Adaptive Scalable Pipelines for Political Event Data Generation

Andrew Halterman
Department of Political Science
Massachusetts Institute of Technology
ahalt@mit.edu

Jill Irvine, Manar Landis
Women's and Gender Studies/International and Area Studies
University of Oklahoma
{jill.irvine, Manar.K.Landis-1}@ou.edu

Phanindra Jalla, Yan Liang, Christan Grant
School of Computer Science
University of Oklahoma
{yliang, Phanindra.Jalla, cgrant}@ou.edu

Mohiuddin Solaimani
Department of Computer Science
The University of Texas at Dallas
mxs121731@ou.edu

***Abstract*—Political event data has been increasingly important for researchers to study and predict global events. Until recently the majority of political events were hand-coded from text, limiting the timeliness and coverage of event data sets. Recent systems have successfully employed big data systems for extracting events from text. These automated event systems have been limited by either the slow performance or high infrastructure demands. In this work, we present a new approach to big data systems that allow for faster extractions when compared to existing systems. We describe a modular system, Biryani, that adaptively extracts events from batches of documents. We use distributed containers to process streams of incoming documents. The number of containers processing documents can be increased or reduced depending on the number of available resources. The optimal configuration for event extraction is learned, and the system adapts to maximize the throughput of coded documents. We show the adaptability through experiments running on laptops and multiple commodity machines. We use this system to extract a new political event data set from several terabytes of text data.**

I. Introduction

The availability of large corpora of online news documents has made it possible for computer and social scientists to study human political behavior at scales that were previously impossible. One of the primary bottlenecks in deriving meaning from text documents is the resource demands of the natural language processing and information extraction that need to be performed. Current document processing pipelines suffer from a range of limitations when it comes to processing hundreds of millions of news articles for social science applications, either in terms of poor performance or ease of use. Document processing pipelines written by social scientists are easily installed and customized, but tend to be single-threaded and slow for corpora larger than a few hundred thousand documents. Frameworks for distributed pipelines that are fully flexible (Apache Nifi, Kubernetes, Spark, [1]) require sophisticated infrastructure and significant technical expertise both to set up and to customize for document processing tasks, which creates a high hurdle for applied use.

In this paper, we present a simple, adaptive pipeline for extracting political events from a large number of news documents. The performance benefits are derived from a low-overhead distributed architecture that adaptively tweaks processing parameters to optimize throughput on the processing machines using a Kalman filter. Its modular, containerized architecture allows the pipeline to be executed on both laptops and large clusters with minimal dependencies. This system greatly outperforms the existing state of the art methods for political event extraction from text [1], with substantively simpler installation and maintenance, which is a high priority for social scientists.

We begin by discussing social scientists' use of event data extracted from text (§ II). We then provide a system overview and describe the components of the system (§ III. We next present the experiment design (§ IV) and the evaluation of the experiment (§ V). We conclude by considering the impact of the experiment results and give future as we conclude directions (§ VII).

II. Background

News articles published on the web give social scientists in academia and government the ability to measure and understand political events around the world [2], [3]. In order to extract meaning from this text, social scientists have developed standardized event ontologies that define political actors and political events in a consistent form [4]. These ontologies represent political events in a framework of "who did what to whom", with standardized codes for different actors (e.g., "government," "military," or "rebel") and the type of action (e.g. "make statement," "protest for leadership change," or "fight with small arms"). To transform raw text into event data, social scientists use specialized dictionary-based coders to extract events from text according to the ontologies.

Early systems used simple dictionary matching techniques to extract events from text [5]. Modern approaches use syntactic information provided by statistical parsers such as Stanford's CoreNLP [6] to better match noun and verb phrases with

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

dictionaries [7]. Processing stories through CoreNLP and the event data extraction pipeline[1] can be very slow, both in "batch" mode and when run on stories streaming in from scrapers downloading news stories from web RSS feeds.[2] In the existing pipeline, as stories are scraped from the web, they are stored in a MongoDB and run through a pipeline to perform CoreNLP annotations, and then run through a pipeline for event extraction, geoparsing, and other annotation tasks [8].

The document processing pipelines in widest use in the open-source event data research community are simple single-threaded pipelines, which struggle to rapidly process large corpora in the hundreds of millions of documents. One approach to improving the processing pipeline is described in Solaimani et. al. [1]. This processes uses a Spark architecture for distributed CoreNLP processing and event extraction. [3] We describe a system that can be executed either as a single machine or a multi-machine distributed system, which achieves better performance than the current state of the art with much lower setup costs and without the need for a Spark cluster and allows researchers to use the same tool for widely varying task sizes.

III. System Overview

The architecture is built on Docker containers, which allow dependencies and software to be self-contained [12]. The processing takes place within distributed containerized CoreNLP instances [6]. Each container pulls documents to process from a central RabbitMQ queue, processes them, and stores them in a SQLite database. This architecture allows the pipeline to run locally on one machine or it may be distributed across a heterogeneous cluster. The completed system is launched using a container orchestration system such as Docker Compose or Kubernetes.[4]

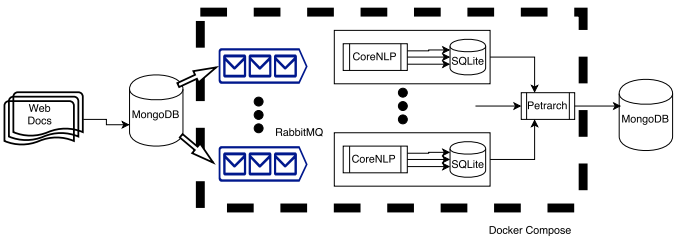

Fig. 1. The Biryani system architecture.

Containerizing and distributing the processing step has several advantages in addition to the ability to distribute across machines: the per-container threads, batch size, and memory can be adapted to increase processing speed, containers can be automatically restarted on failure, and the architecture becomes generalizable to processes that are not natively multi-threaded (e.g. Python).

Figure 1 displays the components of the system. Raw data is uploaded from webpages into a MongoDB. This is a common interface for storing data but it is not essential to the Biryani system. Data is transferred from Mongo or an API to a RabbitMQ. This queue is a persistent storage and the entry point to Biryani. Containers are spun up as consumers of the data queue. The containers run the Stanford CoreNLP process with multiple threads. The data in each container is written to a local SQLite database. Finally, an existing specialized event extraction system [7] can run over the CoreNLP processed text.

A. Optimization

The goal of the pipeline is to improve the throughput in bytes/sec. The primary variables we can manipulate in this architecture to maximize throughput are the number of threads per container and the batch size (in documents) of the task.

To automatically optimize parameters we employ a Kalman filter [13]. The time taken to process documents can be viewed as a random variable, even with the batch size and thread number fixed, the process time will differ each time the process runs. Randomized noise include factors such as how many other processes are running on the system while it are runs the pipeline and exceptions that some documents could cause while being parsed. Therefore, we select the discrete Kalman filter algorithm, which is a Bayesian time series inference model. The Kalman filter uses a series of measurements with an estimated distribution of statistical noise and other inaccuracies, and infers a number or parameters.

The Kalman filter algorithm works in a two-step process. In the prediction step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement is observed, though with with noise and measurement error, these estimates are updated using a weighted average, with more weight being given to estimates with higher certainty. We describe this process in Algorithm 1.

The external parameters to the Kalman Filter are P, the a posteriori error estimate, R, the estimate of measurement variance, and a noise parameter Q. Because we ran the data pipeline on a machine that was not running many other processes, our noise parameter will be relatively small. We therefore set our Q, representing the stability of the system, to be small. The value Z is the size of each batch before the parameters are adjusted. The current a posteriori estimation of the state is represented as X and K is a relative weight given to the measurements and current state estimate. Based on the system configuration, the user may choose different hyper parameters for the Kalman filter to tune the performance better.

IV. Experiments

In this section, we describe the experiments we ran to demonstrate the performance of Biryani. We develop the set

[1]https://github.com/openeventdata/phoenix_pipeline

[2]e.g., https://github.com/johnb30/atlas

[3]This approach and ours both take existing, specialized tools and reformulate them as distributed, scalable tools. An alternative approach would be to use an existing general distributed architecture like Kubernetes or Apache Nifi and modify it to perform the document processing task or to use general tools like [9]–[11].

[4]https://github.com/kubernetes/kubernetes

Algorithm 1 Pseudocode for Kalman 0.75 filter based optimization

$P = 1$
$Q = 10^{-1}$
$K = 0.0$
$R = 0.1^{0.75}$
while batch **in** stream **do**
  $P' = P + Q$
  Z = **len**(batch)
  $K = \frac{P+Q}{P+Q+R}$
  $X' = X + K * (Z - X)$
  $P = (1 - K) * P'$
end while

of experiments to examine the effectiveness of the system compared to an earlier event extraction pipeline [1] and the efficacy of the system optimizations. The data in this system was stored on an Intel® Xeon® CPU X5687 @ 3.60GHz with 16 total cores and 96 GBs of RAM. All processing in these experiments were performed on an Intel® Core™ i7-6950X CPU @ 3.00GHz CPU with 20 total cores and 126 GBs of RAM. The processing machine contains two 6 TB disks and has stated transfer speeds of up to 6 GB/s. For all the experiments we used only 8 cores of the machine and we limit the memory where specified.

We performed all experiments using the English Gigaword corpus [14]. This corpus contains a collection of news wire text data in English that has been acquired over several years by the Linguistic Data Consortium. It contains 4 million documents (12 GB uncompressed) from The New York Times Newswire Service, Agence France Press English Service, Associated Press s Worldstream English Service, and The Xinhua News Agency English Service. This data set mirrors web collection tasks performed by social scientists but is large enough to allow us to test scalability. The data set is preprocessed and added to a MongoDB database offline. Using Gigaword allows us to compare our results with the current state of the art on the same set of documents [1].

Our architecture already has several advantages over the current state-of-the-art. Our system runs without the need for a "Big Data" cluster, which most social scientists do no have access to. Instead, Biryani can be quickly and easily deployed on any modern hardware. The distributed processing approach also allows us to dynamically add and subtract machines to the pool of workers. To further increase our processing speed, we can dynamically change the batch size and threads for each container.

The first experiment aims to discover the optimal batch and thread size for the pipeline, given a particular machine configuration. We performed experiments varying the batch sizes and threads, using a randomized subset of documents of size $1K$and $25K$.

This experiment also logs the total contribution taken by each component in the pipeline, including the Stanford CoreNLP annotators. We note that skew is frequently a cause of performance problems for batch processing systems [15]. A skew in document sizes can leave some batches waiting on a few large documents to complete processing. Additionally, the order of incoming documents can significantly impact the performance of the pipeline. To show the best possible arrangement of document sizes we show the performance time of the pipeline over streams of documents that are randomized and then sorted by size. The sorted document will empirically creating a performance lower bound, because document skew will not occur in batches that are pre-determined to have similar document sizes.

In the next set of experiments, we test the optimizations defined in Section III-A. We compare the various parameters of the optimized pipeline with a static set pipeline configuration.

V. Evaluation

In this section, we describe the results of the experiments described in Section IV. We first give timing numbers for different batch sizes. We compare these total times with times reported by [1]. Next, we look at the break down the proportion of time used for each component. We then examine the Kalman filter performance optimizations.

We first experiment to discover the optimal batch size. We ran each configuration three times and plotted the average time taken for each combination of the batch size and thread. Figures 2 and 3 show darker squares where the values are greater. From the graphs we can approximate that 200 is the optimal batch size for randomized documents of different sizes from Gigaword. This confirms the need to optimize batch size decisions.

Figure 3 shows no significant improvement across all thread sizes. This suggests that Biryani will not see significant gains from optimizing thread sizes.

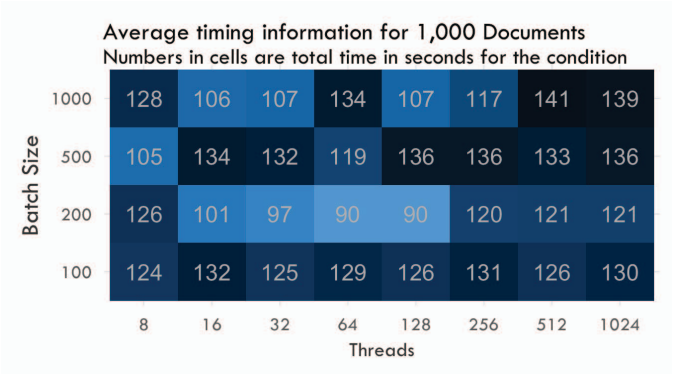

Fig. 2. Average timing information for 1000 Documents.

The time gap in Figure 2 and 3 made us curious to know which part of the pipeline is taking more time. We therefore recorded the timing information of each annotator and other factors (CoreNLP Startup Time, RabbitMQ Delay Time, JSON Object (for inserting to SQLite) with the same sets of documents, batch size and threads used above.The data shows the dependency parse (43.8%) and the creation of a JSON object (34.6%) are the most expensive processes. Lemmatization, tokenization and queue delay make up a small portion, 3.4% of the processing time. The CoreNLP start uptime

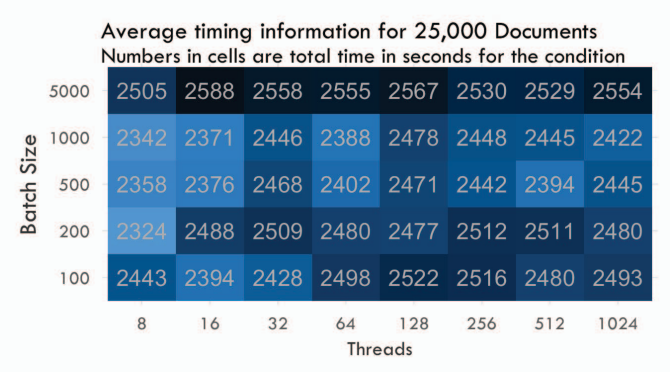

Fig. 3. Average timing information for 25,000 Documents.

represents a significant time portion (11.1%) shows the system needs to carefully select the number of new Docker containers to run.

A. Adaptive Batch Size using Kalman Filter

From the experiments performed in above we found that batch size id the most important feature to optimize. In order to effectively choose how much data should be processed per batch we use a Kalman filter, which we describe in below.

Figure I shows how many bytes of data we are processing for each type of pipeline. 'Kalman 0.25' represents a pipeline with the Kalman filter parameter $R = 0.1^{0.25}$ while other parameters P, K and Q remain unchanged. Note, that the test data subset was reshuffled before by each run of the pipeline.

Pipeline	Docs	Run 1 (s)	Run 2 (s)	Avg (s)	% Gain
Static	150K	13,555	13,619	13,587	-
Kalman 0.75	150K	13,051	13,180	13,115	3.47 %

TABLE I
PERFORMANCE GAIN OF KALMAN FILTER APPROACH OVER 150,000 DOCUMENTS.

Table I shows performance gain of Kalman filter approach over static batch pipeline when performed on 150,000 documents.

VI. PERFORMANCE OF PIPELINE ON LAPTOP CONFIGURATION

We used Google Compute Engine (GKE) to spin up a VM which is similar to a current day laptop configuration. The configuration of VM used was an Intel® Xeon® CPU @ 2.30GHz, 4 cores, and 16GB of RAM. We performed experiments on different pipeline approaches three times each on 12,484 random documents for each run and calculated the average.

Figure 4 shows the performance of Kalman filter approach and the standard error bars of each experiment. In the laptop configuration experiments, we use the static algorithm as a baseline. Kalman filter with a parameter of 0.75 sees the best speed up with a mean increase of 20.33% followed

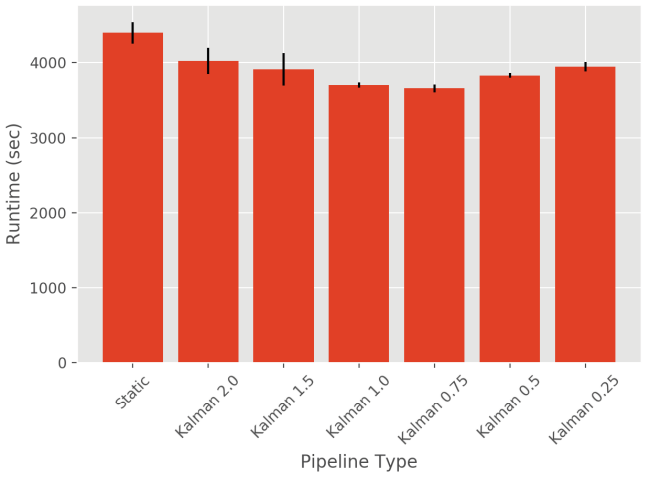

Fig. 4. Kalman Filter approaches over 12,484 sets of Random Documents with standard error shown.

by the 1.0 Pipeline with an 16.41% mean increase. Kalman filter configurations of 0.5 and 0.25 achieved lower speedups of 14.86% and 14.04%, respectively. The lowest performing configurations still had a speedups of 9.41% (Kalman 2.0) and 12.52% (Kalman 1.5). We see that Kalman filter optimization of the event data creation pipeline can also provide speedups to low resource environments.

VII. CONCLUSION

In this paper we describe a pipeline, Biryani, for extracting event data from web documents. Biryani has a container based architecture that adapts to the the available systems and available load to process text. This architecture allows researchers to use whatever resources that they have to process a large data set. We were able to easily deploy into the Azure cloud in addition to the Google Compute Engine for extra processing bandwidth. For future work we plan to optimize based on more system state variables. We will also integrate this processing system with a larger analytic pipeline that is used to query and study the extracted events. The code is available at https://github.com/oudalab/biryani. We will release a new data set under the name TERRIER (Temporally Extended Regularly Reproducible International Event Records) for political scientists to study events.

This work was funded in part by the National Science Foundation under award number SBE-SMA-1539302.

REFERENCES

[1] M. Solaimani, R. Gopalan, L. Khan, P. T. Brandt, and B. Thuraisingham, "Spark-based political event coding," in *Big Data Computing Service and Applications (BigDataService), 2016 IEEE Second International Conference on*. IEEE, 2016, pp. 14–23.

[2] S. P. O'Brien, "Crisis early warning and decision support: Contemporary approaches and thoughts on future research," *International Studies Review*, vol. 12, no. 1, pp. 87–104, 2010.

[3] W. Wang, R. Kennedy, D. Lazer, and N. Ramakrishnan, "Growing pains for global monitoring of societal events," *Science*, vol. 353, no. 6307, pp. 1502–1503, 2016.

[4] P. A. Schrodt, "CAMEO: Conflict and mediation event observations event and actor codebook," *Pennsylvania State University*, 2012.
[5] ——, "TABARI: Textual analysis by augmented replacement instructions," *Dept. of Political Science, University of Kansas,Version 0.7. 3B3*, pp. 1–137, 2009.
[6] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit." in *ACL (System Demonstrations)*, 2014, pp. 55–60.
[7] C. Norris, P. Schrodt, and J. Beieler, "PETRARCH2: Another event coding program," *The Journal of Open Source Software*, vol. 2, no. 9, jan 2017. [Online]. Available: http://dx.doi.org/10.21105/joss.00133
[8] P. A. Schrodt, J. Beieler, and M. Idris, "Three's charm?: Open event data coding with EL: DIABLO, PETRARCH, and the open event data alliance," in *ISA Annual Convention*, 2014.
[9] R. Rak, A. Rowley, W. Black, and S. Ananiadou, "Argo: an integrative, interactive, text mining-based workbench supporting curation," *Database*, vol. 2012, 2012.
[10] M. Perovšek, J. Kranjc, T. Erjavec, B. Cestnik, and N. Lavrač, "TextFlows: A visual programming platform for text mining and natural language processing," *Science of Computer Programming*, vol. 121, pp. 128–152, 2016.
[11] Y. Kano, M. Miwa, K. B. Cohen, L. E. Hunter, S. Ananiadou, and J. Tsujii, "U-compare: A modular NLP workflow construction and evaluation system," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 11–1, 2011.
[12] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.
[13] R. E. Kalman *et al.*, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.
[14] R. Parker, D. Graff, J. Kong, K. Chen, and K. Maeda, "English gigaword," *Linguistic Data Consortium*, 2011.
[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A study of skew in mapreduce applications," *Open Cirrus Summit*, vol. 11, 2011.