RSS Feed Complex Event Detection

Matt Fuller

RSS (really simple syndication) feeds have become largely popular over the last few years. RSS is a web format for publishing frequently updated content such as news articles or blog updates [10]. In this paper we present an RSS feed complex event detection (CED) system and simple SQL language we call RSSQL used for declaring our complex events.

1. Introduction

Traditionally users subscribe to RSS feeds of interest using an RSS feed reader. The RSS feed reader periodically polls the subscribed feeds for updates or items to be displayed to the user. Many RSS feeds usually pertain to a single news source or blog. Others may aggregate various feeds usually on some topic and produce a single RSS feed. Middleware publish-subscribe systems allow users to subscribe a list of keywords of topics. The middleware may crawl RSS feeds and filters relevant items matching the users' subscription [1]. This architecture decouples the publisher from the subscriber. This type of system can be described as stateless [2].

We present an RSS Complex Event Detection (CED) system that allows for more stateful [2] queries on RSS feeds. Our RSS Query Language (RSSQL) is a declarative language used to define events of interest on RSS feeds. We provide a thin client user interface deployed as a web application for interacting with the system to register queries and retrieve the results of these queries. In section 2 we explain how the event detection works and terminology that will be used throughout the paper. In Section 3 we describe our RSSQL language. Section 4 will explain the system architecture of the RSS CED system. In Section 5 we discuss some results which our system produced. Section 6 will discuss some related work. Section 7 provides a conclusion and potential future work for the system. Finally we include an Appendix that will provide a "How To" of building and running our system from the source code.

2. Complex Event Detection & Terminology

A user can register events in our system through our client interface. The events are declared by our RSSQL language which we describe in the following section. Events are defined as activities of interest in our system [3]. For example, a user may only want to be notified when comments are left on a blog entry of a certain topic. A user can define such a query using our RSSQL language. Once the query is registered into our system it will be continuously executed so long as the feeds of interest continue to publish new items.

We refer to individual RSS items as primitive events or atomic occurrences of interest in the system. Complex events are built on top of primitives or other complex events [3]. In our RSS system, primitives that match our filters and operator (which we will describe in the next section) will be referred to as complex events. Our system runs what we will call *RSS Sensors* which periodically poll the RSS feeds for new items or primitives. The primitives that are published on feeds are recognized by the sensors which then push these new items for query execution. The results of the complex events are appended to a file of previously occurred events. At any time a user can retrieve the current results. Each registered query has unique MD5 hash which is used to retrieve the results for a particular query. The commands are explained in 4.5.

3. RSSQL Language

Our RSSQL language allows for more expressive queries of RSS content other than keyword queries. Currently we support a tiny subset of SQL, namely the *SELECT* and *FROM* clauses. Additionally, we support a subset of the *MATCH_RECOGNIZE* clause as described in the paper for pattern matching in sequences of rows [4]. The content within the *MATCH_RECOGNIZE* is where the complex events of interest are expressed. The paper describes two options for outputting matches: All Matches or One row per Match. For our RSS application it makes the most sense to output all matches. One match per row would be a summary row of all the matches and not very useful for our application. The paper describes using regular expressions syntax to define the patterns for matching. Instead of regular expressions to define our complex events, we borrowed event operators from active database research [3]:

And Operator: $and(e_1, e_2,...,e_n ;w)$ outputs a complex event when every e_i occurs within the window w in any order.
Sequence Operator: $seq(e_1, e_2,...,e_n ;w)$ outputs a complex event when every e_i occurs within the window w in sequential order where the end time of the previous event does not overlap with the start time of the net event.
Or Operator: $or(e_1,e_2,...,e_n)$ outputs a complex event whenever any e_i occurs. There is no window frame for this reason.

The following is an example query one may register to be notified of the event where a comment is left on a blog posting of a certain topic.

```
SELECT title, description, link, pubDate
FROM all_news, all_blogs
MATCH_RECOGNIZE(
PATTERN and(A, B; 5)
DEFINE A AS keyword_filter({*}, {"Baseball",
"Scandal"}) AND feed_name = "all_ news"
       B AS keyword_filter({title}, {A.link}) AND
feed_name = "all_blogs"
);
```

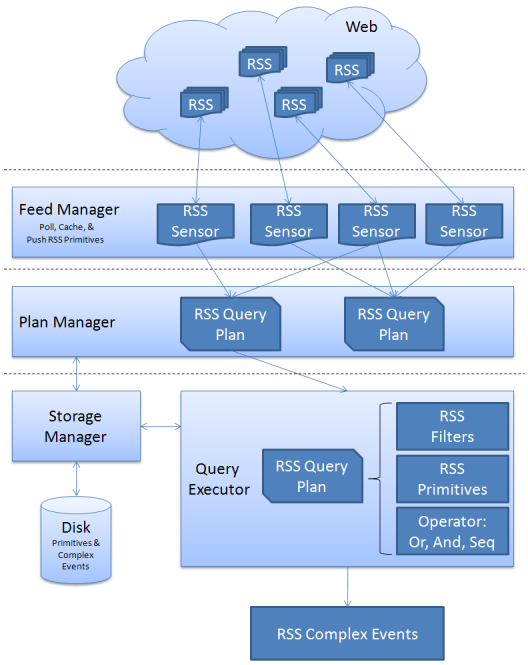
When the query is registered, it registers itself to listen for new items on our all_news and all_blogs views. A view is a logical mapping of one to many different feeds. For example, when new items arrive on all_news we check to see if the items match our keyword filter. After we have found matches on our A variable, we look for matches on our B variable from all_blogs. In this case we assume a comment on an item matched in A will contain the title on an item matched in A. If both A and B occur within the 5 day window, we output a match.

Users can also define their own views on feeds to be used in their query. To create a view or add a new url to the system the user can type:

```
MONITOR feedurl AS viewname;
```

If a Feed Sensor already exists for that feed url, we create a logical mapping from the view to our Feed Sensor. If a Feed Sensor does not exist for the feed url, we create a new Feed Sensor as well as a logical mapping from the view to our new Feed Sensor.

4. System Architecture



4.1 Overview

Our system consists of 4 major components. The first component is the *Feed Manager* which deploys the *RSS Feed Sensors* to poll RSS feeds for updates. The second component is the *Plan Manager* which handles registering *RSS Query Plans* to be continuously executed by the query executor. The third component is the query executor component which handles the execution of a RSS query plan and interacting with the S*torage Manager* to store complex and primitive events. The final component is the C*lient* interface deployed as a thin client web application or a command line tool.

4.2 Feed Manager

The feed manager is responsible for spawning new RSS Sensors and storing their relationships to various views. Each RSS Sensor runs on a separate thread and autonomously monitors a feed url. It maintains a hash set of all previously seen items. At each polling interval, the RSS Sensor polls the RSS feed for new items. For each item, if the item's hash does not exist in the hash set, the Sensor caches the item and pushes it to any interested RSS query plans. Every so often the in-memory cache should be purged to disk by the storage manager.

4.3.1 Plan Manager

The plan manger's role is to store the RSS query plan which is continuously executed. After the query is parsed, the RSS query planner registers the plan with the interested RSS Sensors. When any sensor receives new items, it will push the items to the plan. The plan will call the query executor to run the query. Each plan is given a unique id by the plan manager. When the plan is registered with the plan manager, it has to option to call the query executor on the cached RSS Feed items.

4.3.2 Query Parser

The query parser takes in the RSSQL as text and outputs a syntax tree for further semantic analysis and plan generation. The lexical analyzer and parser are generated using the Java versions of Flex and Bison which are basically equivalent to the more well known Lex and Yacc. The Flex generated scanner separates the text into a stream of tokens defined by regular expressions and passes the tokens to the parser as needed. A grammar for RSSQL similar to BNF is used by Bison to generate a Look Ahead Left to Right (LALR) parser.

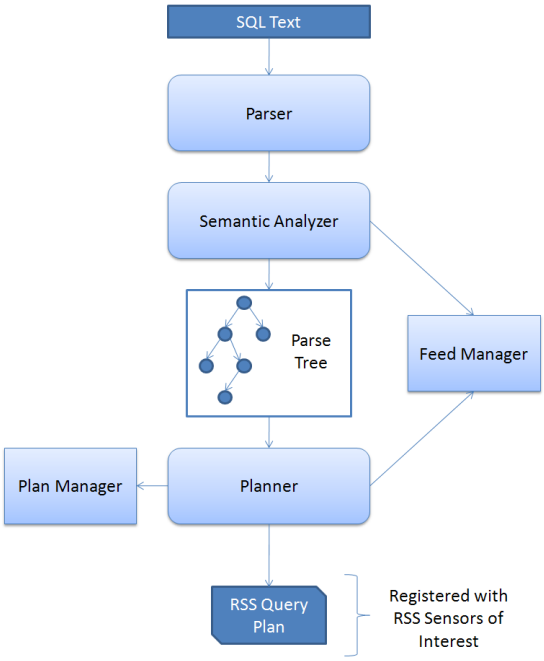
This is a "Yes/No" parser with appropriate error messages. From this we want to build the parse tree. Within each production of the grammar we can construct the necessary nodes to build a parse tree. Later this tree is traversed for semantic analysis and RSS query plan generation. For semantic analysis, as we traverse the

tree, we check that the views specified in the *FROM* clause are registered in the system. Additionally, any column references in the *SELECT* or *DEFINES* clauses are validated against the RSS Sensor schemas to make sure that are valid as well. The RSS Query planner is discussed next.

4.3.3 Query Planner

The Query planner takes as input the valid parse tree and outputs an RSS Query Plan. The role of the RSS Query plan is to listen to the RSS Sensors of interest. When the new items are pushed to the plan it calls on the *Query Executor* to check for a complex event match.

As with the semantic analysis, we traverse the syntax tree to create the plan. We store the projections from the *SELECT* list and views defined in the *FROM* list. Filter chains are created for each *DEFINE AS variable* statement. Filter chains are defined as the predicates that specify a match for a variable.



4.4.1 Query Executor

The query executor's function is to detect events of interest from the primitive events. It takes as input the RSS query plan and the new items from the RSS Sensor. For each primitive item, we check to see what variables the primitive matches. As described in the previous section, a match on the variable is determined by the filter chain. For each variable match, we check with the state of the RSS query plan operator (*and, seq, or*) for a complex event match. If there is a match, we output a complex event. If there is no match, we buffer it in the operator for subsequent events to match. Complex events are sent to the *Query Result Dispatcher* for each plan. Each plan has an associated output stream. Currently, the items are written to a file. But, theoretically, the output stream could be to the output stream of a socket.

4.4.2 Filters

Currently we support two filters in our system. The *keyword filter* takes two vector arguments. The first vector consists on the columns of the RSS Feed we will search on. The second vector contains the keywords. The filter takes a single primitive and outputs either true or false if the primitive contains the specified keywords. The keyword filter has two options for matching: all items or some items.

The price filter takes a product name and price predicate. We experimented with two price sources. The first source we send an HTTP post request to the search page of *http://dealsea.pricegrabber.com* [5].We attempt the parse the html for the price of the item returned in the search result. The html was inconsistent and many times we fail to retrieve a price.

A better source for a price of a product is using Amazon.com API Web Service. Using the Simple Access Object Protocol (SOAP) we send a product name and Amazon returns results relating to that product name [6]. There is always a price if the product exists. We return true or false for the filter if it passes the predicate.

4.5 Client

The client creates a socket connection to our server running with our RSS CED system. For each client connection, the server spawns a worker thread for further client-server communication. The client code writes the RSSQL bytes to the input stream of the socket. We came up with our own simple protocol for sending messages. Using known RPC mechanisms incur two much overhead for our purposes. We have three different request types sent in the form *RequestType: message*

RegisterRequest is used for registering RSSQL queries and Monitor requests.
SimpleCommandRequest is used to request information from the RSS CED system
views: Requests the list of views in the system
feeds: Requests the list of RSS Sensors in the system by URL
[feed name]: Requests the items current and cached for that feed
[view name]: Requests the corresponding Feed URLs associated with the View
ResultsRequest is sent when the user enters events followed by the RSSQL hash. This requests a list of the match complex events.

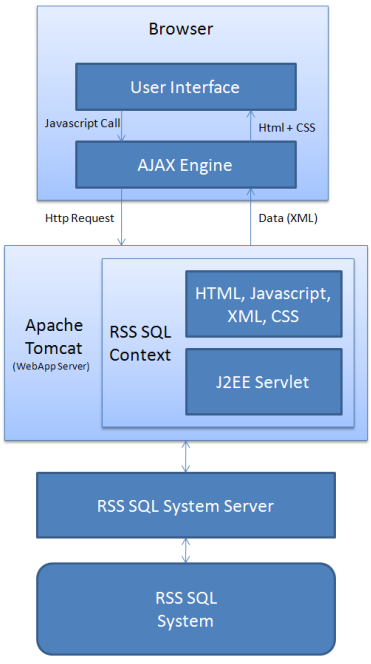
4.5.1 Client Command Line

The command line client is run within a terminal. Users enter their queries at the command line. The input is read

via *stdin*. The server response is written to *stdout*. It runs continuously until the user enters or types "quit".

4.5.2 Thin Client Ajax Interface.

For a better user experience, the client is also deployed as a thin client web application. The web application provides a text area for entering queries and commands as well as a read only text area for displaying the results. The typical HTTP model is synchronous. That is when a browser sends a request to the server it must wait until the server sends a response as HTML in which the page then reloads. If the user is entering many commands, a page reload after each command will become irritable. Instead we use the Ajax model to send asynchronous requests to the server. The user can continuously enter queries and not have to wait for the responses. The dynamic changes on the html are done by manipulating the Document Object Model (DOM) representing the html. The XMLHttpRequest object is used for data retrieval. The logic to manipulate the html is implemented in JavaScript. Getting the Javascript correct and for all types of browsers and browser version is a daunting task. Google provides web toolkit for easily creating Ajax application [7]. The entire web application can be programmed in Java utilizing the standard java packages. Serlvet code is written to handle the request. The Google web toolkit cross-compiles the Java code into Javascript, XML, and HTML. This can then be deployed to Apache Tomcat (which we do).



5. Results

We are currently monitoring approximately 230 RSS feeds obtained from popular news sites such as Reuters, CNN, ABC, Yahoo News, and the Boston Globe. Some of the queries we ran were inspired by the example queries of the Cayuga system as we will explain in the related work section [2]. One query (as described in our RSSQL Language section) was to generate a complex event when someone left a comment about a blog article about a Major League Baseball scandal. We were able to obtain many results for that query as there was a professional baseball player who blogged about the incident. Many people thus commented on his blog yielding results for our query.

For another query we were trying to take a step toward identifying which types of stories new sources were more interested in different presidential candidates. We registered a query to search for articles mentioning any of the 2008 Presidential runners from the two major political parties. The idea would be to have a count aggregate count each variable in the operator to determine which news source favored which political party. By trying to run some of these queries we realized that our language made it very difficult to do so. Languages like Cayuga's can be more expressive but with greater complexity. We will describe more in the next section.

6. Related Work

This system was designed to work with the event detection as described in Combining Proactive and Retroactive Processing for Distributed Complex Event Detection [3]. Although we never integrated, the primitives written to disk by our storage manager could also be sent to the query executor of another system.

Cayuga is a system developed in the Cornell Database Systems group. It is a publish/subscribe system for stateful event monitoring. Some of our queries for our system were inspired by some of the example queries of Cayuga. The Cayuga language appears to be very expressive and can express more than our RSSQL language can. Although our RSSQL was a step towards using simple SQL, it was limited with what it could do.

The subset of the language we used for RSSQL came from the Pattern Matching over Sequences of rows paper [4]. This extension of SQL is very useful for stock data. It performs matching by defining regular expressions and only does the matching on a single table or a view. As our events do not generally appear sequentially in rows of data, we had to loosen that restriction for our application.

Cobra is a content based filtering and aggregator of blogs and RSS feeds. It presents a three tiered system of crawlers to pull the data of many RSS feeds and blogs,

filters to match the published feeds with user subscriptions, and reflectors to provide a personal RSS for the subscribed users [1]. Our system has similarities with the tiers of Cobra. In addition to the filters, we execute our active database operators for complex events.

Yahoo Pipes was another inspiration of our system. Yahoo Pipes is described as a composition tool to aggregate, manipulate, and mashup content from around the web. The output of a Yahoo pipe is a chain simple operators that are combined together to produce a desired output. In a Unix system, simple commands are piped together to produce a desired results. Hence the name, Pipes. Yahoo Pipes is geared for bloggers types and provides a very intuitive user interface to layout the pipe chain [8]. It is a stateless system and executes fully each time one goes to visit the feed. Our system differs in that we look for events over the RSS feeds as well as only execute on the newly arrived items storing the previous matches.

Our system is a continuous query system in the aspect that for each newly published item we execute the queries that are interested in this data to produce results. However unlike a continuous query system such as TinyDB for wireless sensor networks or Borealis for stream processing application, our system remains silent until our complex events occur [3].

7. Future Work & Conclusions

Future work includes working on our RSSQL language so that we can express queries similar to those of the Cayuga system while also keeping the language simple. We should also include more system functions such as the keyword filter. Cayuga had some seemingly intelligent functions such as an *IsPositive* function which returned true or false if the content was of positive sentiment.

To scale, we should run our RSS Sensors on different node. We could create or use a dissemination system such as XPORT [9]. Another problem we encountered was when we polled certain RSS feeds at too frequent of a rate; we got shut off from the source to poll the RSS feed. We should also look into how often to poll certain feeds. Some are updated more than others.

We presented a RSS complex event detection system. Users can register queries to define complex events of interest. Our system continuously polls RSS feeds and pushes the new items to our query executor to match events. We also provide a client command line tool and web interface to interact with our RSS CED system.

8. References

[1] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Welsh. Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds.

[2] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White. Towards Expressive Publish/Subscribe Systems. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT 2006)*, Munich, Germany , March 2006.

[3] M. Akdere. Combining Proactive and Retroactive Processing for Distributed Complex Event Detection.

[4] Anonymous. Pattern Matching in Sequences of Rows. March 2007.

[5] Dealsea. http://dealsea.pricegrabber.com.

[6] Amazon Associate Web Service. http://www.amazon.com.

[7] Google Web Toolkit. http://code.google.com/webtoolkit.

[8] Yahoo Pipes. http://pipes.yahoo.com/pipes.

[9] O. Papaemmanouil, Y. Ahmad, U. Çetintemel, J. Jannotti, Y. Yildirim. Extensible Optimization in Overlay Dissemination Trees. In *SIGMOD*, June 2006.

[10] RSS. http://en.wikipedia.org/wiki/RSS_(file_format).

Appendix

The contents are package in two tars. rssced.tar is the server side of the system. rsscedweb.tar is the Ajax web application. It also contains the war file to deploy as a web app. A list of client commands is listed in the Ajax client as well as in 4.5.

rssced.tar

src (the source code)
events (the directory when primitive events are stored)
lib (any jars needed by the system)
properties (when the system parameters are stored)
rss (when the primitives are stored)
sources (where our system feeds urls are stored)
tools (lex, yacc tools)
build.xml (the ant build file)

Edit any system parameters in the *server.properties* file in the properties directory. Such parameters are the polling intervals for the RSS Sensors, the server port number, and directories of where to store events.

To run the server type: “ant” or “ant server”
To run the command line client type: “ant client”
To compile type: “ant compile” (typing ant server and ant client also compiles)
To generate a new parser and lexer type: “any parser-compile”

rsscedweb.tar

The only directory to be concerned with is
src (the source code)

To generate a web directory type: “ant” or “ant webify”
To generate a war file type: “ant war”

Then just deploy to Apache Tomcat or some other web application server.