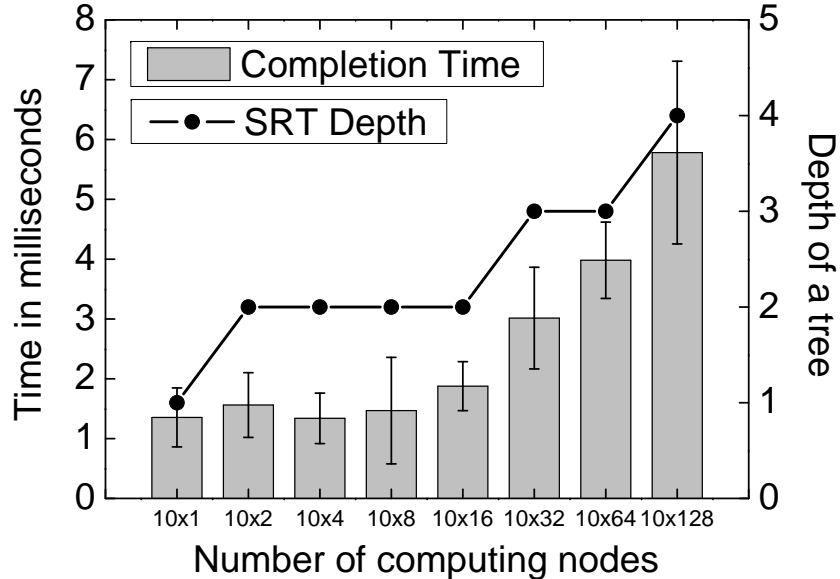
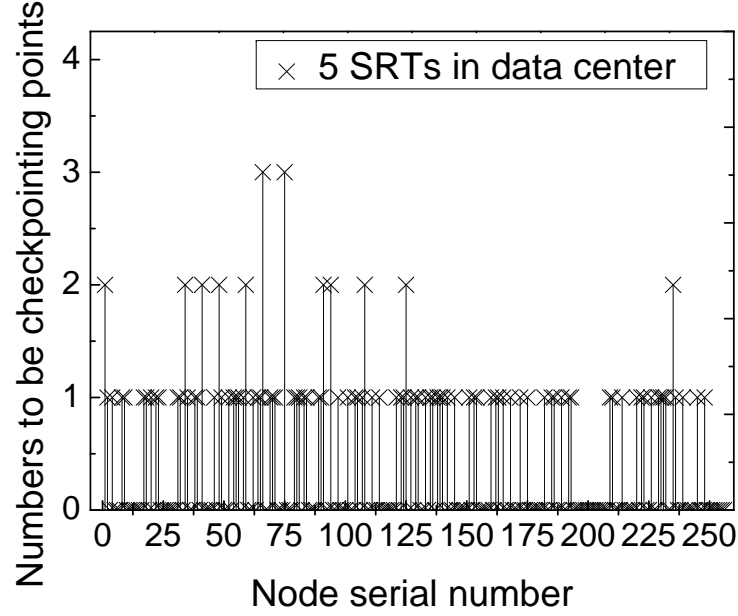
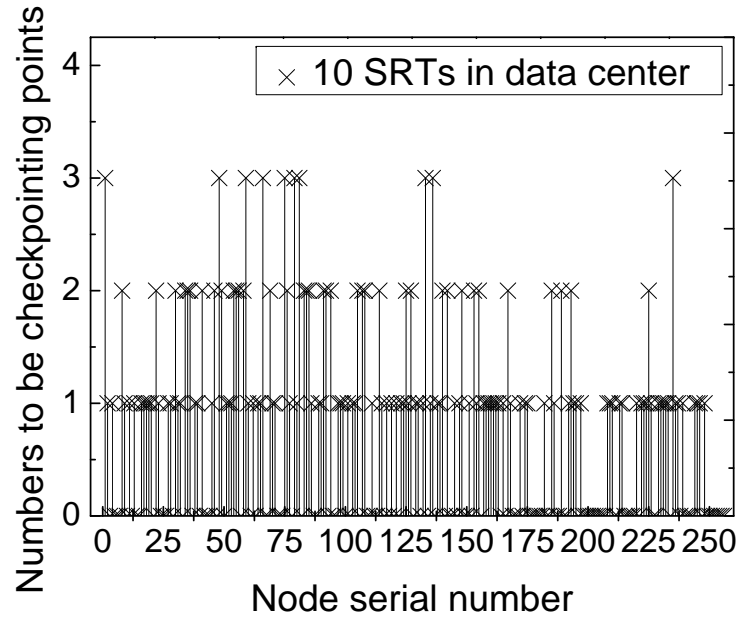


Figure 29: Error detecting time.

interior nodes that perform checkpointing and replay for different SRTs, where X-axis denotes host serial number and Y-axis denotes the accumulative numbers to be interior nodes for different SRTs. Each SRT is responsible for one stream processing



**Figure 30:** Checkpointing and replay load balancing for 5 SRTs.

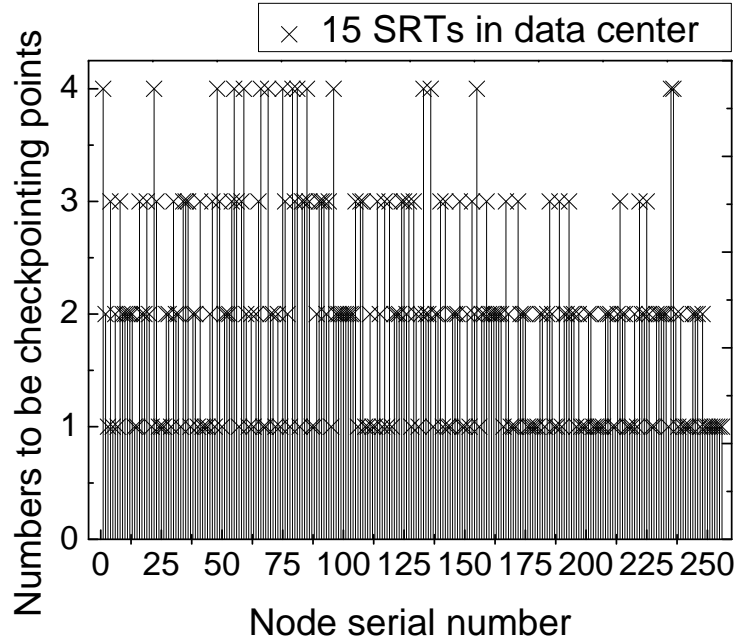


**Figure 31:** Checkpointing and replay load balancing for 10 SRTs.

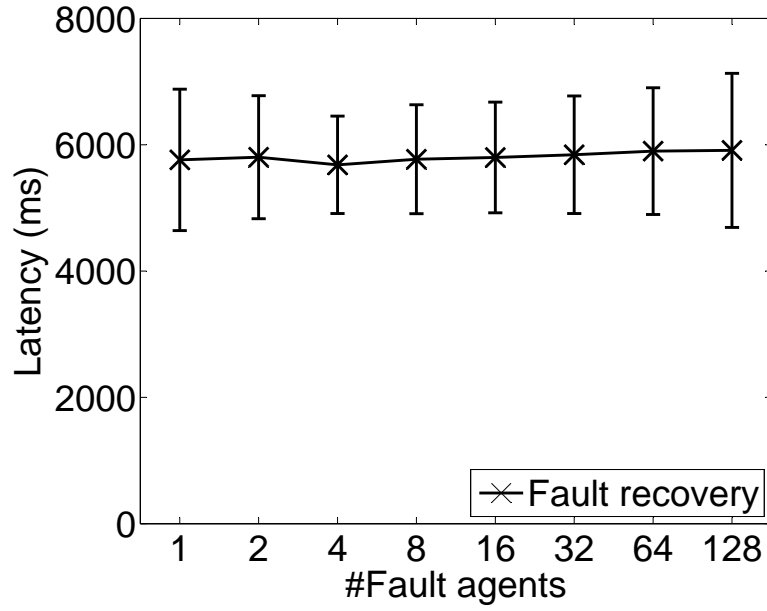
job. From Figure 30, Figure 31, Figure 32, we can see that, each node can be an interior checkpointing and replay point for one SRT job and also be a leaf node for other SRT jobs. Hence, the onus of checkpointing and replay for concurrently running jobs spans roughly evenly over all nodes in ELF.

### 5.6.3 Fault and straggler recovery

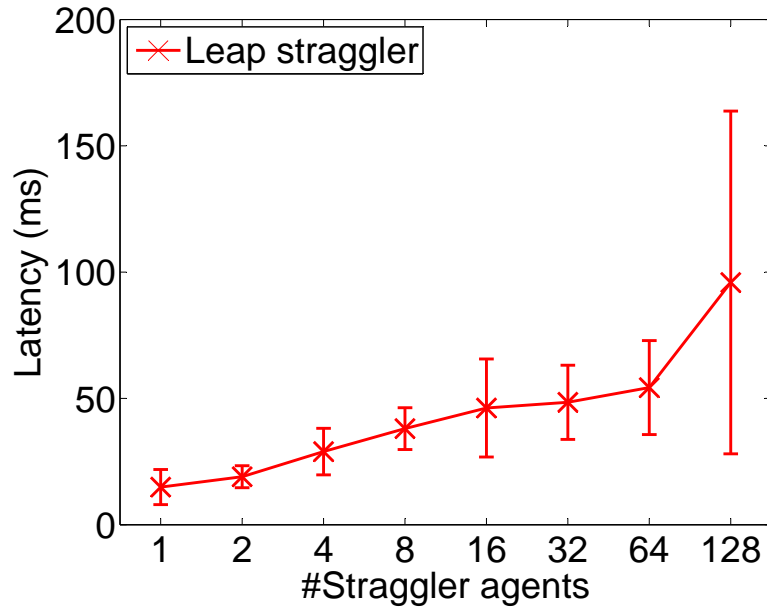
Fault and straggler recovery are evaluated with methods that use human intervention. To cause permanent failures, we deliberately remove some working agents from the datacenter to evaluate how fast ELF can recover. The time cost includes recomputing the routing table entries, rebuilding the SRT links, 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 32:** Checkpointing and replay load balancing for 15 SRTs.



**Figure 33:** Fault recovery.



**Figure 34:** Straggler recovery.

Figure 33 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.

Figure 34 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.

## 5.7 *Discussion*

This section explored the state management of ELF from three perspectives: the way that ELF synchronizes states across distributed machines between stages to guarantee user’s setting consistency, the way that ELF checkpoints node states to recover from transient or permanent failures, and the way that ELF mitigates stragglers.

ELF manages states mainly with synchronization and checkpointing, periodically replicating the task’s in-memory data to other nodes. However, we do not store the complete task state of each checkpoint, which will become increasingly expensive as task state grows. Instead, ELF discards passed batches automatically if no error is detected. Another optimization that ELF makes is to explore task data’s strong temporal and spatial locality by checkpointing each node’s state to its direct children node. The reason lies in that, after recovering from a failure, the children node is mostly likely to substitute the position of failure node in SRT for the follow-up computations.

We show that ELF were successful because we allow users control the most aspects of state issues such as the intensity of consistency, fault recovery and straggler mitigation, and only primarily add a cost in network bandwidth, which is tolerable in many applications.

## CHAPTER VI

### RELATED WORK

In this chapter we review works that relate to our streaming processing solution and discuss in detail how our solution advances the state of the art.

#### ***6.1 Streaming Databases***

The earliest academic systems for stream processing were developed in the database community, such as Aurora [89], Borealis [14], STREAM [20], and SPADE [39], using stateful operators and windows. SPADE provides a toolkit of built-in operators and a set of adapters, targeting the System S runtime. Unlike SPADE or STREAM that uses SQL-style declarative query interfaces, Aurora is based on a dataflow-style “boxes and arrows” paradigm that allows query activity to be interspersed with message processing. Borealis inherits its core stream processing functionality from Aurora. The gap recovery by Hwang [42] that drops error tuples is similar to ELF’s leap straggler recovery mechanism (i.e., ELF also drops error intervals).

Compared to them, ELF contributes: (1) a decentralized model that assigns each job a separate master, to improve the scalability and reliability of streaming systems, particularly when there are numerous concurrent jobs; and (2) the segmentation of streams into mini-batches with intermediate results saved in compressed form in memory, leading to high memory efficiency, particularly when merging streaming data with larger volumes of historical records.

#### ***6.2 MapReduce-style Systems***

Recent work extended the batch-oriented MapReduce model to support continuous stream processing, using techniques like pipelined parallelism, incremental processing

**Table 4:** Current MapReduce projects and related software.

Software	Brief description	Reference
Hive	Hive provides a database query interface to Apache Hadoop.	[78]
Hbase	Scalable distributed database that supports structured data storage for large tables.	[61]
Pig	Pig framework involves a high-level scripting language (Pig Latin) and offers a run-time platform that allows users to execute MapReduce on Hadoop.	[65]
Spark Streaming	Spark Streaming is an extension of the core Spark API. Spark core provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface centered on the RDD abstraction.	[87]
Chukwa	Chukwa is a data collection and analysis framework incorporated with MapReduce and HDFS; the workflow of Chukwa allows for data collection from distributed systems, data processing, and data storage in Hadoop.	[70]
Twister	Twister is a lightweight MapReduce runtime that provides support for iterative MapReduce computations.	[32]
YARN	Apache Hadoop YARN (Yet Another Resource Negotiator) is a cluster management technology.	[80]
MapR	MapR offers the converged data platform with the power of Apache Hadoop, Spark, event streaming, real-time database, and enterprise storage.	[13]

for map and reduce, and minimizing redundant computations. An overview of current MapReduce projects and related software is shown in Table 4. MapReduce model allows an unexperienced programmer to develop parallel programs that are capable of using computers in a cloud. In most cases, programmers are required to specify two functions only: the map function (mapper) and the reduce function (reducer). The mapper regards the *key/value* pair as input and generates intermediate *key/value* pairs. The reducer merges all the pairs associated with the same (intermediate) key and then generates an output.

MapReduce Online [30] pipelines data between map and reduce operators, by calling reduce with partial data for early results. Nova [64] runs as a workflow manager



on top of an unmodified Pig/Hadoop software stack, with data passed in a continuous fashion. Nova claims itself as a tool more suitable for large batched incremental processing than for small data increments processed with low latencies. Incoop [23] applies memorization to the results of partial computations, so that subsequent computations can reuse previous results for unchanged inputs. One-Pass Analytics [52] optimizes MapReduce jobs by avoiding expensive I/O blocking operations such as reloading map output. It exploits main memory to pre-combine map outputs by replacing the sort-merge implementation in MapReduce with a hash-based framework to enable fast in-memory data processing.

iMR [56] offers 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 [55] and Comet [41] run MapReduce jobs on new data every few minutes for “bulk incremental processing”, with all states stored in on-disk filesystem incurring latencies as high as tens of seconds. Spark Streaming [87] divides input data streams into batches and stores them in memory as RDDs [85]. 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.

In contrast to the systems mentioned above, ELF departs from the MapReduce-inherited batch processing nature. It replaces the single master paradigm used by these systems with a more scalable “*many masters*” approach. Perhaps more importantly, by placing ELF agents close to data sources (i.e., webserver), the raw data streams can be processed locally prior to the data shuffling across the network, so as to take advantage of “*data locality*” to avoid unnecessary traffic cost and data movement, particularly for distributed data sources.

### 6.3 Large-scale Streaming Systems

ELF is most akin to recently proposed distributed stream-processing systems like S4 [63], Storm [6], and Muppet[50]. Keyed data events are routed with affinity to processing elements (PEs), which consume the events and do one or both of the following: (1) emit events to other PEs, and (2) publish results. They 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. The most famous platforms include S4, Storm, SQLstream [9], Apache Kafka [49], MillWheel [15] and the like (see Table 5).

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,

**Table 5:** Current large-scale streaming systems and related software.

Software	Brief description	Reference
Storm	Storm is a distributed realtime computation system. It has the advantages of scalable, fault-tolerant, and is easy to set up and operate	[6].
System S	IBM System S provides a programming model and an execution platform for user-developed applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous data streams.	[40]
Splunk	Splunk is a big data analytics platform that rapidly explores, analyzes and visualizes data in Hadoop. It has the advantages of Fast and easy to use, dynamic environments, scales from laptop to datacenter.	[5]
Kafka	Kafka is a distributed publish-subscribe messaging system.	[49]
Sap HANA	SAP HANA was previously called "SAP High-Performance Analytic Appliance". It is a Platform for real-time business, with the advantage of fast in-memory computing and realtime analytics.	[3]

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 window size 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. MeteorShower [81] aims at recovering from large-scale failures, by orchestrating operators' checkpointing activities through 'tokens'. Each operator receiving the token is thereby triggered to checkpoint its state. That is analogous to ELF's job master which sends multicast messages as 'tokens' to synchronize its agents' checkpoints, but ELF's job masters use diverse application-specific tokens' paths instead of a common one. Percolator [68] 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 [69] runs the continuous, stateful operators in Microsoft StreamInsight [16] on a cluster, scaling with load swings through repartitioning or reconfiguring sub-DAG with more or less operators. Google proposed Pregel [58] for iterative graph algorithms. Pregel holds the entire graph in memory distributed across nodes, thus avoiding disk-access latencies. The primary overhead is the communication at the end of each superstep, which is essential to application semantics. MillWheel [15] uses an event-driven API that allows users to specify a directed computation graph for complex streaming computations. Naiad [62] is unique in tying high level-programming pattern (LINQ) to specialized system designs that execute iterative, parallel and cyclic dataflow programs. However, Naiad [62] is more suitable for working set that fits in the aggregate RAM of the cluster than large-scale distributed sets of nodes targeted by ELF. 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.

## **6.4 *Data aggregation systems***

ELF includes an overlay construction containing agents from all data sources and aggregating their states. Concerning overlay construction and data aggregation, PlanetLab has management tools like CoTop [2], CoMon [1], and Mon [53]. CoMon is a web-based general node/slice monitor that monitors most PlanetLab nodes, typically used to examine the resource profiles of individual experiments. CoTop provides a top-like monitoring tool for PlanetLab. Differing from CoMon and CoTop, Mon is an on-demand monitoring service for PlanetLab. Mon constructs a multicast tree on the fly to serve a one-shot query. Mon is a better solution for multicasting user commands to applications rather than locating resources for applications, because it has no prior knowledge about the attributes. Ganglia [60] uses a single hierarchical tree to collect all data of federated clusters. The above tools use a centralized approach without in-network aggregation; hence, all individual data are returned to a local machine, even though only their aggregates are of interest. This has limited scalability with the size of the system and the number of attributes.

Distributed systems have explored scalability for generating aggregated results via some large distributed overlay. Astrolabe [79] provides a generic aggregation abstraction and uses a single static tree to aggregate all states. SDIMS [83] uses the same approach but constructs multiple trees for better scalability and flexibility. Its leaf nodes are physical machines and the internal nodes, represented as virtual nodes, correspond to administrative domains responsible for administrative autonomy and isolation. Unlike SDIMS, which still assumes a single group for the entire system, Moara [48] maintains many groups for aggregation trees based on different query rates and group churn rates, thus reducing bandwidth consumption. Q-Tree [18] targets at multi-attribute composite range queries, by building a single tree and assigning range

intervals on each node in a hierarchical manner.

ELF leverages DHT's low cost to construct overlay and DHT's publish/subscribe flexibility to build bottom-up trees to implement pipeline dataflow and cycle dataflow, so as to lower response times and message overheads. ELF's contribution lies in reconsidering and using the concept in stream processing system but providing user-customizations.

Summarizing, the new contributions ELF made are (1) improved *locality* by pushing batching to the end webserver; (2) *memory efficiency* by saving small duration snapshots in in-memory CBTs; (3) *scalability* by decomposing one master into  $n$  masters for  $n$  jobs running across the network; and (4) *new functionality* like feedback control and coordination cross multiple streaming jobs as inherent system components.

## CHAPTER VII

### CONCLUSION

This dissertation described ELF, a novel decentralized model for streaming computation that changes commonly used “*one master many workers*” architecture to a “*many masters many workers*” architecture. ELF innovations go beyond scalability and performance improvements, to also offer new functionalities, including support for job function changes at runtime and cross-job coordination. Processing consistency and fault recovery are obtained by treating a streaming computation as a series of mini-batch computations on small time intervals. The intermediate results from those batch computations at each interval are stored in memory as compressed buffer trees (CBTs) structure across the datacenter, and are gradually rolling up to a global shared reducer tree (SRT), with results available at the SRT’s root (the job master).

Experimental evaluations demonstrate ELF’s scalability 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. Finally, because ELF bypasses the log collection and storage tier, it can smoothly interoperate with other systems, thus offering a lightweight solution to add stream processing capabilities to the large-scale data collection and batch analytics systems used in web companies and elsewhere.

In the rest of this chapter, we summarize a few of the lessons that influenced this work. Finally, we sketch areas for future work.

#### ***7.1 Lessons Learned***

**The importance of decentralized architecture.** The main thread underlying our work is how to decouple the “one master many workers” centralized architecture into the “many masters many workers” decentralized architecture. Due to the

large volume of streaming data that are continuously generated in realtime, many streaming applications tend to share the data and the platform and run concurrently on top of them. Whereas previous systems have mostly inherited from traditional MapReduce’s centralized architecture to rely on the primary master to track all jobs and tasks resulting in central bottleneck, ELF enhances the scalability by offering each application an independent master to control the application’s behavior while automatically providing fault tolerance.

**The importance of immediate access to data source.** For “big data” applications in particular, datasets are generated in a distributed way. Previous systems usually converge these datasets using a log collecting tier and transfer them to the storage tier. Only on the storage tier, streaming applications have access to the datasets and process them. Another lesson from our work is that we found out that waiting data to be collected, stored and then processed is quite inefficient, and it is low latency that matters, because big data is generated in realtime and is most valuable at its time of generation. Therefore, ELF associates each data source with an ELF agent that has immediate access to the fresh data when it is generated, and thus processed them instantly.

**The importance of integrating large historical records.** One interesting lesson in how to integrate large historical records is to look at bottlenecks. In many cases, a few resources ultimately limit the performance of the application, so giving users control to optimize these resources can lead to good performance. For example, when it comes to store historical records in memory, Walmart’s Muppet and Twitter’s Storm all use `Hashtable`. However, we found out every insert in `Hashtable` require traversing the whole table and perform an operation — *a read, write and update*. Eventually, the accumulated small I/O costs are large and fall into the bottleneck

that limits the throughputs. ELF uses *compressed buffer tree* data structure in memory to store the historical records. The take away is: Integration usually happens on large chunks of data and thus combining many small I/Os into one large I/O and thus enhancing the throughputs.

**The importance of exploring DHT for scalability.** Finally, part of what made ELF scalable is that ELF is built on top of DHTs. Each ELF agent is assigned with a `nodeId` in a circular 128-bit large id space. Using the same hashing function, each application is computed a `appId` in the same id space. The node whose `nodeId` is numerically close to `appId` automatically becomes the node master. Theoretically, when the number of concurrent running applications achieves to a large scale, their masters will be roughly and evenly distributed on all nodes of the overlay.

## 7.2 *Future Work*

There are several smaller, concrete extensions to the ELF framework that would enable a wider array of possible user scenarios.

**Approximate results:** In addition to consistency guarantees and the recomputation of lost work, another way to deal with failures and distributed latency issues is to return approximate partial results. ELF can provide the opportunity by bypassing the missing parents of SRT and simply starting processing before parents are all done. This would need to be coupled to some user-level specification of exactly how consistent they expect the result to be.

**Setting the synchronization interval:** The synchronization interval directly determines the tradeoff between the end-to-end latency, throughput, and the consistency guarantee. It may be useful for ELF to tune it automatically according to the workload variations. This would need a higher-level specification of the service agreement



with the users than just a simple time deadline, and that management construct would lead to interesting new research possibilities.

**CBT store:** In our current implementation, CBT resides in memory only, which might incur data loss if a physical node catastrophically fails. The periodic checkpoints currently in the system only would offer a partial restore. It may be possible to extend in-memory CBT to support continuous checkpointing to local non-volatile storage (disk, NVM, etc.). Storing different versions of the state CBTs is essential for the system to perform lineage-based fault recovery.

Beyond these concrete examples, there are several other longer-term possible extensions to ELF. These extensions would open up a further set of possible explorations in both systems design and in utilization.

**QoS guarantees:** Multiple streaming applications run together on the same cluster for the purpose of data locality, data reuse, and fully utilizing resources, but *how to guarantee their quality of services in the face of resource competitions and performance interference?* Common solutions for sharing a cluster today are either to statically partition the cluster and run one application per partition, or to allocate a set of VMs to each application. However, as streaming applications start being more and more *volatile*, we believe that many of problems will arise in this setting. For example, first, streaming applications usually have complex netlike dataflows, and thus resulting in varied resource requirements at vertices. Second, streaming applications resource demands vary over time, over locality and the like. Third, streaming applications may span tens of thousands of nodes (vertices) running hundreds of jobs. The scale makes multiplexing a cluster complicated. It is difficult for the administrator to tell in advance how much resource each application or each component inside each application should have. For instance, when applications have different priorities, *how should a scheduler evict tasks to make room*

*for more important ones?* As streaming application becomes more diverse and heterogeneous, *what should a resource scheduler look like?* These questions are arising in real Spark, Hadoop deployments, and require new abstractions and analytical work.

**Cloud scale out (SPS):** To benefit from the “*pay-as-you-go*” model in public cloud environments like Amazon EC2 and Rackspace, a stream processing system should have the ability to scale out on demand, by requesting additional VMs at run-time, reacting to changes in processing workload, and repartitioning query operators accordingly. It remains an open question that how a stream processing system can automatically scale out ‘*to the cloud*’. One approach is to scale with load swings through *repartitioning* or *reconfiguring* sub-DAGs with more or less operators, and more or less VMs accordingly.

**Cooperative analytics:** ELF shows the potential to integrate multiple platforms’ intermediate results in a realtime nature, which is highly attractive in practice. For example, Amazon’s *micro-sale* application can run ad-hoc queries over steam state from Twitter’s *hot-topic* application, so as to understand the popularity trend and make instant business decisions to boost the future ‘popularly discussed item’ sales. We believe such cooperative computing will be essential in the future, as most “big data” applications, especially those using business or social data, want to process results in realtime. However, there are many challenges to making it practical, including *how to safely share results across different platforms and meanwhile guarantee privacy*, and *how to prioritize computation across platforms to respect different deadlines*.

**Debugging:** Configuring and deploying big data system is not easy, and as a consequence, they have experienced a wide range of bugs and patches. Many configuration

issues manifest themselves in ways similar to system bugs such as crashes, hangs, silent failures. Users have a difficult time understanding the correctness and the performance of their distributed programs and configurations. One approach we are currently exploring is to leverage the deterministic nature of most cluster computing models to selectively *replay* part of the computation so as to automatically infer configuration requirements, and then provide *hints* to help user avoid misconfiguration vulnerabilities.

**Application/cloud interaction:** As container-based virtualization technology becomes more and more popular, an interesting approach to look at is to enhance container-based virtualization with ‘*elasticity drivers*’ that permits applications to provide input to the underlying hypervisor/cloud infrastructures concerning its resource needs and usage profiles with their consequent effects on their ability to share datacenter systems and devices. Another approach extends per-container elasticity drivers with higher level methods that perform resource arbitration across container ensembles.

We hope that continued experience with ELF will help us address these challenges, and lead to solutions that are applicable both within ELF and to other cluster computing systems.

## REFERENCES

- [1] “Comon.” <http://comon.cs.princeton.edu/>.
- [2] “Cotop.” <http://codeen.cs.princeton.edu/cotop/>.
- [3] “Sap hana.” <http://go.sap.com/index.html>.
- [4] “Sparsehash.” <http://code.google.com/p/sparsehash/>.
- [5] “Splunk.” <http://www.splunk.com/>.
- [6] “Storm.” <https://github.com/nathanmarz/storm.git>.
- [7] “Zql: a sql parser.” <http://zql.sourceforge.net/>, 2002.
- [8] “Scribe.” <http://github.com/facebook/scribe/>, 2008.
- [9] “Sqlstream.” <http://www.sqlstream.com/products/server/>, 2012.
- [10] “Twitter streaming apis.” <https://dev.twitter.com/docs/streaming-apis>, 2012.
- [11] “Zeromq.” <http://zeromq.org/>, 2012.
- [12] “Flume.” <http://flume.apache.org/>, 2013.
- [13] “Mapr.” <https://www.mapr.com/products/apache-hadoop>, 2014.
- [14] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., CHERNIACK, M., HYON HWANG, J., LINDNER, W., MASKEY, A. S., RASIN, E., RYVKINA, E., TATBUL, N., XING, Y., and ZDONIK, S., “The design of the borealis stream processing engine,” in *CIDR*, pp. 277–289, 2005.
- [15] AKIDAU, T., BALIKOV, A., BEKIROGLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., and WHITTLE, S., “Millwheel: Fault-tolerant stream processing at internet scale,” in *VLDB*, 2013.
- [16] ALI, M. H., GERE, C., RAMAN, B. S., SEZGIN, B., and TARNAVSKI, E., “Microsoft cep server and online behavioral targeting,” *Proc. VLDB Endow.*, vol. 2, pp. 1558–1561, Aug. 2009.
- [17] AMUR, H., RICHTER, W., ANDERSEN, D. G., KAMINSKY, M., SCHWAN, K., BALACHANDRAN, A., and ZAWADZKI, E., “Memory-efficient groupby-aggregate using compressed buffer trees,” in *SOCC*, (Santa Clara, CA, USA), ACM, 2013.

- [18] AREFIN, M. A., UDDIN, M. Y. S., GUPTA, I., and NAHRSTEDT, K., “Q-tree: A multi-attribute based range query solution for tele-immersive framework,” in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS ’09, (Washington, DC, USA), pp. 299–307, IEEE Computer Society, 2009.
- [19] ARGE, L., “The buffer tree: A technique for designing batched external data structures,” *Algorithmica*, vol. 37, no. 1, pp. 1–24, 2003.
- [20] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., and WIDOM, J., “Models and issues in data stream systems,” in *PODS*, 2002.
- [21] BABU, S. and WIDOM, J., “Continuous queries over data streams,” *SIGMOD Rec.*, vol. 30, pp. 109–120, Sept. 2001.
- [22] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., and STONEBRAKER, M., “Fault-tolerance in the borealis distributed stream processing system,” in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’05, (New York, NY, USA), pp. 13–24, ACM, 2005.
- [23] BHATOTIA, P., WIEDER, A., RODRIGUES, R., ACAR, U. A., and PASQUIN, R., “Incoop: Mapreduce for incremental computations,” in *SOCC*, (New York, NY, USA), pp. 7:1–7:14, ACM, 2011.
- [24] BRENNAN, L., DEMERS, A., GEHRKE, J., HONG, M., OSSHER, J., PANDA, B., RIEDEWALD, M., THATTE, M., and WHITE, W., “Cayuga: a high-performance event processing engine,” in *SIGMOD*, (New York, NY, USA), pp. 1100–1102, ACM, 2007.
- [25] CASTRO, M., DRUSCHEL, P., KERMARREC, A. M., and ROWSTRON, A. I., “Scribe: a large-scale and decentralized application-level multicast infrastructure,” *IEEE J.Sel. A. Commun.*, vol. 20, no. 8, pp. 1489–1499, 2006.
- [26] CASTRO, M., DRUSCHEL, P., CHARLIE, Y., and ROWSTRON, H. A., “Exploiting network proximity in peer-to-peer overlay networks,” tech. rep., 2002.
- [27] CASTRO, M., JONES, M. B., KERMARREC, A.-M., ROWSTRON, A., THEIMER, M., WANG, H., and WOLMAN, A., “Evaluation of scalable application-level multicast built using peer-to-peer overlays,” in *INFOCOM’03*, IEEE, 2003.
- [28] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., and SHAH, M. A., “Telegraphcq: Continuous dataflow processing for an uncertain world,” in *CIDR*, 2003.
- [29] CHERNIACK, M., BALAKRISHNAN, H., BALAZINSKA, M., CARNEY, D., CETINTEMEL, U., XING, Y., and ZDONIK, S., “Scalable Distributed Stream Processing,” in *CIDR*, (Asilomar, CA), January 2003.

- [30] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEGY, K., and SEARS, R., “Mapreduce online,” in *NSDI*, (Berkeley, CA, USA), USENIX Association, 2010.
- [31] DÉSARMÉNIEN, J., “How to run T<sub>E</sub>X in french,” Tech. Rep. SATN-CS-1013, Computer Science Department, Stanford University, Stanford, California, Aug. 1984.
- [32] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., and FOX, G., “Twister: A runtime for iterative mapreduce,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC ’10, (New York, NY, USA), pp. 810–818, ACM, 2010.
- [33] FACEBOOK <http://www.sec.gov/Archives/edgar/data/1326801/000119312512235588/d287954ds1a.htm>, 2012.
- [34] FOURSQUARE, “Aws case study: foursquare.” <http://aws.amazon.com/solutions/case-studies/foursquare/>, 2012.
- [35] FUCHS, D., “The format of T<sub>E</sub>X’s DVI files version 1,” *TUGboat*, vol. 2, pp. 12–16, July 1981.
- [36] FUCHS, D., “Device independent file format,” *TUGboat*, vol. 3, pp. 14–19, Oct. 1982.
- [37] FURUTA, R. K. and MACKAY, P. A., “Two T<sub>E</sub>X implementations for the IBM PC,” *Dr. Dobbs’s Journal*, vol. 10, pp. 80–91, Sept. 1985.
- [38] GAO, H., HU, J., WILSON, C., LI, Z., CHEN, Y., and ZHAO, B. Y., “Detecting and characterizing social spam campaigns,” in *IMC*, (New York, NY, USA), ACM, 2010.
- [39] GEDIK, B., ANDRADE, H., WU, K.-L., YU, P. S., and DOO, M., “Spade: the system s declarative stream processing engine,” in *SIGMOD*, (New York, NY, USA), pp. 1123–1134, ACM, 2008.
- [40] GEDIK, B., ANDRADE, H., WU, K.-L., YU, P. S., and DOO, M., “Spade: The system s declarative stream processing engine,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’08, (New York, NY, USA), pp. 1123–1134, ACM, 2008.
- [41] HE, B., YANG, M., GUO, Z., CHEN, R., SU, B., LIN, W., and ZHOU, L., “Comet: batched stream processing for data intensive distributed computing,” in *SOCC*, (New York, NY, USA), pp. 63–74, ACM, 2010.
- [42] HWANG, J.-H., BALAZINSKA, M., RASIN, A., CETINTEMEL, U., STONEBRAKER, M., and ZDONIK, S., “High-Availability Algorithms for Distributed Stream Processing,” in *ICDE*, (Tokyo, Japan), April 2005.

- [43] KNUTH, D. E., “The WEB system for structured documentation, version 2.3,” Tech. Rep. STAN-CS-83-980, Computer Science Department, Stanford University, Stanford, California, Sept. 1983.
- [44] KNUTH, D. E., *The T<sub>E</sub>X Book*. Reading, Massachusetts: Addison-Wesley, 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.
- [45] KNUTH, D. E., “Literate programming,” *The Computer Journal*, vol. 27, pp. 97–111, May 1984.
- [46] KNUTH, D. E., “A torture test for T<sub>E</sub>X, version 1.3,” Tech. Rep. STAN-CS-84-1027, Computer Science Department, Stanford University, Stanford, California, Nov. 1984.
- [47] KNUTH, D. E., *T<sub>E</sub>X: The Program*, vol. B of *Computers & Typesetting*. Reading, Massachusetts: Addison-Wesley, 1986.
- [48] KO, S. Y., YALAGANDULA, P., GUPTA, I., TALWAR, V., MILOJICIC, D., and IYER, S., “Moara: Flexible and scalable group-based querying system,” in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware ’08, (New York, NY, USA), pp. 408–428, Springer-Verlag New York, Inc., 2008.
- [49] KREPS, J., NARKHEDE, N., and RAO, J., “Kafka: a distributed messaging system for log processing,” in *NetDB*, (New York, NY, USA), ACM, 2011.
- [50] LAM, W., LIU, L., PRASAD, S., RAJARAMAN, A., VACHERI, Z., and DOAN, A., “Muppet: Mapreduce-style processing of fast data,” *Proc. VLDB Endow.*, vol. 5, pp. 1814–1825, Aug. 2012.
- [51] LAMPORT, L., *L<sup>A</sup>T<sub>E</sub>X: A Document Preparation System. User’s Guide and Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1986.
- [52] LI, B., MAZUR, E., DIAO, Y., MCGREGOR, A., and SHENOY, P., “A platform for scalable one-pass analytics using mapreduce,” in *SIGMOD*, (New York, NY, USA), pp. 985–996, ACM, 2011.
- [53] LIANG, J., KO, S. Y., GUPTA, I., and NAHRSTEDT, K., “Mon: On-demand overlays for distributed system management,” in *Proceedings of the 2Nd Conference on Real, Large Distributed Systems - Volume 2*, WORLDS’05, (Berkeley, CA, USA), pp. 13–18, USENIX Association, 2005.
- [54] LIU, L., PU, C., and TANG, W., “Continual queries for internet scale event-driven information delivery,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 11, pp. 610–628, July 1999.
- [55] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., and YOCUM, K., “Stateful bulk processing for incremental analytics,” in *SOCC*, (New York, NY, USA), pp. 51–62, ACM, 2010.

- [56] LOGOTHETIS, D., TREZZO, C., WEBB, K. C., and YOCUM, K., “In-situ mapreduce for log processing,” in *USENIXATC*, (Berkeley, CA, USA), USENIX Association, 2011.
- [57] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., and HONG, W., “Tag: a tiny aggregation service for ad-hoc sensor networks,” *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 131–146, Dec. 2002.
- [58] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., and CZAJKOWSKI, G., “Pregel: A system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, (New York, NY, USA), pp. 135–146, ACM, 2010.
- [59] MASSIE, M. L., CHUN, B. N., and CULLER, D. E., “The ganglia distributed monitoring system: Design, implementation and experience,” *Parallel Computing*, vol. 30, p. 2004, 2003.
- [60] MASSIE, M. L., CHUN, B. N., and CULLER, D. E., “The ganglia distributed monitoring system: design, implementation, and experience,” *Parallel Computing*, vol. 30, no. 5-6, pp. 817–840, 2004.
- [61] MEDIA, O., *Hbase: The Definitive Guide*. O’Reilly Media, Inc., 2011.
- [62] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., and ABADI, M., “Naiad: a timely dataflow system,” in *SOSP*, (New York, NY, USA), ACM, 2013.
- [63] NEUMEYER, L., ROBBINS, B., NAIR, A., and KESARI, A., “S4: Distributed stream computing platform,” in *ICDMW*, (Washington, DC, USA), pp. 170–177, IEEE Computer Society, 2010.
- [64] OLSTON, C., CHIOU, G., CHITNIS, L., LIU, F., HAN, Y., LARSSON, M., NEUMANN, A., RAO, V. B., SANKARASUBRAMANIAN, V., SETH, S., TIAN, C., ZICORNELL, T., and WANG, X., “Nova: continuous pig/hadoop workflows,” in *SIGMOD*, (New York, NY, USA), pp. 1081–1090, ACM, 2011.
- [65] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., and TOMKINS, A., “Pig latin: a not-so-foreign language for data processing,” in *SIGMOD*, (New York, NY, USA), pp. 1099–1110, ACM, 2008.
- [66] PATASHNIK, O., *BibT<sub>E</sub>Xing*. Computer Science Department, Stanford University, Stanford, California, Jan. 1988. Available in the BibT<sub>E</sub>X release.
- [67] PATASHNIK, O., *Designing BibT<sub>E</sub>X Styles*. Computer Science Department, Stanford University, Jan. 1988.



- [68] PENG, D. and DABEK, F., “Large-scale incremental processing using distributed transactions and notifications,” in *OSDI*, (Berkeley, CA, USA), USENIX Association, 2010.
- [69] QIAN, Z., HE, Y., SU, C., WU, Z., ZHU, H., ZHANG, T., ZHOU, L., YU, Y., and ZHANG, Z., “Timestream: reliable stream computation in the cloud,” in *Eurosys*, (New York, NY, USA), pp. 1–14, ACM, 2013.
- [70] RABKIN, A. and KATZ, R., “Chukwa: a system for reliable large-scale log collection,” in *LISA*, (Berkeley, CA, USA), pp. 1–15, USENIX Association, 2010.
- [71] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., and YU, H., “Opendht: a public dht service and its uses,” in *SIGCOMM’05*, (New York, NY, USA), pp. 73–84, ACM, 2005.
- [72] ROWSTRON, A. I. T. and DRUSCHEL, P., “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *Middleware*, (London, UK, UK), pp. 329–350, Springer-Verlag, 2001.
- [73] SAMUEL, A. L., “First grade T<sub>E</sub>X: A beginner’s T<sub>E</sub>X manual,” Tech. Rep. SATN-CS-83-985, Computer Science Department, Stanford University, Stanford, California, Nov. 1983.
- [74] SOCIAL, G., “An interview with twitter ceo dick costolo at ideas economy: Information.” 2012.
- [75] SPIVAK, M. D., *The Joy of T<sub>E</sub>X*. American Mathematical Society, 1985.
- [76] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., and BALAKRISHNAN, H., “Chord: A scalable peer-to-peer lookup service for internet applications,” in *SIGCOMM’01*, (New York, NY, USA), pp. 149–160, ACM, 2001.
- [77] STONEBRAKER, M., ÇETINTEMEL, U., and ZDONIK, S., “The 8 requirements of real-time stream processing,” *SIGMOD Rec.*, vol. 34, pp. 42–47, Dec. 2005.
- [78] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., and MURTHY, R., “Hive: a warehousing solution over a map-reduce framework,” *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, 2009.
- [79] VAN RENESSE, R., BIRMAN, K. P., and VOGELS, W., “Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining,” *ACM Trans. Comput. Syst.*, vol. 21, pp. 164–206, May 2003.
- [80] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O’MALLEY, O., RADIA, S., REED, B., and BALDESCHWIELER, E., “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC ’13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

- [81] WANG, H., PEH, L.-S., KOUKOU MIDIS, E., TAO, S., and CHAN, M. C., “Meteor shower: A reliable stream processing system for commodity data centers,” in *IPDPS*, (Washington, DC, USA), pp. 1180–1191, IEEE Computer Society, 2012.
- [82] WU, S.-Y., HITT, L. M., CHEN, P.-Y., and ANANDALINGAM, G., “Customized bundle pricing for information goods: A nonlinear mixed-integer programming approach,” *Manage. Sci.*, vol. 54, pp. 608–622, Mar. 2008.
- [83] YALAGANDULA, P. and DAHLIN, M., “A scalable distributed information management system,” in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’04, (New York, NY, USA), pp. 379–390, ACM, 2004.
- [84] YU, Y., ISARD, M., FETTERLY, D., BUDI, M., ERLINGSSON, U., GUNDA, P. K., and CURREY, J., “Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language,” in *OSDI*, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2008.
- [85] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., and STOICA, I., “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing,” in *NSDI*, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.
- [86] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., and STOICA, I., “Spark: cluster computing with working sets,” in *HotCloud’10*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.
- [87] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., and STOICA, I., “Discretized streams: Fault-tolerant streaming computation at scale,” in *SOSP*, (Farmington, PA), ACM, ACM, Sept. 2013.
- [88] ZAHARIA, M., DAS, T., LI, H., SHENKER, S., and STOICA, I., “Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters,” in *HotCloud’12*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2012.
- [89] ZDONIK, S., STONEBRAKER, M., CHERNIACK, M., ÇETINTEMEL, U., BALAZINSKA, M., and BALAKRISHNAN, H., “The aurora and medusa projects,” *Data Engineering*, vol. 51, p. 3, 2003.
- [90] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., and KUBIATOWICZ, J. D., “Tapestry: a resilient global-scale overlay for service deployment,” *IEEE J.Sel. A. Commun.*, vol. 22, pp. 41–53, Sept. 2006.

## VITA

Liting Hu was born in Wuhan, China. She attended public schools in Wuhan, China, received a B.Tech in Computer Science from Huazhong University of Science and Technology, Wuhan, China under the supervision of Dr.Hai Jin in 2007 before coming to Georgia Tech to pursue a doctorate in Computer Science. Her primary research interests are in big data analytics and its intersection with distributed systems. Liting has also done research in datacenter's resource management and system virtualization technology. She is co-advised by Dr.Karsten Schwan and Dr.Matthew Wolf. She spent summers interning at IBM T.J. Watson Research Center, Intel Science and Technology Center for Cloud Computing, Microsoft Research Asia, VMware, and has been working closely with them.