Two Architectures for Parallel Processing of Huge Amounts of Text

Mathijs Kattenberg,[◇] **Zuhaitz Beloki,**[♠] **Aitor Soroa,**[♠] **Xabier Artola,**[♠]
Antske Fokkens,[♣] **Paul Huygen**[♣] **and Kees Verstoep**[♣]
◇SURFsara, Science Park 140, 1098 XG Amsterdam
♠IXA NLP Group, University of the Basque county, 649 Posta kutxa, 20080 Donostia
♣The Network Institute, Vrije Universiteit, Boelelaan 1105, 1081 HV Amsterdam,
mathijs.kattenberg@surfsara.nl, {zuhaitz.beloki,a.soroa,xabier.artola}@ehu.es
{antske.fokkens,p.e.m.huijgen,c.verstoep}@vu.nl

Abstract

This paper presents two alternative NLP architectures to analyze massive amounts of documents, using parallel processing. The two architectures focus on different processing scenarios, namely batch-processing and streaming processing. The batch-processing scenario aims at optimizing the overall throughput of the system, i.e., minimizing the overall time spent on processing all documents. The streaming architecture aims to minimize the time to process real-time incoming documents and is therefore especially suitable for live feeds. The paper presents experiments with both architectures, and reports the overall gain when they are used for batch as well as for streaming processing. All the software described in the paper is publicly available under free licenses.

Keywords: Parallel processing, big data, system architecture

1. Introduction

The amount of textual information available to us today is enormous and is increasing by the day. Current technologies can analyze these texts at a very detailed level, but NLP analyses are time-consuming. We distinguish two issues that need to be dealt with when monitoring textual data that is both abundant and constantly updated. First, the problem of dealing with large amounts of data. Second, the problem of being able to respond immediately when new information comes in.

This paper introduces two architectures that make use of parallel processing to address these issues. Depending on the requirements and quantity of the data this can be done in different ways. Extremely large quantities of data that do not require instant analysis can be processed following a *batch* paradigm, where an entire batch of documents is processed in one go. However, if immediate updates of new information is required, focus lies on a quick turn-out per document. *Streaming* computing (Cherniack et al., 2003) expects documents to arrive at any moment, and aims at reducing the elapsed time needed to process one single document. The two architectures we present each tackle one of these scenarios. The batch processing architecture provides a general setup for NLP pipelines that can be used for processing large quantities of documents distributing jobs over various nodes. The streaming architecture can deal with live feeds using parallellization to optimize processing time per document.

We illustrate the idea behind the architectures through the NewsReader[1] use case, where linguistic analyses are used to form a "history recorder" of the news published around a certain topic. This paper is structured as follows. In Section 2., we describe the background of this work. This is followed by a presentation of the NLP modules we use in Section 3. Sections 4. and 5. introduce the Hadoop architecture for batch processing and the Storm architecture for live streaming, respectively. This is followed by the conclusion in Section 6.

2. Background and related work

Everyday, around 2 million news articles from thousands of sources are published and this number is increasing.[2] Keeping track of all this information, or even a subset of information about a specific domain, is unfeasible without technological support.

NewsReader (Vossen et al., 2014) aims to provide such support by performing detailed linguistic analyses identifying *what* happened to *whom*, *when* and *where* in large amounts of data in four languages, namely, English, Spanish, Italian and Dutch. The extracted information is stored as structured data in RDF (to be specific, as Event-Centric Knowledge Graphs (Rospocher et al., 2016)) so that information specialists can carry out precise search over the data. Vossen et al. (2014) explain how NewsReader functions as a **history recorder**: keeping track and storing every-

[1]`http://www.newsreader-project.eu`

[2]These numbers are based on an estimate by Lexis-Nexis, previously published in (among others) Vossen et al. (2014).

thing that has happened and relating new publications to what is already known from this track record.
Keeping information specialists informed with the latest news, while also allowing them to find connections with events from the past requires efficient processing of already available and incoming information. As such, we distinguish two scenarios. The first scenario is the **batch processing** scenario, where we need to optimize the processing of large amounts of data. This is necessary when initiating the history recorder with news from the past decade or even updating it with all news that was published on a specific day. The second scenario focuses on keeping users up to date with the most recent relevant changes. In this scenario, a new relevant article is published and this incoming article should be analyzed as quickly as possible. This **live streaming** scenario focuses on optimizing processing time for a single document.
The batch processing approach makes use of the Apache Hadoop software library[3] for implementing scalable processing of large collections of data. The Apache Hadoop framework and cluster architecture simplify many issues in parallel computing and allow for scalable and efficient data storage. Apache Hadoop offers a distributed file system and simple programming model that, when used correctly, leads to applications that scale almost linearly with data size, are resilient to machine failures and require little human administration and configuration.
While Apache Hadoop offers great libraries for developing parallel applications, it is less trivial to run an existing regular binary let alone a workflow or pipeline of several related modules. For this use case applications are often run via Hadoop Streaming.[4] The Hadoop streaming approach requires applications to communicate via standard input and output streams and this makes it difficult to signal failures and develop robust pipelines. We avoid this short-coming of Hadoop Streaming by implementing an application using the Cascading[5] software library. This allows us to leverage the benefits of parallel computing using Hadoop but still support a flexible modular pipeline. The details of this approach will be discussed in Section 4.
The live streaming computing approach presented in Section 5. relies on the Apache Storm framework[6] for implementing scalable processing of data streams.

[3]https://hadoop.apache.org/
[4]http://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html
[5]http://www.cascading.org
[6]https://storm.apache.org/

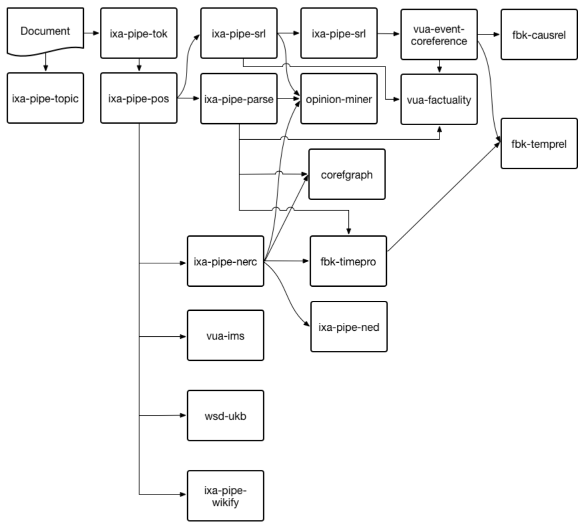

Figure 1: Overview of NLP modules

Storm is a framework for streaming computing whose aim is to implement highly-scalable and parallel processing of data streams. The system presented in Artola et al. (2014) uses Storm to integrate and orchestrate an NLP pipeline comprised by many modules, but it does so following a batch processing paradigm. The streaming approach presented here is an extension of Artola et al. (2014) to deal with streaming scenarios.

3. A pipeline for event recognition

The frameworks we present here are independent of the exact NLP modules that are used in the pipeline. The only requirement posed to the modules is that of using the NLP Annotation Format (Fokkens et al., 2014, NAF) for representing linguistic annotations when using the Storm architecture. The Hadoop setup can in principle work with any setup that uses a compatible representation to communicate between modules.
Table 1 shows the NLP modules used in our experiments, including the type of information they produce and consume. The *input* and *output* columns of the table indicate the information the module requires and the information the module provides, respectively.[7] This information can be used to group the modules on the basis of compatible settings and sequential dependencies for processing texts.
One of the main benefits of grouping the modules based on requirements is that modules that are placed in the same group can be executed in parallel, a fact

[7]These requirements are specified by means of NAF layers. See Fokkens et al. (2014) for more information about NAF.

Acronym	Description	module	input	output
Tok	Tokenizer	ixa-pipe-tok	raw text	sentences, tokens
Topic	Text classification	ixa-pipe-topic	raw text	topics
PoS	PoS tagging	ixa-pipe-pos	tokens	lemmas, PoS-tags
Parse	Parsing	ixa-pipe.parse	lemmas, PoS, tokens	parse trees
Dep	Dependency parsing	ixa-pipe-srl	lemmas, PoS	dependency trees
WSD-ukb	Word Sense Disambiguation	wsd-ukb	lemmas, PoS	lemmas, external refs.
WSD-ims	Word Sense Disambiguation	vua-ims	lemmas, PoS	lemmas, external refs.
NERC	Named Entity Recognition	ixa-pipe-nerc	lemmas, PoS	entities
NED	Named Entity Disamiguation	ixa-pipe-ned	lemmas, PoS, entities	entities
Wikify	Wikification	ixa-pipe-wikify	lemmas, PoS	markables
Time	Time expressions	fbk-timepro	lemmas, PoS, entities, parse trees	Timex3
Coref	Coreference Resolution	corefgraph	lemmas, PoS, parse trees	coreferences
SRL	Semantic Role Labeling	ixa-pipe-srl	lemmas, PoS	semantic roles
eCoref	Event coreference	vua-eventcoreference	srl	coreferences
TempRel	Temporal relations	fbk-temprel	lemmas, PoS, entities, parse trees, coref, timex3	tlink
CausalRel	Causal relations	fbk-causrel	lemmas, PoS, entities, parse trees, coref, timex3, tlink	tlink
Factuality	Factuality	vua-factuality	lemmas, PoS, entities, dependencies, coref	factuality
Opinion	Opinions	opinion-miner	lemmas, PoS, entities, dependencies, parse trees	opinions

Table 1: NewsReader modules for English and their properties

that we exploit in our experiments on streaming processing (see Section 5.). For instance, once the PoS module has identified the Lemmas and PoS-tags, all modules consuming lemmas and PoS annotations can be run in parallel. In our pipeline, this includes the Parse, NERC, Dep, and WSD modules. The NERC is a requisite for some modules, such as the NED, Time and Coref. The Time and Coref modules both also use the output of the Parse module. These modules can thus be placed in the same group and run in parallel as well.

Figure 1 provides a full overview of the NLP modules used in our experiments. The arrows indicate dependencies between modules.[8] A description and relevant references of a previous version of the modules can be found in Agerri et al. (2015).

4. Batch processing with Hadoop

The first architecture we describe aims at optimizing processing time for a large set of documents. It maintains a basic setup where complete documents are passed through the entire pipeline applying individual NLP modules in a fixed order. We first describe Hadoop and the Cascading approach we use. This is followed by the main results of our latest processing task and a discussion about further optimizations.[9]

4.1. Hadoop

The Hadoop (White, 2009) framework is designed to distribute processing of very large datasets across clusters of machines. Hadoop can partition data and computation allowing users to create simple programs that can scale from a single server to thousands of servers. Moreover, it is robust and fault tolerant and can run on commodity hardware.

The APIs are implemented in Java and there is no extensive support to run normal binaries. This can make it challenging to directly support some applications. An example is the NLP pipeline we use that consists of modules programmed in various languages. The next subsection describes how we address this challenge.

4.2. Cascading

The large variety of our NLP modules with different requirements made it non-trivial to place into the MapReduce framework (Dean and Ghemawat, 2008), Hadoop's programming model, without a complete rewrite of all modules. We address this challenge by implementing a Hadoop application using the higher-level Cascading library. Cascading is designed to support complex workflows on Hadoop without needing to implement many individual jobs and implement many low-level map and reduce functions. Using Cascading we define functions on a flow of data (tuples).

[8]This pipeline performs NLP analyses on English. Similar pipelines for Dutch, Spanish and Italian have been developed in NewsReader. We use the English pipeline in our experiments, because it is the most complex of the four.

[9]The basic architecture has previously been described by us in a technical report (Agerri et al., 2015).

This flow is then translated to one or many MapReduce jobs.

For the Newsreader NLP pipeline we defined a data flow consisting of tuples with a document identifier, document NAF XML text and a supporting field indicating whether a NAF document was processed successfully. The tuples in this data flow are first stored on the Hadoop distributed file system (HDFS) using a custom application. During job execution tuples are read from HDFS and supplied as input to many function applications in parallel. The functions take a tuple as input, run a process calling an NLP module and collect the output of this sub-process. Several functions applications in sequence form the NLP pipeline. Due to the nature of the NLP pipeline, both input and output tuples have the same fields: most functions modify only the NAF content.

Our implementation thus 'wraps' NLP modules in functions that create a local process for each step in the pipeline. This process receives a path to the NLP module code which has been distributed using Hadoop's distributed cache. In addition, a path to a local scratch directory and NAF input are provided (either via standard in or via a file path). Our application can integrate any module that can be called from the command-line taking standard input, a path to the module itself and a path to a temporal or scratch directory and produces a NAF document (either via standard out or via a file path). Error handling of the modules is done by monitoring the output streams of the process running the module and by making use of time outs.

Our Cascading application runs the Newsreader pipeline during the map phase of MapReduce. The many map tasks each execute the pipeline in parallel on input documents read from HDFS. One map task may apply the pipeline to one or many documents. This decision is determined by how data is partitioned on HDFS. This allows parallel execution of the pipeline on many documents and makes use of data locality as much as possible.

4.3. Scaling and Evaluation

We used the Cascading architecture to process nearly 2.5 million news articles on the Hadoop cluster of SURFsara.[10] We obtained news articles spanning 11 years of financial news about the car industry from LexisNexis. We ran initial tests investigating the correlation between document length and processing time and observed that processing time increases linearly. This is not surprising given that most modules in our pipeline work on a sentence level. In this set of articles from the same subdomain, writing style is not likely to vary excessively and longer documents will thus typically contain more sentences of similar length and style as shorter documents. We selected the 40% of the articles that consisted of 1,000 - 4,000 characters, because this covers the typical news article length resulting in the 2.5 million articles mentioned above.

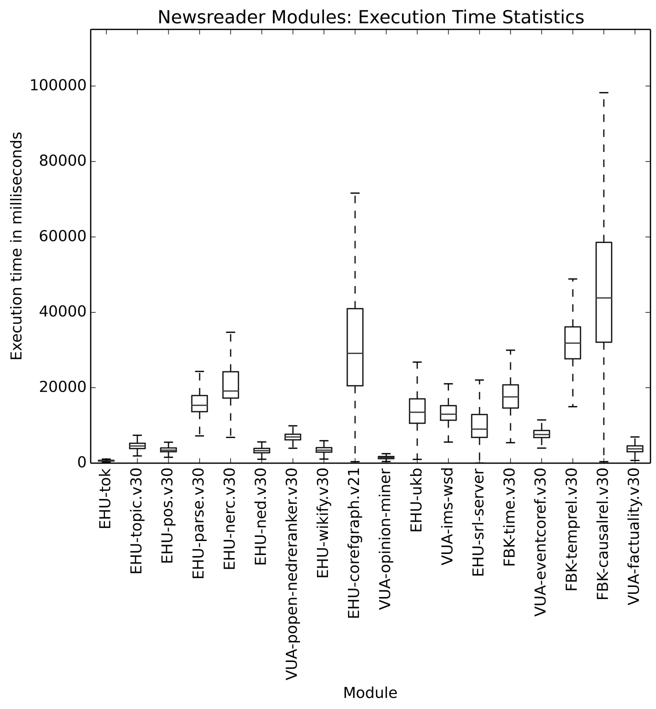

Figure 2: Newsreader Modules: execution time statistics

The total process was divided in 225 jobs resulting in approximately 45 million NLP processing tasks (where one task corresponds to the execution of a single module). Figure 2 provides a boxplot of the execution time of the 45 million NLP processing tasks. The average processing time was 4.4 minutes per document. Overall, it took 198,134 core hours (a single CPU core would take 198,134 hours to process the data) to process the entire dataset. The SURFsara cluster consists of 170 nodes, 1,400 cores and 2 petabytes of data storage. This means that, in case of full capacity, 141.5 hours would be needed to process the full dataset with an average of approximately 17.7 thousand documents per hour.[11]

The source code to the newsreader-hadoop Cascading application is available under a Apache 2.0 license and can be found on GitHub[12]. In addition to the application source a full distribution including all NLP modules is available as download from the GitHub site.

[10] `https://www.surfsara.nl`

[11] Hadoop at SURFsara is shared among other users. In practice, we processed around 4,000 documents per hour.

[12] `https://github.com/sara-nl/newsreader-hadoop`

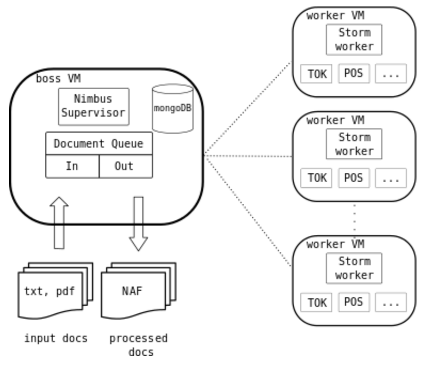

Figure 3: Streaming computing architecture

5. Streaming processing with Storm

Streaming processing deals with scenarios where documents may arrive at any time and have to be processed as fast as possible. In batch processing all the cluster nodes are busy processing a large set of documents, whereas in a streaming processing scenario resources may be idle, waiting for documents to arrive. Therefore, streaming computing focuses on maximizing resource usage for each document minimizing document latency. In our approach, we minimize document latency by running the pipeline modules for one document in parallel.

5.1. A distributed architecture for streaming processing

We implement the streaming architecture using virtual machines (VMs). Virtualization is a widespread practice that increases the server utilization and addresses the variety of dependencies and installation requirements. All the required software and the NLP modules are installed into VMs and can be deployed into a cluster of nodes. We distinguish two types of VMs in the distributed architecture:

- There is one *boss* VM which is the main entry point and receives the documents to be analyzed.
- There are many *worker* nodes performing the actual processing. Instances of NLP modules are deployed among worker nodes, and the execution is performed in parallel.

Figure 3 shows the main architecture for streaming processing. Documents arrive to the boss node and are stored in a queue, which streamlines the document flow and avoids overloading the system. Documents enter the processing chain as soon as there is a worker node ready to accept them. Once entered, they are analyzed using all available resources, that is, the NLP modules of the processing chain are executed in parallel using different worker nodes, whenever possible. The *Nimbus* component in the boss node orchestrates the processing flow and performs the necessary load balancing between the machines in the cluster. The boss node also contains a MongoDB database where partially annotated documents are stored. When the processing chain finishes, the resulting documents are removed from the MongoDB database, stored into an output queue and sent to a central database, where new information is merged with the information represented so far.

Inside the worker nodes, each NLP module is wrapped inside a Storm *bolt*, which performs several tasks. The bolt receives the document identifier as a tuple, and it queries the MongoDB database to retrieve the information pieces needed by the particular module. For example, the coreference module requires terms, constituent trees and named entities to create the coreference chains, and thus the bolt node retrieves only these layers and not retrieve any additional layers (such as dependency trees, factuality or semantic roles) of the NAF document from the database. When the module finishes, the bolt node updates the database to insert the new annotations, and passes the document identifier to the next stage in the processing chain.

We describe the topology in a declarative way, specifying the type of information each NLP module produces and consumes. This leads to a precise definition of the pre- and post-requisites of the modules, which is used to automatically decide the module execution order, including modules that are executed in parallel if there is no dependency between them. Figure 1 presented in Section 3. shows the topology used in our experiments.

Most NLP modules are standalone programs, but some work following a client/server architecture. Servers typically consume many resources making it inefficient to use many instances. To overcome this limitation, the system allows creating specialized worker nodes that only execute some specific tasks. The NED module server requires 8GB of RAM and is deployed into a dedicated worker node used exclusively for NED. Using a custom scheduler inside *Nimbus* we guarantee that Storm will send the NED tuples to the dedicated worker, and that this worker will only perform this particular task.

5.2. Experiments and evaluation

We implemented a Poisson process that simulates a streaming scenario with documents arriving at any moment. Poisson processes emulate events occurring individually at random moments, but tending to occur at an average rate when viewed as a group. Poisson processes contain one parameter, the *rate param-*

Proc. time	Elapsed time	Idle	Latency (doc/sent/token)
290,276	148,156	7,558	148.60/4.18/0.17

Table 2: Streaming processing in seconds of 1000 documents.

eter, which sets the average number of events per unit of time. In our experiments we set the rate parameter to $1,000$ documents within a time frame of 40 hours. The documents contained $35,480$ sentences and $872,393$ tokens.
We created a cluster comprising one boss node (1CPU, 6GM RAM), one dedicated worker node for NED (1CPU, 10GB), and 7 worker nodes for the rest of the modules (2 CPU, 8GB). Table 2 shows the results of the processing. Processing time indicates the time spent when serial processing is used. The elapsed time indicates time when using parallel processing. Latency is the average time needed to process one document/sentence/word.
Processing the documents in a serial fashion would require circa 80 days, each document taking 290 seconds in average. Parallel processing drops this latency to 148.6 seconds, a gain of 50%. Looking at Figure 1 one would expect higher gains because there are up to 6 modules running in parallel. The reason for not obtaining a maximum latency lies in the imbalance among the modules' processing times. Some modules need much more time to complete their task acting as bottlenecks that hurt the overall performance.
Scripts for creating a fully working streaming cluster are publicly available and can be distributed freely.[13]

6. Conclusion and future work

We presented two architectures for batch and streaming scenarios. The architectures serve two different but complementary purposes: batch processing allows dealing with massive amounts of documents in one go; streaming processing allows dealing with the continuous flow of new information.
The Hadoop based batch approach was used to efficiently process a very large quantity of documents (2,498,633), with an overall latency of 4.4 minutes per document providing a potential of processing almost 18 thousand documents per hour. The streaming architecture based on Storm allows dropping the overall document latency by half, which allows quick consumption of new documents into the system.
The parallel execution of unrelated modules within a pipeline application could be explored for batch processing with Hadoop. This might reduce pipeline execution time, but would go against the ideas behind the MapReduce programming model and complicate resource allocation. Another point worth examining is finding the sweet spot of documents processed per map task. One map task executes the NLP pipeline for one or many documents depending on how data is stored on HDFS. Scheduling these tasks incurs overhead but many mappers improve cluster throughput which for multi-tenant clusters is preferred. Finally, the pipeline definition could be improved to support more flexible workflow definitions and better error handling.
As mentioned in Section 5., the optimization by parallellization of modules is not as high as might be aspected due to certain tools forming a bottleneck. One possible way of decreasing the overall latency is to run such demanding modules on stronger nodes in the cluster.[14] We hope to investigate this option in future work.

[13]`https://github.com/ixa-ehu/vmc-from-scratch`

[14]This option was pointed out by one of the reviewers.

7. Acknowledgements

We would like to thank the reviewers for their feedback which helped to improve this paper. Remaining errors are our own. This research was funded by European Union's 7th Framework Programme via the NewsReader project (ICT-316404). This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

8. Bibliographical References

Agerri, R., Aldabe, I., Beloki, Z., Laparra, E., de Lacalle, M. L., Rigau, G., Soroa, A., Fokkens, A., Izquierdo, R., van Erp, M., Vossen, P., Girardi, C., and Minard, A.-L. (2015). Event detection, version 2. Technical report, NewsReader Project.

Artola, X., Beloki, Z., and Soroa, A. (2014). A stream computing approach towards scalable nlp. In Nicoletta Calzolari (Conference Chair), et al., editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland, may. European Language Resources Association (ELRA).

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. (2003). Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Fokkens, A., Soroa, A., Beloki, Z., Ockeloen, N., Rigau, G., van Hage, W. R., and Vossen, P. (2014). NAF and GAF: Linking linguistic annotations. In *Proceedings of the 10th Joint ACL – ISO Workshop on Interoperable Semantic Annotation*.

Rospocher, M., van Erp, M., Vossen, P., Fokkens, A., Aldabe, I., Rigau, G., Soroa, A., Ploeger, T., and Bogaard, T. (2016). Building event-centric knowledge graphs from news. *Journal of Web Semantics*.

Vossen, P., Rigau, G., Serafini, L., Stouten, P., Irving, F., and Hage, W. R. V. (2014). Newsreader: recording history from daily news streams. In *Proceedings of the 9th Language Resources and Evaluation Conference (LREC2014)*, Reykjavik, Iceland, May 26-31.

White, T. (2009). *Hadoop: The Definitive Guide*. "O'Reilly Media, Inc.".