A stream computing approach towards scalable NLP

Xabier Artola, Zuhaitz Beloki, Aitor Soroa

IXA NLP Group, University of the Basque Country, Donostia, Basque Country,
{xabier.artola, zuhaitz.beloki, a.soroa}@ehu.es

Abstract

Computational power needs have grown dramatically in recent years. This is also the case in many language processing tasks, due to overwhelming quantities of textual information that must be processed in a reasonable time frame. This scenario has led to a paradigm shift in the computing architectures and large-scale data processing strategies used in the NLP field. In this paper we describe a series of experiments carried out in the context of the NewsReader project with the goal of analyzing the scaling capabilities of the language processing pipeline used in it. We explore the use of Storm in a new approach for scalable distributed language processing across multiple machines and evaluate its effectiveness and efficiency when processing documents on a medium and large scale. The experiments have shown that there is a big room for improvement regarding language processing performance when adopting parallel architectures, and that we might expect even better results with the use of large clusters with many processing nodes.

Keywords: Distributed NLP architectures, Big Data, Storm

1. Introduction

Nowadays there is a continuous increase of computational power needs due to an overwhelming flow of textual data. This calls for a paradigm shift in computing architecture and large scale data processing. For example, the main goal of the NewsReader[1] project is to perform real-time event detection and extract from text what happened to whom, when and where, removing duplication, complementing information, registering inconsistencies and keeping track of the original sources. The project foresees an estimating flow of 2 million news items per day and the linguistic analysis of those documents needs to be done in a reasonable time frame (one or few hours). The project faces thus an important challenge regarding the scalability of the linguistic processing of texts.

The challenges NewsReader faces fall into a new class of the so called "Big Data" tasks, requiring large scale and intensive processing and which are able to scale efficiently to very big volumes of data (Elsayed et al., 2008; Pantel et al., 2009; Sarmento et al., 2009; Singh et al., 2011; Beheshti et al., 2013; Yu and Chen, 2013; McCreadie et al., 2013; Sakr et al., 2013).

MapReduce (Dean and Ghemawat, 2008) is a programming model framework designed to perform large scale computations and that is able to scale to thousand of nodes in a fault-tolerant manner. However, *MapReduce* follows a *batch* processing model, where computations start and end within a given time frame. *Streaming computing* (Cherniack et al., 2003) represents an alternative programming model for processing a continuous flow of data streams. Streaming computing systems have to deal with very a high level of data throughput still guaranteeing a low level of response latency. This programming model assumes that data are presented to the algorithm as one or more input streams that are processed in order, and only once.

In this paper we describe a series of experiments performed with the goal of analyzing the scaling capabilities of the NewsReader NLP pipeline. We propose a new approach for scalable distributed NL processing across multiple machines. We also evaluate the effectiveness and efficiency of the proposed approach when processing documents on a medium and large scale.

The paper is organized as follows. Section 2 analyzes existing solutions for big data processing and presents the main framework used to implement the NLP pipeline used in the experiments. In section 3, we briefly describe the annotation format used in our framework. Experiments and results are described in sections 4 and 5. Finally, some conclusions are drawn and further work is depicted.

2. Big data for scalable NLP

Processing large amounts of textual data has become a major challenge in the NLP research area. As the majority of digital information is present in the form of unstructured data like web pages or news articles, NLP tasks such as cross-document coreference resolution, event detection or calculating textual similarities often require processing millions of documents in a timely manner (Elsayed et al., 2008; Pantel et al., 2009; Sarmento et al., 2009; Singh et al., 2011; Beheshti et al., 2013; McCreadie et al., 2013). For instance, in (Singh et al., 2011) the authors process a corpus comprising news articles published during the last 20 years. McCreadie et al. (2013) present a distributed framework for event detection that is capable to effectively process thousand of twitter posts every second. The research activities conducted within the NewsReader project strongly rely on the automatic detection of events, which are the core information units present in the news and which support any decision making process that depends on news articles. The research focuses on many challenging aspects such as event detection and modelling, storage and reasoning over events, etc.

Processing massive quantities of data requires designing solutions that are able to run distributed programs across a large cluster of machines. Besides, issues such as parallelization, distribution of data, synchronization between

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

nodes, load balancing and fault tolerance, etc. are of paramount importance. In this section we briefly analyze some of the most widely used frameworks for massive data processing.

The *MapReduce* algorithm arose from the need of the Google company to run straightforward programs with very large input data sets. This need led to the design of a solution where programs are distributed across large machine clusters. Therefore, a new library was designed with the aim of hiding from the user all the logic about the aforementioned issues, letting programmers concentrate their efforts on their applications' logic.

Hadoop is an open-source implementation of *MapReduce* that has been widely used during the last few years. The library is fault tolerant; it knows how to react when a worker node or even the master node fails.

One of the most important characteristic of Hadoop is that of being oriented to perform batch processing, but it leads to serious problems when using it for real-time streaming processing systems. Hadoop SPARK (Zaharia et al., 2010) overcomes this problem by extending Hadoop with new workloads like streaming, interactive queries and learning algorithms. In any case, using Hadoop or SPARK frameworks require reimplementing the NLP algorithms using a programming language from the *MapReduce* family.

2.1. Streaming computing

The main characteristic of the *batch* processing model is that of having a beginning and an end, i.e., processes on a batch setting start and eventually finish their jobs. In a *streaming computing* scenario (Cherniack et al., 2003), however, there exist no beginning nor end in the processing. Instead, the programming model is designed to process messages forever while maintaining high levels of data throughput and a low level of response latency.

S4[2] is an open source, general-purpose, distributed, scalable and partially fault-tolerant platform for developing and running distributed programs to process continuous streams of data. S4 was developed because of an unsuccessful attempt to adapt Hadoop to deal with applications consuming large real-time streams of data. Therefore, it was concluded that a library that would work for both batch and stream processing was not viable. S4 offers the flexibility to deploy new algorithms as needed in research environments, while scalability and high availability requested by production environments are taken into account.

The main units in the design of this system are the Processing Elements (PEs). The PEs encapsulate the functionality of each logical piece of processing. The only way of communication between PEs is by sending messages, making the system derive from a combination of MapReduce and the Actors model (Hewitt et al., 1973). A high level of encapsulation and transparency is achieved by this model, resulting in a high level of simplicity. However, S4 lacks a cluster balancing system, making the system unbalance over time.

Storm[3] is an alternative to S4 for streaming computing. It was created to satisfy the needs of a distributed and scalable real-time computation framework. The main design goals of Storm are the following:

- Make the design friendly and easy to understand.
- Provide a simple Application Programming Interface for processing data streams.
- Allow to set scalable clusters with high availability using commodity hardware.
- Minimize latency by supporting local memory reads and avoiding disk I/O bottlenecks.
- Use a symmetric architecture, where all nodes have the same responsibility and functionality.
- Use a pluggable architecture.

The main abstraction structure of Storm is the topology, top level abstractions which describe the processing that each message passes through. The topology is represented as a graph where nodes are processing components, while edges represent the messages between them. Topology nodes fall into two categories: the so called *spout* and *bolt* nodes. *Spout* nodes are the entry points of a topology and the source of the initial messages to be processed. *Bolt* nodes are the actual processing units, which receive messages, process them, and pass the processed messages to the next stage in the topology.

The data model of Storm is the *tuple*, i.e., each *bolt* node in the topology consumes[4] and produces tuples. The tuple is an abstraction of the data model, and is general enough to allow any data to be passed around the topology.

In Storm, each node of the topology may reside on a different physical machine; the Storm controller (called *Nimbus*) is the responsible to distribute the tuples among the different machines, and guarantees that each message traverses all the nodes in the topology.

It is important to note that in Storm you can have several instances of each topology node, thus allowing actual parallel processing. Following the so called *parallelism hint*, it is possible to specify how many instances of each topology node will be actually running.

When it comes to the issue of cluster management, Storm uses a centralized model like Hadoop. There is a master node, called Nimbus, and multiple worker nodes, known as Supervisors. The Nimbus is responsible for creating Supervisor instances through the cluster and assigning a task or a set of tasks to each of them. It is also its job to monitor the cluster for failures. Supervisors manage all the input and output events of a worker node and start/stop task processes as necessary. Storm, like S4, uses ZooKeeper to manage communication inside the cluster. It also performs automatic rebalancing to compensate the processing load between the nodes.

3. The NLP Annotation Format

The experiments described in this paper involve the integration of many NLP tools into a common framework.

[2] http://incubator.apache.org/s4/

[3] http://storm.incubator.apache.org/

[4] Unlike *spout* nodes, which are the initial nodes and therefore do not consume tuples.

```
<NAF>
 <!-- text layer -->
 <text>
  <wf id="w1" offset="0" length="4">John</wf>
  <wf id="w2" offset="5" length="6">taught</wf>
  <wf id="w3" offset="12" length="11">mathematics</wf>
  <wf id="w4" offset="24" length="2">in</wf>
  <wf id="w5" offset="27" length="3">New</wf>
  <wf id="w6" offset="31" length="4">York</wf>
 </text>
 <!-- term layer -->
 <terms>
  <term id="t1" lemma="John" pos="R">
   <span><target id="w1"/></span>
  </term>
  <term id="t2" type="open" lemma="teach" pos="V">
   <span><target id="w2"/></span>
  </term>
  ...
 </terms>
 <!-- entity layer -->
 <entities>
  <entity id="e1" type="person">
   <references>
     <!--John-->
    <span><target id="t1"/></span>
   </references>
  </entity>
  <entity id="e2" type="location">
   <!--New York-->
   <references>
    <span><target id="t5"/><target id="t6"/></span>
   </references>
   <externalReferences>
    <externalRef
      reference="http://dbpedia.org/page/New_York_City"
      confidence="0.8"/>
   </externalReferences>
  </entity>
 </entities>
</NAF>
```

Figure 1: Excerpt of a NAF document showing the text, term and entity layers.

One key issue for the integration of diverse NLP modules is the definition of a common annotation format, which guarantees the correct interoperability among the tools. In this work we use the so-called NLP Annotation Format (NAF) (Fokkens et al., 2014), which is designed to be the standard format for exchanging information between linguistic processing tools within the NewsReader project.

NAF follows the main principles of LAF as outlined in (Ide et al., 2003). Like LAF, NAF aims at maximum flexibility, processing efficiency and reusability. It is a layered, extensible format where each tool incrementally adds its output while maintaining all information that was present in its input. NAF has shown to be suitable for a complex pipeline combining tools developed at different sites in the NewsReader project.

Nowadays there exists a wide range of representation schemas for annotating documents with linguistic information. Although a detailed analysis including a comparison of those schemas with NAF is clearly out of the scope of the present paper, let us briefly depict the main reasons which led the NewsReader project to adopt NAF for NLP annotation. On the one hand, NAF allows combining the representations of multiple semantic modules, including the relations between alternative analyses. On the other hand, the complex tasks carried out in the project require the definition of semantic layers for annotations such as factuality statements for which no current standard implementation exists. Besides, NAF is specifically designed to work on distributed environments where NLP modules produce annotations on the same document in parallel.

NAF comprises several annotations over a text at different linguistic levels (morphosyntactic, syntactic, semantic). The following general rules are met in all layers:

- `<span>` elements are used to define the range of linguistic elements to which an annotation applies.
- Linguistic annotations of a particular level always span elements of previous levels.
- Linguistic annotations of different levels are not mixed.

The "levels" in the general rules refer to different types of linguistic information, which can be different groupings of linguistic entities (e.g. tokens vs terms vs chunks), relations between linguistic entities (e.g. dependencies, semantic roles), or information about a specific linguistic entity (e.g. disambiguated word sense).

The most basic level in NAF is the text layer which assigns identifiers to tokens in the text. The term layer defines basic terms (lexical units) which consist of one or more tokens in the case of multiword expressions. Further layers (chunks, entities, etc.) typically consist of one or more terms. `span` elements are used to refer to specific elements in lower layers. For instance, in NAF multiword expressions are described by a single term, which spans to the ids of the tokens that compose the expression. NAF provides the following layers to represent the output of common NLP tasks:

The header contains metadata about the input document, such as its public ID, the URI, creation time, etc. The header also records information about all the LP modules used to produce the annotations in the NAF document.

The raw layer contains the input document verbatim. Because the input text may contain many characters which are invalid in XML, the raw layer is enclosed within a `CDATA` section.

The text layer contains the tokens of the document. Optionally, sentence, paragraph and page boundaries are also indicated. This layer is the result of sentence splitting and tokenization.

The terms layer contains words and multiwords. It also includes information such as part-of-speech, references to other resources such as wordnet senses, compound elements, etc. Since (multi)words consist of tokens, they refer to tokens in the *text layer*.

The chunks layer contains chunks of words, such as noun phrases, prepositional phrases, etc. Since chunks consist of words, they refer to words in the *terms layer*. Each chunk has a *head*, which is also an item in the terms layer.

The dependency layer contains dependency relations between words. These relations refers to terms in the *terms layer*.

The entity layer contains entity mentions. Entity mentions have an entity type (person, organization, etc.) and are linked to instances from external resources such as Wikipedia or Dbpedia.

The coreference layer contains clusters of term spans which refer to the same entity.

The semantic role layer, including predicates and arguments associated to it.

The time expression layer identifies time expressions mentioned on the text.

The factuality layer encodes the veracity or *factuality* of events as mentioned in the text. This information is useful for recognizing whether the events mentioned in the text actually happened (factual events), did not happen (contrafactual events), or there is some uncertainty about the event happening or not.

Figure 1 shows an excerpt of a NAF document comprising three layers: text, terms and entities. The characteristic of being multi-layered makes NAF to be particularly well suited to work on a distributed and parallel environment. Processing modules represent their output in different NAF layers and usually they do not modify the annotations of the lower layers. Therefore, several processors can create new annotations to the same document in parallel as far as there is no dependence between them.

4. Experiment setting

In this section we describe the different settings established to carry out the experiments. Scalable NLP processing requires parallel processing of textual data. The parallelization can be effectively performed at several levels, from deploying copies of the same LP processor among servers to the reimplementation of the core algorithms of each module using multi-threading, parallel computing. This last type of fine-grained parallelization is clearly out of the scope of the present work, as it is unreasonable to reimplement all the modules needed to perform complex tasks as event mining. We rather aim to process huge amount of textual data by defining and implementing an architecture for NLP processing which allows the parallel processing of documents. First of all, let us explain how we interpret Storm concepts, such as *spout* and *bolt* nodes, and tuples, in the case of our particular language processing scenario:

- The *spout* node is a process which reads a text document and sends it to the first bolt of the topology.
- The *bolt* nodes are wrapper programs which receive input tuples, call the actual NLP modules for processing, and send the output tuples to the next stage in the topology.
- The tuples in our Storm topology comprise two elements, a document identifier and the XML serialization of the NAF document encoded as a string.

Layer	100 docs.	1,000 docs.
tokens	138,803	1,185,933
senses	42,519	390,948
entities	10,804	75,349

Figure 3: Number of annotations obtained for each layer.

We created a small NLP pipeline comprising four modules: a tokenizer (TOK), a part of speech tagger (POS), a Named Entity Recognition and Classification module (NERC), and a Word-Sense Disambiguation module (WSD). All this modules are based on the IXA-pipeline processing framework (Agerri et al., 2014).

Initially the four modules are executed following a pipeline architecture, i.e., each module running sequentially one after the other. This setting is the baseline system and the starting point for our analysis.

On a second experiment, we implement a Storm topology following again a pipeline approach. This setting is similar to the baseline system but has a main advantage. When a module finishes the processing, it passes the annotated document to the next step, and starts processing the next document. Therefore, in this setting there are as many documents processed in parallel as stages in the pipeline. Because we have a pipeline comprising 4 modules, the pipeline is able to process 4 documents at the same time.

On a final setting, we experiment creating many instances of some selected *bolt* nodes, therefore allowing the parallel execution of them. Figure 2 illustrates this final experiment setting.

All the experiments were performed on a PC machine with an Intel Core i5-3570 3.4GHz processor with 4 cores and 4GB RAM, running on Linux.

5. Results

We experimented the NLP pipeline with 1000 documents, each one comprising an average of 1200 words and 50 sentences. We performed experiments with a subset of 10 documents ($16,208$ words, 682 sentences), a subset of 100 documents ($138,803$ words, $5,416$ sentences) and with the complete set of 1000 documents ($1,185,933$ words, $48,746$ sentences). Figure 3 shows the number of annotations obtained for each layer after the processing.

Figure 4 shows the time elapsed in processing the documents. The first six rows correspond to the processing of 10 documents, the next six rows to the processing of 100 documents and the last six rows to the processing of 1000 documents. As the figure shows, the baseline system runs at a performance of about 100 words per second. The simple Storm topology yields a performance gain of less than 13%, which is less than expected. When analyzing the result, we realized that there is an unbalance regarding the time spent by each module. The 96% of the processing time is spent by the WSD module, which is by far the module needing more time to complete its task. Although the Storm topology can in principle multiply the performance by a factor of four, in practice all the computing is concentrated in one single node, which severely compromises the overall performance gain.

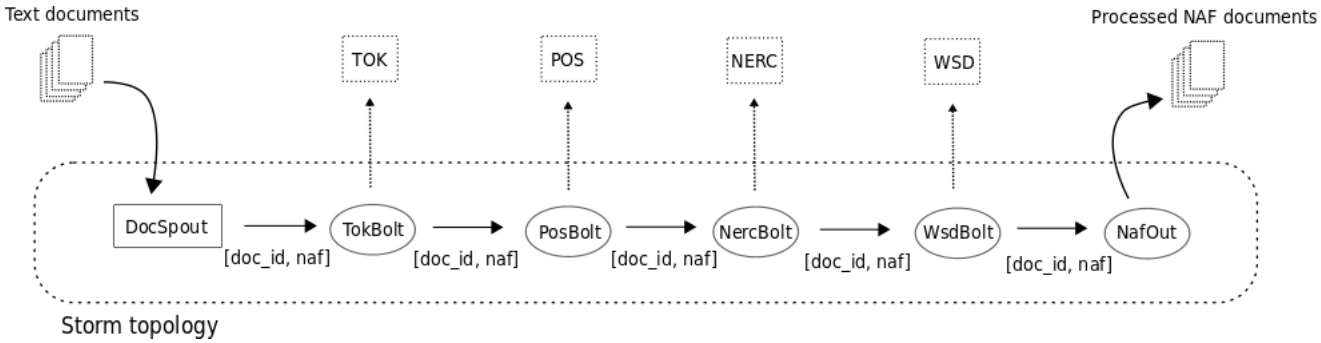

Figure 2: The Storm topology of our pipeline. It is composed of a *spout*, four *bolts* that perform the actual NL processing, and a special bolt *NafOut* that writes the resulting NAF documents to XML files. The NLP *bolts* are wrapper programs that call the corresponding external NLP modules to do their job.

	Total time	words/s	sent/s	Gain
10 documents				
pipeline	2m42s	99.8	4.2	-
Storm	2m25s	111.5	4.7	%10.4
$Storm_2$	1m29s	182.9	7.7	%45.4
$Storm_4$	1m32s	175.3	7.4	%43.0
$Storm_5$	1m28s	182.5	7.7	%45.3
$Storm_6$	1m22s	195.4	8.2	%49.0
100 documents				
pipeline	21m16s	108.8	4.2	-
Storm	18m43s	123.5	4.8	%12.0
$Storm_2$	10m48s	214.3	8.4	%49.3
$Storm_4$	7m46s	297.6	11.6	%63.5
$Storm_5$	7m44s	299.1	11.7	%63.7
$Storm_6$	7m48s	296.1	11.6	%63.3
1000 documents				
pipeline	3h15m16s	101.2	4.2	-
Storm	2h50m21s	116.0	4.8	%12.8
$Storm_2$	1h40m37	196.5	8.1	%48.5
$Storm_4$	1h14m25s	265.6	10.9	%61.9
$Storm_5$	1h10m45s	279.3	11.5	%63.8
$Storm_6$	1h11m37s	276.0	11.3	%63.3

Figure 4: Performance of the NLP pipeline in different settings: *pipeline* is the basic pipeline used as baseline; *Storm* is the same pipeline executed as a Storm topology; $Storm_2$ represents a Storm pipeline with 2 instances of the WSD module ($Storm_4$ has 4 instances, $Storm_5$ 5, and $Storm_6$ 6).

With these points in mind, we experimented four alternatives (dubbed $Storm_2$, $Storm_4$, $Storm_5$, and $Storm_6$), with respectively 2, 4, 5 and 6 instances of the WSD module running in parallel. The results in Figure 4 show that running multiple instances of WSD does increase the overall performance significantly. The major gain is obtained with five instances of WSD, with an increase of 63% in the overall performance. More WSD instances do not help improving the results, which is expected given the fact that the machine used for the experiments has 4 CPU cores.

6. Conclusion

In this paper a new approach for scalable distributed language processing across multiple machines using Storm has been proposed. We have described and evaluated the efficiency of a series of experiments carried out with the goal of analyzing the scaling capabilities of the NewsReader NLP pipeline.

These initial experiments have shown that there is a big room for improvement regarding language processing performance when adopting parallel architectures such as Storm. With the use of large clusters with many nodes, we might expect a significant boost in the performance of overall NLP processing.

The experiments conducted in this paper are only an approach on the way to get a much more sophisticated distributed environment for NLP. Our next objectives focus on experimenting with a larger scale setup in three different aspects: running the experiments in a multi-node cluster with high computing capabilities, enhancing the general system architecture and designing more sofisticated topologies and algorithms.

The hardware used for these experiments was useful only for testing purposes. In the future we aim to have a cluster composed of several nodes, an essential scenario to make the most of the distributed architecture designed for the pipeline. This will allow us to experiment with a much larger input document set as well.

As we are developing a totally distributed and highly scalable system, several architecture-related issues come out. One of them is the input method that will receive text documents and send them to the pipeline. To accomplish that, we foresee the need of a distributed message queue system as the input. Another issue is the fact that too much data traffic is produced between each NLP module, since a full NAF document with all the layers' annotations must be sent from each module for every document to be processed. This could be avoided using a distributed database like MongoDB and retrieving and storing only the annotation layers required and produced by each module.

Similarly, we have in mind a couple of topology design improvements to be delved in the future:

- Use of non linear topologies. The experiments described here follow a pipeline approach, but in principle we could also run them on non linear topologies, where two modules are processing the same document at the same time. Non linear topologies require considering the following aspects:

We need to clearly identify the pre- and post-requisites of each module, thus deducting the indications as to which modules must precede which and which modules can be run in parallel on the same document.

We need a special *bolt* which receives input from many NLP module *bolts* (each one conveying different annotations on the same document) and *merges* all this information producing a single, unified document.

- Granularity-based splitting of documents. NLP modules work at different levels of granularity. For instance, a POS tagger works at sentence level, the WSD module works at paragraph level, whereas a coreference module works at document level. We want to experiment splitting the input document into pieces of the required granularity, so that the NLP modules can quickly analyze those pieces, thus increasing the overall processing speed.

Acknowledgment

We are grateful to the anonymous reviewers for their insightful comments. This work has been partially funded by the NewsReader (FP7-ICT- 2011-8-316404) and SKaTer (TIN2012-38584-C06-02) projects. Zuhaitz Beloki's work is funded by a PhD grant from the University of the Basque Country.

7. References

Rodrigo Agerri, Josu Bermudez, and German Rigau. 2014. IXA pipeline: Efficient and ready to use multilingual NLP tools. In *Proceedings of the 9th Language Resources and Evaluation Conference (LREC2014)*.

Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, STOC '96, pages 20–29, New York, NY, USA. ACM.

Seyed-Mehdi-Reza Beheshti, Srikumar Venugopal, Seung Hwan Ryu, Boualem Benatallah, and Wei Wang. 2013. Big data and cross-document coreference resolution: Current state and future opportunities. *CoRR*, abs/1311.3987.

Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January.

Jeffrey Dean and Sanjay Ghemawat. 2008. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January.

Tamer Elsayed, Jimmy J. Lin, and Douglas W. Oard. 2008. Pairwise document similarity in large collections with mapreduce. In *ACL (Short Papers)*, pages 265–268. The Association for Computer Linguistics.

Antske Fokkens, Aitor Soroa, Zuhaitz Beloki, Niels Ockeloen, German Rigau, Willem Robert van Hage, and Piek Vossen. 2014. NAF and GAF: Linking linguistic annotations. In *To appear in Proceedings of 10th Joint ACL/ISO Workshop on Interoperable Semantic Annotation (ISA-10)*.

Sebastian Hellmann, Jens Lehmann, Sören Auer, and Martin Brümmer. 2013. Integrating nlp using linked data. In *Proceedings of the 12th International Semantic Web Conference (ISWC)*.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Nancy Ide, Laurent Romary, and Éric Villemonte de La Clergerie. 2003. International standard for a linguistic annotation framework. In *Proceedings of the HLT-NAACL 2003 Workshop on Software Engineering and Architecture of Language Technology Systems (SEALTS)*. Association for Computational Linguistics.

Richard McCreadie, Craig Macdonald, Iadh Ounis, Miles Osborne, and Sasa Petrovic. 2013. Scalable distributed event detection for twitter. In *Proceedings of IEEE International Conference on Big Data*.

Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. 2009. Web-scale distributional similarity and entity set expansion. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2 - Volume 2*, pages 938–947, Stroudsburg, PA, USA. Association for Computational Linguistics.

Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. 2013. The family of mapreduce and large scale data processing systems. *CoRR*, abs/1302.2966.

Lus Sarmento, Alexander Kehlenbeck, Eugnio C. Oliveira, and Lyle H. Ungar. 2009. An approach to web-scale named-entity disambiguation. In Petra Perner, editor, *MLDM*, volume 5632 of *Lecture Notes in Computer Science*, pages 689–703. Springer.

Sameer Singh, Amarnag Subramanya, Fernando Pereira, and Andrew McCallum. 2011. Large-scale cross-document coreference using distributed inference and hierarchical models. In *Association for Computational Linguistics: Human Language Technologies (ACL HLT)*.

Wei Yu and Junpeng Chen. 2013. The state-of-the-art in web-scale semantic information processing for cloud computing. *CoRR*, abs/1305.4228.

Matei Zaharia, N. M. Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. Technical report, EECS Department, University of California, Berkeley, May.