StreamWeb: Real-Time Web Monitoring with Stream Computing

Toyotaro Suzumura\textsuperscript{1,2} and Tomoaki Oiki\textsuperscript{2}
\textsuperscript{1} IBM Research – Tokyo
\textsuperscript{2} Tokyo Institute of Technology
We propose a real-time web monitoring system called “StreamWeb” that handles the large amounts of streaming social data available from the Web and analyzes that data in real time on top of a stream computing platform.
Outline

- Background and Motivation
- Stream Computing and System S
- Real-Time Web Monitoring System
- System Evaluation
- Concluding Remarks and Future Work
Background – Growth of Streaming Social Data

- 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*, the *Facebook Open Stream API*, and so forth. This trend will greatly affect the world and lead to innovative services.
Motivation: Real-Time Web Monitoring

- We need real-time web monitoring system that handles the large amounts of streaming data available from the Web and analyzes that data in real time for such examples as real-time pandemic prediction, marketing, economic indicator (GDP and Consumer Price Index, ...),
Problem Statement
Prior Arts is built as a monolithic architecture and special-purpose application

- **Social Web Monitoring**
  - Google Flu Trends ([http://www.google.org/flutrends/](http://www.google.org/flutrends/))
    - Ginsberg (Google), Detecting influenza epidemics using search engine query data, Nature 2008
  - Earthquake Real-time Monitoring from Twitter [Sakaki, WWW’10]
    → Built as a monolithic architecture and special-purpose application

- **MapReduce Programming Model** [Dean, OSDI’04]
  - We focus on the real-timeliness and response times as well as the throughput, and MapReduce and Hadoop are unsatisfactory.

[Sakaki, WWW2010] Earthquake shakes Twitter users: real-time event detection by social sensors
[Dean, OSDI2004] MapReduce: Simplified Data Processing on Large Clusters
Outline

- Background and Motivation
- Stream Computing and System S
- Real-Time Web Monitoring System
- System Evaluation
- Concluding Remarks and Future Work
Stream Computing and System S

- **System S**: a Stream Computing Middleware developed by IBM Research (productized as “InfoSphere Streams” now)
- A middleware platform that processes massive amount of data on the memory rather than storing data on the disk like traditional model

**Traditional Computing**

- Fact finding with data-at-rest

**Stream Computing**

- Insights from data in motion
System S Programming Model

Source Adapters

Operator Repository

Sink Adapters

Application Programming (SPADE)

Platform optimized compilation
SPADE : Advantages of Stream Processing as Parallelization Model

- A stream-centric programming language dedicated for data stream processing
- Streams as first class entity
  - Explicit task and data parallelism
  - Intuitive way to exploit multi-core and multi-nodes
- Operator and data source profiling for better resource management
- Reuse of operators across stored and live data
- Support for User-Defined Operator (UDOP) implemented in either C/C++ or Java
A SPADE Example

[Application]
SourceSink trace

[Nodepool]
Nodepool np := (“host1”, “host2”, “host3”)

[Program]
// virtual schema declaration
vstream Sensor (id : id_t, location : Double, light : Float, temperature : Float, timestamp : timestamp_t)

// a source stream is generated by a Source operator – in this case tuples come from an input file
stream SenSource ( schemaof(Sensor) )
    := Source( ) [ “file:///SenSource.dat” ] {}
    -> node(np, 0)

// this intermediate stream is produced by an Aggregate operator, using the SenSource stream as input
stream SenAggregator ( schemaof(Sensor) )
    := Aggregate( SenSource <count(100),count(1)> ) [ id . location ]
    { Any(id), Any(location), Max(light), Min(temperature), Avg(timestamp) } 
    -> node(np, 1)

// this intermediate stream is produced by a functor operator
stream SenFunctor ( id: Integer, location: Double, message: String )
    := Functor( SenAggregator ) [ log(temperature,2.0)>6.0 ]
    { id, location, “Node ”+toString(id)+ “ at location ”+toString(location) } 
    -> node(np, 2)

// result management is done by a sink operator – in this case produced tuples are sent to a socket
Null := Sink( SenFunctor ) [ “udp://192.168.0.144:5500/” ] {}
    -> node(np, 0)
InfoSphere Streams Runtime

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation.
Outline

- Background and Motivation
- Stream Computing and System S
- Real-Time Web Monitoring System
- System Evaluation
- Concluding Remarks and Future Work
StreamWeb: Real-Time Web Monitoring with Stream Computing

- We propose real-time web monitoring system called “StreamWeb” that handles the large amounts of streaming social data available from the Web and analyzes that data in real time on top of a stream computing platform.
System Requirements for StreamWeb

- **Generality and Extensibility**
  - The system needs to add and monitor additional data sources as new data sources become available.
  - The system needs to support for various analytics algorithms and two Web Services: Pushed-based Web Service (e.g. Twitter Streaming API) and Pull-based Web Services (e.g. Twitter Search Service).

- **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.

- **Performance and Scalability**
  - The system should scale as the volume of data becomes large.
  - The system should handle major surges dynamically since the number of messages varies depending on the time of day and the situation, such as when a special event is taking place.
# Overall StreamWeb Architecture

## Visualization Tier
- **Web Browser**
  - **Web Application** (e.g. Visualization via Map)
  - **Web Application** (e.g. only display Statistics)

## Real-time Analytics Tier
- **Real-time Analytics Engine**
  - **Streaming Data Collector**
  - **Streaming Translator**
  - **Web Scraping**

## External Web Services
- **External Web Services (I)**
  - **Streaming Web Service (w/ Streaming API)**
  - **Web Service (w/ RESTAPI or RSS)**
  - **Web Sites (w/o API)**

## External Web Services (II)
- **Map** (e.g. Google Map, Yahoo Map)
- **SNS** (e.g. Facebook)
- **Photo Sharing** (e.g. Flickr)

---

**IEEE ICWS 2011 (International Conference on Web Services)**
Real-Time Analytics Tier

- This tier is comprised of **Streaming Data Collector (SDC)** and **Real-Time Analytics Engine (RAE)**.

- **Implementation on top of SPADE**
  - 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.

- **Support for various Web services,**
  - Type I that already support streaming API
  - Type II that provides data access via REST or SOAP
  - Type III existing websites without special APIs.

SDC is only responsible for handling continuous data from external components. We need extra translation components for Type II and Type III sources.
Sample Real-Time Analytics Engine

- 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.
Streaming Data Collector

- For the XML format:
  - 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 the JSON format:
  - We also created a dedicated SPADE operator for parsing JSON-format data using the C++-based JSON Parser.
How to realize sample services with Twitter Search Service / Streaming API?

**Twitter Streaming API**
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.

**Twitter Search Service**

1. Retrieve a list of posted messages from Twitter Search Service:
   - 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 *Mecabu* 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.

**Twitter Streaming API**: http://dev.twitter.com/pages/streaming_api_methods
**Twitter Search Service**: http://search.twitter.com/api/
Implementation (1/2)

SPADE Program

Twitter Streaming API
- Source Connector
- Post Parser

Twitter Search Service
- Post Retrieval

Functor
- Split
- Post Filter
- Geometry Coder

Post Parser
- Filter
- Geometry Coder

Visualization Tier
- Barrier

Node assignment on physical compute nodes
- streams01
- streams02
- streams03
- streams04
- streams05
- streams06
- streams07
Implementation (2/2)

- **SourceConnector** connects to a Twitter Streaming Server via the Twitter Streaming API, and then continues to fetch the posted messages in the JSON format.

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

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

- **GeometryCoder** returns a list of messages with geographic information for the latitude and longitude.
Visualization Tier Example

This user interface 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.
このネコがWindowsの新発売日を知らなかった
http://japan.internet.com/linuxtoday/20091021/5.html
Performance Evaluation

- **Experiment I** tested the *system scalability* while monitoring various numbers of keywords.
- **Experiment II** tested the *performance optimization* with System S’s node re-allocation.
- 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.
Experimental Environment

**Total Nodes:** 7 nodes (streams01 – streams07)

**Spec. for Each Node:** 2.7-GHz AMD Athlon 1640B uniprocessor, 1GB memory, CentOS 5.2 (Linux Kernel 2.6.18.-92)

**Network:** 1Gbps Network

**Software:** InfoSphere Streams 1.1

**Data Access Method:** spritzer level of the Twitter Streaming API

**Data:** 41,237 posted messages (50,432 KB) for the 1 hour from 0:00 to 0:59 on 2009/10/18

**Experimental Setting:**
The emulation for reusing the messages via the Twitter Streaming API was handled by the first UDOP operator, the “SourceConnector” that avoid a bottleneck in sending the posted messages from a file or network socket by storing the messages in memory.
Experimental I: Throughput as the number of nodes increased

- The system could process **more than 25,000 messages per second with 3 nodes.** The throughput was saturated around 3 nodes due to a bottleneck in the Split operation that distributes the data to the multiple nodes.

- By increasing the number of monitoring keywords from 1 to 1024 that leads to more computational load, the throughput becomes better in linear way up to 6 nodes.
Experiment II: Optimizing Throughput by Changing Node Allocation

- 6 nodes (streams01 to streams06) used as compute nodes and 1 node (streams07) used for the Socket, Functor, and Split operations.

- The node *streams07* becomes the bottleneck since it is busy trying to send a sufficient number of requests to all the compute nodes.

- By re-allocating the Functor and Split operators to streams07, and allocating the Socket operator to streams06, the throughput became better from 20000 to around 250000 messages per second.
Concluding Remarks and Future Work

Concluding Remarks

– We proposed a real-time Web monitoring system called “StreamWeb” built on top of a stream computing platform, System S.

– Our first application of the StreamWeb system tracks vast amount of streaming Twitter messages and displays them according to their originating locations on Google Maps.

– We only showed 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.

Future Work

– We will use other data sources other than Twitter and build more applications and complex analytics, and explore other performance optimizations.
Questions

Thank You