StreamWeb: Real-Time Web Monitoring with Stream Computing

Toyotaro Suzumura\textsuperscript{1,2} and Tomoaki Oki\textsuperscript{2}
IBM Research – Tokyo\textsuperscript{1}, Tokyo Institute of Technology\textsuperscript{2}
toyo@jp.ibm.com, oiki.t.aa@m.titech.ac.jp

ABSTRACT

A new trend involves Web services such as Twitter beginning to publish streaming Web APIs that enable partners and end users to retrieve streaming data. By combining such push-based Web services and existing pull-based Web services, it is now possible for us to understand the current status or trends of the world in a more real-time way, such as real-time tracking of infectious disease, real-time crime prediction, or real-time marketing, and so various innovative business services are possible. For a system architecture to implement such services, the services are normally built from the scratch, and the performance and scalability depend upon the engineers’ skills. In this paper we propose a real-time Web monitoring system called “StreamWeb” on top of a stream computing system called System S developed by IBM Research. The StreamWeb system allows developers to easily describe their analytical algorithms for a variety of kinds of Web streaming data without worrying about the performance and scalability, and provides real-time and scalable Web monitoring for massive amounts of data. As an experimental proof-of-concept application, we built an application that monitors a list of keywords in the Twitter streaming data, and that displays any messages including the specified keywords onto a map of the physical location (from Google) where the message was posted. Our system can handle nearly 30 thousand Twitter messages per second on a system with 8 computing nodes. This prototype application confirms that we can build real-time Web monitoring systems while satisfying the needs for high software productivity and for system scalability.

1. INTRODUCTION

In the Web 2.0 era, many Web services have started to publish their Web service APIs using SOAP or REST or both. This openness has enabled new innovative Web services by mashing up Web services. Recently a major trend involves Web services with streaming APIs that allows end users or partners to retrieve real-time streaming data published by those Web services. Examples include the Twitter Streaming API [20], the Facebook Open Stream API, and augmented reality services such as the recently launched Sekai Camera. The authors believe that this trend will greatly affect the world and lead to innovative services. For example, a new service can obtain live voices from regular citizens as a real-time stream.

An especially interesting new type of streaming API is used for a microblogging service. According to Wikipedia, microblogging is defined as a form of multimedia blogging that allows users to send brief text updates or micromedia such as photos or audio clips and publish them, either to be viewed by anyone or by a restricted group that can be chosen by the user. Microblogging services have also emerged as an important source of real-time news updates in recent crisis situations. The short updates allow users to post news items quickly in real-time, reaching the audience in seconds. In particular, Twitter has been the most successful microblogging service to date and has experienced explosive growth, even attracting some celebrities. Twitter attracted a total of 44.5 million unique visitors worldwide in June, 2009 according to [19]. Twitter publishes all of the posted messages into what they call a “public timeline” if the users are making public “tweets” (the basic message units of Twitter).

To build microblogging services, there are several problems that must be addressed. One of the problems is the volume of data. The data volume from streaming APIs can become very large, can increase rapidly, and there is a strong need to process the data in real time. For example, with Twitter Streaming, even at the lowest volume called “spritzer”, the volume of data each day is approximately 1 GB. This access level uses some filtering, but other access methods such as “firehose” are already running around 100 GB per day.

In this paper, we address this problem of large amounts of data and make the following contributions. First we built a real-time Web monitoring system called StreamWeb on top of IBM’s stream computing system called System S [1][2][3]. Second, we show how we can easily develop a highly scalable and high-performance application using StreamWeb, by building a keyword monitoring and mapping application (using Google Maps) from the large amounts of streaming Twitter messages.

The remainder of the paper is organized as follows. First, the motivation and problem statement of this research are presented in Section 2. Section 3 describes the system requirements and design of our real-time Web monitoring system StreamWeb. In Section 4 we describe the implementation of the StreamWeb system based on the design described in Section 3. In Section 5, we present performance evaluation on how well our system scales for streaming Twitter messages and how flexibly we can optimize performance. Discussion appears in Section 6, and related work is reviewed in Section 7, followed by our conclusions and future directions in Section 8.

2. Motivation

In this section, we consider three motivations for our research.

Batch: Existing efforts for Web mining basically use batch execution approaches that store all of the data and then process it with distributed execution engines and programming models such as MapReduce [11]. However, in situations where the data volume is large and response time or latency is critical, the store-and-process model is not an acceptable solution.

Streaming API: However, as more streaming APIs are becoming widely available, it is increasingly possible to analyze the real-time status of the world. Some existing systems such as Google Trends already provide nearly real-time information, but the update frequency is still around several minutes.

Middleware: For programming models and software infrastructure, there is little middleware that addresses the problems. The
Data Stream Management System (DSMS) [1][3][5][6][7] has emerged as a stream processing engine that handles real-time data from various sensor sources. There has been no attempt to apply DSMS to the Web domain, and thus our effort described in this paper is a first attempt.


To realize our goals, we built a real-time Web monitoring system called StreamWeb, based on the ideas of stream computing. Our prototype handles the large amounts of streaming data available from the Web and analyzes that data in real time.

The fundamental idea of stream computing is that incoming data should be processed “on the fly” within memory without being stored into relatively slow secondary storage such as a hard disk. This paradigm is opposed to the “store-and-process” model that is used in batch-type computations on large amounts of data stored on disks or in a distributed file system. Representatives of current popular programming models and implementations in this field are MapReduce.

We can distinguish stream computing from traditional batch-type systems based on on-the-fly versus store-and-process. However, there are other system requirements when building a scalable and distributed stream processing platform, including programming models, runtime architectures, optimal job placement and scheduling, and load shedding. In our research, our StreamWeb system is built on top of IBM’s System S [1][2][3], which is currently under development by IBM Research. Since System S and its programming model called SPADE [1] has flexible extension points, we can use it to build a highly robust system specifically for real-time Web monitoring while leveraging the existing functionalities such as its distributed capability. Before going into the details of the system architecture, we first introduce one of our target application scenarios in the next section.

3.1 Application Scenarios

These days many microblogging services are available such as Twitter and the Facebook Open Stream API. The fundamental nature of microblogging is that individual tweets are not really meaningful when compared with the content described in a blog. For example, people may tweet only single words, such as “hi” or “tired”. However, when assembling the collections of tweets, we can understand the macro-trends of the larger world, such as how an influenza outbreak is expanding.

One of the most successful microblogging services, Twitter, uses a public timeline model that can show nearly all of the tweets. For Twitter, the number of users was 44.5 million as of Sept. 2009. By leveraging the real-time and crowd characteristics, we can understand the current status in a near-real-time fashion. In addition, the microblogging services such as Twitter have profile sections that allow users to describe their locations. Some people even publish precise location information including latitude and longitude when they use applications with devices such as the iPhone. By leveraging that location information with the tweets, we can understand what is going on in various locations and areas. Here are some examples:

Rea-Time Pandemics Prediction: By tracking words such as “flu” and “cold”, we can understand where the flu and other infectious diseases are appearing in real time, and use this information to respond effectively to epidemics [15][16][17][18].

Real-Time Marketing: By tracking the trends of new commercial products such as TV’s or the iPhone, a company can monitor its reputation in real time.

Real-Time Economic Status: Economic indicators play critical roles when central banks and government decide on financial and economic policies. Reports on crucial indicators are typically lag one month or more after significant events. By improving the timeliness and accuracy of this public data in the era of the information age better policy decisions would be possible. One possible data source could be microblogging services. Some tweets reflect the real-time economic changes. For example, if many people tweet about losing jobs, it may provide input about the unemployment rate, or an increase in tweets about expensive tomatoes may reflect changes in the consumer price index. There are many other possible application scenarios such as real-time weather monitoring, earthquake reporting, or detecting terrorist incidents.

3.2 Requirements for StreamWeb

In this section we list our requirements for the StreamWeb system.

Target Services: There are two basic types of Web services. One is a Web service that allows other software components to retrieve data as a pushed stream. A representative service of this type is the Twitter Streaming API [20]. The other is a traditional Web service that allows only pull-type retrieval such as the Twitter Search Service. Although push-based services are becoming more common, pull-based services are still dominant. Our system should handle both of these types of service. Also, the data formats returned by such services are either JSON or XML format, so our system should handle both of these data formats.

Performance: The system should scale as the volume of data becomes large. For Twitter, the number of messages varies depending on the time of day and the situation, such as when a special event is taking place. The system should handle major surges dynamically. We also assume there are multiple streaming services, including microblogging services such as Twitter, and pull-based Web services.

Programmability and Software Productivity: The system needs to provide an easy-to-use programming model that allows end users to write new analytical algorithms without worrying about the performance and scalability issues. The leading approach for handling large volumes of data is MapReduce [11]. However, we focus on the real-timeliness and response times as well as the
throughput, and MapReduce and Hadoop are unsatisfactory. The developer should be able to focus only on the analytical algorithms, perhaps selecting from built-in functions and connecting them for basic operations.

**Extensibility:** The system needs to add and monitor additional data sources as new data sources become available. Many Web services have started to publish their streaming APIs, but we cannot assume that all Web services will publish their APIs anytime soon. Therefore, we should be able to deal with the existing push-based Web services.

### 3.3 System Components for StreamWeb

The overall system architecture is illustrated in Figure 1. The system is divided into 3 basic parts, the external components, a real-time analytics tier, and a visualization tier. The analytics and visualization tiers are implemented within our system, and the external components are provided by the external Web services. The details of each layer are described in the next sections.

#### 3.3.1 External Components

As shown in the Figure 1, we assume two types of external applications, external Web services Types I and II. Type I includes streaming web services such as microblogging streaming data (such as Twitter, the Flickr Streaming API, or the Sekai Camera API). The Type II external Web services are more static Web services that provide analytical algorithms for static locations such as the Flickr Web service and other social network services such as Facebook.

#### 3.3.2 Real-time Analytics Tier

The real-time analytics tier has two components, the streaming data collector and the real-time analytics engine. A Streaming Data Collector (SDC) receives streaming data from some Web service such as Twitter. With the streaming API, SDC opens a connection with a streaming server and holds the connection until it is closed, perhaps by an error. An SDC handles incoming data sequentially. Meanwhile, the real-time analytics engine executes the registered streaming analytical algorithms from the SDC.

#### 3.3.3 Visualization Tier

The visualization tier is responsible for visualizing the incoming data provided by the Real-Time Analytics Tier. Typical implementations of this visualizing tier are as a Web application using HTML and JavaScript or as a standalone visualization application. If it is in the form of a Web application accessible from a Web browser, it handles a series of Ajax requests generated with JavaScript in the Web browser and immediately returns a list of posted messages.

### 4. Implementation of StreamWeb

The implementation of the system is built on top of the IBM Stream Computing System called System S [1]. The Streaming Data Collector and Real-Time Analytics Engine are implemented using SPADE [1] and run on top of System S. When the volume of incoming data becomes large, the streaming data collector will scale for the increased traffic thanks to System S’s design. Before going into the details of the system, we give an overview of System S and its programming language, SPADE.

#### 4.1 Overview of System S and SPADE

System S [1][2][3] is large-scale, distributed data stream processing middleware under development at IBM Research. It processes structured and unstructured data streams and can be scaled to a large numbers of compute nodes. The System S runtime can execute a large number of long-running jobs (queries) in the form of data-flow graphs described in its special stream-application language called SPADE (Stream Processing Application Declarative Engine) [1]. SPADE is a stream-centric and operator-based language for stream processing applications for System S, and also supports all of the basic stream-relational operators with rich windowing semantics. Here are the main built-in operators mentioned in this paper:

- **Functor:** adds attributes, removes attributes, filters tuples, maps output attributes to a function of input attributes
- **Aggregate:** window-based aggregates, with groupings
- **Join:** window-based binary stream join
- **Sort:** window-based approximate sorting
- **Barrier:** synchronizes multiple streams
- **Split:** splits the incoming tuples for different operators
- **Source:** ingests data from outside sources such as network sockets
- **Sink:** publishes data to outside destinations such as network sockets, databases, or file systems

SPADE also allows users to create customized operations with analytic code or legacy code written in C/C++ or Java. Such an operator is a UDOP (User-Defined Operator), and it has a critical role in providing flexibility for System S. Developers can use built-in operators and UDOPs to build data-flow graphs. The Appendix has an example SPADE program for StreamWeb. After a developer writes a SPADE program, the SPADE compiler creates executables, shell scripts, other configurations, and then assembles the executable files. The compiler optimizes the code with statically available information such as the current status of the CPU utilization or other profiled information. System S also has a runtime optimization system, SODA[3]. For additional details on these techniques, please refer to [1][2][3].

SPADE uses code generation to fuse operators with PEs. The PE code generator produces code that (1) fetches tuples from the PE input buffers and relays them to the operators within, (2) receives tuples from operators within and inserts them into the PE output buffers, and (3) for all of the intra-PE connections between the operators, it fuses the outputs of operators with the downstream inputs using function calls. In other words, when going from a SPADE program to the actual deployable distributed program, the logical streams may be implemented as simple function calls (for fused operators) for pointer exchanges (across PEs in the same computational node) to network communications (for PEs sitting on different computational nodes). This code generation approach is extremely powerful because simple recompilation can go from a fully fused application to a fully distributed version, adapting to different ratios of processing to I/O provided by different computational architectures (such as blade centers versus Blue Gene).
However, we built our own parser to translate fixed interval components for Type II and Type III continuous data from external components. We need extra type of access via REST or SOAP, and that already support streaming API. RAE is dependent on the application, so we will describe depending on the incoming traffic.

4.2 Real-Time Analytics Tier Implementation

The Real-Time Analytics Tier is basically composed of two components, the Streaming Data Collector (SDC) and the Real-Time Analytics Engine (RAE). Both of the components are implemented using SPADE and run on top of System S. As the incoming data volume increases, both components can scale depending on the incoming traffic, thanks to System S’s design. RAE is dependent on the application, so we will describe a for sample implementation in the next section. As mentioned in the design section, we need to support various Web services, Type I that already support streaming API, and Type II that provides data access via REST or SOAP, and also Type III existing websites without special APIs. Even though we need to support these three type of Web services, SDC is only responsible for handling continuous data from external components. We need extra translation components for Type II and Type III sources. Type II is a pull-based service, so we implemented a streaming translator that repeatedly retrieve data from a pull-based Web service at a fixed interval. For Type II, we have not yet implemented a special translator, but we are using an existing “Web scraping” solution [21]. SDC handles two data formats, XML or JSON. For XML, we built our own parser, the Streaming XML Parser dedicated to incoming XML messages. Existing XML parsers such as Xerces assume that the parser retrieves the XML data from a file. However, for streaming data, we should avoid storing the incoming XML data in any file. For this reason we implemented our own parsing component for the incoming XML data. This component finds a target tags, and continues reading the content until the corresponding end tag appears. For the JSON format, we also created a dedicated SPADE operator for parsing JSON-format data using the C++ JSON Parser.

4.3 Samples of Real-Time Analytics Engine

Since the real-time analytics engine is dependent on the application scenario, such as those described in Section 3.1, this section describes initial tests of the prototype system. The scenario is that the system obtains streaming messages from the Twitter service, monitors for specified keywords, and then maps the messages including those keywords onto Google Maps in real-time. To realize this service, we used two Web services available in Twitter. One is a traditional Web service called the Twitter Search Service that returns a list of messages with the keywords specified in the HTTP request. The other is the Twitter Streaming API[20]. Here are the steps that identify the location data with latitudes and longitudes for these two services. The Twitter Streaming API is basically simpler since the returned messages have more of the information required for this scenario.

Twitter Search Service

1. Retrieve a list of posted messages from Twitter: The system sends an HTTP request with the target keywords for monitoring and receives a list of messages. This is pull-based, so it repeatedly sends request to the search service.
2. Retrieve the user profiles via the Twitter API (since the returned messages include only the user names).
3. Each returned user profile includes a user location. Some users with iPhones also publish their exact locations, so Step 2 can be skipped. For Japanese users, the system uses the morphological analysis tool Mecab to get the name of the city from the location data in the profile data.
4. The internal dictionary identifies the latitude and longitude for the user location. (From the statistics provided by [19], 44% of the users have no location data, resulting in the location (0,0).

Twitter Streaming API

1. The system obtains all of the posted messages from the Twitter Streaming API and filters them against the specified keywords. These results include the user profile data.
2. Go directly to Steps 3 and 4 of the Twitter Search Service.
Implementation

Next we describe how we implemented this service in SPADE and System S. An overview of a dataflow graph is shown in Figure 1 and a code snippet from the SPADE program is shown in the Appendix. The dataflow graph uses 3 built-in operators, the Funct, Split, and Barrier operators, and 5 UDOPs (User-defined operators). The UDOP operators are implemented in C++ and the lines of code for each operator appear in Table 1. They are relatively small and developers focus only on the input data, the core processing algorithm, and the output data. Parallelism and scalability need not be considered.

1. **SourceConnector** connects to a Twitter Streaming Server via the Twitter Streaming API and then continues to fetch the posted messages in the JSON format. This data format is also available in XML, but we used JSON for better performance.

2. **PostParser** parses the incoming JSON messages with a JSON Parser implemented in C++.

3. **PostFilter** obtains posted messages from PostParser, and transmits only the messages with the specified keywords

4. **GeometryCoder** returns a list of messages with geographic information for the latitude and longitude.

5. (Only for Twitter Search Service) **PostRetrieval** has two parameters, a word to be monitored and an identifier. The identifier is used for the internal stream. This operator contacts the Twitter Search Service with the specified word to be monitored in the parameter section. PostRetrieval returns tuples using a schema called PostSchema. This schema has 6 elements: an identifier (String), an update time (Long), an id (String), the content (String), a user name (String), and a jobid (String). All of the data corresponds to a Twitter post.

The notable thing is that by writing SPADE code with the required UDOP operators and built-in operators, the SPADE compiler creates a set of programs, a Makefile, and shell scripts that can be run on a cluster of nodes. The generated SPADE program is shown in the Appendix. The detailed syntax and programming model of SPADE [1] is not described in this paper, but here is the basic syntax for defining an input stream, operator, and output stream. This code snippet defines a user-defined operator called “UserCustomizedOperator” that receives tuples from an incoming stream called “InputStreamA” defined earlier, and outputs a stream called “OutputStreamA” with two kinds of data, the id (Integer) and text (String). The user-defined operator, UserCustomizedOperator, is implemented using C++ in other sources files.

```
stream OutputStreamA (id: Integer, text: String) 
  := Udob(InputStreamA)["UserCustomizedOperator "]() 
  ->node(sourcepool, 1), partition["part1"]
```

Also an operator can have additional meta-information for node and partition. The node parameter specifies which nodes the operator is assigned to, and the partition parameter allows the fusion of multiple operators. When fusing multiple operators, they are executed as one process and communicate with each other using lightweight function calls. By using the same keyword for a partition, the operations are fused into a single process. Simply by changing the node and partition operators, we can obtain optimal performance for a particular computation environment.

<table>
<thead>
<tr>
<th>Name of UDOP</th>
<th>Lines of C++ code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceConnector</td>
<td>192</td>
</tr>
<tr>
<td>PostParser</td>
<td>82</td>
</tr>
<tr>
<td>PostFilter</td>
<td>131</td>
</tr>
<tr>
<td>GeometryCoder</td>
<td>81</td>
</tr>
</tbody>
</table>

Table 1. Lines of Code for User-Defined Operators

Visualization Tier

We implement a prototype visualization tier as a Web application. Our Web application was implemented in Python using the Python Twisted Library. For a production system, this could be replaced with a higher performance and more scalable Web server such as Apache or Lighttpd. The Web server can handle a list of posted messages from the SDC every 20 Seconds. To detect duplicated messages, the Web server has a data management module that maintains a list of the posted messages for a period of time. A JavaScript running in a Web browser asynchronously connects to the Web server to retrieve the data. The data consists of pairs of a posted message and its location (longitude and latitude), and the JavaScript displays the data using Google Maps via the Google Maps API. Since the Ajax library retrieves many data pairs each time, the JavaScript displays the data with some random delay from 0 to 1 second. This random delay avoids the overlap of multiple messages if from the same location.

4.4 User Interface

The user interface as shown in Figure 3 and Figure 4 displays twitter messages at the locations where they were posted. We used the Google Maps API and Ajax components in JavaScript to asynchronously connect to the Web server (implemented using the Python Twisted library) and retrieve the posted messages and the location data. The user interface allows users to add or delete words to be monitored. When the user adds a word, it is added to the list of watched words. When the user changes the mode to watch mode, then the SDC component starts to monitor the listed words. Figure 3 shows a world map that displays the Twitter messages that include the word “iPhone” in real-time, and Figure 4 displays Japanese posts on a map of Japan.

5. Performance Evaluation

In this section we conducted a performance evaluation and assessed the scalability of the system, and also demonstrated how to use SPADE to optimize the performance when it falls below expectations. The application structure is shown in Figure 11 and the SPADE program is described in the Appendix. For the performance experiment, we used only the push-based Twitter Streaming API rather than the Twitter Search Service.

5.1 Experimental Environment

We used a set of 8 compute nodes (HP ProLiant ML 115), each of which is a 2.7-GHz AMD Athlon 1640B uniprocessor with 1 GB of RAM running Cent OS 5.2 (kernel 2.6.18-92.1.12.el5). The hostname of each node starts with “streams” followed by a number from 01 to 08. The streams08 node is used for the Web server that displays the posted Twitter messages, and the other 7
5.2 Experimental Data
For our experiments, we collected posted messages at the spritzer level of the Twitter Streaming API from 0:00 to 0:59 on 2009/10/18, once per minute. The Twitter Streaming API has three levels based on decreasing volumes of data, firehose, gardenhose, and spritzer. The firehose level returns all of the public messages, but this is only available to the certain users. The spritzer level allows anybody to access streaming messages without any permission, but the returned messages are only a sample of the posted messages. Therefore in our experiment, we artificially increased the amount of data based on the fetched messages by sending the same messages repeatedly. Our original data from spritzer was 41,237 posted messages (50,432 KB) for the 1 hour of our original test. Also, to avoid a bottleneck in sending the posted messages from a file or network socket, we stored the messages in memory. The emulation for reusing the messages via the Twitter Streaming API was handled by the first UDOP operator, the “SourceConnector” in Figure 11.

5.3 Experiments and Results
We performed two experiments, Experiment I and Experiment II as follows. Both of the experiments used the same dataflow graph in Figure 11 and the SPADE program of the Appendix. There were slight modifications in the assignment of nodes to operators during the performance optimization. In the SPADE program depicted in Figure 11, the first operator called “SourceConnector” parses JSON data using the C++ JSON Parsing Library, and this parsing is the heaviest processing, so we distributed this work among the nodes from streams01 to streams06.

**Experiment I** tested the system scalability while monitoring various numbers of keywords.

**Experiment II** tested the performance characteristics as the number of monitored keywords increased.

**Experimental I: Throughput as the number of nodes increased**
The Experiment I performance results are shown in Figure 5 and Figure 6. The X axis shows the number of nodes, and the Y axis shows the throughput in messages per second. In the first test, the number of keywords to be monitored was set to 1. As shown in the graph, the system could process more than 25,000 messages per second with 3 nodes. The throughput was increased up to 3 nodes, but additional nodes had no benefit. With 6 nodes, the throughput was actually decreased to around 21,500 messages per second. This saturation effect is due to a bottleneck in the Split operation (Figure 11) that distributes the data to the multiple nodes, appearing because this operation needs to send relatively large messages to each node. The computations in this test are relatively small when compared to the communication costs since this test monitors only one word. This situation could be improved by using a network with higher throughput, such as Infiniband or
a 10-Gb network. In the next test, we increased the computational load by monitoring more words, up to 1,024 words. The graph shown in Figure 6 shows the linear scalability against the number of nodes, showing that the Split operation is no longer a bottleneck. In this experiment we used 6 compute nodes from streams01 to stream06, and 1 node, streams07 for the Functor Split, and Socket operations.

**Experiment II: Throughput while monitoring various numbers of keywords**

In this experiment we studied the performance characteristics while varying the number of keywords being monitored. The graph in Figure 7 shows the throughput with 6 compute nodes while varying the number of keywords from 1 to 4,096. The maximum throughput is around 20,000 messages per second. After 512 keywords, the throughput is saturated. The experimental settings are the same as in Experiment I, with 6 nodes (streams01 to streams06) used as compute nodes and 1 node (streams07) used for the Socket, Functor, and Split operations. The CPU usage in this experiment is shown in Figure 8. To study the CPU utilization, we look at two nodes, streams01, which does computations like JSON parsing, and stream07, which does the Socket, Functor, and Split operations. In this experiment, the CPU utilization at streams01 is around 80% up to 512 keywords, but beyond 512 keywords, it increases to 100%. At the same time, streams07, which includes the Split operator for distributing the data to multiple nodes, consumes 100% of its CPU cycles up to 512 keywords. This is because the compute nodes are not saturated and streams07 is busy trying to send a sufficient number of requests to each of them. This shows that the node streams07 becomes the bottleneck with its 3 operations.

To address the bottleneck in streams07, we divided the 3 operations between 2 different nodes, streams06 and streams07. The streams06 node takes care of the Socket operator to handle the incoming Twitter messages, and the streams07 node handles the Functor and Split operators. Now there are only 5 compute nodes. SPADE and System S are designed so that these kinds of changes in node assignment can be done easily. After this modification, the experimental results are as shown in Figure 9. The graph shows this node assignment outperforms the previous assignment since it processes more than 25,000 messages per second up to 128 words. The throughput comparison in Figure 12 shows this difference.

**6. Discussion**

In this section we discuss three topics that should be improved or addressed: handling bursty situations, visualization, and limitations on the data sources.

**Handling Bursty Situations:** In our performance experiments described in Section 5, we assume that the messages arrive at the average rate, but in reality there are bursts of activity when unusual events occur, such as an earthquake, a major election, or Halloween. Although our system can increase its throughput by adding more nodes and the configuration is fairly easy, with limited computation resources such as the 8 nodes in our experiments, we need to consider other techniques for handling such bursts. In the DSMS field, load shedding is a hot topic, and allows for shedding extra jobs or processing only some fraction of the incoming messages. Many load shedding techniques have been proposed from random sampling to semantic filtering, but we want to support multiple approaches, since the appropriate filters depend on the keywords being monitored. For example, if a government agency is trying to prevent crimes or terrorist attacks, then random sampling is not suitable.

**Visualization and its Performance:** The current JavaScript-based user interface does not scale with the number of messages. There are several ways to address this problem.

**More precise location information:** The location data is grouped for a series tweets. Also, we could leverage the existing profiling Web services such as IDDY, or other SNS services such as Facebook. However, this raises private concerns.

**Limitations on Data Sources:** We only used the Twitter service in our experiments. In this service, current statistics indicate that only 5% of the users create 80% of the tweets. If the analysis depends on a single data source, then there is a strong bias, so we should evaluate other microblogging services or trusted websites.

**7. Related Work**

**Trend Analysis:** In terms of trend analysis, several commercial vendors publish trend analysis results, such as Google Trends and Google Flu Trend. Our application is similar, but the existing efforts are dedicated to particular domains and seem to be built in an ad hoc fashion. Twitter also provides its own trend analysis such as Buzztter that displays the top-k topics. Our system aims at generalizing such services as a system and programming model with real-time capability and scalability. In addition, our system seeks to provide more timely information while leveraging various information sources.

**Mapping Various Kinds of Information onto Maps:** Great efforts have been made and many services exist that map various kinds of information to map systems such as Google Maps, For example, Crandall et al. [11] proposed a system for mapping Flickr photos with geo-tag onto Google Maps. However, as far as the authors know, no existing services focus on mixing the location data with microblogging data.

**Stream Computing Middleware:** Stream Computing is often described as a Data Stream Management System (DSMS) to be compared with a traditional database system (DBMS, Database Management System). In addition to System S, our stream computing platform, other systems include Borcalis and Telegraph CQ [1][3][5][6][7]. As noted, System S uses a declarative language that lets developers add their user-defined functions, but some other systems are more limited since they use extended SQL-like syntax, such as CQL (Continuous Query Language)[8]. As far as the authors know, there have been no attempts to apply stream computing to the Web mining area, although there was related work in ITS (Intelligent Transport System) [7].

**8. Conclusions and Future Directions**

In this paper we described a real-time Web monitoring system built on top of System S. This is a first attempt to apply and generalize stream computing for the Web domain. Our StreamWeb system tracks vast amount of streaming Twitter messages and displays them according to their originating locations on Google Maps. This paper only describes one instance of streaming data sources, but our defined architecture is general and flexible, so we could build other innovative applications to
find new knowledge in real-time. For future work, we will use other data sources other than Twitter and build more applications, and explore other performance optimizations such as the load shedding mentioned in the Discussion section.

REFERENCES
[12] Why we twitter: understanding Microblogging usage and communities, International Conference on Knowledge Discovery and Data Mining (KDD 2007)
[19] Twitter Statistic Data in Comscore

Appendix: SPADE Program

```
(Application) /* application name */
streamtwitter

(Nodepools) /** define a set of compute nodes */
nodepool sourcepool[] := ["stream07", "stream06", "stream05"]
nodepool main[] := ["stream01", "stream02", "stream03", "stream04", "stream05", "stream06", "stream07"]

(Program) /** application body */
/** the number of compute nodes given by the command argument */
#define PROCESSOR_SIZE 1
/** data schema, StreamPostSchema is defined in the other file*/
$vstream AnnotatedPostStream(schemaFor(StreamPostSchema), identifier(String),jobid(String))
stream RawPostStream(text(String))
  := Udp0(["SourceConnector"][])
  -> node(sourcepool,0)
stream IndexedRawPostStream(index:Long,text:String)
  := Functor([RawPostStream][index]] index := seqNum()1)
  -> node(sourcepool,1), partition["divider"]

/*** distribute the post messages to multiple nodes with the Split operator. SPADE has a for-loop syntax to avoid creating redundant streams ***/
for_begin @i 0 to PROCESSOR_SIZE
  stream SomeRawPostStream(index:Long,text:String)
  := vstream
  -> node(sourcepool,1)

/*** bundle is a syntax for bundling multiple streams ***/
bundle LocationedPostStreams := 0
for_begin @i 0 to PROCESSOR_SIZE
  stream PostStream@i(schemaFor(StreamPostSchema))
    := Udp0([SomeRawPostStreams][i]"PostParser")
  stream FilteredPostStream@i(schemaFor(AnnotatedPostSchema))
    := Udp0([PostStream@i]"PostFilter")
    -> node(main.mod(@i),PROCESSOR_SIZE), partition["ali@i"]
  stream LocationStream@i(location:String)
    := Functor([FilteredPostStream@i][index := user_location])
    -> node(main.mod(@i),PROCESSOR_SIZE), partition["ali@i"]
  stream GeometryStream@i(latitude:Double,longitude:Double)
    := Udp0([LocationStream@i][latitude := longitude,longitude := latitude])
    -> node(main.mod(@i),PROCESSOR_SIZE), partition["ali@i"]
  stream LocationedPostStream@i(schemaFor(AnnotatedPostSchema),latitude:Double,longitude:Double)
    := Barrier([FilteredPostStream@i][latitude := longitude,longitude := latitude])
    -> node(main.mod(@i),PROCESSOR_SIZE)

LocationedPostStreams := LocationedPostStream@i
for_end

/*** this stream is exported to other SPADE programes ***/
#export properties [category:twitter]
stream ConvertedPostStream(schemaFor(ResultDataSchema))
  := Functor(LocationedPostStreams[])
  {"uplatitude := latitude,downlatitude := latitude, leftlongitude := longitude, rightlongitude := longitude, value := content,appendix := user_name + ":" + user_location, starttime := time, endtime := time, type := "twitterpost"}
  -> node(sourcepool,0)
```