Streaming Temporal Graphs

Eric Llewellyn Goodman

University of Colorado at Boulder, ya_krutoi@hotmail.com

Follow this and additional works at: https://scholar.colorado.edu/csci_gradetds

Part of the Computer Sciences Commons

Recommended Citation
https://scholar.colorado.edu/csci_gradetds/203

This Dissertation is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
Streaming Temporal Graphs

by

Eric L. Goodman

B.S. Computer Science and Math, Brigham Young University, 2003

M.S. Computer Science, Brigham Young University, 2005

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2019
This thesis entitled:
Streaming Temporal Graphs
written by Eric L. Goodman
has been approved for the Department of Computer Science

Dr. Dirk Grunwald

Dr. Daniel Massey

Dr. Eric Keller

Dr. Qin Lv

Date ________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Goodman, Eric L. (Ph.D., Computer Science)

Streaming Temporal Graphs

Thesis directed by Dr. Dirk Grunwald

We provide a domain specific language called the Streaming Analytics Language (SAL) to write concise but expressive analyses of streaming temporal graphs. We target problems where the data comes as an infinite stream and where the volume is prohibitive, requiring a single pass over the data and tight spatial and temporal complexity constraints. Also, each item in the stream can be thought of as an edge in a graph, and each edge has an associated timestamp and duration.

A real-world problem that is a streaming temporal graph is cyber security data. Machines communicate with each other within a network, forming a streaming sequence of edges with temporal information. As such, we elucidate the value of SAL by applying it to a large range of cyber-related problems. With a combination of vertex-centric computations that create features per vertex, and subgraph matching to find communication patterns of interest, we cover a wide spectrum of important cyber use cases. As an example, we discuss Verizon’s Data Breach Investigations Report, and show how SAL can be used to capture most of the nine different categories of cyber breaches. Also, we apply SAL to discovering botnet activity within network traffic in 13 different scenarios, with an average area under the curve (AUC) of the receiver operating characteristic (ROC) of 0.87.

Besides SAL as a language, as another contribution we present an implementation we call the Streaming Analytics Machine (SAM). With SAM, we can run SAL programs in parallel on a cluster, achieving rates of a million netflows per second, and scaling to 128 nodes or 2560 cores. We compare SAM to another streaming framework, Apache Flink, and find that Flink cannot scale past 32 nodes for the problem of finding triangles (a subgraph of three interconnected nodes) within the streaming graph. Also, SAM excels when the subgraphs are frequent, continuing to find the expected number of subgraphs, while Flink performance degrades and under-reports. Together, SAL and SAM provide an expressive and scalable infrastructure for performing analyses on streaming temporal graphs.
Acknowledgements

A big thanks goes to my advisor, Dirk Grunwald, who helped me hone this dissertation over a lengthy period of time. The path I took was shaped by him and allowed me to create a much better product and a more relevant research contribution.

I also thank the committee members for their time and input. This includes not only the current committee of Daniel Massey, Eric Keller, Sangtae Ha, and Qin Lv, but also former members John Black and Sriram Sankaranarayana.

Sandia National Laboratories funded my doctoral degree through their university programs, for which I am grateful. I especially appreciate the unwavering support and patience from my managers, Judy Spomer and Michael Haass.

My parents were instrumental in fostering my curiosity and desire to learn, which lead me down this path of educational advancement. I still haven’t invented the transmat beam, but that will be my next project.
Contents

Chapter

1. Introduction 1

2. The Streaming Analytics Language 5
   2.1 Case Study 14
   2.2 Static Vertex Attributes 22
   2.3 Temporal Subgraph Matching 23

3. SAL as a Query Language for Cyber Problems 29
   3.1 Crimeware 30
   3.2 Cyber-Espionage 33
   3.3 Denial of Service 34
   3.4 Point of Sale 36
   3.5 Privilege Misuse 37
   3.6 Web Applications 39
   3.7 Summary 40

4. Related Work 41
   4.1 Vertex-centric Computing 42
   4.2 Graph Domain Specific Languages 47
      4.2.1 Green-Marl 48
# Table of Contents

6.3 Summary .................................................... 91

7 Temporal Subgraph Matching .................................. 92
   7.1 SAM Parameter Exploration .............................. 94
   7.2 Apache Flink ............................................. 96
   7.3 Comparison Results: SAM vs Flink ................... 98
   7.4 Simulating other Constraints ............................ 103
   7.5 Summary .................................................. 104

8 Conclusions .................................................... 106

Bibliography ..................................................... 108

Appendix

A SAL and Flink Programs ....................................... 125
   A.1 Machine Learning Example .............................. 125
   A.2 Apache Flink Triangle Implementation ................ 126
Tables

Table

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>A table of vertex-centric frameworks and various attributes.</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>This table shows the conciseness of the SAL language. For the Disclosure pipeline (discussed in Section 2.1), the machine learning pipeline (discussed in Section 6.1), and the temporal triangle query (discussed in Chapter 7) the amount of code needed is reduced between 10-25 times.</td>
<td>63</td>
</tr>
<tr>
<td>5.2</td>
<td>A listing of the major classes in SAM and how they map to language features in SAL. We also make the distinction between consumers, producers, and feature creators. Producers all share a parallelFeed method that sends the data to registered consumers, which is how parallelization is achieved. Each of the consumers receive data from producers. If a consumer is also a feature creator, it performs a computation on the received data, generating a feature, which is then added to a thread-safe feature map.</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>A more detailed look at the AUC results. The second column is the AUC of the ROC after training on the first half of the data and then testing on the second half. The third column is reversed: trained on the second half and tested on the first half.</td>
<td>87</td>
</tr>
</tbody>
</table>
Figures

1.1 This dissertation is an intersection of four different research areas and presents a domain specific language (DSL) to express streaming temporal graph and machine learning queries. 2

2.1 This figure represents a common pattern for vertex-centric computations in SAL programs. A stream of edges is partitioned into $N$ vertices that are then operated on by $o$ operators that compute over a sliding window of length $n$ items. 6

2.2 Illustration showing the subgraph nature of a Watering Hole Attack. The target machine accesses a popular site, Bait. Shortly after accessing Bait, Target begins communicating to a new machine, the Controller 10

2.3 This figure demonstrates the use of the STREAM BY statement to logically divide a single stream into multiple streams. The STREAM BY statement is following by a FOREACH GENERATE statement that creates estimates on the top two most frequent destination ports and their frequencies. The FILTER statement uses this information to downselect to IPs where the top two ports account for 90% of the traffic. 16

2.4 This figure demonstrates the use of the TRANSFORM and COLLAPSE BY statements to define parts of the Disclosure pipeline. 19
3.1 A subgraph showing the flow of messages for data exfiltration. A controller sends a small control message to an infected machine, which then sends large amounts of data to a drop box. ................................................. 33

3.2 A subgraph showing an XSS attack. .................................................. 39

4.1 SPARQL Query number two from the Lehigh University Benchmark. .............. 57

5.1 Architecture of the system: data comes in over a socket layer and is then distributed via ZeroMQ. .......................................................... 64

5.2 The push pull object. ................................................................. 65

5.3 Traditional Compressed Sparse Row (CSR) data structure. ......................... 72

5.4 Modified Compressed Sparse Row (CSR) data structure. .......................... 72

5.5 An overview of the algorithm employed to detect subgraphs of interest. Asynchronous threads are launched to consume each edge. The consume threads then perform five different operations. Another set of threads are pulling requests from other nodes for edges within the Request Communicator. Finally, one more set of threads pull edges from other nodes in reply to edge requests. .................... 75

6.1 This graphic shows the AUC of the ROC curve for each of the CTU scenarios. We first train on the first half of the data and test on the second half, and then switch. ................................. 87

6.2 Weak Scaling Results: For each run with \( n \) nodes, a total of \( n \) million netflows were run through the system with a million contiguous netflows per node randomly selected from the CTU dataset. There were two types of runs, with renamed IPs (to create the illusion of a larger network) or with randomized IPs. For the renamed IPs, there continues to be improved throughput until 61 nodes/976 cores. For the randomized IPs, the scaling continues through the total 64 nodes / 1024 cores. .... 89
6.3 This shows the weak scaling efficiency. The efficiency continues to degrade for the renamed IPs, but for the randomized IPs, the efficiency hovers a little above 0.5. This is largely due to the fact that the randomized IPs almost perfectly load-balances the work, while the renamed IPs encounters load imbalances because of the power-law nature of the data.

7.1 We vary the number of pull threads and the number of push sockets (NPS) for the RequestCommunicator to determine the best setting for maximum throughput. The RequestMap object utilizes the RequestCommunicator during the process(edge) function.

7.2 Shows the maximum throughput for SAM and Flink. We vary the number of nodes with values of 16, 32, 64, and 120. We vary the number of vertices, |V|, between 25,000 and 150,000. Flink has the best throughput with 32 nodes, after which adding additional nodes hurts the overall aggregate rate and the number of found triangles continues to decrease. SAM is able to scale to 120 nodes.

7.3 Another view of the data where we plot the aggregate throughput versus the total number of triangles to find. This view emphasizes that SAM does better with more complex problems, i.e. more triangles.

7.4 Another look at the maximum throughput for SAM and Flink. Here we relax the restriction that the number of reported triangles must be higher than 85% of expected, but keep the latency restrictions that the computation must complete within 15 seconds of termination of input. Flink struggles increasing to 64 nodes, and then drops consistently with 120 nodes. SAM continues to increase throughput through 120 nodes.
7.5 This graph shows the ratio of found triangles divided by the number of expected triangles given by Theorem 2. The rate for each method was the same as from Figure 7.4. SAM stays fairly consistent throughout the runs, with an average of 0.937. For Flink, the ratio decreases as the number of nodes increases. The average ratio starts at 0.873 with 16 nodes, and then decreases to an average of 0.521 with 120 nodes.

7.6 We keep the first edges of queries \( (kq) \) with probabilities \( \{0.01, 0.1, 1\} \). We also vary \( |V| \) to range from 25,000 to 150,000. This figure is a plot of edges/second versus the number of nodes.

7.7 We vary the rate at which the first edges of queries are kept \( (kq = \text{keep queries}) \) and plot the edges per day versus the rate at which triangles occur.
Chapter 1

Introduction

Graphs, sets of entities interconnected with edges, are a popular and powerful method for representing many problems. Cyber systems [109, 154], where machines communicate with each other through network interfaces, naturally create interaction graphs over time. Additionally, social networks [54, 140], the electrical grid [79], protein-protein interaction networks [193, 103], neuronal dynamics [31], and many other domains are all naturally modeled as graphs. While there is a rich history and theory behind graphs, analyzing their temporal nature is less well understood, and the temporal dimension can be critical to analyzing and detecting patterns of interest. In particular, activity within cyber systems can be naturally described as a graph with temporal information on the edges. The machines in the network are the nodes, while communications between machines form edges with several properties, one of them being the interval of time during which the communication occurred.

Another dimension to consider is the data generation rate. For example, cyber data produced on high speed networks allow for 100 Gb/s links and multiple-Tb/s switches, creating a vast stream of data that is challenging to analyze. A category of research called streaming and semi-streaming [144, 59] has produced algorithms with strict spatial and temporal constraints to deal with infinite streams, but these operators have largely been narrowly customized to specific problems, and not incorporated into a larger framework for data analysis. There is also a wealth of projects with streaming APIs [12, 190, 188, 95, 172, 11, 78, 13, 10, 64], but a standardized, efficient way for expressing streaming analysis has not arisen.
Figure 1.1: This dissertation is an intersection of four different research areas and presents a domain specific language (DSL) to express streaming temporal graph and machine learning queries.

The main contribution of this dissertation is to combine within one domain specific language (DSL) the ability to succinctly express

- graph operations,

- temporal patterns,

- machine learning pipelines, and

- streaming operators.

We call this DSL the *Streaming Analytics Language* or SAL. SAL can be applied to any stream of data. The only requirement is that the stream be a sequence of tuples. However, to flesh out the details of SAL, we focused our efforts on analyzing network traffic and detecting malicious behavior.

Network analysis has all the components previously discussed. Machines interacting with each other on a network produce a graph, where the nodes are Internet Protocol endpoints (IP’s) and the edges are communications between IP’s. Each communication has a start time and a duration, fulfilling the temporal aspect of the problem. Additionally, there are communication patterns among nodes, i.e. subgraphs, that are of interest and may indicate malicious behavior, and these patterns have a temporal dimension to them that are vital to correctly identifying the
subgraph. For instance, say node $A$ talks to node $B$ at time $t_1$ and $B$ talks to $C$ at time $t_2$. You might infer causality if $t_1 < t_2$, but the opposite is not true, even though the topological structure is the same either way. There are several approaches that utilize the topological graph structure of cyber interactions [91, 97, 53, 113, 178, 109, 191, 218, 206, 145] or the temporal dimension of the communication [23, 53, 196, 109, 191, 85, 206, 164]. We seek to combine these ad-hoc approaches within a unifying framework, namely SAL.

In terms of machine learning, there are many approaches that use machine learning to classify network traffic. For example:

- botnet detection [138, 23, 84, 223, 198, 105, 184]
- intrusion detection [30, 185, 124, 167]
- classifying network traffic by application [101, 26, 146, 212, 57, 214]
- insider threat detection [134, 69, 92]

An overall theme behind these works is to extract meaningful features and then perform machine learning, either in an unsupervised or supervised approach. The contribution of this work is to express these myriad approaches within a single DSL and framework.

Finally, cyber systems quickly become a streaming problem as the size of the network grows. Data is generated with such speed that often a single pass over the data is all that can be accomplished. Additionally, tight spatial and temporal complexity constraints are required to keep pace with data creation, and to not overwhelm limited computational resources. There are a number of streaming operators that have been published [16, 72, 132, 46, 221, 14, 48, 16, 221, 46, 48, 42, 131]. SAL provides these streaming operators as first-class citizens in the language.

SAL can be categorized into two types of computations: vertex-centric and subgraph matching. For vertex-centric, the data is partitioned by vertices in the graph, and calculations are computed based upon the tuples belonging to the vertex. More formally, let $T$ be an infinite sequence of tuples that have a source vertex and a destination vertex, i.e. each tuple is an edge with
vertices defined along with potentially other fields. Let $V$ be the set of all vertices. Let $T_v$ be the sequence of tuples where $t \in T_v \Rightarrow \text{source}(t) = v \lor \text{target}(t) = v$ for some $v \in V$. We refer to vertex-centric computations as any function $f$ that is computed continuously over each sequence $T_v$ for some subset of $V$.

Subgraph matching is finding subgraphs that fit topological, vertex, and edge constraints. Topological constraints define a subgraph isomorphism problem between a graph $G = (V, E)$ and a query graph $H = (V', E')$. The task is to find all subgraphs $g \in G$ such that there is a bijection $f$ that maps the vertices and edges of $g$ to the vertices and edges of $H$. Namely, let $g = (\hat{V}, \hat{E})$ where $\hat{V} \subseteq V$ and $\hat{E} \subseteq E \cap (\hat{V} \times \hat{V})$. Then for $g$ to be isomorphic to $H$, there must exist a bijection $f : \hat{V} \to V'$ such that $(a, b) \in \hat{E} \Rightarrow (f(a), f(b)) \in E'$. Vertex constraints define further requirements on the properties of the vertices of $g$ that must be fulfilled for a match. Often, features are derived for vertices using vertex-centric computations, and these features are then used to further refine the subgraph query. Edge constraints define requirements on the fields within an edge and between edges. For example, one can compare the start time of one edge, $\text{start}(e_0)$ to the start time of another edge, $\text{start}(e_1)$, to ensure temporal ordering such as $\text{start}(e_0) < \text{start}(e_1)$. These two components of SAL, vertex-centric and subgraph matching, provide an expressive base from which to succinctly describe many of the analyses discussed earlier.

Besides the language itself as a contribution, we also present a scalable implementation that translates SAL into C++ code that runs in parallel on a cluster of machines. We call this interpreter the Streaming Analytics Machine, or SAM. The code for SAM is over 19,000 lines, and scales to 128 nodes or 2560 cores and can process one million edges per second from streaming netflow data.

The rest of this dissertation is organized as follows: Chapter 2 introduces SAL and the various ways of expressing streaming computations. Chapter 3 gives example SAL programs to express a variety of cyber use cases. Chapter 4 discusses related work. Chapter 5 covers the implementation of SAM, and how SAL maps to SAM. Chapter 6 discusses performance achieved via vertex-centric computations, both in terms of scalability and classifier accuracy. Chapter 7 presents our results with subgraph matching. Chapter 8 concludes.
Chapter 2

The Streaming Analytics Language

In this chapter we present the Streaming Analytics Language (SAL), define the different aspects of the language, and describe case studies and examples to illustrate the concepts. SAL is a combination of imperative statements used for feature extraction and declarative, SPARQL-like statements, used to express subgraph queries.

The streaming aspect of this work introduces important constraints on the design of SAL and what operations we allow to be expressed. We utilize streaming algorithms, which is an area of research where the desire is to keep the space and temporal requirements to be polylogarithmic, i.e. $O((\log n)^k)$ for some $k$ where $n$ is the size of the sliding window over which computations are performed [144]. Sometimes those constraints, in particular the temporal complexity, are relaxed [130]. Several algorithms have been published within these constraints:

- K-medians [16]
- Frequent Items [72]
- Mean/Frequency Counts [132, 46, 221]
- Quantiles [14]
- Rarity [48]
- Variance [16, 221]
- Vector norms [46]
• Similarity [48]
• Count distinct elements [42, 141]

We expose these algorithms as first-class citizens in SAL.

The motivation behind the tight spatial bounds is that we are operating over an infinite stream of data where the ingest rates prohibit storing the entirety of the data within a sliding window. For example, a common structure of many vertex-centric SAL programs is represented in Figure 2.1. There is a stream of edges that is partitioned into \( N \) streams representing a stream for each vertex. Each vertex has \( o \) operators applied to it, each of which uses a window of length \( n \). Assume that the \( o \) operators utilize different aspects of the edge data, i.e. they don’t share the same data, and that each element of the window requires \( b \) bytes. Non-streaming approaches would require \( N \times o \times n \times b \) bytes while streaming approaches require \( N \times o \times \log(n) \times b \) bytes. Given example numbers of a million vertices, 10 operators, a window size of 5000, and 4 bytes per vertex, non-streaming approaches would necessitate about 190 GBs while streaming approaches require only 500 MBs. While 190GBs could reasonably fit within memory of a single system, it doesn’t take too much additional complexity to make the program unable to reside within memory. Our implementation of operators within SAL all have polylogarithmic spatial complexity.

Figure 2.1: This figure represents a common pattern for vertex-centric computations in SAL programs. A stream of edges is partitioned into \( N \) vertices that are then operated on by \( o \) operators that compute over a sliding window of length \( n \) items.
While some streaming algorithms are defined to operate on the entire stream, we focus on the sliding window model, where only recent inputs contribute to feature calculation. We believe the sliding window model is more appropriate for cyber data, where the environment is constantly changing over time and threats evolve and signatures are dynamic. The sliding window model can also be subcategorized into either a window defined by a particular time duration, or by the last \( n \) items. Currently SAL only supports expressing windows over the last \( n \) items, but many of the underlying algorithms are easily adapted to temporal windows, so it would not be difficult to add support for temporal windows.

These streaming operators are specified with \texttt{FOREACH} statements, which have the following form:

\[
<\text{FeatureName}> = \texttt{FOREACH} <\text{StreamName}> \texttt{GENERATE} <\text{Operator}>
\]

For all logical streams contained by \(<\text{StreamName}>, \text{the } <\text{Operator}> \text{ is applied to each stream, creating a time-varying feature indexed by each logical stream.}

The operators that can be specified are listed below:

- **ave**: Returns the estimated average value over the sliding window using an exponential histogram approach \[47\]. If not using default values, the operator has the form \( \text{ave}(f, n, k) \) where \( f \) is the name of the field in the tuple, \( n \) is the size of the sliding window, and \( k \) is a parameter of the exponential histogram model that determines the size of each bin in the histogram.

- **sum**: Returns the estimated sum over the sliding window. It also uses an exponential histogram and has the same form \( \text{sum}(f, n, k) \).

- **var**: The estimated variance. We again use the exponential histogram approach with the same signature, \( \text{var}(f, n, k) \).

- **topk**: Returns the estimated top \( k \) most frequent items. We use an implementation of Golab et al. \[73\]. It has the form \( \text{topk}(f, n, b, k) \) where \( f \) is the targeted field, \( n \) is the size
of the sliding window, \( b \) is the size of the basic window, and \( k \) is how many frequent items to report.

- **median**: The estimated median value. Arasu and Manku [14] provide an approach for estimating quantiles to find a median value.

- **countdistinct**: This creates an estimate on the number distinct elements. Metwally et al. [141] and Clifford and Cosma [42] produced algorithms that can be used for this calculation.

Another related area of research is semi-streaming [144, 59], which is a relaxation from polylog spatial requirements to \( O(N \cdot \text{polylog} N) \) for graph algorithms where \( N \) is the number of vertices in the graph. Many graph algorithms cannot be calculated on anything less than \( O(N \cdot \text{polylog} N) \) space, hence the need to relax the requirement. While we do not implement any semi-streaming algorithms, it should be noted that many SAL vertex-centric programs have the general form of a semi-streaming operation. For example, there are \( N \) different streams, one for each vertex. For each of the \( N \) streams we compute a \( O(\text{polylog} n) \) spatial complexity streaming operation, where \( n \) is the window size. Overall, the entire pipeline has spatial complexity \( O(N \cdot \text{polylog} n) \), which closely resembles the definition of semi-streaming.

We will introduce SAL with an example, provided by Listing 2.1, which specifies a query to match a type of malicious pattern known as a “Watering Hole Attack”. In a Watering Hole Attack, an attacker infects websites (e.g. via malvertisements) that are frequented by the targeted community. When someone from the targeted community accesses the website, their machine becomes infected and begins communicating with a machine controlled by the attacker. Figure 2.2 illustrates the attack as set of nodes and edges.
Listing 2.1: SAL Code Example

```sal
// Preamble Statements
WindowSize = 1000;

// Connection Statements
Netflows = VastStream("localhost", 9999);

// Partition Statements
PARTITION Netflows BY SourceIp, DestIp;
HASH SourceIp WITH IpHashFunction;
HASH DestIp WITH IpHashFunction;

// Defining Features
Top1000 = FOREACH Netflows GENERATE topk(DestIp, 100000, 10000, 1000);

// Subgraph definition
SUBGRAPH ON Netflows WITH source(SourceIp)
AND target(DestIp)
{
    trg e1 bait;
    trg e2 controller;
    start(e2) > end(e1);
    start(e2) - end(e1) < 10;
    bait in Top1000;
    controller not in Top1000;
}
```

In this SAL program there are five parts:

- Preamble Statements
- Partition Statements
Figure 2.2: Illustration showing the subgraph nature of a Watering Hole Attack. The target machine accesses a popular site, Bait. Shortly after accessing Bait, Target begins communicating to a new machine, the Controller.

- Connection Statements

- Feature Definition

- Subgraph Definition

Preamble statements allow for global constants to be defined that are used throughout the program. In the above listing, line 2 defines the default window size, i.e. the number of items in the sliding window.

After the preamble are the connection statements. Line 8 defines a stream of netflows called Netflows. VastStream tells the SAL interpreter to expect netflow data of a particular format (we use the same format for netflows as found in the VAST Challenge 2013: Mini-Challenge 3 dataset [201]). Each participating node in the cluster receives netflows over a socket on port 9999. The VastStream function creates a stream of tuples that represent netflows. For the tuples generated by VastStream, keywords are defined to access the individual fields of the tuple, listed below. A more complete definition can be found in the data description provided by VAST.

- SamGeneratedId: Each netflow is assigned a unique ID by SAM.

- Label: This field is for when a netflow has a label, such as when we are processing labeled
data, or when SAM assigns a label during testing. This is often used for supervised machine learning where individual netflows are labeled as malicious or benign.

- **TimeSeconds**: The number of seconds since the epoch plus microseconds as a decimal value.

- **ParseDate**: A date that is easier to read.

- **DateTime**: A numeric version of **ParseDate**

- **IpLayerProtocol**: The IP protocol number.

- **IpLayerProtocolCode**: The text version of the IP protocol.

- **SourceIp**: The source IP.

- **DestIp**: The destination IP.

- **SourcePort**: The source port.

- **DestPort**: The destination port.

- **MoreFragments**: Used when the flow is a long running data stream. Nonzero when more fragments will follow.

- **CountFragments**: If nonzero, indicates that this record is not the first for the flow.

- **DurationSeconds**: An integer indicating the time in seconds that the flow existed.

- **SrcPayloadBytes**: The sum of bytes that are part of the payload coming from the source.

- **DestPayloadBytes**: The sum of bytes that are part of the payload coming from the destination.

- **SrcTotalBytes**: The total bytes coming from the source.

- **DestTotalBytes**: The total bytes coming from the destination.
- **SrcPacketCount**: The total number of packets coming from the source.

- **DestPacketCount**: The total number of packets coming from the destination.

- **RecordForceOut**: Indicates that the record was pushed out before shutdown, and may be incomplete.

There are several different standard netflow formats. SAL currently supports only one format, the one used by VAST [201]. Adding other netflow formats is a straightforward task. In fact, SAL can easily be extended to process any type of tuple. Using our current SAL interpreter, SAM (see Chapter 5), the process is to define a C++ `std::tuple` with the required fields and a function object that accepts a string and returns an `std::tuple`. Once a mapping is defined from the desired keyword (e.g. VastStream to the `std::tuple`), this new tuple type can then be used in SAL connection statements. The mapping is defined via the Scala Parser Combinator [175] and is discussed in more detail in Chapter 5.

Following the connection statements is the definition of how the tuples are partitioned across the cluster. Line 8 specifies that the netflows should be partitioned separately by SourceIp and DestIp. Each node in the cluster acts as an independent sensor and receives a separate stream of netflows. These independent streams are then re-partitioned across the cluster. In this example, each node is assigned a set of Source IP addresses and Destination IP addresses. IP addresses are assigned to nodes using a common hash function. The hash function used is specified on lines 9 and 10, called the **IpHashFunction**. This is another avenue for extending SAL. Other hash functions can be defined and mapped to SAL constructs, similar to how other tuples can be added to SAL. The process is to define a function object that accepts the tuple type and returns an integer, and then map the function object to a keyword using the Scala Parser Combinator.

The next part of the SAL example program defines features. On line 13 we use the `FOREACH GENERATE` statement to calculate the most frequent destination IPs across the stream of netflows. The second parameter defines the total window size of the sliding window. The third parameter defines the size of a basic window [73], which divides up the sliding window into smaller
chunks. The fourth parameter defines the number of most frequent items to keep track of. In this case, the **FOREACH GENERATE** statement operates on a single stream. In Section 2.1 we will examine how the same statement can operate on multiple streams.

Finally, lines 19 and 20 define the subgraph of interest in terms of declarative, SPARQL-like statements that have the form `<node> <edge> <node>`. Lines 21 and 22 define temporal constraints on the edges, specifying that `e1` comes before `e2` and that from the end of `e1` to the start of `e2`, the total time is twenty seconds. Also, lines 23 and 24 define constraints on the vertices based on the previously defined feature **Top1000**. Vertex **bait** should be a frequent destination while **controller** should be infrequent.

We will cover other Domain Specific Languages in Section 4.2 but as a quick note, the language probably closest to the non-subgraph portion of SAL is Pig Latin. Pig Latin is an imperative language developed as a procedural analog to SQL. The developers of Pig found that imperative map-reduce style approaches were popular for data intensive parallel tasks, but too low-level and rigid, requiring significant development time and enabling low levels of reuse. Also, the declarative syntax of SQL was unnatural to the data analysts. Pig Latin’s purpose is to provide a high-level language that is easy to describe the flow of data over a set of operations, but that uses more abstract, higher-level constructs than map-reduce.

Pig Latin describes the flow of data over a set of operations, similar to SAL. However, given SAL’s focus on streaming data instead of batch processing, the semantics of the two languages diverge. The biggest difference is the **STREAM BY** statement. While Pig processes batches of tuples, one of SAL’s main use case is to divide a never-ending stream of tuples into separate substreams, to each of which a pipeline of operations is applied. Pig has a **FOREACH GENERATE** statement, but it’s purpose is to transform a set of tuples into another set of tuples, whereas for SAL, the semantics are to create a feature for each substream of data.

Concerning the graph-matching portion of SAL, we elected to use a declarative approach, where the user describes the subgraph of interest rather than specifying a procedural query plan for how the subgraphs are found. The language closest to our approach is SPARQL (a recursive
acronym for SPARQL Protocol and RDF Query Language), which is query language over Resource Description Framework (RDF) [1] data. In preliminary work [76] we mapped SPARQL code onto parallel vertex-centric programming frameworks, which are discussed in more detail in Section 4.1. From that experience we reasoned that it is easier to declare subgraphs of interest rather than create the plan to find them. Depending on the order of execution, the computational requirements can be orders of magnitude different. In our opinion, it is better to let a query optimizer handle the organization of the subgraph query plan.

While we use some concepts of SPARQL, namely basic graph patterns that describes the topological structure of the subgraph, there are aspects of SAL that significantly diverge from SPARQL and necessitate a separate approach. The differences between the requirements of SAL and SPARQL-related efforts are described in more detail in Section 4.6.

For the rest of this chapter we will do the following:

- Look at a case study of using SAL based on the problem of finding botnets in Section 2.1
- Integrate with static vertex attributes in Section 2.2
- Present temporal subgraph queries in Section 2.3

2.1 Case Study

In this section, we examine one approach for creating a classifier for detecting botnets, namely Disclosure [23]. This example will help elucidate how SAL can be used to create succinct representations of machine learning pipelines that previously were developed in an ad-hoc fashion. Also, it will demonstrate what cannot be expressed by SAL if we want to remain within the bounds of streaming calculations. Some of the features created by Disclosure require algorithms that do not comply with the desired constraints of streaming and semi-streaming. As such we have decided to not include those features within SAL at this time. If space is not a concern, SAL can always be extended to use non-streaming approaches that do not have polylogarithmic space constraints. In
any case, our intent with SAL is to winnow the stream of data to something more manageable for more intensive study. The full program is provided in Listing 2.2.

<table>
<thead>
<tr>
<th>Listing 2.2: SAL Disclosure Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Netflows = VastStream(&quot;localhost&quot;, 9999);</td>
</tr>
<tr>
<td>2 PARTITION Netflows By SourceIp, DestIp;</td>
</tr>
<tr>
<td>3 HASH SourceIp WITH IpHashFunction;</td>
</tr>
<tr>
<td>4 HASH DestIp WITH IpHashFunction;</td>
</tr>
<tr>
<td>5 VertsByDest = STREAM Netflows By DestIp;</td>
</tr>
<tr>
<td>6 Top2 = FOREACH VertsByDest GENERATE topk(DestPort, 10000, 1000, 2);</td>
</tr>
<tr>
<td>7 Servers = FILTER VertsByDest By Top2.value(0) + Top2.value(1) &gt; 0.9;</td>
</tr>
<tr>
<td>8 FlowsizeAveIncoming = FOREACH Servers GENERATE ave(SrcTotalBytes);</td>
</tr>
<tr>
<td>9 FlowsizeAveOutcoming = FOREACH Servers GENERATE ave(DestTotalBytes);</td>
</tr>
<tr>
<td>10 FlowsizeVarIncoming = FOREACH Servers GENERATE var(SrcTotalBytes);</td>
</tr>
<tr>
<td>11 FlowsizeVarOutcoming = FOREACH Servers GENERATE var(DestTotalBytes);</td>
</tr>
<tr>
<td>12 UniqueIncoming = FOREACH Servers GENERATE countdistinct(SrcTotalBytes);</td>
</tr>
<tr>
<td>13 UniqueOutgoing = FOREACH Servers GENERATE countdistinct(DestTotalBytes);</td>
</tr>
<tr>
<td>14 DestSrc = STREAM Servers By DestIp, SourceIp;</td>
</tr>
<tr>
<td>15 TimeLapseSeries = FOREACH DestSrc TRANSFORM</td>
</tr>
<tr>
<td>16 (TimeSeconds - TimeSeconds.prev(1)) : TimeDiff;</td>
</tr>
<tr>
<td>17 TimeDiffVar = FOREACH TimeLapseSeries GENERATE var(TimeDiff);</td>
</tr>
<tr>
<td>18 TimeDiffMed = FOREACH TimeLapseSeries GENERATE median(TimeDiff);</td>
</tr>
<tr>
<td>19 DestOnly = COLLAPSE TimeLapseSeries By DestIp For TimeDiffVar, TimeDiffMed;</td>
</tr>
<tr>
<td>20 AveTimeDiffVar = FOREACH DestOnly GENERATE ave(TimeDiffVar);</td>
</tr>
<tr>
<td>21 VarTimeDiffVar = FOREACH DestOnly GENERATE var(TimeDiffVar);</td>
</tr>
</tbody>
</table>

Many botnets have one or a small set of command and control servers that issue commands to a large number of infected machines, that then perform attacks such as distributed denial-of-service, stealing data, spam [183], or even use the computational resources for Bitcoin mining [102]. Disclosure focuses on identifying command and control (C&C) servers within a botnet.
The first part of the Disclosure pipeline is to identify servers. They define servers as an IP address where the top two ports account for 90% of the flows. Server definition is handled by lines 5 - 7. Line 5 demonstrates the use of a `STREAM BY` statement. It logically divides the input stream (Netflows) into multiple streams by a set of keys, which in this case is a single key, `DestIp`. The `FOREACH GENERATE` statement on line 6 then operates on each of these individual streams. It calculates an estimate of the top two ports that receive traffic for each destination IP using the `topk` operator. We then follow that with a filter on line 7. The `value(n)` function returns the frequency of the `n`th most frequent item (zero-based indexing). Netflows where the destination IP has two ports that account for 90% of the traffic to that IP are allowed through. Figure 2.3 demonstrates the flow graphically.

![Diagram](image)

Figure 2.3: This figure demonstrates the use of the `STREAM BY` statement to logically divide a single stream into multiple streams. The `STREAM BY` statement is followed by a `FOREACH GENERATE` statement that creates estimates on the top two most frequent destination ports and their frequencies. The `FILTER` statement uses this information to downselect to IPs where the top two ports account for 90% of the traffic.

Once the servers have been determined, the authors of Disclosure hypothesized that flow size distributions for C&C servers are distinguishable from benign servers. An example they give is that C&C generally have a limited number of commands, and thus flow sizes are drawn from a small set of values. On the other hand, benign servers will generally have a much wider range of values. To detect these differences between C&C servers and benign servers, they create three different types
of features based on flow size:

- **Statistical Features**
- **Autocorrelation**
- **Unique flow sizes**

For the statistical features, they extract the mean and standard deviation for the size of incoming and outgoing flows for each server. This is easy to express in SAL, and can be found on lines 8-11. For each IP in the set of servers, we use the `FOREACH GENERATE` statement combined with either the `ave` operator or `var` operator.

The Disclosure authors also generate features using autocorrelation on the flow sizes. The idea is that C&C servers often have periodic behavior that would be evident from an autocorrelation calculation. For each server, the flow sizes coming in with each netflow can be thought of as a time series. They divide this signal up into 300 second intervals and calculate the autocorrelation of the time series. Unfortunately, we are not aware of a polylogarithmic streaming algorithm for calculating autocorrelation. As such, we currently do not allow autocorrelation to be expressed within SAL; mixing algorithms with vastly different spatial and temporal complexity requirements would negate many of the benefits of the language. For a problem with $N$ vertices, calculating autocorrelation for the each vertex would require $O(N \cdot n)$ space where $n$ is the length of the window. Applying polylogarithmic streaming operators to $N$ vertices would require $O(N \cdot \log(n)^k)$ space, offering significant savings and calling into question the scalability of Disclosure to handle large problems. That said, there is nothing preventing an extension of SAL for an autocorrelation operation, just the documentation would need to be clear about the complexity.

The final set of features based on flow-sizes involves finding the set of unique flow sizes. The hypothesis is that botnets have a limited set of messages, and so the number of unique flow sizes will be smaller than a typical benign server. To find an estimate on the number of unique flow sizes, one can use the `countdistinct` operator, as shown on lines 12-13. However, Disclosure goes a step further than just the number of unique flow sizes. They create an array with the counts for
each unique element and then compute unspecified statistical features on this array. While this is not exactly expressible in SAL, and having an exact answer would break our space constraints, one could use TopK to get an estimate on the counts for the most frequent elements.

Besides features based on flow size, Disclosure also computes features focusing on the client access patterns. The hypothesis is that all the bots accessing a particular C&C server will exhibit very similar behavior, while the behavior of clients accessing benign servers will not be so uniform. Disclosure defines a time series for each server-client pair by calculating the inter-arrival times between consecutive connections. For example, if we had $n$ netflows that occurred at times $t_0, t_1, \ldots, t_n$, then the series would be $t_1 - t_0, t_2 - t_1, \ldots, t_n - t_{n-1}$. To specify this time series in SAL, one uses the TRANSFORM statement. The TRANSFORM statement allows one to transform from one tuple representation to another.

For this example, we need to transform from the original netflow tuple representation to a tuple that has three values: the $\text{SrcIp}$, $\text{DestIp}$, and the inter-arrival time. On line 14, we use the STREAM BY statement to further split the stream of netflows into source-destination IP pairs. Line 16 then follows with the the TRANSFORM statement that calculates the inter-arrival time. Since $\text{SourceIp}$ and $\text{DestIp}$ are the keys defined by the STREAM BY statement, those values are included by default as the first two values of the newly defined tuple. This part of the pipeline is represented in the left side of Figure 2.4.

On line 16 we introduce the $\text{prev}(i)$ function. The $\text{prev}(i)$ function returns the value of the related field $i$ items back in time. So $\text{TimeSeconds}.\text{prev}(1)$ returns the value of $\text{TimeSeconds}$ in the tuple that occurred just previous to the current tuple, thus giving us the inter-arrival time. The colon followed by $\text{TimeDiff}$ gives a label to the tuple value and can be referred to in later SAL statements (e.g. line 17).

Now that we have the time series expressed in SAL, we can then add the feature extraction methods that Disclosure performs on the inter-arrival times. Disclosure calculates the minimum, maximum, median, and standard deviation. The median and standard deviation can be expressed in SAL, and are found on lines 17 and 18.
Figure 2.4: This figure demonstrates the use of the **TRANSFORM** and **COLLAPSE BY** statements to define parts of the Disclosure pipeline.

However, maximum and minimum are not currently supported in SAL. The reason is that max and min provably require $O(n)$ space where $n$ is the size of the window when computing over a sliding window\(^4\). As discussed previously, we made the design decision to limit operators to be polylogarithmic so that when we compute over $N$ vertices, the total space complexity is $O(N \cdot \log(n)^k)$ instead of $O(N \cdot n)$. When computing over the entire data stream, to find the max/min one can keep track of one number, but the sliding window adds complexity as the max/min expires. A mixture between the two models, over the entire stream and sliding windows, might be sufficient for creating features. For example, one could use the scheme presented below to estimate the maximum value:
Here, we create a sequence of max values, $\text{max}_0, \text{max}_1, \ldots$, where each $\text{max}_i$ is calculated during the presentation of $n$ consecutive items; however, only two values need to be stored at any one time, $\text{max}_i$ and $\text{max}_{i+1}$. When querying for the current max after $x$ items, then the index of the maximum value for item $x$ is

$$f(x) = \begin{cases} 0, & \text{if } x < \frac{n}{2}. \\ \lfloor \frac{2}{n} x - 1 \rfloor, & \text{otherwise}. \end{cases} \quad (2.1)$$

where $n$ is the size of the window. This scheme would allow for constant space requirements and $O(n)$ processing time; however, it remains to be seen if max/min features extracted in this manner are useful in practical situations and we leave it as future work.

For the features derived from the time series to be applicable to classifying servers, we need to combine the features from all the clients. To do so, we no longer use \texttt{SourceIp} as one of the keys to split the data flow by using the \texttt{COLLAPSE BY} statement. The \texttt{BY} clause contains a list of keys that are kept. Unspecified keys are removed from the key set. On line 19 \texttt{DestIp} is kept while \texttt{SourceIp} is not specified, meaning that it is no longer used as a key to split the data flow.

The semantics of the \texttt{COLLAPSE BY} statement is the following: Let $k_+$ be the tuple elements that are kept by \texttt{COLLAPSE}, and let $k_-$ be the tuple elements that are no longer used
as a key. For each tuple $t$ that appears in the stream $S$ of data, let $k_+(t)$ be the subtuple with only tuple elements from $k_+$, and let $k_-(t)$ be the subtuple with only tuple elements from $k_-$. Also, let $r(t)$ be the subtuple with remaining elements that are neither in $k_+(t)$ nor in $k_-(t)$. Define $L$ to be the set of unique $k_+(t)$ for all $t \in S$, i.e.

$$L = \bigcup_{t \in S} k_+(t)$$  \hspace{1cm} (2.2)

For each $l \in L$, we define another set, $M_l$:

$$M_l = \bigcup_{t \in S, k_+(t) = l} k_-(t)$$  \hspace{1cm} (2.3)

**COLLAPSE BY** creates a mapping for each set $M_l$, mapping the elements of $M_l$ to the most recently seen $r(t)$ associated with each $m \in M_l$. These mappings can then be operated on by the **FOREACH GENERATE** statement.

As an example, say we have the following **DestIp**, **SourceIp** pairs with the following values for **TimeDiffMed**:

<table>
<thead>
<tr>
<th>Netflow #</th>
<th>DestIp</th>
<th>SourceIp</th>
<th>TimeDiffMed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>192.168.0.1</td>
<td>192.168.0.100</td>
<td>1.5</td>
</tr>
<tr>
<td>$i + 1$</td>
<td>192.168.0.2</td>
<td>192.168.0.100</td>
<td>5</td>
</tr>
<tr>
<td>$i + 2$</td>
<td>192.168.0.1</td>
<td>192.168.0.101</td>
<td>2.1</td>
</tr>
<tr>
<td>$i + 3$</td>
<td>192.168.0.1</td>
<td>192.168.0.100</td>
<td>2</td>
</tr>
</tbody>
</table>

The following graphic illustrates the result of applying line [19] to the above data:
After processing tuple $i$, there will be one map, for destination IP 192.168.0.1, and it will have one key-value pair, $<192.168.0.100, 1.5>$. After tuple $i + 1$, there will be two maps, one for 192.168.0.1 and one for 192.168.0.2. The map for 192.168.0.2 will have the key-value pair $<192.168.0.100, 5>$. After tuple $i + 2$, there will be another element in the map for 192.168.0.1, namely $<192.168.0.101, 2.1>$. After tuple $i + 3$, in the map for 192.168.0.1, the value for key 192.168.0.100 is replaced with 2.

Once we have collapsed back to DestIp, we can then calculate statistics across the set of clients for each server. The Disclosure paper does not specify which statistics are calculated, but we assume standard operators such as average, variance, and median were used, all of which can be expressed in SAL. Lines 20 - 21 provide some examples.

That concludes our exploration of how SAL can be used to express concepts from a real pipeline defined in another paper. While there are some operations that cannot be expressed, e.g. the autocorrelation and max/min features, most of the features could be expressed in SAL. We believe SAL provides a succinct way to express streaming machine learning pipelines.

To use this pipeline for machine learning purposes, a SAL pipeline can have two operating modes: training and testing. In the training mode, the pipeline is run against a finite set of data, usually with labels, and any features created by the pipeline will be appended per input tuple. This feature set along with the labels is used to train a classifier offline. The testing phase then applies the pipeline to an online stream of data, transforming each input tuple into defined feature set, and then applies the trained classifier to the feature set to assign a label.

### 2.2 Static Vertex Attributes

The previous section outlined features that can be defined and that depend upon the stream of data to calculate them. However, there may exist attributes associated with vertices that remain relatively static. A common use case we’ve found is knowing when a vertex is a local IP address and when it is outside the firewall. This is helpful in defining subgraphs of interest, further refining the meaning and constraining what is considered. For the Watering Hole Attack, we can add the
LocalInfo = VertexInfo(<connection info>)

and then within the subgraph section we can add

```plaintext
trg in Local;
bait not in Local;
controller not in Local;
```

This allows us to create the constraints that trg is behind the firewall while bait and controller are outside the firewall, further clarifying the intent of the query.

The general format for in and not in is `<node variable> <in|not in> <vertex info>` The in and not in check to see if the node variable is a defined key within the static vertex attributes. We can think of the Static Vertex Attributes as a table with a primary key. Attributes of the table are accessed via the following form: `<Attribute> (<key>). <Fieldname>`. For example, if we have field Type that informs what the purpose of an IP is, we can add the following types of constraints:

```plaintext
LocalInfo(trg).Type == "Server"
LocalInfo(pos).Type == "POS"
```

Where the first line indicates that the vertex trg is a Server and the second line indicates that the vertex pos is a POS or point-of-sale machine.

We will see more examples of Static Vertex Attributes when we walk through how SAL can express common cyber queries in Chapter 3.

2.3 Temporal Subgraph Matching

While the previous section focused on the streaming operators expressed as imperative statement in SAL, here we provide more details on subgraph pattern matching, which we later show to be polynomial in the number of possible edges over a sliding window. Subgraph matching uses declarative statements with a syntax inspired by SPARQL [166]. A SAL subgraph matching query
has the following pieces:

- **Graph Definition**: SAL defines a stream of tuples, e.g. **Netflows**, but in order to perform subgraph matching, SAL needs to know what constitutes a vertex within the tuple. In particular, we support directed graphs where each edge has a direction, emanating from a source and terminating at a target. The definition of the graph is specified with a statement of the form

\[
\text{SUBGRAPH ON } <\text{Stream}> \text{ WITH } \text{source}(<\text{Source}>) \text{ AND } \text{target}(<\text{Target}>) \{ \ldots \}
\]

The tuple fields `<Source>` and `<Target>` must be of the same type. In our implementation there is a `std::tuple` that is mapped to a particular tuple type in SAL. The two fields must be of the same type in the `std::tuple`. Within the curly braces, `{ ... }`, the following three types of statements are made.

- **Edge Definition**: These are statements that define the topological structure of the graph and come in the form `<node> <edge> <node>`. Each element is a variable name for a node or an edge. Variable names are alphanumeric but begin with an alphabetic character.

- **Edge Constraints**: Using the keywords defined for the tuple, one can form a constraint on edges in terms of a comparison of two arithmetic expressions:

\[
<\text{EdgeConstraint}> := <\text{Expr}><\text{Comp}><\text{Expr}>

<\text{Expr}> := <\text{Term}>
| <\text{Expr}> + <\text{Term}>
| <\text{Expr}> - <\text{Term}>

<\text{Term}> := <\text{Factor}>
| <\text{Term} > * <\text{Factor}>
| <\text{Term} > / <\text{Factor}>

<\text{Factor}> := (<\text{Exp}>)| - <\text{Factor}>
| <\text{Variable} > . <\text{Field}>

\text{start}(<\text{Variable}>)|\text{end}(<\text{Variable}>)|<\text{RealNumber}>
|<\text{String}>

<\text{Comp}> := < | <= | >= | ==
\]

In order to evaluate correctly, the variable name must match a variable specified for an edge in an Edge Definition statement.
• Vertex Constraints: One can also define constraints on vertices:

\[
< \text{VertexConstraint} > := < \text{node} > < \text{VertexComp} > < \text{FeatureName} > | \\
< \text{node} > < \text{VertexComp} > < \text{VertexInfo} > | \\
< \text{VertexInfo} > ( < \text{node} > ), < \text{FieldName} > \\
< \text{Comp} > < \text{RVertexInfo} >
\]

\[
< \text{VertexComp} > := \text{in} \mid \text{not in}
\]

\[
< \text{RVertexInfo} > := < \text{String} > \mid < \text{Float} > \mid < \text{Integer} >
\]

Vertex constraints have three basic forms. The first form checks to see if a vertex is found in a previously defined feature, e.g. \textit{bait in Top1000} checks to see if the vertex \textit{bait} is found in the feature \textit{Top1000}. In order to evaluate correctly, the node variable must refer to a vertex variable defined in an Edge Definition statement, the feature variable must refer to a variable defined by a Feature Definition Statement and the key of the feature must be of the same type as the vertex type. The second form is similar to the first, except it checks to see if the vertex variable is a key in a defined VertexInfo table. The third form accesses a field of a VertexInfo table with the vertex variable as a key and checks that the it satisfies comparison operation.

With overall process for subgraph definition outlined, we now discuss the complexity of the problem. We begin with a discussion on subgraph isomorphism and how that problem relates to temporal subgraph matching.

Subgraph isomorphism, the decision problem of determining for a pair of graphs, \(H\) and \(G\), if \(G\) contains a subgraph that is isomorphic to \(H\), is known to be NP-complete. Additionally, the problem of finding all matching subgraphs within a larger graph is NP-hard [122]. However, when we consider a streaming environment with bounds on the temporal extent of the subgraph, the problem becomes polynomial. We introduce some definitions to lay the groundwork for discussing complexity.
**Definition 1.** Graph: $G = (V, E)$ is a graph where $V$ are vertices and $E$ are edges. Edges $e \in E$ are tuples of the form $(u, v)$ where $u, v \in V$.

**Definition 2.** Temporal Graph: $G = (V, E)$ is a **temporal graph** where $V$ are vertices and $E$ are temporal edges. Temporal edges $e \in E$ are tuples of the form $(u, v, t, \delta)$ where $u, v \in V$ and $t, \delta \in \mathbb{R}$ and represent the start time of the edge and its duration, respectively.

**Definition 3.** Streaming Temporal Graph: A graph $G$ is a **streaming temporal graph** when $G$ is a temporal graph where $V$ is finite and $E$ is infinite.

To express subgraph queries against a temporal graph, we need the notion of temporal constraints on edges. We previously introduced constraints on edges which have the general form of two arithmetic expressions involving tuple fields, i.e.

$$< Constraint > := < Expr > < Comp > < Expr >$$

Temporal constraints are simply constraints where the expressions involve the temporal fields of the tuples. For convenience we define $\text{start}()$ to return the time that the edge started in seconds, and $\text{end}()$ as the time that the edge ended. Otherwise, to obtain a value of a tuple field, one uses the form $< variable > . < field >$, e.g. $e1.TimeSeconds$. Now we define **Temporal Subgraph Queries** using temporal constraints.

**Definition 4.** Temporal Subgraph Query: For a temporal graph $G$, a temporal subgraph query is composed of a graph $H$ with edges of the form $(u, v)$, where each of the edges may have temporal constraints defined. The result of the query is all subgraphs $H' \in G$ that are isomorphic to $H$ and that fulfill the defined temporal constraints.

Below is an example triangular temporal subgraph query:
The first part of the query with the variables $x_1$, $x_2$, and $x_3$ defines the subgraph structure $H$. In this example, we have defined a triangle. The second part of the query defines the temporal constraints. Edge $e_1$ starts before $e_2$, $e_2$ starts before $e_3$, and the entire subgraph must occur within a 10 second window from the start of the first edge to the start of the last edge. For temporal queries where there is a maximum window specified, then the problem of finding all matching subgraphs is polynomial. As far as we are aware, no one else has examined the complexity of subgraph matching within a streaming setting with temporal constraints.

**Theorem 1.** For a streaming temporal graph $G$, let $n$ be the maximum number of edges that arise during a time duration of length $\Delta q$. A temporal subgraph query $q$ with $d$ edges and maximum temporal extent $\Delta q$ and computed on an interval of edges of length $\Delta q$ has temporal and spatial complexity of $O(n^d)$.

**Proof.** For an interval of length $\Delta q$, there are at most $n$ edges. Assume each of those $n$ edges satisfy one edge of $q$. Then there are at most $n$ potential matching subgraphs in the interval. Now assume for each of those $n$ potential matching subgraphs, each of the $n$ edges satisfies another of the edges in $q$. Then there are $n^2$ potential matching subgraphs. If we continue for $d$ iterations, we have a max of $n^d$ matching subgraphs. Thus to find and store all the matching subgraphs, it would require $O(n^d)$ operations and $O(n^d)$ space to store the results. □
The point of Theorem 1 is to show that subgraph matching on streaming data is tractable, and that over any window of time that is about the length of the temporal subgraph query, computing over that window has polynomial complexity, as long as the number of edges in queries has some maximum value.

In this section we’ve introduced the main principles of expressing a subgraph query: defining the source and target fields, edge expressions, edge constraints, and vertex constraints. In the next chapter we will see examples of SAL applied to common patterns arising in cyber security.
Chapter 3

SAL as a Query Language for Cyber Problems

This chapter covers several cyber use cases and analyzes how they can be expressed as SAL programs. To ensure broad coverage, we use Verizon’s Data Breach Investigations Report (DBIR) of 2018 [203] as the basis for our discussion. The DBIR presents an overview of cyber security incidents and breaches. They define an incident as a security event that compromises the integrity, confidentiality or availability of an information asset. A breach is more substantial, and defined as an incident that results in the confirmed disclosure, not just potential exposure, of data to an unauthorized party. As of the 2018 report, Verizon had recorded 333,000 incidents and 16,000 breaches. They group these observed cyber attacks into nine different categories, namely

- Crimeware
- Cyber-Espionage
- Denial of Service
- Lost and Stolen Assets
- Miscellaneous Errors
- Payment Card Skimmers
- Point of Sale
- Privilege Misuse
• Web Applications

Verizon reports that 94% of incidents and 90% of breaches fall within the nine categories. They have another category, **Everything Else**, for attacks that don’t fit the nine. We will discuss how SAL could be used to define the patterns that comprise each category. However, we do not cover Lost and Stolen Assets (e.g. a stolen laptop) and Payment Card Skimmers (e.g. a skimming device physically implanted on an ATM), as these categories relate to physical security, which is out of scope. Also, we do not cover Miscellaneous Errors, which largely deals with misconfigurations (e.g. default passwords on a database).

For the code listings in this section, to avoid frequent repetition we assume that a **Netflows** stream has already been created, and that the data has been partitioned. For an example see lines 5-10 of Listing 2.1.

### 3.1 Crimeware

The Crimeware category has instances of malware that were largely *opportunistic in nature and are financially motivated*, but that didn’t fit a specific pattern. An example of this is the ZeroAccess botnet [209]. The ZeroAccess botnet had its heyday in 2012 when it had over one million infected hosts at its disposal. The motivations behind ZeroAccess were purely financial, with much of the revenue coming from click fraud. The botmasters would send lists of urls of advertisements, and the controlled machines would simulate browsers clicking on those advertisements. They would then collect the ad revenue from the generated activity. At its height, researchers estimate that ZeroAccess garnered around $100,000 a day from click fraud.

To detect click fraud in general, one could start with the hypothesis that the click-through behavior of bots is different than humans. It is probably difficult to know what traffic constitutes a click through, but we can monitor how clients interact with web servers. Humans interact with the web in Poisson-like or bursty distributions [202], while a bot would likely create relatively constant activity.
We begin by creating a definition of a web server in Listing 3.1, similar to our approach for Disclosure in Section 2.1. However, we focus on IPs where the majority of the traffic is directed towards port 80.

**Listing 3.1: Definition of Web Servers**

```plaintext
1 VertsByDest = STREAM Netflows BY DestIp;
2 Top1 = FOREACH VertsByDest GENERATE topk(DestPort,10000,1000,1);
3 Servers = FILTER VertsByDest BY Top1.value(0) > 0.6;
4 Servers = FILTER Servers BY Top1.key(0) == 80;
```

Also similar to the Disclosure use case, we use a `STREAM BY` statement to create logical streams of source/destination pairs in Listing 3.2. We then find the inter-arrival times and calculate statistics, again similar to Disclosure.

**Listing 3.2: Calculating Inter-arrival Times**

```plaintext
1 DestSrc = STREAM Servers BY DestIp, SourceIp;
2 TimeLapseSeries = FOREACH DestSrc TRANSFORM
3 (TimeSeconds - TimeSeconds.prev(1)) : TimeDiff;
4 TimeDiffVar = FOREACH TimeLapseSeries GENERATE var(TimeDiff);
5 TimeDiffMed = FOREACH TimeLapseSeries GENERATE median(TimeDiff);
6 TimeDiffAve = FOREACH TimeLapseSeries GENERATE ave(TimeDiff);
```

In contrast to Disclosure, we are interested in statistics about the clients, not the servers. We want to categorize clients into two bins: bot-generated and human-generated behavior. As such we collapse on the source IP, and calculate features based on the aggregates in Listing 3.3.
Listing 3.3: Calculating Time-based Client Statistics

1 \textbf{SrcOnly} = \textbf{COLLAPSE} \textbf{TimeLapseSeries} \textbf{BY \ SourceIp FOR} \textbf{TimeDiffVar, TimeDiffMed, TimeDiffAve} ;
2 \textbf{AveTimeDiffVar} = \textbf{FOREACH} \textbf{SrcOnly GENERATE} \textbf{ave} (\textbf{TimeDiffVar}) ;
3 \textbf{VarTimeDiffVar} = \textbf{FOREACH} \textbf{SrcOnly GENERATE} \textbf{var} (\textbf{TimeDiffVar}) ;
4 \textbf{AveTimeDiffMed} = \textbf{FOREACH} \textbf{SrcOnly GENERATE} \textbf{ave} (\textbf{TimeDiffMed}) ;
5 \textbf{VarTimeDiffMed} = \textbf{FOREACH} \textbf{SrcOnly GENERATE} \textbf{var} (\textbf{TimeDiffMed}) ;
6 \textbf{AveTimeDiffAve} = \textbf{FOREACH} \textbf{SrcOnly GENERATE} \textbf{ave} (\textbf{TimeDiffAve}) ;
7 \textbf{VarTimeDiffAve} = \textbf{FOREACH} \textbf{SrcOnly GENERATE} \textbf{var} (\textbf{TimeDiffAve}) ;

From these generated features, one could use a clustering-based approach to understand the data. In an ideal world, there would be one cluster for human-generated traffic and another for bot-generated traffic. However, it is more likely that an iterative process is required. In the case of the ZeroAccess bot, it was a peer-to-peer-based bot, and communication between peers used four particular ports: 16464, 16465, 16470, and 16471. In the early phases of analyzing data, this kind of peculiar communication pattern might become apparent from clustering on a first pass, and one can add a filter to look for that specific pattern as in Listing 3.4.

Listing 3.4: Suspicious Ports

1 \textbf{VertsByDest} = \textbf{STREAM} \textbf{Netflows BY DestIp} ;
2 \textbf{Top10Ports} = \textbf{FOREACH} \textbf{VertsByDest GENERATE} \textbf{topk} (\textbf{DestPort},10000,1000,10) ;
3 \textbf{BadClients} = \textbf{FILTER} \textbf{VertsByDest BY 16464 in Top10Ports} ;
4 \textbf{BadClients} = \textbf{STREAM} \textbf{BadClients BY SourceIp} ;
5 \textbf{Top1} = \textbf{FOREACH} \textbf{VertsByDest GENERATE} \textbf{topk} (\textbf{DestPort},10000,1000,1) ;
6 \textbf{Servers} = \textbf{FILTER} \textbf{VertsByDest BY Top1.value(0) > 0.6} ;
7 \textbf{Servers} = \textbf{FILTER} \textbf{Servers BY Top1.key == 80} ;
8 \textbf{Servers} = \textbf{FILTER} \textbf{Servers BY SourceIp in BadClients}

On line 4 we switch the key to stream by from \textbf{DestIp} to \textbf{SourceIp} so that on line 8 we can test if the \textbf{SourceIp} of the \textbf{Servers} traffic is in \textbf{BadClients}. At the end, \textbf{Servers} is filtered down to netflows where the destination IP is characterized as a web server and the client IP has
generated traffic on port 16464.

The ZeroAccess bot is a good example of how SAL could be used to test out hypotheses. One starts with an idea of what click fraud looks like from network behavior, and then we can begin to extract useful data and calculated features that allow researchers to test their hypotheses.

3.2 Cyber-Espionage

According to the Verizon report, this category involves malicious activity linked to state-affiliated actors and/or exhibiting the motive of espionage. They also state that 93% of breaches are attributed to state-affiliated groups or nation-states. In another report, Symantec advises that the number of targeted attack groups, i.e., groups that are professional, highly organized, and target specifically rather than indiscriminately, grew at a rate of 29 groups a year between the years of 2015 to 2017, from a total of 87 to 140. The evidence points to growing sophistication and technical capabilities of adversarial groups.

A common activity of cyber-espionage is the exfiltration of data. Joslyn et al. provide an exfiltration query that we express in terms of SAL. Figure 3.1 presents the overall graph structure. Listing 3.5 gives the SAL code. We assume a Static Vertex Attributes table named Local has

![Figure 3.1: A subgraph showing the flow of messages for data exfiltration. A controller sends a small control message to an infected machine, which then sends large amounts of data to a drop box.](image)
been defined and is a single column with IP addresses. The presence of the IP address in the table implies that the address is behind the firewall. Lines 4 and 5 defines the topology of the graph. The temporal constraints of lines 6 and 7 define the temporal ordering, that \( e_1 \) happens before \( e_2 \), and that time between the end of \( e_1 \) and the start of \( e_2 \) should be less than ten seconds. Lines 8 and 10 specify that drop and controller are not local, while target is a machine behind the firewall. Finally, line 11 states that the number of bytes coming from the controller to the target is relatively small (\(< 20\) bytes) while the data going from target to dropbox is large (\(> 1000\) bytes).

![Listing 3.5: Exfiltration Query](image)

3.3 Denial of Service

Denial of Service is an attack where the intent is to deny intended users access to a networked resource. Often, Denial of Service is a distributed undertaking, with large botnets employed to send enormous volumes of traffic to the victim, overwhelming resources so that legitimate users are unable to reach the site. This is known as Distributed Denial of Service, or DDoS. Jonker
et al. [108] note that one third of all active /24 networks experienced DDoS attacks over a two year period. Attacks have been seen passing 1 terabit per second, with an attack against Github peaking at 1.35 Tbps [56]. DDoS occurs with alarming frequency, with Jonker at al. reporting about 30,000 attacks per day.

The primary means of conducting a DDoS is through a botnet. There are even DDoS-as-a-Service offerings that allow you to rent a botnet for a DDoS attack [173]. One structure for botnets is the command and control variant (C&C). Below we list a potential query to identify a subgraph associated with denial of service. There is one controller node that sends out a message to multiple infected machines, which then send messages to a target node. The average size of the messages of the infected machines is small (e.g. a SYN flood). This query is a subgraph of the intended behavior, where we have many infected machines.
### Listing 3.6: Denial of Service Query

```
VerbsBySrc = STREAM Netflows BY SourceIp;
FlowsizeAve = FOREACH VerbsBySrc GENERATE ave(SrcPayloadBytes);
SmallMessages = FILTER VerbsBySrc BY FlowsizeAve < 10;
SUBGRAPH ON Netflows WITH source (SourceIp) AND target (DestIp) {
  controller e1 node1;
  controller e2 node2;
  node1 e3 target;
  node2 e4 target;
  start(e1) < start(e2);
  start(e2) < start(e3);
  start(e3) < start(e4);
  start(e4) - start(e1) < 10;
  node1 in SmallMessages;
  node2 in SmallMessages;
}
```

### 3.4 Point of Sale

Point of Sale (POS) intrusions cover attacks that are conducted remotely but ultimately target POS terminals and controllers. During a POS transaction, communication of the sensitive information is largely encrypted. The vulnerable time when credit card and other information is not encrypted is when it is loaded into memory of a POS terminal. Thus, most POS-associated malware scans the memory of running processes and performs a regular expression search for credit card information, which is then aggregated and communicated back to the attacker.

The Target Breach [180], which occurred in 2013, was one of the first large-scale POS breaches, where 40 million card credentials and 70 million customer records were stolen. While each POS breach has some unique elements, in particular how the POS malware is distributed across a
retailer’s network, the general pattern of how data is collected and communicated back to the attacker has many commonalities:

- POS malware skims the memory of running processes, using regular expressions to discover credit card information.

- Found credential information is sent to a server. In the case of the Target Breach, a small set of compromised servers behind Target’s firewall aggregated the information from a large number of POS terminals. The compromised servers sent the aggregated credit card information to an external address controlled by the attacker.

A basic SAL query that could cover this pattern is provided in Listing 3.7. In this example, we have a Static Vertex Attributes table name `Local` that has two columns, the key column with the IP addresses, and another column with label `type`. This allows us to check if an IP is behind the firewall, and if it is a server or a POS machine.

```
Listing 3.7: POS Subgraph Query

1 SUBGRAPH ON Netflows WITH source(SourceIp)
2     AND target(DestIp)
3 {
4     pos e1 server;
5     server e2 drop;
6     Local(pos).type == "POS";
7     Local(server).type == "Server";
8     drop not in Local;
9 }
```

### 3.5 Privilege Misuse

The privilege misuse category largely deals with insider threat. A brochure from the FBI lists several examples of insider threats, with a common theme of collecting many documents,
electronically or physically, sometimes on the order of hundreds of thousands. One pattern that could be considered suspicious is downloading/accessing many documents within a short period of time. For this query, we have a different graph than what we’ve used previously. We need enhanced semantic information than what is usually provided by netflows. Namely we need a way of identifying document accesses by individuals. This requires that we have authentication in place to determine who is generating the traffic, and that we can identify when a particular document is being accessed. For many companies and institutions, this type of infrastructure is already in place.

For the purposes of discussion, we will assume that there are tuples generated with the given fields:

- **PersonId**: the identifier of the person accessing the document.
- **DocumentId**: the identifier assigned to the document.
- **TimeSeconds**: the time in seconds from epoch that the access occurred.

We refer to this stream of tuples as a **DocumentStream**. We will assume that there is a hash function defined that will partition the data by PersonId. The SAL program in Listing 3.8 finds the average time between document accesses, and creates a filtered list of users that have average time between document accesses less than 0.1 seconds.

```
Listing 3.8: Malicious Insider Query

1 Documents = DocumentStream("localhost", 9999);
2 PARTITION Documents BY PersonId;
3 HASH PersonId WITH PersonHashFunction;
4 ByPerson = STREAM Documents BY PersonId;
5 TimeLapseSeries = FOREACH ByPerson TRANSFORM
6   (TimeSeconds - TimeSeconds.prev(1)) : TimeDiff;
7 TimeDiffAve = FOREACH TimeLapseSeries GENERATE ave(TimeDiff);
8 Downloaders = FILTER TimeLapseSeries BY TimeDiffAve < 0.1;
```
3.6 Web Applications

Web application attacks are defined as *any incident in which a web application was the vector of attack*. An attack that fits this category is that of Cross-site Scripting (XSS). A XSS attack is possible when a web site accepts unsanitized user input that is then displayed back on the website. For example, if a site accepts comments, a malicious actor can insert java script code that is then added to the HTML page as a comment. When other users visit the compromised url, the malicious javascript is run, opening the door for theft of cookies, eventlisteners that register keystrokes, and phishing through the use of fake/modified logins.

![Figure 3.2: A subgraph showing an XSS attack.](image)

The general pattern is similar to a Watering Hole Attack. Here a client browser accesses a frequent site or server. The downloaded HTML has malicious javascript embedded, which causes the client browser to access a relatively rare site and upload data to the attacker controlled site. Figure 3.2 presents the subgraph of interest. However, we can add additional details to make the query more specific to XSS. Listing 3.9 provides a query for XSS attacks in SAL. Lines 1-3 create a definition of a server, similar to the Disclosure query. On line 4 we create a reference to a Static Vertex Attributes table, **DomainInfo**. DomainInfo has two columns, the key column being IP, and another column being the associated domain. This allows us to test that the domain of the server or frequent site is not equal to the domain of the malicious IP (line 12).
It is also important to point out are lines 15 and 16 which accesses the application protocol being used. Most netflow formats do not include application protocol information; however, for some of the more common protocols (e.g. HTTP, HTTPS, SMTP) tools like Wireshark are relatively good at decoding application layer protocols. Thus it is conceivable that the netflow tuples could be augmented with common application level protocols.

**Listing 3.9: XSS Attack Query**

```java
1 VertsByDest = STREAM Netflows BY DestIp;
2 Top2 = FOREACH VertsByDest GENERATE topk(DestPort, 10000, 1000, 2);
3 Servers = FILTER VertsByDest BY Top2.value(0) + Top2.value(1) > 0.9;
4 DomainInfo = VertexInfo("domainurl", 5000);

5 SUBGRAPH ON Netflows WITH source (SourceIp) AND target (DestIp) {
6   client e1 frequent;
7   client e2 malicious;
8   frequent in Servers;
9   client not in Servers;
10  DomainInfo(frequent).domain != DomainInfo(malicious).domain
11  start(e2) start(e1) <= 10;
12  start(e1) < start(e2);
13  e1.ApplicationProtocol == "http"
14  e2.ApplicationProtocol == "http"
15 }
```

### 3.7 Summary

In this chapter we’ve applied SAL to a broad range of cyber problems, using Verizon’s Data Breach Investigations Report as the basis for exploring coverage. SAL covers a wide range of use cases within the cyber realm.
Chapter 4

Related Work

Given that we are combining aspects of graph theory, machine learning, and streaming into one domain specific language, there is a large body of related work.

In terms of graph research, there are several different related-subareas:

- Vertex-centric Computing: This a popular way to express parallel graph operations based on the Bulk Synchronous Parallel paradigm [200]. While a useful approach for batch processing of static graphs, its application to streaming graph computations is less relevant. We discuss this avenue of research in Section 4.1.

- Graph Domain Specific Languages: A number of DSLs have been developed to express graph algorithms. However, these DSLs largely target shared-memory architectures, operate on static graphs, and do not have the ability to express machine learning pipelines. Graph DSLs are discussed in Section 4.2.

- Linear Algebra-based Graph Systems: Many graph algorithms can be expressed as computations on matrices and as such, systems have been developed that adopt this approach. These approaches are discussed in Section 4.3.

- Graph APIs: In Section 4.4 we discuss other libraries that tackle graph computations, but aren’t necessarily DSLs.

- Graph computations using GPU: In Section 4.5 we discuss approaches that target the GPU for graph computations.
• SPARQL: The semantic web \cite{21} is a graph with a rich set of edge types. As such, the query language for the semantic web, SPARQL, has as its basis **basic graph patterns** that are essentially subgraphs. SPARQL influenced the design of SAL. Related efforts are discussed in section 4.6.

Machine learning is a rich and active research area. We discuss common frameworks in Section 4.7. Data Stream Management Systems (DSMS) are designed to handle streaming data as opposed to static data. The initial approaches were largely ports of relational database concepts to streaming data. Lately a plethora of projects with distributed APIs for handling streaming have become available, largely through Apache projects. These efforts are discussed in Section 4.8. We also discuss Network Telemetry in Section 4.9 which is an area dealing with network monitoring using network infrastructure devices such as switches.

### 4.1 Vertex-centric Computing

The idea of vertex-centric programming was popularized by Google with their paper on Pregel \cite{31}, a distributed graph platform with operations similar to MapReduce. Vertex-centric programming is based on Valiant’s Bulk Synchronous Parallel (BSP) \cite{200} paradigm. BSP prescribes a method for organizing computation in order to both predict parallel performance, and to simplify the task of parallelizing algorithms. There are three phases:

1. **Concurrent computation:** Each node performs computation independent of all other nodes. Memory access occurs only from the nodes’ local memory.

2. **Communication:** The nodes communicate with each other.

3. **Barrier Synchronization:** All nodes are stalled until the communication phase is over.

Algorithms described in terms of vertex-centric computation have a similar pattern. For example, consider the PageRank algorithm. Each vertex is assigned an initial PageRank value, and then the algorithm iterates until the PageRank values stabilize:
(1) Each vertex takes its current PageRank score, divides by the number of outgoing edges, and then sends that value to all of its neighbors. This is the communication phase of BSP.

(2) Progress halts until all vertices have received messages with updated page rank values from their neighbors. This is the synchronization phase of BSP.

(3) Each vertex re-computes the PageRank score based upon the updated values from its neighbors. This is the concurrent computation phase of BSP.

In essence, vertex-centric programming is BSP where the communication phase is limited to nodes connected by edges in the graph.

Since Pregel was developed, a number of other platforms have adopted the vertex-centric programming model. Below is a list:

- Giraph is the open source Apache version of Pregel [93].

- GraphLab [127] [126] and PowerGraph [74] arose from the same group at Carnegie Mellon. GraphLab started as a shared-memory implementation. PowerGraph moved the approach to a distributed platform and used the vertex-centric approach for parallelization. PowerGraph uses MPI as the underlying mechanism for parallelization and is restricted to in-memory computations with no fault tolerance. The latest instantiation, GraphLab Create, backs away from enabling custom algorithm development, but instead defines a python interface backed by C++ code. GraphLab Create added support for out-of-core computations.

- GraphX [210] extends Spark’s Resilient Distributed Dataset (RDD) to form a new abstraction called the Resilient Distributed Graph (RDG). While not strictly vertex-centric, they show through their API calls to updateVertices and aggregateNeighbors that they can implement the patterns defined by Pregel and PowerGraph. Because GraphX is built on top of Spark, it is distributed and the RDG mechanism provides fault-tolerance.
• Grace [165] is an in-memory graph management system. It is relatively unique in that it includes support for transactions and ACID semantics. Besides an API for performing updates to the graph (e.g. vertex or edge addition or deletion), the programming paradigm is also vertex-centric.

• Graph Processing System (GPS) [171, 99] is Pregel-like distributed system which added these features: 1) API support for global computations, 2) dynamic repartitioning, and 3) distribution of high-degree vertices. In [99] they add Green-Marl [98] as a domain specific language that is then mapped to GPS’s code base. Green-Marl is a DSL for expressing graph algorithms, which we discuss in more detail in Section 4.2.1.

• GraphChi [119] examines out-of-core graph computations on a single node by performing operations on sets of edges that are stored sequentially on disk. Disabling random access increases the number of edges read from disk, but the speed of the sequential access more than compensates.

• CuSha [116] is a vertex-centric graph processing system that is CUDA-based. To overcome the problem of irregular memory accesses, resulting in poor GPU performance, they use two graph representations: \textit{G-Shards} and \textit{Concatenated Windows}. \textit{G-Shards} uses the shard concept from GraphChi [119] to organize edges into ordered sets for memory access patterns better suited for GPU computations. \textit{Concatenated Windows} is a modification of \textit{G-Shards} for sparse graphs. The size of a shard in \textit{G-Shards} is inversely related to sparsity: the sparser the graph, the smaller the shard size on average. This can lead to GPU under-utilization. \textit{Concatenated Windows} combines shards into the same computation window.

• GraphGen [156] is an FPGA approach for vertex-centric computations. It takes a vertex-centric graph specification and compiles that into the targeted FPGA platform. They show speedups between 2.9x and 14.6x faster than CPU applications.

• X-Stream [170] can perform both in-memory and out-of-core computations on graphs, but
only for a single shared-memory machine. The novelty of this approach is that programming paradigm is edge-centric rather than vertex-centric. By adopting an edge-centric model, they can avoid some of the overhead incurred by GraphChi [119], which must pre-sort its shards by source vertex.

- VENUS [38] introduces optimizations to the original out-of-core approach of GraphChi [119], such as handling the structure of the graph, which doesn’t change, differently from vertex values, which change over the computation.

- Pregelix [29] was one of the first distributed approaches that allowed out-of-core workloads for vertex-centric programming. They employ database-style storage management and map vertex-centric programs to a query plan of joins and computations on tables.

- CHAOS [169], another distributed out-of-core approach, is a generalization of the X-Stream approach [170] for a distributed setting. The main assumption behind Choas is that the network is not the bottleneck, but instead reading from disk. Thus if a node needs a shard from a remote node, the latency is about the same as reading from a local drive. They show weak scaling results to 32 nodes, processing a graph 32 times larger than on one node, but only taking 1.61 times longer.

- PowerLyra’s [37] contribution is to recognize that processing low-degree vertices should be done differently than high-degree vertices in the gather-apply-scatter approach (BSP). High-degree vertices have their edges partitioned across the cluster, while low degree vertices are contained entirely within one node.

- GraphD [211] is also another distributed out-of-core approach. They show significant improvement over Pregelix and CHAOS for certain classes of problems. Part of the reason behind the speedup is GraphD’s ability to eliminate external-memory joins by using message combiners.

- FlashGraph [219] designs its vertex-centric computations around an array of solid-state
drives (SSDs). Vertex information is kept in memory, while edges are read-only and reside on the SSDs. Their performance compares favorably to in-memory approaches, and they are able to compute on a 129 billion edge graph on a single node.

- G-store \cite{118} is another vertex-centric approach utilizing flash drives. They outperform FlashGraph by up to $2.4 \times$ and can run some algorithms on a trillion edge graph in tens of minutes. They improve the cache hit ratio on multi-core CPUs using a tile-based physical grouping on disks. They also use a slide-cache-rewind approach where G-Store tries to utilize data that has already been fetched.

Table \ref{tab:vertex-cent} provides a summary of all the vertex-centric methods discussed above. It shows which approaches run on only a single node or if an approach can run on multiple nodes in a distributed setting. Also, it shows which frameworks are in-memory only and which can utilize out-of-core or computations on flash drives.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Single Node</th>
<th>Multiple Nodes</th>
<th>In-memory Only</th>
<th>Out-of-core</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Giraph</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphLab</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerGraph</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>GraphX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>GPS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>GraphChi</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-Stream</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Venus</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pregelix</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAOS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>PowerLyra</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>FlashGraph</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>G-Store</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: A table of vertex-centric frameworks and various attributes.

None of the above methods provide any native support for temporal information. However, it would be possible to encode temporal information on the edges (most provide support for defining arbitrary data structures on the edges and vertices). Then it would be incumbent upon the
programmer to properly use this information to implement various forms of the temporal logic.
The biggest impediment to using vertex-centric computing for a streaming is that it is by nature
a batch process. Adapting it to a streaming environment would likely entail creating overlapping
windows of time over which to perform queries in batch. Of course, SAL is a much higher level of
abstraction than any of the interfaces provided by these frameworks.

4.2 Graph Domain Specific Languages

A major contribution of this work is the presentation of a domain specific language for
streaming temporal graph computations that allow succinct algorithmic descriptions. In this section
we give an overview of other domain specific languages that specifically target graph computations.
We discuss the following works:

- Green-Marl [98]
- Ligra [181]
- Galois [152]
- Gemini [220]
- Grazelle [83]
- GraphIt [217]

A limitation for most of these DSLs, except for Gemini, is that they target shared memory
machines. While some of the concepts could be generalized to distributed memory approaches,
their current implementations do not include support for distributed settings. Another work, called
Gluon [49], is an approach that can be used to convert existing shared-memory infrastructures into
distributed frameworks. Gluon provides a lightweight API that allows an existing shared memory
graph system to partition the graph across a cluster. The primary requirement is that the shared
memory graph system provides a vertex-centric programming model (discussed in Section 4.1).
creators of Gluon provide a performance evaluation of Ligra, Galois, and IrGL (a GPU, single-node graph system) [159] that have been extended to a distributed setting, called D-Ligra, D-Galois, and D-IrGL. Overall they show that the D-Galois and D-IrGL outperform Gemini.

Regardless of whether these DLS approaches can be adjusted to a distributed setting, a fundamental difference between this set of work and our own is the streaming aspect of our approach and domain. An underlying assumption of the approaches in this section is that the data is static. The algorithmic solutions, partitioning, operation scheduling, and underlying data processing rely upon this assumption.

4.2.1 Green-Marl

Green-Marl [98] attempts to increase programmer productivity through the use of high level constructs that can be used to succinctly describe graph algorithms. One of these constructs is traversals over the graph, either breadth-first or depth-first. Below is their implementation of betweenness centrality using Green-Marl. They iterate over each node and perform breadth-first search from the node to find the number of shortest paths through each node. They make use of InBFS and InRBFS, which iterate through nodes in breadth-first and reverse breadth-first order, respectively.

<table>
<thead>
<tr>
<th>Listing 4.1: Green-Marl Betweenness Centrality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Procedure Compute_BC(G: Graph, BC: Node Prop&lt;Float&gt;(G)) {</td>
</tr>
<tr>
<td>2 G.BC = 0; // initialize BC</td>
</tr>
<tr>
<td>3 Foreach(s: G.Nodes) {</td>
</tr>
<tr>
<td>4 // define temporary properties</td>
</tr>
<tr>
<td>5 Node Prop&lt;Float&gt;(G) Sigma;</td>
</tr>
<tr>
<td>6 Node Prop&lt;Float&gt;(G) Delta;</td>
</tr>
<tr>
<td>7 s.Sigma = 1; // Initialize Sigma for root</td>
</tr>
<tr>
<td>8 // Traverse graph in BFS-order from s</td>
</tr>
</tbody>
</table>
When BFS and DFS are not sufficient to express a desired computation, the framework allows you to access the vertices and edges and write the for-loops yourself. For many situations, Green-Marl provides implicit parallelism, with the compiler automatically parallelizing loops. Besides the BFS and DFS traversals and implicit parallelism, Green-Marl also provides various collections, deferred assignment (allows for bulk synchronous consistency), and reductions.

4.2.2 Ligra

Ligra [181] has more flexible mechanisms than Green-Marl for writing graph traversals. While Green-Marl only allows BFS and DFS, Ligra can operate over arbitrary sets of vertices on each iteration. This allows it to naturally and succinctly express algorithms for radii estimation and Bellman-Ford shortest path, which do not use BFS or DFS traversals.

4.2.3 Galois

The researchers behind Galois [152] argue that BSP-style computations are insufficient for performance across a variety of input graphs and graph algorithms. They present a light-weight
infrastructure that can represent the APIs of GraphLab [127], GraphChi [119], and Ligra [181]. Galois also provides fine-grained scheduling mechanisms through the use of priorities per task.

4.2.4 Gemini

Gemini [220] is a graph DSL similar to Ligra, but with an implementation that addresses shortcomings of graph computations in a distributed setting. In McSherry et al. [139], the authors note that a single threaded approach beats many scalable distributed graph frameworks. Gemini adopts several approaches to make scalability efficient for graph computations:

- Adaptive push/pull vertex updates depending upon sparsity/density of active edge set: When there sparse/few active edges, pushing updates to vertices along outgoing edges is generally more efficient while for dense/many active edges, it is more efficient to pull vertex updates from neighboring vertices.

- Chunk-based partitioning: They try to capture locality within the graph structure by using a simple scheme for creating contiguous chunks of vertices.

- CSR and CSC: Similar to SAM, they index edges using compressed sparse row and compressed sparse column format. However, they employ bitmaps and a doubly compressed sparse column approach to reduce the memory footprint of both data structures.

- NUMA-aware: They apply their chunk-based partitioning recursively in an attempt to factor in NUMA-characteristics of today’s multicore architectures.

- Work-stealing: While the overall computation is a bulk-synchronous parallel computation, they employ types of work-stealing within a node to better load balance computation intranode.

With these optimizations, Gemini is able to outperform other distributed graph implementations between $8.91 \times$ and $39.8 \times$ on an eight node cluster connected with 100 Gbps Infiniband.
4.2.5 Grazelle

Grazelle \[83\] focuses on pull-based graph algorithms where an outer loop iterates over destination vertices and an inner loop over in-edges. They optimize this pattern with a scheduler-aware interface for the parallel loops that allow each thread to execute consecutive iterations and a vector-sparse format that overcomes some of the limitations of compressed-sparse data structures, particularly with unaligned memory accesses that occur with low-degree vertices. They compare their performance on single node against four other graph systems with improvements ranging between \(4.6 \times\) and \(66.8 \times\).

4.2.6 GraphIt

GraphIt \[217\] is another domain specific language for graph applications. They make the argument that computations on graphs are difficult to reason about. For example, the performance of an implementation of a graph algorithm can vary widely depending on the structure of the underlying graph. Their DSL, GraphIt, separates the what (the end result) from the how (the schedule of operations). This allows programmers to specify a correct algorithm, and then separately tinker with the specific optimization strategies. They also present an autotuner that will automatically search for optimizations.

GraphIt allows programmers to reason about three different tradeoffs in optimizations: locality, work efficiency, and parallelism. Often, one can optimize one dimension at the detriment of the others. For example, load-balancing is often an issue with graph computations. One can spend more time and instructions (hurting work efficiency) to enable better load-balancing and locality. It is often difficult to know a priori which optimizations are appropriate for a given algorithm and set of data. GraphIt allows programmers (and the autotuner) to specify eleven different optimizations which have varying effects on the tradeoff space. These strategies include: DensePull, DensePush, DensePull-SparsePush, DensePush-SparsePush, edge-aware-vertex-parallel, edge-parallel, bitvector, vertex data layout, cache partitioning, NUMA partitioning, and kernel fusion.
A GraphIt program has two sections: the algorithm specification and the scheduling optimizations. The algorithm API is divided into set operators, vertexset operators, and edgeset operators. Low level details such as synchronization and deduplication are abstracted away and dealt with by the compiler. Lines of the algorithm specification can be annotated with labels (of the form #label#), which are latter referenced in the scheduling section where specific optimizations can be applied.

4.3 Linear Algebra-based Graph Systems

Many algorithms on graphs are readily described in terms of matrix operations using the adjacency matrix, $A$, of the graph. For instance, to find the number of triangles in the graph, it is $\text{Trace}(A^3)/6$ (assuming an undirected graph with no self-loops). Each $i,j$ element in $A^3$ is the number paths of length three going from $i$ to $j$. Elements along the diagonal describe three-paths going from a node back to itself, i.e. triangles. Since each node reports its participation in the triangle, and does so twice (in each direction), we divide by six to get the total number of unique triangles. There are many other examples of graph algorithms that can be described as operations on the adjacency matrix [70] [114].

While many graph algorithms can be expressed as operations on sparse matrices, it is unclear if that abstraction is best in terms of programmer productivity or parallel performance. Satish et al. [174] begins to look at these issues, though mostly addressing performance issues with Combinatorial BLAS [32] and other approaches. For our work, it is unclear how to account for temporal properties on the edges. Multiplying an adjacency matrix with itself does not account for the temporal constraints on the edges. Also, the question remains how streaming operations would be implemented. On what version of the adjacency matrix would the operations be performed?

Frameworks that express graph algorithms as linear algebra include following:

- Combinatorial BLAS [32]
- Knowledge Discovery Toolbox [128]
• Pegasus [110]

• GraphMat [192]

We discuss these approaches below.

4.3.1 Combinatorial BLAS

Buluç and Gilbert [32] present one of the first libraries for performing graph computations in terms of linear algebra primitives. They name it Combinatorial BLAS, combinatorial because combinatorics and graph theory are strongly related, and BLAS, after the canonical Basic Linear Algebra Subroutines, a well known library of successful primitives packages. Since adjacency matrices are the primary target of the library, distributed sparse matrices are first class citizens in Combinatorial BLAS. However, there are other functions that use dense matrices, and dense and sparse vectors. Similar to BLAS, they define a set of primitive functions. For example, SpGEMM is a function that multiplies two sparse matrices. In their paper introducing Combinatorial BLAS, the authors provide details for two algorithms implemented with the new API, betweenness centrality and Markov clustering.

4.3.2 Knowledge Discovery Toolbox

Another notable work using a linear algebra-type approach includes the Knowledge Discovery Toolbox (KDT) [128]. KDT uses Combinatorial BLAS as computational kernels but presents Python as the API.

4.3.3 Pegasus

Pegasus [110] implements a sparse matrix-vector multiplication (GIM-V or generalized iterated matrix-vector multiplication) on top of Hadoop and describes how this single primitive is sufficient for algorithms such as PageRank, random walk with restart, diameter estimation, and connected components.
4.3.4 GraphMAT

GraphMat [192] provides a way of expressing vertex-centric programs which are then mapped to matrix-style computations. They argue that this approach garners the productivity of vertex-centric programming while retaining the performance of optimized sparse matrix operations. They show speedups between $1.1 \times - 7 \times$ over other graph frameworks.

4.4 Graph APIs

In this section we discuss other APIs that have been developed for processing graph algorithms:

- Parallel Boost Graph Library (PBGL) [81]
- The MultiThreaded Graph Library (MTGL) [22]
- Small-world Network Analysis and Partitioning (SNAP) [18]
- STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation [17]
- Grappa [149] [148]

The Parallel Boost Graph Library (PBGL) [81] builds upon the Boost Graph Library (BGL) [182], using concepts such as generic programming and the visitor pattern. Like the Standard Template Library, BGL and PBGL extensively uses templates for class types so that the same algorithm can be applied to different graph representations. For example, you could have an adjacency-matrix representation of a graph or a compressed sparse row graph but breadth-first search would be written generically enough to apply to both representations.

Another concept important to BGL and PBGL is the visitor pattern. Visitors allow programmers to modify algorithms to extend functionality in a fine-grained way. For example, the *examine vertex* function of the BFS visitor could be used to increment a counter on each vertex in
order to count the number of paths going through each vertex in a betweenness calculation. In this way, it is similar to the BFS traversals of Green-Marl and Ligra, just expressed in different way.

The MultiThreaded Graph Library [22] was directly inspired by PBGL, using many of the same concepts such as generic templates and visitors. However, instead of a distributed setting as with PBGL, MTGL focuses on shared-memory supercomputers such as the Cray XMT.

Another set of shared-memory approaches are SNAP [18] and STINGER [17]. Both SNAP and STINGER come from George Tech. SNAP is a library for analyzing primarily static graphs (but also some support for dynamic graphs) on shared memory machines. The primary building blocks are called graph kernels, which include BFS, connected components, minimum spanning tree. It is unclear what constitutes a kernel, and what is the set of expressible algorithms given a set of kernels. While SNAP was more focused on static graph analysis, STINGER is for dynamically changing graphs. Again, STINGER is limited to shared memory systems. With STINGER, they developed several dynamic graph algorithms for connected components [136], clustering coefficient [55], and betweenness centrality [80].

Grappa [149, 148] is an effort to create the illusion of a shared memory machine on a distributed cluster. They create a latency tolerant software layer on top of commodity hardware to support data-intensive applications such as graph problems and mitigate latency problems associated with small messages, poor locality, and the need for fine-grained synchronization. In essence, they want to recreate the nice features of the Terra MTA without the exorbitant cost for specialized hardware [8]. In [148] they show how they can implement various frameworks such as GraphLab and Spark and achieve greater performance than obtainable with native algorithms.

4.5 GPU Graph Systems

Programming graph computations on GPUs are particularly challenging due to the usually random memory access patterns of many graph algorithms. However, several optimizations can be employed to enable GPU-facilitated computations. Totem [68] and GStream [176] are some of the first works to process graphs with GPUs where the size of the graph is larger than what is avail-
able on the device. IrGL \cite{159} is an intermediate-level program representation that allows graph algorithms to be described that are then translated into CUDA code with a compiler. The compiler automatically utilizes three types of optimizations for data-driven problems to allow efficient utilization of the GPU. Groute \cite{19} provides an asynchronous programming model that can run on multi-GPU systems. Pan et al. also provide a framework for graph computations on multi-GPU systems, published at the same time as Groute. While an important body of work, to date GPU graph systems focus on static graph analysis and do not consider streaming.

4.6 SPARQL

SPARQL\cite{66} (a recursive acronym for SPARQL Protocol and RDF Query Language) is a SQL-like declarative language for querying the semantic web. It is used to express subgraphs of interest in an RDF (Resource Description Framework) dataset. RDF is described in terms of triples: a subject, a predicate, and an object. This can be thought of as describing a graph, where the subject and object are nodes in the graph and the predicate is a labeled directed edge between the two. SPARQL influenced our syntax in how temporal subgraphs are expressed in SAL.

Below is an example SPARQL query, query number 2 from the popular Lehigh University Benchmark (LUBM) \cite{86}: It finds graduate students who are a member of a department in the same university where they received their undergraduate degree. Figure 4.1 gives a visual representation of the query graph.

<table>
<thead>
<tr>
<th>Listing 4.2: LUBM Query 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX rdf: <a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#">http://www.w3.org/1999/02/22-rdf-syntax-ns#</a></td>
</tr>
<tr>
<td>PREFIX ub: <a href="http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#">http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#</a></td>
</tr>
<tr>
<td>WHERE</td>
</tr>
<tr>
<td>{ ?X rdf:type ub:GraduateStudent .</td>
</tr>
<tr>
<td>?Y rdf:type ub:University .</td>
</tr>
<tr>
<td>?Z rdf:type ub:Department .</td>
</tr>
<tr>
<td>?X ub:undergraduateDegreeFrom ?Y }</td>
</tr>
</tbody>
</table>
The main building block of a SPARQL query is the basic graph pattern (BGP). The where clause in LUBM query 2 is a BGP with six triple patterns. At its core is a triangle with additional edges defining the vertex types. In general, SPARQL is well suited for specifying subgraphs of interest, though it does have problems when the query graph is automorphic as this will produce duplicate answers [77]. Besides BGPs, SPARQL also allows has other constructs such as unions, negations, optionals, aggregations, and property paths.

The SPARQL standard does not address temporal aspects of the graph. There has been some work in extending RDF and SPARQL to encompass temporal notions. In Guiterrez et al. [89, 88] they present an extension of RDF to include temporal intervals. They provide a sketch of a query language. Each edge has a temporal interval associated with it, \([t_1, t_2]\), or just \([T]\) for short. You can then use those intervals in queries. Using one of their examples, if you want to find students that have taken a Masters-level course sometime since 2000, you would express the query

\[
(?X, \text{takes}, ?C) : [?T], (?C, \text{type}, \text{Master}) : [?T], 2000 \leq ?T, ?T \leq \text{Now}
\]

In Tappolet and Bernstein [197], they present a query language \(\tau\)-SPARQL for temporal
queries which is directly translatable to standard SPARQL queries. However, $\tau$-SPARQL is not sufficiently expressive for our needs. It is suited to finding edges valid at a given time, or returning intervals of time when edges were valid, but not to expressing constraints on the temporal ordering of edges relative to each other in a subgraph query.

Much of the semantic web research has focused on relatively static collections of RDF triples with querying and reasoning implemented with that assumption. Another avenue has explored streaming environments, where RDF triples are generated continuously. Several benchmarks are commonly used to compare streaming RDF frameworks:

- LSBench [121] is a social network-based data generation with posts, comments, likes, follows etc., that you would typically find with social network activity.

- SRBench [215] has as its basis in the Linked Open Data cloud [137] with real-world sensor data, LinkedSensorData [117], which is published U.S. weather data.


Approaches that specifically target streaming RDF and continuous SPARQL queries include C-SPARQL [62], Wukong [179], Wukong+S [216], EP-SPARQL [9], SPARQLStream [35], and CQELS [120]. While the streaming aspect of this problem and the task of finding subgraphs overlap with our goals, there are several aspects not addressed which our work targets. The temporal ordering of edges is ignored; they find subgraphs within windows of time, but do not specify the relative times of when edges occur, which is an important dimension when describing cyber attacks. Also, they do not compute attributes on the streaming data, which can then be used as features for a machine learning solution, or as a filtering mechanism for subgraph selection. RDF triples allow for a rich set of edge types; however, for cyber problems, we have edges with multiple attributes, which is awkward to express in RDF, the mechanism being reification [204]. Finally, with the exception of Wukong+S [216] that has scaling numbers to 8 nodes, none of the streaming RDF/SPARQL approaches are distributed.
4.7 Machine Learning

In terms of machine learning, there are several popular frameworks:

- Scikit-learn is a python code base with a range of machine learning approaches. It is a useful tool for machine learning and data mining research and projects, but it is largely limited to computations on a single node without a much support for parallelization.

- Apache Mahout focuses on distributed computations using a linear algebra framework and mathematically focused Scala DSL.

- Deep learning frameworks: With the popularity of deep learning, several software frameworks have been developed that allow the programmer to specify neural network architectures and modify hyperparameters. Some frameworks of note include Tensorflow, Pytorch, and Keras. All of these frameworks largely concentrate on employing neural networks on a single node with one or more gpus available.

All of these approaches are valuable, allowing common machine learning tasks to be utilized and explored with well-designed APIs; however, none address the streaming aspect of this work.

4.8 Data Stream Management Systems

Data Stream Management Systems (DSMS) are designed to specifically tackle streaming data. One of the first efforts to address data management for streaming data is the Aurora project. In the Aurora system model, a variety of data sources feed into a directed acyclic graph (DAG) of processing operations. Operators can be one of filter, map (a generalized projection operator), union (merging two or more streams into a single stream), bsort (buffered sort, an approximate sort operator), aggregate (applying a function to a window of tuples), join (joins two stream together over a specified time window), resample (similar to join but with a generalized time window), and split (implicit operation deduced from the DAG). The DAG feeds into three types of endpoints: continuous queries, views, and ad-hoc queries.
While the Aurora project focused on stream management within a single node, Medusa expanded that to handle a distributed setting, covering both the situation where the computational resources are under a single administrative domain (intra-participant distribution) and the more general case when services are provided by multiple autonomous participants (inter-participant distribution). For intra-participant distributions, Aurora is deployed on multiple nodes, and local decisions and pair-wise interactions between nodes is used to reconfigure the workload when resource-usage becomes unbalanced. For inter-participant distributions, Medusa uses the economic principles of an agoric system to adjust the load management. In such a system participants provide services such as data sources or analytical capabilities, and participants form contracts where they pay for those services. The idea is that market forces will drive the system to an efficient balance of computational resources.

The Borealis project superseded the Aurora and Medusa projects, combining the two foci into one. Borealis adds the ability to revise queries when corrections to data streams are provided. It also allows dynamic query modification, e.g. switching to less expensive operations when the system becomes overloaded. Finally, Borealis adds a framework for optimizing various QoS metrics in a heterogeneous network comprised of servers and sensors. This family of work, Aurora, Medusa, and Borealis, address an array of requirements for a streaming system, but focus largely on porting SQL-like operations to a streaming environment, and they do not support streaming operators such as topk as first-class citizens.

IBM developed a streaming language, Streams Processing Language (SPL), and its predecessor, SPADE. SPL is the main avenue for programming applications in the IBM Streams platform. The main abstraction of SPL is a graph where the edges are streams of data (tuples), and the nodes are operators. SPL is very flexible, allowing for new operators to be defined using language concepts from C, Java, SQL, Python, and ML. Comparing SAL to SPL, SAL is much more domain specific. SAL comes with native support for streaming and semi-streaming operators. SPL does not support these type of operators as first-class citizens, but given the expressiveness of the language, they could conceivably be programmed into a custom operator.
There are many recent frameworks that provide streaming APIs:

- Apache Storm [190]
- Apache Spark [188]
- Apache Flink [12]
- Apache Heron [95]
- Apache Samza [172]
- Apache Gearpump [13]
- Apache Kafka [64]
- Apache Apex [10]
- Google Cloud Dataflow [78]

Many of these could potentially be the basis of a backend implementation for SAL. Zhang et al. [216] report success in adopting Storm as the backend of a streaming C-SPARQL engine [62], while Spark has significantly longer latencies. We initially targeted Spark streaming as an implementation backend for SAL but found that the overhead of keeping state from one discretized portion of the stream to the next was prohibitive. In Chapter 7 we report on performance using Apache Flink for streaming temporal subgraph matching. We found mixed results comparing SAM and Flink, largely dependent on the frequency of occurring subgraphs. Regardless, our main contribution in this work is the domain specific language of SAL. SAM was developed as a prototype implementation, and it is certainly possible to develop another implementation using these frameworks as a backend.

### 4.9 Network Telemetry

In our work with cyber data, we focus on netflow data. Netflow data is a summary of packet data going through a switch. Packets are organized by a 5-tuple key: source IP, destination
IP, source port, destination port, and protocol. Statistics are gathered around related packets that share the same 5-tuple key. Previously, the netflow aggregations were either performed using custom silicon (fixed-function ASICs) on high-end routers, or the packet traffic was mirrored to remote collectors for analysis [186]. However, recently there is trend towards programmable packet forwarding, opening the door for hybrid computations between the telemetry plane (the switch) and the analysis plane. P4 [27], short for *Programming Protocol-independent Packet Processors*, is a domain specific language to express computations on network traffic data. Switches such as the Barefoot Tofino [151] are P4 enabled, enabling the performance of fixed-function ASICs but with the benefit of being reconfigurable.

Recent work has demonstrated the benefits of hybrid telemetry/analysis plane computations. Sonata [87] defines a declarative language for expressing network traffic computations, and then automatically partitions the computations between the data and analysis planes. Flowradar [125] strikes a balance between cheap switches with a paucity of resources and a remote collector with ample resources to enable netflow calculations without sampling. Another work, *Flow [187]*, has mechanisms for concurrent and dynamic queries by shifting aggregation functions from the telemetry plane to the analysis plane.

This work in network telemetry is a natural path forward for SAL. Our current work focuses on network traffic data after netflow calculations have already been performed. In essence, we are operating entirely in the analysis plane. However, opening the door for custom calculations at the packet level create opportunities for analysts to develop new features for SAL pipelines. Also, we can create a unifying approach that can deploy SAL programs not only to the analysis plane, but some of the computations can be moved up into the telemetry plane. Finally, most of the current work in network telemetry focuses on data coming from one switch. Combining data from multiple switches is largely unexplored. Our work with SAL and SAM has started in this direction. In Chapters 6 and 7 we show coordination of up to 128 nodes, each acting as a separate sensor.
Chapter  5

Implementation

In this chapter we discuss how we go from SAL code to an implementation that runs in parallel across a cluster. To translate SAL code, we used the Scala Parser Combinator Library [175]. This allowed us to express the grammar of SAL and map those elements to C++ code that utilizes a prototype parallel library that we wrote to execute SAL programs. We call this library the Streaming Analytics Machine, or SAM, which is over 19,000 lines of code. Table 5.1 below shows the difference between directly writing C++ code utilizing SAM, or writing the same program in SAL. Writing in SAL uses about 10-25 times fewer lines of code.

This chapter is divided up into three sections. We first talk about how data is partitioned across the cluster via SAM in Section 5.1. Then we discuss vertex-centric computations such as feature creation in Section 5.2. Finally we discuss the data structures and algorithm used for subgraph matching in Section 5.3.

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disclosure</td>
<td>520</td>
<td>20</td>
</tr>
<tr>
<td>Machine Learning Pipeline</td>
<td>482</td>
<td>34</td>
</tr>
<tr>
<td>Temporal Triangle Query</td>
<td>238</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.1: This table shows the conciseness of the SAL language. For the Disclosure pipeline (discussed in Section 2.1), the machine learning pipeline (discussed in Section 6.1), and the temporal triangle query (discussed in Chapter 7) the amount of code needed is reduced between 10-25 times.
Figure 5.1: Architecture of the system: data comes in over a socket layer and is then distributed via ZeroMQ.

5.1 Partitioning the Data

SAM is architected so that each node in the cluster receives tuple data (see Figure 5.1). For the prototype, the only ingest method is a simple socket layer. In maturing SAM, other options such as Kafka [65] would be an obvious alternative. We then use ZeroMQ [6] to distribute the netflows across the cluster. The pattern used to distribute the data occurs more than once within SAM, and is encapsulated into a **ZeroMQ Push Pull Communicator** object, which we will refer to as **Communicator**. The Communicator uses the push pull paradigm of ZeroMQ, where producers submit data to push sockets and consumers collect from corresponding pull sockets. In general, push sockets bind to an address and the pull sockets connect to an address, allowing multiple pull sockets to collect messages from the same address.

Behind the scenes, ZeroMQ opens multiple ports in the dynamic range (49152 to 65535) for each ZMQ socket that is opened logically in the code. Nevertheless, for some uses of the Communicator we found that contention using the ZMQ sockets created significant performance bottlenecks. As such, we created the option to specify multiple push sockets per node, and anytime a node needs to send data to another node, it randomly selects one of the push sockets to use.
Figure 5.2 gives an overview of the design. If there are $n$ nodes in the cluster and $p$ push sockets per node, in total there are $(n-1)p$ push sockets. However, for the partitioning phase, only one push socket was needed. The need to have multiple push sockets arose because of subgraph matching, and is discussed in more detail in Section 5.3.

When multiple push sockets were necessary, we also found that instantiating multiple threads to collect the data increased throughput. $t$ threads are dedicated to pulling data from the pull sockets. Again for the partitioning phase, one thread was sufficient. In our experiments, we found 4 push sockets per node and 16 total pull threads worked best for most situations. However, this is probably a consequence of the number of cores per node, which in this case was 20. These results are discussed more in Section 7.1.

As discussed in Chapter 2, the data is partitioned with SAL statements of the form:

```
HASH <TupleField> WITH <HashFunction>;
```

SAM uses the specified hash function and applies it to the tuple field and sends it to the proper node. For all of our experiments, we have the following two statements:

```
HASH SourceIp WITH IpHashFunction;
HASH DestIp WITH IpHashFunction;
```
We do this since almost all computations for cyber analysis, our initial target domain, involves aggregating statistics and features per IP address. For each netflow that a node receives, it performs a hash (shared across the cluster) of the source IP in the netflow mod the cluster size to determine where to send the netflow. Similarly, the node hashes the destination IP mod the cluster size and sends the IP to that location. Thus for each netflow a node receives over the socket layer, it sends the same netflow twice over ZeroMQ. Each node is then prepared to perform calculations on a per IP basis. SAL can be extended to use other hash functions, as discussed in Chapter 2.

That concludes our discussion of how tuple data is partitioned across the cluster. Once the data is partitioned, we can then perform queries involving vertex-centric computations and subgraph matching.

5.2 Vertex-centric Computations

The partitioning phase can be thought of as defining vertices in the graph. For cyber data, by partitioning by source IP and destination IP each node in the cluster has access to all the tuples/edges related to a particular IP. Thus we can easily calculate statistics regarding each IP with no further communication within the cluster for these downstream calculations.

There are three different interfaces (defined as abstract or base classes) that are key to vertex-centric computations:

- Data Source: These classes are how the cluster receives data. As mentioned before, we implemented one avenue to ingest data through socket communications called **ReadSocket**. ReadSocket implements the **AbstractDataSource** class, which has two methods: **connect** and **receive**. For ReadSocket, **connect** opens the socket for communication while **receive** starts a loop to ingest the data. This data is then made available to Consumers through the Producer interface, discussed next.

- Producers: These objects produce tuples that can be consumed by other classes called Consumers. The important methods are **registerConsumer**, which allows a Consumer
to express interest in the output of a Producer, and **parallelFeed**. Pseudocode for the **parallelFeed** method is shown in Algorithm 1. It emits tuples to consumers in parallel.

- Consumers: Consumers implement the **AbstractConsumer** class that has the method **consume(tuple)**. Producers call this method to supply the consumers with tuples to ingest.

```
Algorithm 1  Producer: Feeding Tuples in Parallel
1:  function PRODUCER.parallelFeed(edge)
2:      mutex.lock()
3:      queue[numItems] ← edge
4:      ++numItems
5:      if numItems ≥ queueLength then
6:          for all c ∈ consumers do ▷ Iterate over all consumers
7:              for all e ∈ queue do ▷ Parallel loop
8:                  c.consume(e)
9:          end for
10:      end if
11:      mutex.unlock()
12:  end function
```

Many SAL statements and operators map to either consumer and/or producer interfaces and to an underlying C++ class that provides the specific functionality. Table 5.2 gives an overview.

<table>
<thead>
<tr>
<th>C++ Name</th>
<th>Consumer</th>
<th>Producer</th>
<th>Feature Creator</th>
<th>Maps To</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadSocket</td>
<td>x</td>
<td>x</td>
<td>Feature Creator</td>
<td>FlowStream</td>
</tr>
<tr>
<td>ZeroMQPushPull</td>
<td>x</td>
<td>x</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>Filter</td>
<td>x</td>
<td>x</td>
<td></td>
<td>FILTER</td>
</tr>
<tr>
<td>Transform</td>
<td>x</td>
<td>x</td>
<td></td>
<td>TRANSFORM</td>
</tr>
<tr>
<td>CollapsedConsumer</td>
<td>x</td>
<td></td>
<td></td>
<td>COLLAPSE</td>
</tr>
<tr>
<td>Project</td>
<td>x</td>
<td></td>
<td></td>
<td>COLLAPSE</td>
</tr>
<tr>
<td>EHSum</td>
<td>x</td>
<td>x</td>
<td></td>
<td>sum</td>
</tr>
<tr>
<td>EHHave</td>
<td>x</td>
<td>x</td>
<td></td>
<td>ave</td>
</tr>
<tr>
<td>EHVariance</td>
<td>x</td>
<td>x</td>
<td></td>
<td>var</td>
</tr>
<tr>
<td>TopK</td>
<td>x</td>
<td>x</td>
<td></td>
<td>topk</td>
</tr>
</tbody>
</table>

Table 5.2: A listing of the major classes in SAM and how they map to language features in SAL. We also make the distinction between consumers, producers, and feature creators. Producers all share a **parallelFeed** method that sends the data to registered consumers, which is how parallelization is achieved. Each of the consumers receive data from producers. If a consumer is also a feature creator, it performs a computation on the received data, generating a feature, which is then added to a thread-safe feature map.
Often consumers also generate features. Each node creates a thread-safe feature map to collect features that are generated. The function signatures is as follows:

```java
updateInsert(string key, string featureName, Feature f)
```

The `key` is generated by the key fields specified in the `STREAM BY` statement. The `featureName` comes from the identifier specified in the `FOREACH GENERATE` statement. For example, in the below SAL snippet, the key is created by using a string hash function on the concatenation of the source IP and destination IP. The identifier is `Feature1` as specified in the `FOREACH GENERATE` statement.

```sal
DestSrc = STREAM Netflows BY SourceIp, DestIp;
Feature1 = FOREACH DestSrc GENERATE ave(SrcTotalBytes);
```

The scheme for ensuring thread-safety in the feature map comes from Goodman et al. [75].

The `Project` class provides the functionality of the `COLLPASE` statement. The `Project` class creates features similar to the Feature Creator classes, but they cannot be accessed directly through the API. The features are `Map` features, in other words the sets $M_l$ using the terminology from Section 2.1. These `Map` features are added to the same feature map used for all other generated features. These `Map` features can then be used by the `CollapsedConsumer` class, which can be specified to calculate statistics on the map for each kept key in the stream.

### 5.3 Subgraph Matching

In this section we focus on the data structures, processes, and algorithms that enable a subgraph query. For discussion, we repeat the Watering Hole Attack query previously given in Listing 2.1.
Listing 5.1: Subgraph for Watering Hole Attack

```cpp
SUBGRAPH ON Netflows WITH source(SourceIp) AND target(DestIp)
{
  target e1 bait;
  target e2 controller;
  start(e2) > end(e1);
  start(e2) - end(e1) < 10;
  bait in Top1000;
  controller not in Top1000;
}
```

Subgraph queries are translated from the SPARQL-like syntax to a sequence of C++ objects, namely of type `EdgeExpression`, `EdgeConstraintExpression`, and `VertexConstraintExpression`. In the Watering Hole Attack query, the first two topological statements become EdgeExpressions, in essence, assigning labels to vertices and edges. The next two lines define temporal constraints on the edges which become EdgeConstraintExpression’s. The last two lines are examples of VertexConstraintExpression’s. SAM takes the EdgeExpressions and TimeEdgeExpressions and develops an order of execution based upon the temporal ordering of the edges. In this example, $e_1$ occurs before $e_2$.

The approach we have taken assumes there is a strict temporal ordering of the edges. For our intended use case of cyber security, this is generally an acceptable assumption. Most queries have causality implicit in the question. For example, the Watering Hole Attack is a sequence of actions, with the initial cause the maladvertisement exploiting a vulnerability in a target system, which then causes other actions, all of which are temporally ordered. As another example, botnets with a command and control server, commands are issued from the command and control server, which then produces subsequent actions by the bots, which again produces a temporal ordering of the interactions. Our argument is that requiring strict temporal ordering of edges is not a huge
constraint, or at the very least covers many relevant questions an analyst might ask of cyber data. We leave as future work an implementation that can address queries that do not have a temporal ordering on the edges.

It should be noted that while SAM, the implementation, currently requires temporal ordering of edges, SAL, the language, is not so constrained. SAM can certainly be expanded to include other approaches that do not require temporal ordering of edges.

Once the subgraph query has been finalized with the order of edges determined, SAM is now set to find matching subgraphs within the stream of data. Before discussing the actual algorithm, we will first discuss the data structures used by SAM.

5.3.1 Data Structures

We make use of a number of thread-safe custom data structures, namely

- Compressed Sparse Row (CSR)
- Compressed Sparse Column (CSC)
- ZeroMQ Push Pull Communicator (Communicator)
- Subgraph Query Result Map (ResultMap)
- Edge Request Map (RequestMap)
- Graph Store (GraphStore)

In parenthesis is how we will refer to them in the remainder of this thesis.

5.3.1.1 Compressed Sparse Row

CSR is common way of representing sparse matrices with the earliest known description a 1967 article [32]. The data structure is useful for representing sparse graphs since graphs can be represented with a matrix, the rows signifying source vertices and the columns destination vertices.
Instead of a $|V| \times |V|$ matrix, where $|V|$ is the number of vertices, CSR’s have space requirement $O(|E|)$, where $|E|$ is the number of edges. In the case of sparse graphs, $|E| \ll |V| \times |V|$.

Figure 5.3 presents the traditional CSR data structure. There is an array of size $|V|$ where each element of the array points to a list of edges. For array element $v$, the list of edges are all the edges that have $v$ as the source. With this index structure, we can easily find all the edges that have a particular vertex $v$ as the source. However, a complete scan of the data structure is required to find all edges that have $v$ as a destination, leading to the need for the compressed sparse column data structure, described in the next section.

CSR’s as presented work well with static data where the number of vertices and edges remain constant. However, our work requires edges that expire, and also for the possibility that vertices may come and go. To handle this situation, we create an array of lists of lists of edges, with each array element protected with a mutex. Figure 5.4 shows the overall structure. Instead of an array of size $|V|$, we create an array of bins that are accessed via a hash function. When an edge is consumed by SAM, it is added to this CSR data structure. The source of the edge is hashed, and if the source has never been seen, a list is added to the first level. Then the edge is added to the list. If the vertex has been seen before, the edge is added to the existing edge list. Additionally, SAM keeps track of the longest duration of a registered query, and any edges that are older than the current time minus the longest duration are deleted. Checks for old edges are done anytime a new edge is added to an existing edge list.

The CSR has mutexes protecting each bin, and any access to a bin must first gain control of the mutex. This will be important later when we describe the algorithm for finding subgraph matches. Regarding performance, as each bin has its own mutex, the chance of collision and thread contention is very low. In metrics gathering, waiting for mutex locks of the CSR data structure has never been a significant factor in any of our experiments.
Figure 5.3: Traditional Compressed Sparse Row (CSR) data structure.

Figure 5.4: Modified Compressed Sparse Row (CSR) data structure.
5.3.1.2 Compressed Sparse Column

The Compressed Sparse Column (CSC) is exactly the same as the CSR, the only difference is we index on the target vertex instead of the source vertex. Both the CSR and CSC are needed, depending upon the query invoked. We may need to find an edge with a certain source (CSR), or an edge with a certain target (CSC).

5.3.1.3 ZeroMQ PushPull Communicator

This object was discussed previously in Section 5.1. Subgraph matching led to the modification of multiple push sockets and multiple pull threads. There are two different communicators used: and Request Communicator and an Edge Communicator. The Request Communicator sends requests to other nodes for edges fitting a particular pattern. The Edge Communicator sends matching edges back. It is the Edge Communicator that benefits most from having multiple push sockets and threads. In our experiments, we found 4 push sockets per node and 16 total pull threads worked best for the Edge Communicator in most situations, which makes sense because each socket had 20 cores. These results are discussed more in Section 7.1.

The Communicator allows for callback functions to be registered. These callback functions are invoked whenever a pull socket receives data. We use callback functions to enable proper handling of requests by the Request Communicator and edges with the Edge Communicator. These are discussed in more detail in Sections 5.3.2.2 and 5.3.2.3.

5.3.1.4 Subgraph Query Result Map

The ResultMap is used to store intermediate query results. There are two central methods, add and process. The method add takes as input an intermediate query result, creates additional query results based on the local CSR and CSC, and then creates a list of edge requests for other edges on other nodes that are needed to complete queries. The method process is similar, except in this case it takes an edge that has been received by the node and it checks to see if that edge satisfies any existing intermediate subgraph queries. The details of processing intermediate results
and edges used will be discussed in greater depth in Section 5.3.2.

The ResultMap uses a hash structure to store results. It is an array of vectors of intermediate query results. The array is of fixed size, so must be set appropriately large to deal with the number of results generated during the maximum time window specified for all registered queries, otherwise excessive time is spent linearly probing the vector of intermediate results. There are mutexes protecting each access to each array slot. For each intermediate result created, it is indexed by the source, the target, or a combination of the source and target, depending on what the query needs to satisfy the next edge. The index is the result of a hash function, and that index is used to store the intermediate result within the array data structure.

5.3.1.5 Edge Request Map

The RequestMap is the object that receives requests for edges that match certain criteria, and sends out edges to the requestors whenever an edge is found that matches the criteria. There are two important methods to mention: addRequest(EdgeRequest) and process(Edge). The addRequest method takes an EdgeRequest object and stores it. The process method takes an Edge and checks to see if there are any registered edge requests that match the given edge.

The main data structure is again a hash table. The approach to hashing is similar to the ResultMap, as it depends on whether the source is defined in the request data structure, the target is defined, or both. The index is then used to find a bin with an array of lists of EdgeRequest objects. Similar to the ResultMap, whenever process is called, edge requests that have expired are removed from the list.

5.3.1.6 Graph Store

The GraphStore object uses all the previous data structures, orchestrating them together to perform subgraph queries. It has a pointer to the ResultMap and to the RequestMap. It also has pointers to two Communicators, one for sending/receiving edges to/from other nodes (Edge Communicator), and another for sending/receiving edge requests (Request Communicator).
GraphStore maintains the CSR and CSC data structures. How all these pieces work together is described in the next section.

### 5.3.2 Algorithm

![Diagram of algorithm](image)

Figure 5.5: An overview of the algorithm employed to detect subgraphs of interest. Asynchronous threads are launched to consume each edge. The consume threads then performs five different operations. Another set of threads are pulling requests from other nodes for edges within the Request Communicator. Finally, one more set of threads pull edges from other nodes in reply to edge requests.

The algorithm employed to find matching subgraphs employs three sets of threads:

- GraphStore consume threads.
- The threads associated with the Request Communicator.
- The threads associated with the Edge Communicator.
5.3.2.1 **GraphStore Consume(Edge) threads**

The GraphStore object’s primary method is the `consume` function. GraphStore is tied into a producer, and any edges that the producer generates is fed to the `consume` function. For each `consume` call, an asynchronous thread is launched. We found that occasionally a call to `consume` would take an exorbitant amount of time, orders of magnitude greater than the average time. These outliers were a result of delays in the underlying ZeroMQ communication infrastructure, proving difficult to entirely prevent. As such, we added the asynchronous approach, effectively hiding these outliers by allowing work to continue even if one thread was blocked by ZeroMQ.

Each consume thread performs the following actions:

1. Adds the edge to the CSR and CSC data structures.
2. Processes the edge against the ResultMap, looking to see if any intermediate results can be further developed with the new edge.
3. Checks to see if this edge satisfies the first edge of any registered queries. If so, adds the results to the ResultMap.
4. Processes the edge against the RequestMap to see if there are any outstanding requests that match with the new edge.
5. Steps two and three can result in new edges being needed that may reside on other nodes. These edge requests are accumulated and sent out.

For steps 2-4, we provide greater detail below:

**Step 2: ResultMap.process(edge):** ResultMap’s `process(edge)` function is outlined in Algorithm 2 on page 79. At the top level, the `process(edge)` dives down into three `process(edge, indexer, checker)` calls. Intermediate query results are indexed by the next edge to be matched. The `srcIndexFunc, trgIndexFunc, bothCheckFunc` are lambda functions that perform a hash function against the appropriate field of the edge to find where relevant intermediate results are
stored within the hash table of the ResultMap. The srcCheckFunc is a lambda function that takes as input a query result. It checks that the query result’s next edge has a bound source variable and an unbound target variable. Similarly, trgCheckFunc returns true if the query result’s next edge has an unbound source variable and bound target variable. The bothCheckFunc returns true if both variables are bound. These lambda functions are passed to the process(edge, indexer, checker) function, where the logic is the same for all three cases by using the different lambda functions.

The process(edge, indexer, checker) function (line 12) on page 79 uses the provided indexer to find the location (line 14) of the potentially relevant intermediate results in the hash table (class member alr). Line 15 finds the time of the edge. We use this on line 20 to judge whether an intermediate result has expired and can be deleted. On line 18 we iterate through all the intermediate results found in the bin alr[index]. We use the checker lambda function to make sure the intermediate result is of the expected form, and if so, we try to add an edge to the result. The addEdge function (line 23) adds an edge to the intermediate result if it can, and if so, returns a new intermediate result but leaves the pre-existing result unchanged. This allows the pre-existing intermediate result to match with other edges later. On line 25 we add the new result to the list, gen, of newly generated intermediate results. On line 31 we pass gen to processAgainstGraph.

The processAgainstGraph function is used to see if there are existing edges in the CSR or CSC that can be used to further complete the queries. The overall approach is to generate frontiers of modified results, and continue processing the frontier until no new intermediate results are created. Line 39 sets the frontier to be the beginning of gen’s iterator. Then we push back a null element onto gen on line 40 to mark the end of the current frontier. Line 41 continues iterating until there are no longer any new results generated. Line 42 iterates over the frontier until a null element is reach. Lines 45 and 46 find edges from the graph that match the provided intermediate result on the frontier. Then on line 47 we try to add the edges to the intermediate result, creating new results that are added to the new frontier on line 50. Eventually the updated list of intermediate results is returned to the process(edge, indexer, checker) function. This
The function add_nocheck is a private method that differs from the public add in that it doesn’t check the CSR and CSC, since that step has already been taken. It also generates edge requests for edges that will be found on other nodes, and that list is returned on line 35 and then in the overarching process function on line 9.

**Step 3: Checking registered queries:** When a GraphStore object receives a new edge, it must check to see if that edge satisfies the first edge of any registered queries. The process is outlined in Algorithm 3. The GraphStore.checkQueries(edge) function is straightforward. On line 3 we iterate through all registered queries, checking to see if the edge satisfies the first edge of the query (line 4), and if so, adds a new intermediate result to the ResultMap (line 5).

The ResultMap’s add function begins on line 11. We may create intermediate results where the next edge belongs to another node, so we initialize a list, requests, to store those requests (line 12). We need to check the CSR and CSC if the new result can be extended further by local knowledge of the graph. As such we make a call to processAgainstGraph on line 13. On line 16 we iterate through all generated intermediate results. If the result is not complete, we calculate an index (line 18), and place it within the array of lists of results (line 20) with mutexes to protect access. If the result is complete, we send it to the output destination (line 23). At the very end we return a list of requests, which is then aggregated by GraphStore’s checkQueries function, and then returned on line 8 and eventually used by step 5 to send out the requests to other nodes.

**Step 4: RequestMap.process(edge)** The purpose of RequestMap’s process(edge) is to take a new edge and find all open edge requests that match that edge. The process is detailed in Algorithm 4. The opening logic of the RequestMap.process(edge) function is similar to that of ResultMap’s process function. There are three lambda functions for indexing into RequestMap’s hash table structure, which is an array of lists of EdgeRequests (ale). There are also three lambda functions for checking that an edge matches a request. The indexer and the checker for each of the three cases are each passed to the process(edge, indexer, checker) function on lines 2-4. The process(edge, indexer, checker) function iterates through all of the edge requests with the same index as the current edge (line 11), deleteing all that have expired (line 13). If the edge
Algorithm 2 ResultMap: Processing a New Edge

1: function ResultMap.process(edge)
2:     requests, a list of edge requests.
3:     requests += process(edge, srcIndexFunc)
4:         srcCheckFunc)
5:     requests += process(edge, trgIndexFunc,
6:         trgCheckFunc)
7:     requests += process(edge, bothIndexFunc,
8:         bothCheckFunc)
9:     return requests
10: end function
11:
12: function ResultMap.process(edge, indexer, checker)
13:     gen, a list of intermediate results generated.
14:     index ← indexer(edge)
15:     currentTime ← getTime(edge)
16:     mutexes[index].lock()
17:     requests, a list of edge requests.
18:     for all r ∈ alr[index] do
19:         if r.isExpired(currentTime) then
20:             erase(r)
21:         else
22:             if checker(r) then
23:                 newR ← r.addEdge(edge)
24:                 if newR not null then
25:                     gen.push_back(newR)
26:             end if
27:         end if
28:     end for
29:     mutexes[index].unlock()
30:     gen ← processAgainstGraph(gen)
31:     for all g ∈ gen do
32:         requests += add_nocheck(g)
33:     end for
34:     return requests
35: end function
36:
37: function ResultMap.processAgainstGraph(gen)
38:     frontier ← gen.begin()
39:     gen.push_back(null)
40:     while frontier ≠ gen.end() do
41:         while frontier ≠ null do
42:             if !frontier.complete() then
43:                 ++frontier
44:                 foundEdges = csr.findEdges(frontier)
45:                 foundEdges += csc.findEdges(frontier)
46:                 for all edge ∈ foundEdges do
47:                     if newR not null then
48:                         gen.push_back(newR)
49:                     end if
50:                 end if
51:             end if
52:         end while
53:     end while
54:     frontier ← frontier.erase()
55:     gen.push_back(null)
56:     end while
57:     return gen
58: end function
Algorithm 3 Checking Registered Queries

1: function GraphStore.checkQueries(edge)
2:    requests, a list of edge requests.
3:    for all q ∈ queries do
4:        if q.satisfiedBy(edge) then
5:            requests += resultMap.add(Result(edge))
6:        end if
7:    end for
8:    return requests
9: end function

10: function RequestMap.add(result)
11:    requests, a list of edge requests.
12:    gen, a list of intermediate results generated.
13:    gen.push_back(result)
14:    gen ← processAgainstGraph(gen)
15:    for all g ∈ gen do
16:        if !g.complete() then
17:            index = g.hash()
18:            mutexes[index].lock()
19:            alr[index].push_back(g)
20:            mutexes[index].unlock()
21:        else
22:            output(g)
23:        end if
24:    end for
25:    return requests
26: end function
matches the request, the edge is then sent to another node using the EdgeCommunicator (line 16).

Algorithm 4 RequestMap: Processing a New Edge

```
1: function RequestMap.process(edge)
2:     process(edge, srcIndexFunc, srcCheckFunc)
3:     process(edge, trgIndexFunc, trgCheckFunc)
4:     process(edge, bothIndexFunc, bothCheckFunc)
5: end function
6:
7: function RequestMap.process(edge, indexer, checker)
8:     index ← indexer(edge)
9:     currentTime ← getTime(edge)
10:    mutexes[index].lock()
11:    for all e ∈ ale[index] do
12:        if e.isExpired(currentTime) then
13:            erase(e)
14:        else
15:            if checker(e, edge) then
16:                edgeCommunicator.send(edge)
17:            end if
18:        end if
19:    end for
20:    mutexes[index].unlock()
21: end function
```

5.3.2.2 Request Communicator Threads

Another group of threads are the pull threads of the Request Communicator. These threads pull from the ZMQ pull sockets dedicated to gathering edge requests from other nodes. One callback is registered, the requestCallback. The requestCallback calls two methods: addRequest and processRequestAgainstGraph. The addRequest method registers the request with the RequestMap. The method processRequestAgainstGraph checks to see if any existing edges are already stored locally that match the edge request.

5.3.2.3 Edge Communicator Threads

The last group of threads to discuss are pull threads of the Edge Communicator and its callback function, edgeCallback. Upon receiving an edge through the callback, ResultMap.process(edge)
is called, which is discussed in detail in section 5.3.2.1. The method `process(edge)` produces edge requests, which are then sent via the Request Communicator, the same as Step 5 of `GraphStore.consume(edge)`.

5.4 Summary

In this chapter we’ve covered the implementation of the Streaming Analytics Machine or SAM, which is a library used for expressing both streaming machine learning pipelines and temporal subgraph queries. SAM is over 19,000 lines of code. The Streaming Analytics Language, SAL, maps onto SAM classes, allowing SAL to run in parallel across a cluster.
Chapter 6

Vertex Centric Computations

In this chapter we discuss performance of vertex-centric computations expressed in SAL, i.e. programs that do not utilize subgraph matching. We discuss both the performance achieved for a trained classifier obtained from a SAL pipeline and the scaling observed on a cluster of size 64 nodes.

6.1 Classifier Results

To validate the value of SAL in expressing pipelines and in filtering out benign data, we used a simple program where we stream the netflows two ways, using two `STREAM BY` statements, one keyed by destination IP and the other by source IP. Then we create features based on the `ave` and `var` streaming operators for each of the fields:

1. `SrcTotalBytes`
2. `DestTotalBytes`
3. `DurationSeconds`
4. `SrcPayloadBytes`
5. `DestPayloadBytes`,
6. `SrcPacketCount`
7. `DestPacketCount`
This results in 28 total features. The entire program can be found in Appendix A.1.

We apply this pipeline to CTU-13 [63], which contains 13 botnet scenarios. CTU-13 has some nice characteristics, including:

- Real botnet attacks: A set of virtual machines were created and infected with botnets.
- Real background traffic: Traffic from their university router was captured at the same time as the botnet traffic. The botnet traffic was bridged into the university network.
- A variety of protocols and behaviors: The scenarios cover a range of protocols that were used by the malware such as IRC, P2P, and HTTP. Also some scenarios sent spam, others performed click-fraud, port scans, DDoS attacks, or Fast-Flux.
- A variety of bots: The 13 scenarios use 7 different bots.

Since the SAL program inherently encodes historical information (each generated feature is calculated on a sliding window of data), we can’t treat each netflow independently and thus can’t create random subsets of the data as is usual in a cross-validation approach. We instead split each scenario into two parts. We want to keep the number of malicious netflows about the same in each part (we need enough examples to train on), so we find the point in the scenario timewise where the malicious examples are balanced. Namely we have two sets, $P_1$ and $P_2$, where $\forall n_1 \in P_1, \forall n_2 \in P_2, \text{TimeSeconds}(n_1) < \text{TimeSeconds}(n_2)$ and $|\text{Malicious}(P_1)| \approx |\text{Malicious}(P_2)|$, where $\text{TimeSeconds}$ returns the time in seconds of the netflow and $\text{Malicious}$ returns a subset of the provided set of all the malicious netflows in the provided set. With each scenario split into two parts, we train on the first part and test on the second part. Then we switch: train on the second and test on the first. We make use of a Random Forest Classifier [28] as implemented in scikit-learn [163].

After generating the features as specified by the SAL program in Appendix A.1 we performed a greedy search over the features to down-select to the most important ones. The process is outlined in Algorithm 5 but the general notion is to add features one at a time until no improvement is
Algorithm 5 Greedy Process for Selecting Features

\[ \text{features} \leftarrow \text{Features from SAL program} \]
\[ \text{selected} \leftarrow \emptyset \]
\[ \text{bestAUC} \leftarrow 0 \]
\[ \text{prevAUC} \leftarrow -1 \]
\[ \text{improvement} \leftarrow \text{True} \]

while improvement \leftarrow \text{True} do
  \[ \text{iterBestAUC} \leftarrow 0 \]
  \[ \text{iterBestFeature} \leftarrow \text{None} \]
  for \( f \in \text{features} \) do
    \[ \text{candidateSet} \leftarrow \text{selected} \cup f \]
    \[ \text{candidateAUC} \leftarrow \text{Average AUC using candidateSet} \]
    if candidateAUC > \text{iterBestAUC} then
      \[ \text{iterBestAUC} \leftarrow \text{candidateAUC} \]
      \[ \text{iterBestFeature} \leftarrow f \]
    end if
  end for
  if iterBestAUC > bestAUC then
    \[ \text{bestAUC} = \text{iterBestAUC} \]
    \[ \text{features} = \text{features} - \text{iterBestFeature} \]
    \[ \text{selected} = \text{selected} \cup \text{iterBestFeature} \]
  else
    \[ \text{improvement} = \text{False} \]
  end if
end while
found in the average AUC over the 13 scenarios.

We found that the following eight features provided the best performance across the 13 scenarios. They are listed in the order they were added using the greedy approach above.

1. Average **DestPayloadBytes**
2. Variance **DestPayloadBytes**
3. Average **DestPacketCount**
4. Variance **DestPacketCount**
5. Average **SrcPayloadBytes**
6. Average **SrcPacketCount**
7. Average **DestTotalBytes**
8. Variance **SrcTotalBytes**

Using the above eight features, Figure 6.1 shows the AUC of the ROC for each of the 13 scenarios. In four of the scenarios, 1, 3, 6, and 11, training on either half was sufficient for the other half, with AUCs between 0.926 and 0.998. Some scenarios, namely 2, 5, 8, 12, and 13, the first half was sufficient to obtain AUCs between 0.922 and 0.992 on the second half, but the reverse was not true. For scenario 10, training on the second half was predictive of the first half, but not the other way. Scenario 9 had AUCs of 0.820 and 0.887, which is decent, but lackluster compared to the other scenarios. The classifier had issues with scenarios 4 and 7. Scenario 7 did not perform well, probably because there were only 63 malicious netflows, as can be seen in Table 6.1. We are not sure why the classifier struggled with scenario 4.

### 6.2 Scaling

For our scaling experiments, we made use of Cloudlab [168], which is a set of clusters distributed across three sites, Utah, Wisconsin, and South Carolina, where researchers can provision
Figure 6.1: This graphic shows the AUC of the ROC curve for each of the CTU scenarios. We first train on the first half of the data and test on the second half, and then switch.

Table 6.1: A more detailed look at the AUC results. The second column is the AUC of the ROC after training on the first half of the data and then testing on the second half. The third column is reversed: trained on the second half and tested on the first half.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>AUC first</th>
<th>AUC second</th>
<th>Netflows</th>
<th>Botnet Netflows</th>
<th>% Bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.944</td>
<td>0.926</td>
<td>2,824,636</td>
<td>40,961</td>
<td>1.45%</td>
</tr>
<tr>
<td>2</td>
<td>0.930</td>
<td>0.887</td>
<td>1,808,122</td>
<td>20,941</td>
<td>1.16%</td>
</tr>
<tr>
<td>3</td>
<td>0.998</td>
<td>0.996</td>
<td>4,710,638</td>
<td>26,822</td>
<td>0.57%</td>
</tr>
<tr>
<td>4</td>
<td>0.476</td>
<td>0.775</td>
<td>1,121,076</td>
<td>2,580</td>
<td>0.23%</td>
</tr>
<tr>
<td>5</td>
<td>0.922</td>
<td>0.843</td>
<td>129,832</td>
<td>901</td>
<td>0.69%</td>
</tr>
<tr>
<td>6</td>
<td>0.972</td>
<td>0.976</td>
<td>558,919</td>
<td>4,630</td>
<td>0.83%</td>
</tr>
<tr>
<td>7</td>
<td>0.604</td>
<td>0.761</td>
<td>114,077</td>
<td>63</td>
<td>0.06%</td>
</tr>
<tr>
<td>8</td>
<td>0.992</td>
<td>0.886</td>
<td>2,954,230</td>
<td>6,127</td>
<td>0.21%</td>
</tr>
<tr>
<td>9</td>
<td>0.820</td>
<td>0.887</td>
<td>2,087,508</td>
<td>184,987</td>
<td>8.86%</td>
</tr>
<tr>
<td>10</td>
<td>0.666</td>
<td>0.950</td>
<td>1,309,791</td>
<td>106,352</td>
<td>8.12%</td>
</tr>
<tr>
<td>11</td>
<td>0.996</td>
<td>0.995</td>
<td>107,251</td>
<td>8,164</td>
<td>7.61%</td>
</tr>
<tr>
<td>12</td>
<td>0.903</td>
<td>0.796</td>
<td>325,471</td>
<td>2,168</td>
<td>0.67%</td>
</tr>
<tr>
<td>13</td>
<td>0.938</td>
<td>0.770</td>
<td>1,925,149</td>
<td>40,003</td>
<td>2.08%</td>
</tr>
</tbody>
</table>
a set of nodes to their specifications. This allowed us to create an image where our code, SAM, was deployed and working, and then replicate that image to a cluster size of our choice. In particular we make use of the Clemson system in South Carolina. The Clemson system has 16 cores per node, 10 Gb/s Ethernet and 256 GB of memory. We were able to allocate a cluster with 64 nodes.

Figure 6.2 shows the weak scaling results. Weak scaling examines the solution time where the problem size scales with the number of nodes, i.e. there is a fixed problem size per node. For our experiments, each node was fed one million netflows. Thus, for \( n \) nodes, the total problem size is \( n \) million netflows. Each node had available the entire CTU dataset concatenated into one file. Then we randomly selected for each node a contiguous chunk of one million netflows. For each randomly selected chunk, we renamed the IP addresses to simulate a larger network instead of replaying the same IP addresses, just from different time frames. For comparison we also ran another round with completely random IP addresses, such that an IP address had very little chance of being in multiple netflows. This helped us determine if scaling issues on the realistic data set were from load balancing problems or some other issue. Each data point is the average of three runs.

As can be seen from Figure 6.2, the renamed IP set of runs peaks out at 61 nodes or 976 cores where we obtain a throughput of 373,000 netflows per second or 32.2 billion per day. For the randomized IP set of runs, the scaling is noticeably steeper, reaching a peak throughput of 515,000 netflows per second (44.5 billion per day) with 63 nodes. Figure 6.3 takes a look at the weak scaling efficiency. This is defined as \( E(n) = T_1/T_n \), where \( T_i \) is the time taken by a run with \( i \) nodes. Efficiency degrades quicker for the renamed IP set of runs while the randomized IP runs hover close to 0.4. We believe the difference is due to load balance issues. For the randomized IP set of runs, the work is completely balanced between all 64 nodes. The renamed IP runs are more realistic, akin to a power law distribution, where a small set of nodes account for most of the traffic. In this situation, it becomes difficult to partition the work evenly across all the nodes.

As far as we are aware, we are the first to show scalable distributed network analysis on a cluster of size 64 nodes. The most direct comparison to other works in terms of scaling is Bumgardner and Marek [33]. In this work, they funnel netflows through a 10 node cluster running
Figure 6.2: Weak Scaling Results: For each run with $n$ nodes, a total of $n$ million netflows were run through the system with a million contiguous netflows per node randomly selected from the CTU dataset. There were two types of runs, with renamed IPs (to create the illusion of a larger network) or with randomized IPs. For the renamed IPs, there continues to be improved throughput until 61 nodes / 976 cores. For the randomized IPs, the scaling continues through the total 64 nodes / 1024 cores.

Figure 6.3: This shows the weak scaling efficiency. The efficiency continues to degrade for the renamed IPs, but for the randomized IPs, the efficiency hovers a little above 0.5. This is largely due to the fact that the randomized IPs almost perfectly load-balances the work, while the renamed IPs encounters load imbalances because of the power-law nature of the data.
They call their approach a hybrid stream/batch system because it uses Storm to stream netflow data into a batch system, Hadoop, for analysis. The stream portion is what is most similar to our work. Over the Storm pipeline, they achieve a rate of 234,000 netflows per second, or about 23,400 netflows per second, per node. Our pipeline with ten nodes achieved a rate of 13,900 netflows per second, per node. However, their pipeline is much simpler. They do not partition the netflows by IP and calculate features. The only processing they undergo during streaming is adding subnet information to the netflows, which does not require partitioning across the cluster.

There is other work which performs distributed network analysis, but their focus is on packet-level data. As such it is difficult to compare as their metrics are in terms of GBs/second, and of course the nature of packet analysis is significantly different than netflow analysis.

In terms of botnet identification, Botfinder and Disclosure report single node batch performance numbers. Our approach on a single 16-core node achieved a rate of 30,500 netflows per second. For Botfinder, they extract 5 features on a 12 core Intel Core i7 chip, achieving a rate of 46,300 netflows per second. Disclosure generates greater than nine features (the text is ambiguous in places, referencing unspecified statistical features) on a 16 core Intel Xeon CPU E5630. They specify that they run the feature extraction for one day’s worth of data in 10 hours and 53 minutes, but they do not clearly state if that is on both data sets they chose to evaluate or just one. If it is both, the rate is roughly 40,000 netflows per second. Both of these approaches are batch processes.

In the work of hybrid batch/stream system, Bumgardner and Marek note a ten fold increase in throughput going from streaming to batch processing. For the stream processing phase, they achieved a throughput of 234,000 netflows per second (on ten nodes). They then run MapReduce jobs on the stored data in batch, achieving rates between 1.7 million netflows per second to 2.6 million netflows per second (again on ten nodes). Since our approach is streaming, comparing to rates achieved by batch processes may not be a fair characterization. During stream processing, the network becomes the limiting factor while during batch processing, the network becomes less
of an issue.

6.3 Summary

In this chapter we’ve exercised the vertex-centric portion of SAL. We developed a SAL program for the CTU-13 [63] dataset and gathered both classifier and scaling performance. The trained classifier obtained an average AUC of the ROC of 0.87. In an operational setting, this could be used to greatly reduce the amount of traffic needing to be analyzed. SAL can be used as a first pass over the data using space and temporally-efficient streaming algorithms. After down-selecting, more expensive algorithms can be applied to the remaining data.

In terms of scalability using SAM, we are able to scale to 61 nodes and 976 cores, obtaining a throughput of 373,000 netflows per second or 32.2 billion per day. On completely load-balanced data, we obtain greater efficiency out to 64 nodes than the real data, indicating that SAM could be improved with a more intelligent partitioning strategy. In the end, we’ve shown SAL can be used to succinctly express machine learning pipelines and that our implementation can scale to large problem sizes.
Chapter 7

Temporal Subgraph Matching

In this chapter we explore the performance obtained by SAL and SAM with regards to temporal subgraph matching. We focus on the temporal triangle query presented earlier in Chapter 2 but repeated in its entirety in Listing 7.1 below for convenience. The query is looking for a set of three vertices, \( x_1 \), \( x_2 \), and \( x_3 \), where there are three edges forming the triangle: \( x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_1 \). Also, edge \( e_1 \) happens before \( e_2 \), which happens before \( e_3 \). All together the edges must occur within a ten second window. We chose this query because it complex enough to meaningfully exercise the framework, but simple enough that it could be a component of a query.

The contributions of this chapter are the following:

- We present scaling performance of SAM, showing that it continues to increase throughput to 128 nodes or 2560 cores, the maximum cluster size cluster we could reserve.

- We also compare SAL and SAM to Apache Flink [12], which is another framework designed and built for streaming applications. The comparison reveals that SAM and Flink process different strengths dependent on the parameter regime. SAM does better when triangles occur frequently (roughly greater than five triangles per second per node), while Flink achieves the best throughput when triangles are rare. Also, while SAM scales to 128 nodes, Flink plateaus after 32 nodes.

- SAL again provides savings in terms of programming complexity. The full SAL program for temporal triangles requires 13 lines of code while a direct implementation using SAM
directly takes 238 lines and the Apache Flink approach is 228 lines long.

- We simulate vertex constraints that would filter out many intermediate results. In this situation SAM obtains a rate of 1 million netflows per second or 91.8 billion netflows per day.

```java
Netflows = VastStream("localhost", 9999);
PARTITION Netflows By SourceIp, DestIp;
HASH SourceIp WITH IpHashFunction;
HASH DestIp With IpHashFunction;
SUBGRAPH ON Netflows WITH source (SourceIp) AND target (DestIp)
{
    x1 e1 x2;
    x2 e2 x3;
    x3 e3 x1;
    start(e3)     start(e1) <= 10;
    start(e1) < start(e2);
    start(e2) < start(e3);
}
```

For all of our experiments in this chapter we again use Cloudlab [168]. In particular we make use of the Wisconsin cluster using node type c220g5, which has two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz and 192 GB ECC DDR4-2666 memory per node. We run experiments up to 128 nodes or 2560 cores. This is in contrast to Chapter 6 which had 16 cores per node.

Section 7.1 explores hyperparameters that were imperative for SAM scaling. Section 7.2 discusses the Apache Flink implementation of the triangle query that we used for comparison. Then in Section 7.3 we present the comparison between SAM and Flink running the same query. Section 7.4 simulates vertex constraints that would filter out many initial intermediate results, giving us an idea of the performance when the subgraph pattern is relatively rare.
Figure 7.1: We vary the number of pull threads and the number of push sockets (NPS) for the RequestCommunicator to determine the best setting for maximum throughput. The RequestMap object utilizes the RequestCommunicator during the `process(edge)` function.

### 7.1 SAM Parameter Exploration

In doing performance analysis of SAM and the algorithm outlined in Section 5.3.2, we found that a large portion of the time was spent in the `process(edge)` function of the RequestMap object. In particular, the time spent sending edges owned by one node to other nodes was significant. As such it became the focus of performance enhancements.

The solution we developed, discussed earlier in Section 5.3.1.3 and shown in Figure 5.2, was to create multiple push sockets to send data to an individual node, and also multiple pull threads to receive the data. Having multiple push sockets prevents hot spots, where multiple threads are trying to use the same socket. Multiple pull threads allows data ingestion to happen in parallel.

To find the ideal combination of push sockets and pull threads, we performed the following experiment. We ran the temporal triangle query of Listing 2.3 with the following parameters:

- The table capacity for the ResultMap, CSR, CSC, and RequestMap were all set to be 2 million.
- The number of unique vertices, $|V|$, was set to 50,000.
- The rate at which edges were produced was 1000 per second.
• The width of the time window was 11 seconds. If an item is more than 11 seconds old, it is deleted. Since the length of the query was 10 seconds, this gives us a little buffer if data comes in late from other nodes.

• We allocated a 64 node cluster on Wisconsin. As mentioned earlier, these nodes have 20 cores.

With the above parameters, we then varied the number of pull threads from 1 to 24 and the number of push sockets from 1 to 8. Figure 7.1 displays the average time spent in `RequestMap.process(edge)`, the function that calls `RequestCommunicator.send(message)` to send out edges to other nodes according to recorded requests. The dominating factor to reduce the time spent in `RequestMap.process(edge)` was to increase the number of pull threads to around 16 threads. No sizeable benefit is garnered by adding more than 16 threads, likely due to the fact that each node had 20 total cores. We did do some experiments with more than 24 threads, but the constant thread context switching became overwhelming and SAM could not keep pace.

The other trend is some benefit by adding multiple push sockets. There is a large jump in performance from 1 to 2 push sockets. The trend continues, and is especially clear when there is just one pull thread. However, at 16 pull threads, the distinction becomes more muddled.

There are some oddities we are currently at a loss for explaining. For example, when the number of push sockets is 3, there is a large dip when pull threads is 10. This behavior appears to be repeatable but we have not investigated thoroughly. Regardless, using 16 pull threads and 4 push sockets appears to be a good parameter setting for this experiment, and that is what we used for the Flink/SAM comparison in Section 7.3.

As a result of our technique, we were able to increase both the network and processor utilization. Using one push socket per node in the cluster and one pull thread for all receiving sockets, we sent about 20-30 thousand packets a second. With 16 pull threads and 4 push sockets, we pushed that up to about 225 thousand packets a second. Also, we increased how much the CPU is being utilized. It went from about 2-3 cores to almost full utilization at 15-20 cores. With this increased
utilization we were able to obtain scaling on problems to 128 nodes, as we will see in Sections 7.3 and 7.4.

7.2 Apache Flink

We have currently mapped SAL into SAM as the implementation. SAL is converted into SAM code using the Scala Parser Combinator [175]. However, we wanted to compare how another framework would perform. It is certainly possible to map SAL into other languages and frameworks. A prime candidate for evaluation is Apache Flink [12], which was custom designed and built with streaming applications in mind. In this section, we outline the approach taken for implementing the triangle query of Listing 2.3 using Apache Flink. Once the groundwork has been laid, the actual specification of the algorithm is relatively succinct. The java code using Flink 1.6 is presented in Listing 7.2. The full code is provided in Appendix A.2.
Listing 7.2: Apache Flink Triangle Implementation

```java
final StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(numSources);
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

NetflowSource netflowSource = new NetflowSource(numEvents, numIps, rate);
DataStreamSource<Netflow> netflows = env.addSource(netflowSource);

DataStream<Triad> triads = netflows
    .keyBy(new DestKeySelector())
    .intervalJoin(netflows.keyBy(new SourceKeySelector()))
    .between(Time.milliseconds(0),
            Time.milliseconds((long) queryWindow * 1000))
    .process(new EdgeJoiner(queryWindow));

DataStream<Triangle> triangles = triads
    .keyBy(new TriadKeySelector())
    .intervalJoin(netflows.keyBy(new LastEdgeKeySelector()))
    .between(Time.milliseconds(0),
            Time.milliseconds((long) queryWindow * 1000))
    .process(new TriadJoiner(queryWindow));

triangles.writeAsText(outputFile, FileSystem.WriteMode.OVERWRITE);
```

Lines 1–4 define the streaming environment, such as the number of parallel sources (generally the number of nodes in the cluster) and that event time of the netflow should be used as the timestamp (as opposed to ingestion time or processing time). Lines 6–7 create a stream of netflows. `NetflowSource` is a custom class that generates the netflows in the same manner as was used for the SAL/SAM experiments. Lines 9–14 finds triads, which are sets of two connected
edges that fulfill the temporal requirements. The variable `queryWindow` on line 13 was set to be ten in our experiments, corresponding to a temporal window of 10 seconds.

Line 14 uses an interval join. Flink has different types of joining approaches. The one that was most appropriate for our application is the interval join. It takes elements from two streams (in this case the same stream `netflows`), and defines an interval of time over which the join can occur. In our case, the interval starts with the timestamp of the first edge and it ends `queryWindow` seconds later. This join is a self-join: it merges the `netflows` data stream keyed by the destination IP with the `netflows` data stream keyed by the source IP. In the end we have a stream of pairs of edges with three vertices of the form \( A \rightarrow B \rightarrow C \).

Once we have triads that fulfill the temporal constraints, we can define the last piece: finding another edge that completes the triangle, \( C \rightarrow A \). Lines 16-21 performs another interval join, this time between the triads and the original `netflows` data streams. The join is defined to occur such that the time difference between the starting time of the last edge and the starting time of the first edge must be less than `queryWindow` seconds.

Overall, the complete code is 228 lines long, and can be found in Appendix A.2. This number includes the code that would likely need to be generated were a mapping between SAL and Flink completed. This includes classes to do key selection (`SourceKeySelector`, `DestKeySelector`, `LastEdgeKeySelector`, `TriadKeySelector`), the `Triad` and `Triangle` classes, and classes to perform joining (`EdgeJoiner` and `TriadJoiner`). However, the code for the `Netflow` and the `NetflowSource` classes were not included, as those classes would likely be part of the library as they are with SAM. For comparison, the SAL version of this query is 13 lines long.

7.3 Comparison Results: SAM vs Flink

In this section we compare the performance of SAM with the custom Apache Flink implementation. In both cases, we run the triangle query of Listing 2.3 which matches triangles that occur within ten seconds of the start time of the first edge. We examine weak scaling, wherein the number of elements per node is kept constant as we increase the number of nodes. In all runs, each
node is fed 2,500,000 edges. The edges are netflows, as that is our intended initial target domain. There are many different formats for netflows. We use the format used by the VAST Challenge 2013: Mini-Challenge 3 dataset [201], where each netflow has 19 fields. However, for this experiment we only change the source and destination IPs, which are randomly selected from a pool of $|V|$ vertices for each netflow generated. We find the highest rate achievable by each approach by incrementally increasing the rate at which edges are produced until the throughput is too great for the method to keep pace.

For the Apache Flink runs we set both the job manager and task manager heap sizes to be 32 GBs. The number of taskmanager task slots was set to 1. As we only ever ran one Flink job at a time, this seemed to be the most appropriate setting. We set the number of sources to be the size of the cluster.

We can obtain an estimate on the number triangles as a function of $r$, the rate of edges produced per second, $|V|$, the number of unique vertices, $|E|$, the number of edges, $t$, the length of the query in seconds, and $n$, the number of nodes in the cluster.

**Theorem 2.** Let $r$ be the rate at which edges are presented per node. Let $|V|$ be the number of vertices from which target and source vertices are randomly selected. Let $t$ be the length of the query in seconds for the triangle query from Listing 2.3. Let $n$ be the number of nodes in the cluster. Assuming a continuous stream of edges, if we select a contiguous set of $|E|$ edges from each node from the stream of edges, the total number of expected triangles fulfilling the query of Listing 2.3 is

$$n \left( \frac{n r t}{|V|} \right)^2.$$

**Proof.** For each edge $e_a$ that is presented in the stream, there is a set, $E_b$, of edges that occur after $e_a$ that fulfill the temporal constraints, where $|E_b| = n r t$. For each $e_b \in E_b$, the probability of the $\text{target}(e_a)$ being equal to $\text{source}(e_b)$ is $\frac{1}{|V|}$. Thus the expected number of triads matching the first two edges of Listing 2.3 for each edge $e_b$ is $\frac{n r t}{|V|}$. For each matching triad, there is a set, $E_c$, of edges that occur after $e_b$ that fulfill the temporal constraints. However, since triads are expected to occur uniformly over the time period $t$, and the query must be completed within $t$ seconds, each
triad does not have the same expected number of edges in $E_c$. For the first triad, the expected number of matching edges is $\frac{nnrt}{|V|^2}$, because the first triad has the entire $t$ seconds worth of edges to match against, and each edge must fulfill both $target(e_b) = source(e_c)$ and $target(e_c) = source(e_a)$.

The last triad occurs near the end of the time interval, and is expected to have no edges to match against, so there are 0 expected triangles. The series of expected triangles from the first triad to the last forms an arithmetic sequence, which is of length $\frac{nnrt}{|V|}$, so overall the total number of triangles is $\frac{1}{2|V|} \left( \frac{nnrt}{|V|} \right)^2$. Since there are $n$ nodes, each producing $|E|$ edges, then the total number of expected triangles over all the edges is $\frac{n|E|}{2|V|} \left( \frac{nnrt}{|V|} \right)^2$.

For both implementations, we kept increasing the rate until the run time exceeded $|V|/r + 15$ seconds, where $r$ is the rate of edges produced per second. Latencies longer than 15 seconds usually indicated that the implementation was unable to keep pace. Also, we stopped increasing the rate when the number of triangles was less than 85% of the expected value. Generally the solutions stay above this value unless they are struggling.

Figure 7.2 shows the overall max aggregate throughput achieved by SAM and Flink. We varied the number of vertices, $|V|$, to be 5,000, 10,000, 25,000, 50,000, 100,000, and 150,000. For $|V| = 5,000$ or 10,000, SAM obtained the best overall rate, continuing to garner increasing aggregate throughput to 120 nodes (we could only obtain 120 nodes for this experiment, the next section reports 128 node performance). Flink struggled with more than 32 nodes. While Flink continued to report results, the percentage of expected triangles fell dramatically to about 60-70% for 64 nodes, and around 50% for 120 nodes. While Flink was not able to scale past 32 nodes, it showed the best overall throughput for $|V| = 50,000$ and 150,000. $|V| = 25,000$ was near the middle ground, with Flink performing slightly better, with 32 nodes obtaining an aggregate rate of 60,800 edges per second while SAM obtained 60,000 edges per second on 120 nodes.

Figure 7.3 provides another view of the data. We show the max aggregate throughput versus a measure of the complexity of the problem: the total number of triangles. The graph emphasizes the fact that SAM performs better with more complex problems, i.e. with more total triangles to
Figure 7.2: Shows the maximum throughput for SAM and Flink. We vary the number of nodes with values of 16, 32, 64, and 120. We vary the number of vertices, $|V|$, between 25,000 and 150,000. Flink has the best throughput with 32 nodes, after which adding additional nodes hurts the overall aggregate rate and the number of found triangles continues to decrease. SAM is able to scale to 120 nodes.

Figure 7.3: Another view of the data where we plot the aggregate throughput versus the total number of triangles to find. This view emphasizes that SAM does better with more complex problems, i.e. more triangles.
Figure 7.4: Another look at the maximum throughput for SAM and Flink. Here we relax the restriction that the number of reported triangles must be higher than 85% of expected, but keep the latency restrictions that the computation must complete within 15 seconds of termination of input. Flink struggles increasing to 64 nodes, and then drops consistently with 120 nodes. SAM continues to increase throughput through 120 nodes.

Figure 7.5: This graph shows the ratio of found triangles divided by the number of expected triangles given by Theorem 2. The rate for each method was the same as from Figure 7.4. SAM stays fairly consistent throughout the runs, with an average of 0.937. For Flink, the ratio decreases as the number of nodes increases. The average ratio starts at 0.873 with 16 nodes, and then decreases to an average of 0.521 with 120 nodes.
find.

Figure 7.4 displays the max aggregate throughput again, but this time without the restriction that the number of reported triangles must be higher than 85% of expected. This figure emphasizes the fact that Flink struggles after 32 nodes. Figure 7.5 examines the ratio of found triangles divided by expected triangles for each approach. SAM stays fairly consistent, with an average of 93.7% of expected. Flink decreases as the number of nodes increases, starting with 87.3% of expected on average with 16 nodes, then decreasing to 52.1% of expected with 120 nodes.

To summarize this section, SAM and Flink have competing strengths. SAM excels when the triangle formation rate is greater than 5 per second per node, and SAM scales to 120 nodes. Flink does better when triangles are relatively rare, but cannot scale past 32 nodes. Both approaches show some loss due to network latencies. If some data arrives too late, it is not included in the calculations for either SAM or Flink. However, SAM has an advantages in this regard, showing consistent results throughout the processor count range which is higher than Flink.

7.4 Simulating other Constraints

We envision SAL being used to not only select queries based on the subgraph structure, but also on vertex-based constraints. The Watering Hole query of Listing 2.1 is an example. There are constraints defined on both the bait and controller vertices. In this section we simulate vertex constraints by randomly dropping the first edge of the triangle query, i.e. we process the netflow through SAM, but we force the edge to not match the first edge of the triangle query. This will give us an idea of SAM’s performance when the subgraph queries are more selective. The triangle query of Listing 2.3 as specified creates an intermediate result for every consumed netflow since there are no vertex constraints.

We compare SAM’s performance over three settings: keeping the first edge all of the time, 1% of the time, and .1% of the time. We vary $|V|$ to be 25,000, 50,000, 100,000, and 150,000. Figure 7.6 presents the results. The general trends are as expected. As we increase $|V|$, the number of triangles created decreases, and the rate goes up. Also, as we increase the rate at which edges
are kept causes a decrease in the maximum achievable rate, again because more triangles are being produced, which causes a greater computational burden. The maximum achievable rate is 1,062,400 netflows per second, when we keep 0.01% of the first edges and \(|V| = 150,000\).

Figure 7.6: We keep the first edges of queries \((kq)\) with probabilities \(\{0.01, 0.1, 1\}\). We also vary \(|V|\) to range from 25,000 to 150,000. This figure is a plot of edges/second versus the number of nodes.

As the number of triangles is likely the predominant factor in determining maximum throughput, another way of looking at the data is to plot the rate versus triangles/second, which we do in Figure 7.7. We only examine the runs with 128 nodes, and vary the number of kept queries. Also, in this figure we calculate the rate as edges/day. As expected, the rate at which triangles occur is inversely proportional to maximum throughput. The maximum rate achieved is 91.8 billion edges per day.

7.5 Summary

In this chapter we presented performance of SAM on subgraph queries, namely a temporal triangle query, and compare SAM to a custom solution written for Apache Flink. SAM excels when triangles occur frequently (greater than 5 per second per node) while Flink is best when triangles are rare. For all parameter settings, SAM showed improved throughput to 128 nodes or 2560 cores, while Flink’s rate started to decrease after 32 nodes. For more selective queries,
Figure 7.7: We vary the rate at which the first edges of queries are kept ($kq =$ keep queries) and plot the edges per day versus the rate at which triangles occur.

SAM is able to obtain an aggregate rate 1 million netflows per second. While we have presented SAM as the implementation for SAL, Apache Flink shows promise as another potential underlying backend. Future work understanding the performance differences between both platforms will likely be beneficial for both approaches.

We also have further evidence that programming in SAL is an efficient way to express streaming subgraph queries. SAL requires only 13 lines of code, while programming directly with SAM is 238 lines and the Flink approach is 228 lines.
In this dissertation we have presented a new domain specific language called the Streaming Analytics Language or SAL. SAL ingests streaming data in the form of tuples, and allows analysis that combines graph theory, streaming, temporal information, and machine learning. In particular we examined streaming cyber data and showed how SAL can be used to express a wide range of queries relevant to cyber security and detecting malicious patterns. We used Verizon’s Data Breach Investigation Report as a backdrop for discussing categories of attacks involving computer networks, and showed how SAL can be utilized to detect malicious patterns. We also showed that SAL programs are succinct, requiring 10-25 fewer lines of code than writing using our library, the Streaming Analytics Machine or SAM, or with another framework such as Apache Flink.

As another contribution, we have presented SAM which is one possible implementation target of SAL. We examined the scaling performance of SAM on two different problems: a vertex-centric computation to detect botnet activity and a temporal subgraph query. We showed scaling to 61 nodes or 976 cores for the vertex-centric computations, a scaling result not seen for related netflow analysis. We also garnered an average area under the curve of the receiver operating characteristic of 0.87 in detecting botnet activity. For subgraph matching, we showed scaling to 128 nodes or 2560 cores for temporal triangle finding. An alternative approach using custom written Apache Flink code could not scale past 32 nodes, and SAM had an overall higher throughput when the triangles occurred greater than 5 per second per node.

For future work in terms of the underlying implementation, a fecund avenue could be to
combine the strengths of our approach, SAM, with that of streaming frameworks such as Apache Flink. We showed that SAM excels at subgraph matching when the pattern is frequent; however, Flink was much more efficient with rare patterns. If we can tune the approach based on the distribution present in the data, SAL would be able to handle high throughput streaming data with greater robustness.

In terms of the expressibility of SAL, we primarily focused on homogeneous data - all data sources were of the same type, namely netflows. However, cyber has many different levels and layers that could be combined in the same workflow. For example, host-based intrusion detection systems can generate alerts that can then be combined with network-level analysis. In terms of Point of Sale breaches, unusual memory activity by a process on a Point of Sale machine might be a weak indicator, but when combined with corresponding network traffic that indicates exfiltration from the same machine, could be a powerful way to combine alerts into the same picture. SAL could be used to succinctly express this combination of patterns, allowing for fewer false positives and overall stronger indicators of malicious activity.

Finally, network telemetry is another avenue of research where SAL could be extended. Our work so far has assumed that netflow generation has already been taken care of at variety of sensors. We then ingest the netflows and perform our analysis with SAL. However, switches today allow for configurable programmable packet forwarding, opening opportunities for SAL to be compiled as a combination of switch computing resources and an analytics plane.
Bibliography


[40] François Chollet et al. Keras. https://keras.io, 2015.


[58] FBI. The insider threat.


118


[140] Natarajan Meghanathan, editor. Graph Theoretic Approaches for Analyzing Large-Scale Social Networks. IGI Global, Hershey, PA, 2018.


Appendix A

SAL and Flink Programs

This Appendix contains complete code samples of SAL and Apache Flink programs used in this work.

A.1 Machine Learning Example

This SAL query is used on the CTU-13 dataset [63] as discussed in Chapter [6].

Listing A.1: Machine Learning Program

```
Edges = VastStream("localhost", 9999);
PARTITION Edges By SourceIp, DestIp;
HASH SourceIp WITH IpHashFunction;
HASH DestIp With IpHashFunction;
VerticesByDest = STREAM Edges BY DestIp;
VerticesBySrc = STREAM Edges BY SourceIp;
feature0 = FOREACH VerticesBySrc GENERATE ave(SrcTotalBytes,10000,2);
feature1 = FOREACH VerticesBySrc GENERATE var(SrcTotalBytes,10000,2);
feature2 = FOREACH VerticesBySrc GENERATE ave(DestTotalBytes,10000,2);
feature3 = FOREACH VerticesBySrc GENERATE var(DestTotalBytes,10000,2);
feature4 = FOREACH VerticesBySrc GENERATE ave(DurationSeconds,10000,2);
feature5 = FOREACH VerticesBySrc GENERATE var(DurationSeconds,10000,2);
feature6 = FOREACH VerticesBySrc GENERATE ave(SrcPayloadBytes,10000,2);
```
A.2 Apache Flink Triangle Implementation

This is the full Apache Flink program used to compare against the SAL triangle query as discussed in Section 7.2. We present the Netflow class, the NetflowSource class, and the actual benchmark code. The Netflow class represents the VAST netflow format [201]. NetflowSource extends the RichParallelSourceFunction class which allows the generation of netflow data across the Flink cluster. Only the benchmark code was counted in the total line count for the Apache Flink implementation.
Listing A.2: Apache Flink Netflow Class

```java
package edu.cu.flink.benchmarks;

public class Netflow {
    public int samGeneratedId;
    public int label;
    public double timeSeconds;
    public String parseDate;
    public String dateTimeString;
    public String protocol;
    public String protocolCode;
    public String sourceIp;
    public String destIp;
    public int sourcePort;
    public int destPort;
    public String moreFragments;
    public int countFragments;
    public double durationSeconds;
    public long srcPayloadBytes;
    public long destPayloadBytes;
    public long srcTotalBytes;
    public long destTotalBytes;
    public long firstSeenSrcPacketCount;
    public long firstSeenDestPacketCount;
    public int recordForceOut;

    public Netflow(int samGeneratedId,
                    int label,
                    ...)
```
double timeSeconds,
String parseDate,
String dateTimeString,
String protocol,
String protocolCode,
String sourceIp,
String destIp,
int sourcePort,
int destPort,
String moreFragments,
int countFragments,
double durationSeconds,
long srcPayloadBytes,
long destPayloadBytes,
long srcTotalBytes,
long destTotalBytes,
long firstSeenSrcPacketCount,
long firstSeenDestPacketCount,
int recordForceOut
{
    this.label = label;
    this.timeSeconds = timeSeconds;
    this.parseDate = parseDate;
    this.dateTimeString = dateTimeString;
    this.protocol = protocol;
    this.protocolCode = protocolCode;
    this.sourceIp = sourceIp;
    this.destIp = destIp;
    this.sourcePort = sourcePort;
    this.destPort = destPort;
    this.moreFragments = moreFragments;
    this.countFragments = countFragments;
    this.durationSeconds = durationSeconds;
    this.srcPayloadBytes = srcPayloadBytes;
    this.destPayloadBytes = destPayloadBytes;
    this.srcTotalBytes = srcTotalBytes;
    this.destTotalBytes = destTotalBytes;
    this.firstSeenSrcPacketCount = firstSeenSrcPacketCount;
    this.firstSeenDestPacketCount = firstSeenDestPacketCount;
    this.recordForceOut = recordForceOut;
}

public String toString()
{
    return timeSeconds + "," + sourceIp + "," + destIp;
}

Listing A.3: Apache Flink NetflowSource Class

package edu.cu.flink.benchmarks;

import org.apache.flink.api.common.functions.RuntimeContext;
import org.apache.flink.streaming.api.functions.source.∗;
import org.apache.flink.streaming.api.watermark.Watermark;
import java.time.Instant;
import java.util.Random;

/**
* This creates netflows from a pool of IPs.

 */

public class NetflowSource extends RichParallelSourceFunction<Netflow> {

    /// The number of netflows to create.
    private int numEvents;

    /// How many netflows created.
    private int currentEvent;

    /// How many unique IPs in the pool
    private int numIps;

    /// Used to create the netflows with randomly selected ips
    private Random random;

    /// Time from start of run
    private double time = 0;

    /// The time in seconds between netflows
    private double increment = 0.1;

    /// When the first netflow was created
    private double t1;

    public int samGeneratedId;
    public int label;
    public double timeSeconds;
    public String parseDate;
    public String dateTimeString;
public String protocol;
public String protocolCode;
public String sourceIp;
public String destIp;
public int sourcePort;
public int destPort;
public String moreFragments;
public int countFragments;
public double durationSeconds;
public long srcPayloadBytes;
public long destPayloadBytes;
public long srcTotalBytes;
public long destTotalBytes;
public long firstSeenSrcPacketCount;
public long firstSeenDestPacketCount;
public int recordForceOut;

/**
 * @param numEvents The number of netflows to produce.
 * @param numIps The number of ips in the pool.
 * @param rate The rate of netflows produced in netflows/s.
 */
public NetflowSource(int numEvents, int numIps, double rate)
{
    this.numEvents = numEvents;
    currentEvent = 0;
    this.numIps = numIps;
    this.increment = 1 / rate;
    //this.random = new Random();
label = 0;
parseDate = "parseDate";
dateTimeString = "dateTimeString";
protocol = "protocol";
protocolCode = "protocolCode";
sourcePort = 80;
destPort = 80;
moreFragments = "moreFragments";
countFragments = 0;
durationSeconds = 0.1;
srcPayloadBytes = 10;
destPayloadBytes = 10;
srcTotalBytes = 15;
destTotalBytes = 15;
firstSeenSrcPacketCount = 5;
firstSeenDestPacketCount = 5;
recordForceOut = 0;

@Override
public void run(SourceContext<Netflow> out) throws Exception {
    RuntimeContext context = getRuntimeContext();
    int taskId = context.getIndexOfThisSubtask();
    if (currentEvent == 0) {
        t1 = ((double) Instant.now().toEpochMilli()) / 1000;
        this.random = new Random(taskId);
    }
while (currentEvent < numEvents) {
    long t = Instant.now().toEpochMillis();
    double currentTime = ((double)t) / 1000;
    if (currentEvent % 1000 == 0) {
        double expectedTime = currentEvent * increment;
        double actualTime = currentTime - t1;
        System.out.println("Expected time: "+ expectedTime +
            "Actual time: "+ actualTime);
    }

double diff = currentTime - t1;
    if (diff < currentEvent * increment) {
        long numMilliseconds = (long)(currentEvent *
            increment - diff) * 1000;
        Thread.sleep(numMilliseconds);
    }

int source = random.nextInt(numIps);
int dest = random.nextInt(numIps);
String sourceIp = "node" + source;
String destIp = "node" + dest;
Netflow netflow = new Netflow(currentEvent, label, time,
    parseDate, dateTimeString,
    protocol, protocolCode,
    sourceIp, destIp, sourcePort,
    destPort, moreFragments,
    countFragments,
durationSeconds, srcPayloadBytes,
  destPayloadBytes, srcTotalBytes,
  destTotalBytes, firstSeenSrcPacketCount,
  firstSeenDestPacketCount,
  recordForceOut);

  out.collectWithTimestamp(netflow, t);
  out.emitWatermark(new Watermark(t));
  currentEvent++;
  time += increment;
}
}

@Override
public void cancel()
{
  this.currentEvent = numEvents;
}
}
```java
public class BenchmarkTrianglesNetflow {

    private static class SourceKeySelector implements KeySelector<Netflow, String> {
        @Override
        public String getKey(Netflow edge) {
            return edge.sourceIp;
        }
    }

    private static class DestKeySelector implements KeySelector<Netflow, String> {
        @Override
        public String getKey(Netflow edge) {
            return edge.destIp;
        }
    }

    private static class LastEdgeKeySelector implements KeySelector<Netflow, Tuple2<String, String>> {
```

```java
@Override
public Tuple2<String, String> getKey(Netflow e1) {
    return new Tuple2<String, String>(e1.destIp, e1.sourceIp);
}

private static class Triad
{
    Netflow e1;
    Netflow e2;

    public Triad(Netflow e1, Netflow e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    public String toString() {
        String str = e1.toString() + "\n" + e2.toString();
        return str;
    }

    /**
     * Key selector that returns a tuple with the source of the first
     * edge and the destination of the second edge.
     */
    private static class TriadKeySelector
```
implements KeySelector<Triad, Tuple2<String, String>>
{
    @Override
    public Tuple2<String, String> getKey(Triad triad)
    {
        return new Tuple2<String, String>(triad.e1.sourceIp, triad.e2.destIp);
    }
}

private static class Triangle
{
    Netflow e1;
    Netflow e2;
    Netflow e3;

    public Triangle(Netflow e1, Netflow e2, Netflow e3)
    {
        this.e1 = e1;
        this.e2 = e2;
        this.e3 = e3;
    }

    public String toString()
    {
        String str = e1.toString() + " "+ e2.toString() + " "+ e3.toString();
        return str;
    }
}
private static class EdgeJoiner
    extends ProcessJoinFunction<Netflow, Netflow, Triad> {
    private double queryWindow;

    public EdgeJoiner(double queryWindow) {
        this.queryWindow = queryWindow;
    }

    @Override
    public void processElement(Netflow e1, Netflow e2,
      Context ctx, Collector<Triad> out) {
      if (e1.timeSeconds < e2.timeSeconds) {
        if (e2.timeSeconds - e1.timeSeconds <= queryWindow) {
            out.collect(new Triad(e1, e2));
        }
      }
    }

private static class TriadJoiner
    extends ProcessJoinFunction<Triad, Netflow, Triangle> {
    private double queryWindow;

    public TriadJoiner(double queryWindow) {
        this.queryWindow = queryWindow;
    }
@Override

public void processElement(Triad triad, Netflow e3,
        Context ctx, Collector<Triangle> out)
{
    if (triad.e2.timeSeconds < e3.timeSeconds) {
        if (e3.timeSeconds - triad.e1.timeSeconds <= queryWindow) {
            out.collect(new Triangle(triad.e1, triad.e2, e3));
        }
    }
}

public static void main(String[] args) throws Exception {

    Options options = new Options();
    Option numNetflowsOption = new Option("nn", "numNetflows", true,
            "Number of netflows to create per source.");
    Option numIpsOption = new Option("nip", "numIps", true,
            "Number of ips in the pool.");
    Option rateOption = new Option("r", "rate", true,
            "The rate that netflows are generated.");
    Option numSourcesOption = new Option("ns", "numSources", true,
            "The number of netflow sources.");
    Option queryWindowOption = new Option("qw", "queryWindow", true,
            "The length of the query in seconds.");
    Option outputFileOption = new Option("out", "outputFile", true,
            "Where the output should go.");
    Option outputNetflowOption = new Option("net", "outputNetflow", true,
"Where the netflows should go (optional)."

Option outputTriadOption = new Option("tria d", " outputTriads", true,
  "Where the triads should go (optional).")

numNetflowsOption.setRequired(true);
numIpsOption.setRequired(true);
rateOption.setRequired(true);
numSourcesOption.setRequired(true);
queryWindowOption.setRequired(true);
outputFileOption.setRequired(true);

options.addOption(numNetflowsOption);
options.addOption(numIpsOption);
options.addOption(rateOption);
options.addOption(numSourcesOption);
options.addOption(queryWindowOption);
options.addOption(outputFileOption);
options.addOption(outputNetflowOption);
options.addOption(outputTriadOption);

CommandLineParser parser = new DefaultParser();
HelpFormatter formatter = new HelpFormatter();
CommandLine cmd = null;

try {
  cmd = parser.parse(options, args);
} catch (ParseException e) {
  System.out.println(e.getMessage());
  formatter.printHelp("utility-name", options);
int numEvents = Integer.parseInt(cmd.getOptionValue("numNetflows"));
int numIps = Integer.parseInt(cmd.getOptionValue("numIps"));
double rate = Double.parseDouble(cmd.getOptionValue("rate"));
int numSources = Integer.parseInt(cmd.getOptionValue("numSources"));
double queryWindow = Double.parseDouble(
    cmd.getOptionValue("queryWindow"));
String outputFile = cmd.getOptionValue("outputFile");
String outputNetflowFile = cmd.getOptionValue("outputNetflow");
String outputTriadFile = cmd.getOptionValue("outputTriads");

// get the execution environment
final StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(numSources);
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

NetflowSource netflowSource = new NetflowSource(numEvents, numIps,
    rate);
DataStreamSource<Netflow> netflows = env.addSource(netflowSource);

if (outputNetflowFile != null) {
    netflows.writeAsText(outputNetflowFile,
        FileSystem.WriteMode.OVERWRITE);
}

DataStream<Triad> triads = netflows
    .keyBy(new DestKeySelector());
.intervalJoin(netflows.keyBy(new SourceKeySelector()))
.between(Time.milliseconds(0),
    Time.milliseconds((long) queryWindow * 1000))
.process(new EdgeJoiner(queryWindow));

if (outputTriadFile != null) {
    triads.writeAsText(outputTriadFile,
        FileSystem.WriteMode.OVERWRITE);
}

DataStream<Triangle> triangles = triads
    .keyBy(new TriadKeySelector())
    .intervalJoin(netflows.keyBy(new LastEdgeKeySelector()))
    .between(Time.milliseconds(0),
        Time.milliseconds((long) queryWindow * 1000))
    .process(new TriadJoiner(queryWindow));

    triangles.writeAsText(outputFile, FileSystem.WriteMode.OVERWRITE);

    env.execute();
}