The Report committee for Simon Hafner

Certifies that this is the approved version of the following report:

Typesafe NLP Pipelines on Spark

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

Jason Baldridge

Co-Supervisor: _____

Katrin Erk

Typesafe NLP Pipelines on Spark

by

Simon Hafner, B.A.

Report

Presented to the Faculty of the Graduate School

of the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

December 2014

Typesafe NLP Pipelines on Spark

by

Simon Hafner, M.A.
The University of Texas at Austin, 2014

SUPERVISOR: Jason Baldridge
CO-SUPERVISOR: Katrin Erk

Natural language pipelines consist of various natural language algorithms that use the annotations of a previous algorithm to compute more annotations. These algorithms tend to be expensive in terms of computational power. Therefore it is advantageous to parallelize them in order to reduce the time necessary to analyze a large document collection. The goal of this project was to develop a new framework to encapsulate algorithms such that they may be used as part of a pipeline without any additional work.

The framework consists of a custom-built data structure called Slab which implements type safety and functional transparency to integrate itself into the Scala programming language. Because of this integration, it is possible to use Spark, a MapReduce framework, to parallelize the pipeline on a cluster. To assess the performance of the new framework, a pipeline based on the OpenNLP library was created. An existing pipeline implemented in UIMA, an industry standard for natural language pipeline frameworks, served as a baseline in terms of performance. The pipeline created from the new framework processed the corpus in about half the time.

Contents

Chapter 1

Introduction

Natural language pipelines consist of different natural language algorithms that use the annotations of a previous algorithm to compute more annotations. These pipelines are demanding in terms of computational power at both complexity of the annotations as well as the amount of documents processed. The annotations may refer to either text or other annotations and the annotations should be indexed for short access times. Natural language algorithms tend to be expensive in terms of computational power, and therefore it is advantageous to parallelize them in order to reduce the time cost to analyze a large document collection.

The goal of this project is to develop a new framework to encapsulate algorithms such that they may be used as part of a pipeline without any additional work. One main task of the framework is to collect the data generated by the algorithms and presenting this data to the next algorithm in a standardized way. The concept of a standardized way is known as an API. The other main task is to parallelize the processing of the documents to speed up the pipeline, if the algorithms used allow it.

The project focuses on using existing libraries to harvest the benefits of these libraries as they improve. The inclusion of existing features also keeps the core of the project simple, which reduces the possible amount of bugs. There is also little learning curve for people who are already familiar with these tools. So the libraries included should have a spread in the community to give the project a kickstart in finding traction.

1.1 The two tasks

The project can be reduced to the two tasks of data representation and parallelization. The data representation problem is unique to pipelines, while the parallelization does not exhibit any special properties which would differentiate it from other projects enough to not be able to use standard concepts and tools.

1.1.1 Data Representation

> Get your data structures correct first, and the rest of the program will write itself.
> - David Jones

Language Analysis with a computer is usually done with different algorithms joined together. The joining is usually done via a waterfall model, where each algorithm takes the output from a previous algorithm and uses that output and the document itself to generate new output. The waterfall model implies that there is no feedback from a later algorithms to earlier stages. The word pipeline comes from UNIX pipelines, where different specialized programs are joined together to perform a task. The data format is plaintext, where the elements are separated by newlines. The tools can be interchanged to perform different tasks.

NLP pipelines have a similar idea, where different parts can be plugged in as required. However, most pipelines tend to consist only of a single library. The data format of pipelines is also more complex than what usual UNIX tools process. A parse tree of the sentence "Tom ate an apple" can still be expressed with newlines, but the lines are not regular anymore, they are context-free. The meaning of a line depends on what comes before it.

Tom N1
ate V1
an D1
apple N2
. S1

DP1 D1 N2
VP1 V1 DP1
CP1 N1 VP1

SN CP1 S1

The framework has to define how e.g. the parse tree is represented in data to join the parts of the pipeline together. The data structure to store the annotations has to be general enough to allow to store simple tags or complex parse trees. If there are restrictions on which types can be stored, it is easier to reason about the behavior of the data, which is important for e.g. fast access. The preferred behavior of the data structure would be to enable special behavior if the data type is known and still allow for unknown data types.

There are two representations to each data structure, the internal and the external representation. The internal representation refers to how the data structure is represented inside the programming language and how it is accessed. Depending on the programming language, it might also ensure type safety, e.g. when you expect to retrieve a string from the data structure, the type safety ensures you get a string in any circumstance. The external data representation describes how the data is serialized and sent to another process, which may be on the same machine or on the other side of the world. It is important to have an external representation of the data to achieve serialization, as the data will have to be sent to other computers.

In a language which does not have type safety a data structure to store the annotations can be achieved with a map which uses the class as key and the annotations as values in a list. Such a construct is also possible in a language that ensures type safety, but it is not as simple. Maps can only contain a single type as value, so the developer has to manually specify the type of the element after retrieving it. As an additional feature, it is possible to check if a required type is available in the data structure, but the check requires additional code. This project will use a third route with a list-like data structure that stores types as well. A single type can be accessed by calculating at which position the element of a certain type is at compile time.

To generate the external representation, there are two basic ways to serialize annotations. Either the document itself is modified to contain the annotations or the annotations are stored with character offsets outside the document. This project uses the second approach using offsets, because it does have fewer limits on which annotations can be stored.

UIMA encapsulates the annotations into a custom data structure called Common Analysis Structure (CAS). (Ferrucci and Lally, 2004) UIMA DUCC on top of UIMA Scaleout

can be used to process a lot of documents in parallel. The C&C parser uses a less general annotation structure, which is less complex and faster. (Curran et al., 2007)

1.1.2 Parallelization

> Whenever possible, steal code. - Tom Duff

Because of practical concerns, the algorithms should be parallelized if possible. Parallelization decreases the runtime of a pipeline proportionally to the number of processors added, with a small additional cost for the coordination overhead. A typical desktop machine has currently four processor cores. A cloud computing solution has thousands available. An algorithm may run 10 seconds per documents. With tens of thousands of documents, the pipeline may take multiple days to complete without parallelization. With the help of parallelization, the process may take mere hours.

Parallelization requires to separate the workload into small pieces that can be distributed to different machines. The pieces in case of a pipeline are algorithms that are applied to a document. The algorithm can be run on multiple processors at the same time on different documents to split the computation time. Naturally, the split only works if the algorithm is independent of other documents. Topic modeling is an algorithm that needs to see the full document at the same time, so it cannot be parallelized by simple means. However, previous steps to the topic model such as tokenizing can still be parallelized.

There are two main ways to address parallelization, the reactive and the batch style. The reactive style has a well-known representation, the actors. Each actor runs independently and parallel to each other, the communication happens via messages which are sent by actors to other actors. Any actor may send messages to any other actor, or even to itself. The batch style is more restrictive and fans out similar work to different computers and collects the results back into the main machine. Batching is a one-way street, which makes it less general but also easier to develop, because it is simpler. There are also other ways of parallelization, but these two have gained traction in the last few years.

The two abstractions over parallelization treat each operation under the assumption that it is executed on a remote machine. The other possible assumption would be that each operation is run on the same machine, which would impose a subset of the restrictions of the assumption of remote execution. (Waldo et al., 1997) Therefore it would not be possible to run code which has been developed with the assumption it would be run on only one

machine on multiple machines in parallel.

1.2 Overview

In the second chapter, the important concepts are explained. The basic idea of functional programming (FP), referential transparency, and its implications. FP imposes shape on how programs can be developed and it helps reasoning how code will behave under different conditions. The same can be said about type safety, which also places restrictions in order to prevent undesired behavior. The parallelism paradigms, actors and map/reduce, build on FP to enable parallelism without shared memory, which leads to bugs because it tends to exhibit unexpected behavior. The data routing, bringing the correct input to the correct function, is closely coupled to how the data structure is built.

The related projects mentions other projects with similar goals and how and if they implement the concepts. It also explains the tools used in this project and why they were chosen.

The chapter on representations explains why the data representation is such an essential part of any NLP pipeline framework. It clarifies the different types of representation and why a specific type of annotation is the only one which satisfies all relevant concepts. The final part of this chapter is how exactly the data structure is implemented and where the complexities lie.

For a case study, the OpenNLP library (Baldridge, 2005) is implemented using the new library implemented in this project. The UIMA connectors of the OpenNLP library serve as a comparison to an existing industry standard. To serve as a measure, the OANC corpus (Fillmore et al., 1998) is run over a cluster of machines.

Chapter 2

Concepts

This section explains the concepts referenced in this thesis. The concepts are functional programming, which defines how you section code, type safety, which catches programmer mistakes, actors and mapreduce, which are concepts how to implement parallelism, and data routing, which describes how data is routed from annotator to annotator. Most of those concepts are commonly used, except for data routing, which is similar to data flow, but it involves a bit more.

2.1 Functional Programming

Functional programming is a specific style of programming, that is characterized by referential transparency. Referential transparency means that each subroutine can be replaced by its results. These subroutines are called functions. This concept is similar to locality, where a subroutine should only interact with a defined subset of the whole program. in functional programming, this subset is defined as only the arguments. Functional programming also prohibits mutation of the arguments, because changing the arguments would break referential transparency.

Functional programming has various advantages. By enforcing locality, debugging a project developed with functional style is easier because there are no unexpected dependencies between different functions. In parallel programming, the most problematic part are the dependencies outside the function. When two functions depend on shared data and they are not guaranteed to run sequentially, there will be conflicts, which will lead to

bugs. Functional programming restricts the shared state to immutable data. This limitation restricts the development of parallel programs to specific patterns. These patterns help to structure the code in such a way that it is easier to reason about side effects which could affect other threads when running in parallel. A function defined in functional style can be run in parallel an infinite amount of times without affecting other runs, which makes parallelizing trivial.

Another implication by the functional style is partial application, also known as currying. A function that uses multiple arguments to calculate its results can be passed only a single argument, which leads to a new function that requires one argument less. An example would be a function that takes a model and a document and computes the sentence annotations. The partial application of said function with only the model leaves us with a function that takes a document and returns annotations, which is basically an annotator.

2.2 Type Safety

Type safety is the concept to ensure each function is passed the arguments it expects, to make sure that e.g. not a string and a number are added together. In many languages, type safety can be circumvented by an operation called casting, which changes the type of a variable without changing its structure. Casting breaks type safety, because the compiler cannot ensure a variable is of the correct type.

Type safety exists to catch programmer mistakes, at the cost of required type annotation. A compiler may infer some of those type annotations, but not all of them. Some compilers even require full type annotation, without any inferring. The compiler then matches up all type annotations and see if they form a logical whole. If not, a compiler error is thrown.

Because type safety ensures the correct arguments, this project uses the concept to make sure each annotator receives the annotations it requires.

2.3 Actors and MapReduce

Actors and MapReduce are both coding conventions for parallel programming. The actor paradigm follows the original object-oriented concept from Smalltalk, where object may only communicate with other objects via messages. (Goldberg and Robson, 1983) And the elements of pipelines can also be regarded as objects. (Sinha et al., 2010)

To parallelize a pipeline, each object, called actor in this context, runs in its own thread. The MapReduce concept is based on functional programming, where each function can be assured to not affect any other function, because the functions ensure locality, which means that they may not depend on anything but their arguments. (Lin and Dyer, 2010)

There may be additional restrictions placed on the messages in the actor-based system. The Akka framework also restricts the messages to be immutable, because the messages may be resent or even multiplied.(Gupta, 2012) The immutability allows e.g. to resend messages which may have been lost. Without immutability, it cannot be guaranteed that a message has not arrived at the destination object before being resent and already been mutated, which would lead to undefined behavior.

The locality of MapReduce allows for functions to be called again in case their results are lost due to network failure, if the results are cached. Because functional programming involves copying over modification, the caching of intermediary data does not involve the expensive copying of information and can be done without additional processor time.

A pipeline is built of multiple analyzers, each its own unit of computation. Depending on the convention used, each analyzer is either an actor or a function. An actor would receive the document to be analyzed as well as any previous annotations as message and send the new annotations together with the old ones and the document either back to a master agent or forward it to the next agent in the pipeline. In the MapReduce style, the input is passed as arguments and the output is then passed to the next analyzer, like a transform of a list in the functional style.

In a pipeline, the single unit of computation is the actor or the function, depending on programming paradigm. A function can be easily expressed as an actor, while the other way round is more expensive computationally. The reason is that a function may not change state, but an actor can. So for each call, a new actor has to be created to ensure that no modifications happened to the actor.

2.4 Data Routing

Data Routing is the challenge to pass the correct annotations to the correct annotator. Some annotators only produce annotations, a notable example being the sentence detector which is only based on the input document. Other analyzers, such as a tokenizer, work per sentence, so they need access to the data which describes the sentence borders.

Data routing touches both data representation and parallelism. It requires the data representation to be able to distinguish between the different types of annotation. For parallelism, it enables to have more paths than just the linear pipeline. This option is not explored, because it is complex and can only make a difference in an actor-based parallelism setup.

Data routing should also enable using e.g. sentence annotations from an analyzer from one framework and the tokenizer from another framework and have them seamlessly work together. The two annotators often have different types for the sentence annotation, e.g. the sentence detector produces org.foo.Sentence, while the tokenizer expects org.bar.Sentence. Sadly, there is no direct way to connect the two annotators, so a conversion function which converts from org.foo.Sentence to org.bar.Sentence is required.

To reduce the conversion overhead, it is possible to define a common type, e.g. org.example.Sentence. Then the sentences from the sentence detector should be of type org.example.Sentence, with further specification of the type as org.foo.Sentence. The tokenizer, if the only information required are the sentence borders, can request org.example.Sentence, which defines the access to the sentence border information. When both libraries are aware of org.example.Sentence, it is possible to have a direct connection.

The UIMA project has a different route to circumvent the requirement for type reformatting. The types for e.g. sentences are passed to the annotators via type descriptor configuration files. Because the type system is runtime based, it is possible to inject a different type which operates the same without the requirement for a common interface.

2.5 Data Structure

The data structure should implement the concepts of functional programming, type safety and data routing. Functional programming is implemented by declaring the construct immutable and passing all information from one stage to the next. Type safety can be achieved by ensuring the compiler has knowledge of the types. This knowledge can be passed by using hlists, which will be explained at a later stage. The data routing is also part of how hlists can be accessed.

There are different approaches on how to store the annotation data, some of which involve yielding some of the concepts this project deems relevant for the more general implementation. The more specialized implementations have their own advantages, such

as a simpler format and/or higher speed.

To illustrate how annotations are represented in terms of data, a short pipeline consisting of a sentence detector, a tokenizer, a PoS tagger, and a parser will be explained on a small example document. The sentence detector annotates sentences, the tokenizer word boundaries, and the PoS tagger part of speech tags. The parser builds the parse tree of the sentence.

The first step is the sentence detection, which is trivial in this case, because the periods line up with the sentences. There are no other uses for periods in this paragraph, such as Mr. Smith.

Tom ate an apple. But not quite, because Jerry was faster.

There are different ways to indicate where the sentences start and end. The way presented here is by having a line break between sentences. This representation works well for large chunks of data, such as sentences. It is also very simple, and is a data format that is used by a lot of the basic UNIX tools. Nearly every programming language has a function or a construct to read a file line by line.

Tom ate an apple.
But not quite, because Jerry was faster.

For tokens, there is a lot more single entities of data. The tokenizer splits the sentences into tokens. This format is simple to parse, and it is reasonably readable for a human. The sentence boundary needs to be stored as well, and is represented as an empty line.

Tom
ate
an
apple
.

The PoS tagger needs two sets of information: the sentence boundaries and the tokens. This information is stored in the data structure with line breaks, and can be retrieved in a simple manner. The PoS tagger adds a tag to each token, by appending it to the line the token is in. The tag is separated by a space, some projects use a tab instead.

Tom N

ate V

an D

apple N

. S

The data format used in this example so far is simple to read and develop with. However, if we would like to represent the sentence tree structure constructed by the parser, a more sophisticated format is required. There are different ways to represent the sentence tree structure. One would be to continue applying the prior format and add the tree connections in a second paragraph, one per line. The sentences are also separated by line breaks. In the case where there is no parse tree, because the sentence is an injection, there are two line breaks after the sentence.

Tom N1

ate V1

an D1

apple N2

. S1

DP1 D1 N2

VP1 V1 DP1

CP1 N1 VP1

SN CP1 S1

For this specific problem, it is also possible to use a lisp-like list representation to serialize the data. The representation is possible because parse trees do not have intersecting branches, at least the ones used in most NLP.

(SN (CP (N Tom) (VP (V ate) (DP (D an) (N apple)))) (S .))

It is also possible to encode the same data with XML, which looks different because of the difference in data structure annotations (<e pos="N">x</e> vs. (N x)), but the logical structure is the same.

```
<e pos="SN">
  <e pos="CP">
```

```
    <e pos="N">Tom</e>
    <e pos="VP">
      <e pos="V">ate</e>
      <e pos="DP">
        <e pos="D">an</e>
        <e pos="N">apple</e>
      </e>
    </e>
  </e>
  <e pos="S">.</e>
</e>
```

Another element of the pipeline annotates named entities, such as proper names. To annotate the names found in the lisp-like structure, it is necessary to add another 'type' of lists to specifically say which lists are from the parse tree and which tree nodes are named entities. The addition of this distinction makes the list representation unfeasible to add the NE (named entity) data. The XML tags work well and it is simple to add a new tag family <ne></ne> to the XML document.

```
<e pos="SN">
  <e pos="CP">
    <ne>
      <e pos="N">Tom</e>
    </ne>
    <e pos="VP">
      <e pos="V">ate</e>
      <e pos="DP">
        <e pos="D">an</e>
        <e pos="N">apple</e>
      </e>
    </e>
  </e>
  <e pos="S">.</e>
</e>
```

However, as soon as two elements overlap partially, the XML structure is not feasible anymore. The following example is invalid XML.

<e>te<f>s</e>t</f>

All these data structures violate the concept of immutability. The original document is not retrievable anymore after the first annotator. To implement this concept, it is possible to save the annotations outside the actual text, with references to the annotation starts and ends. This way to encode the information is barely readable for humans, but easy to read for the machine. This method works in almost every case and is simple, which is why it is the strategy used in a lot of frameworks, and will also be adopted by this project. This format is called the standoff format.

Tom ate an apple.

N 0 3
V 4 6
D 8 9
N 11 15
S 16 16

There are various implementations of the standoff format. It is possible to have the annotations in a different file, and in any serialization format. The GrAF format (Ide and Suderman, 2007) stores the annotations in different files. It has the advantage of accessing each annotation on its own without the requirement of parsing all other annotations as well.

The format represented here is also possible with the annotations and the document in the same file. The format would get more complex however, because there needs to be an indicator of which annotations are sentences and which ones are part of speech tags. In a closed world annotation, which does only incorporate a given set of annotators, it is possible to skip the indicator and use positions to indicate of which type an annotation has.

Tom ate an apple.

sentences
0 16

PoS

N 0 3

V 4 6

D 8 9

N 11 15

S 16 16

The annotations in the previous example have a hierarchical structure too, which is lost in the pure standoff annotation. To store the tree structure, GrAF was developed. (Ide and Suderman, 2007) GrAF is a data structure based on a graph, where the nodes are the annotations and the edges are the hierarchical structure. This project focuses on the standoff annotation, although it is possible to use a graph with typed elements as well.

There is also an inline format for multiple annotations, which inserts the annotation information directly into the text. The C&C parser makes use of such a format to be extremely efficient. (Curran et al., 2007) However, it is extremely specialized, because it destroys the original data and adds the annotations in a non-general way. The tokenizer & stemmer adds the information separated by pipes.

<c> Tom|Tom ate|eat an|an apple.|apple.

Later stages, such as the pos-tagger, supertagger and parser add their corresponding annotations with additional pipes. The format is very efficient in terms of space, but it has a lot of assumptions built-in, such as that all PoS tags are at the same position like tokens. The assumption holds perfectly in the case of the C&C parser, but the assumption that each following annotator adds only information of the length of token spans cannot hold for all kinds of annotations. E.g. an NER annotator can produce an annotation that holds multiple tokens.

The newlines are inserted for readability and are not part of the original format.

<c> Tom|Tom|NNP|I-NP|I-PER|N
ate|eat|VBD|I-VP|O|(S[dcl]\NP)/NP
an|an|DT|I-NP|O|NP[nb]/N
apple.|apple.|NN|I-NP|O|N

To achieve the principle of immutability, which is represented by keeping the original document intact, the only way is to use standoff annotation. The line-wise annotation

destroys the original document. Even if we were to add the original document to the data structure, it would be difficult to reference the annotations back to the original documents. The inline annotation is similar to the line-wise annotation, except it uses spaces instead of linebreaks and pipes instead of spaces.

Chapter 3

Related Projects

3.1 UIMA

UIMA is the industry standard for NLP components on the Java Virtual Machine (JVM). There is a C++ implementation as well, but not as far-reaching as on the JVM. The basic unit of the UIMA parts is an Analysis Engine. Each analysis engine processes an incoming document with annotations and adds further annotations to the document. The annotations with the document are encapsulated in a CAS, a Common Analysis Structure (Ferrucci and Lally, 2004).

An analysis engine consists of three basic steps: initialization, processing and cleanup. The initialization consists of e.g. loading the training data into the analysis engine. The processing step loads the document from the CAS to be processed and creates annotations based on the training data. These annotations are then added to the CAS. The processing step can be repeated an infinite number of times. The cleanup step, if required, purges the analysis engine from the memory if it is no longer required.

And each of these parts requires the output from the previous one, so the CAS containing the annotations has to be passed from one annotator to the next. This structure of annotators is encapsulated by an aggregate analysis engine which contains multiple analysis engines which process the document serially.

The UIMA components can follow referential transparency, but they don't have to. There is a parameter in the annotator description which indicates whether an annotator modifies the CAS. The design decision to allow modification of the data structures was

made for speed reasons.

Type safety in UIMA does not come from the compiler; it is a custom implementation from UIMA because it was developed before Java had the type features it has today. The type system does not check the soundness of the pipeline at compile-time. UIMA DUCC does not output an error message when the pipeline misses the sentence detector, the first element in the pipeline.

UIMA offers various implementations of parallelism. Behemoth implements a map reduce style parallelism on top of Hadoop for UIMA. Behemoth also supports other annotators, such as GATE or solr.(Nioche, 2010) However, it is not part of the Apache foundation. UIMA DUCC is the official cluster management software and was constructed for the jeopardy bot Watson (Brown et al., 2013). It treats a full pipeline like an actor and sends single CAS to each worker, not each annotator like an actor.

The data routing is solved via the custom type system, which uses type strings. To request an annotation, the type string is passed to the corresponding function, which then returns the annotation list associated with that type string. The type string to request annotations is retrieved as a configuration parameter. The routing is effectively implemented with the configuration files.

The data structure stores the basic annotation with the span, which can be exported as an iterator to gain access to all the annotations. The annotations are also stored indexed. The index is by span by default, but can be exchanged with another index which uses a different indexing strategy if needed.

The UIMA type model is deemed not expressive enough to encode a full ontology. (Verspoor et al., 2009) The main issue of the authors is the inability to index based on the ontology types, which have complex inheritance patterns. The authors intended to store the ontology relations as types in the UIMA type system. This approach might be possible with Scala types.

3.2 Behemoth

Behemoth is one of the components which implements parallelism for UIMA. (Nioche, 2010) The main goals of behemoth are to combine different NLP framework components, so they are similar to this project, although they work directly with UIMA pears instead of custom code. They also incorporate adapters to GATE, solr, and UIMA. The basic structure of the

project is very similar to UIMA, with a custom type system.

The pipelines are encapsulated into a function to use in Hadoop (White, 2009). The functions in Hadoop have similar constraints to functions in functional programming, but the implementation is different from usual functions. The functions itself are pure enough, in terms of encapsulating the side effects of the pipelines.

The annotations are stored in a struct with begin, end, and type fields. The type field is represented as a string. The behemoth framework does not have any inner associated with the types, there is no type safety in any kind. It relies on the type safety inside the frameworks it uses.

The parallelization structure is map/reduce, as implemented by Hadoop.

The framework converts annotations from the various components into its own format. It does not convert from its own format back into the UIMA CAS format, so it is more targeted to mix and match pipelines rather than single annotators.

The data routing is not relevant in Behemoth, because it does not implement any transformation from its own format into annotators. The only exception here are the document filters, which operate on full documents and optionally remove a document from the collection to be processed.

3.3 DUCC

DUCC is the official parallelism implementation for UIMA. It is based on UIMA AS, which is a parallelism framework for UIMA. UIMA AS implements the multi-machine parallelization, and DUCC implements the cluster administration. DUCC itself does not implement any annotators, the only concept it touches are actors. (Brown et al., 2013)

To run DUCC, a collection reader which converts the documents to CAS is required. The CAS are then sent one-by-one to the workers, which is similar to an actor-based pattern with a message queue.

3.4 Spark

Spark (Zaharia et al., 2010) implements map/reduce with an API which allows to use basic functions and parallelize them, without adapting the functions to a specialized API. After implementing the desired transformations on a collection from the standard library, it is possible to reuse the same code on a spark collection. When reducing the collection, there

are some more specialized APIs, which are not present on the default collections. These specialized functions exist for performance reasons.

Spark splits the work into partitions for each transformation. One such transformation is the sentence detector, which creates new annotations added to the document. Each transformation of a partition is executed in a task. When there are multiple transformations, they are converted to a single transformation, to avoid creating new tasks.

This project uses Spark for parallelization. The API allows to use the functions which create annotations directly in Spark without the need to implement a specialized API. Spark also offers good performance.

3.5 Scala

Scala is a programming language which runs on the JVM. (Odersky et al., 2004) It is chosen because it is one of the programming languages supported by Spark and its expressiveness, which is very helpful for parallelized NLP applications. (Miller et al., 2011) Scala also supports typeclasses, which are used to ensure the type safety in the data structure. (Odersky, 2006)

Chapter 4

Representations for Natural Language Annotations

This project implements the presentation for NL annotation in its custom format called a Slab. Slabs store the annotations and provide the data routing between the annotators. The Slabs are the core of the project, because they enable developers to write pipelines without the requirement for additional code which directs annotations from annotator to annotator.

The final code implementing the data structures is available online via https://github.com/reactormonk/epic.

4.1 Implementation

The standoff single file format is similar to how annotations are logically stored in a programming language. The type of the annotations has to be stored as well as the data.

Storing a single collection of annotations, e.g. only PoS tags, is simple. Any programming language supports some kind of struct, which is a data structure that has named fields, and each field holds an element. So a PoS tag has a 'tag' field, which store the actual tag. For the standoff annotation discussed above, there is also a 'begin' and an 'end' field, which store the character offsets for each. All annotations are then stored in a container such as an array. The elements can be accessed by iterating over them.

To store a second type of annotation, such as sentences, a second type of container is required. With only a single container which has a special PoS tag 'sentence', it is required

to check at runtime which element is a sentence and which a PoS tag and act accordingly, but this approach is not the ideal path in this project, because it uses a static language, where the compiler can help a developer to know the type of an element before runtime to ensure no PoS tag is mistaken for a sentence. The main focus of a Slab is to ensure type safety, which means that each element has a determined type and cannot be mistaken for another element without the compiler messaging the error to the developer.

Also, if a library based on Slabs adds named entity annotations to the collection of annotations, the code might break because there are unexpected annotations which do not correspond to the known types. The defensive approach would be to ignore all unknown annotations, but it might be that an annotation is close enough to be recognized as one of the known types (e.g. if it has a tag field) but it still isn't the exact same and the misinterpretation may lead to errors.

Each container of annotations should be retrievable from the Slab via its type, and a call for 'sentence' should return all tags of type 'sentence'. There are two basic approaches to store the different annotation containers.

One approach is to store the type information of the containers, which can later be used to find the correct annotation container when the type information is passed to the Slab. Because of the association of type information to a collection of annotations, the collections can be stored in a map (also known as hash or dictionary). However, because a map assumes that all values are of the same type, the type of the returned collections has to be manually changed in order for the compiler to know which type the returned annotations are.

The other approach is to store the annotation collections inside a heterogeneous list, which does not have the assumption that all values are the same type. (Kiselyov et al., 2004) An hlist is a recursive data structure where the type of all elements is known to the compiler before runtime, at compile time. Utilizing the hlist, it is possible to store different types of annotations without manually informing the compiler which type a value has. The manual assignment is called 'casting' and is often considered bad programming style.

An hlist is built similar to a linked list, except at type level. Each element is either a cons, which consists of the current value (head) and the rest of the list (tail), or a nil, which is the empty list. An empty list is represented by changing the name from 'nil' to 'hnil' because of the naming convention used in hlists.

```
HNil
```

The first value of the list is represented with a cons and the empty list as the tail, signaling that this is the last element. The hcons is the cons of an hlist.

```
HCons(1, HNil)
```

Any number of elements may be prepended, similar to a linked list.

```
HCons("test", HCons(1, HNil))
```

The type of the empty list is 'HNil'. The type is built recursively, so the list 'HCons(1, HNil)' has the type 'HCons[Int, HNil]'. The type of 'HCons("test", HCons(1, HNil))' is 'HCons[String, HCons[Int, HNil]]'.

The values in an hlist can be accessed with recursive functions, similar to the functions used in linked lists. However, the function is constructed at compile time to match the specific hlist type. E.g. the function to select a value of a specific type has two subfunctions, one that selects the value and the other that recurses further down the hlist. The first function matches 'HCons[Value, Any]', the other 'HCons[Any, HCons[Any, Any]]', where 'Any' describes a value of any type. Note that there is no matching function for the empty list – the only function that can end the recursion is the one that finds the value. So if the compiler cannot construct the stack of functions because the function which ends the recursion cannot be found, because there is no such value in the hlist, the compilation fails as expected.

4.2 Scala Implementation

The Scala implementation used in this project is mostly written by Miles Sabin in the shapeless project. The full list of contributors is available via https://github.com/milessabin/shapeless/graphs/contributors. The implementation follows the basic structure outlined in the previous subsection. There are a few implementation details due to limitations of the Scala compiler.

The operations on hlists use implicit typeclasses to construct the recursive functions needed for the traversal of the hlist. Scala supports typeclasses via implicits. (Odersky, 2006; Oliveira et al., 2010) For the pattern matching on the hlist, implicit functions are used. As an example, the Selector operator will be discussed, which selects a single element

from the hlist based in its type. The code is slightly different from the code in the shapeless library, but it still works as expected.

```
trait Selector[L <: HList, U] {
  def apply(t: L): U
}

object Selector {
  implicit def select[H, T <: HList]: Selector[H :: T, H] =
    new Selector[H :: T, H] {
      def apply(l : H :: T) = l.head
    }

  implicit def recurse[H, T <: HList, U]
    (implicit st : Selector[T, U]): Selector[H :: T, U] =
      new Selector[H :: T, U] {
        def apply(l : H :: T) = st(l.tail)
      }
}

object ImplicitOps {
  implicit class Ops[L <: HList](l : L) {
    def sel[U](implicit selector : Selector[L, U]) : U = selector(l)
  }
}
```

The trait that defines a selector which takes an HList or an HCons and return an element of type U.

```
trait Selector[L <: HList, U] {
  def apply(t: L): U
}
```

This function which matches in the case where the head is of the requested type. There are two types, the head, H, and the tail of the hlist, called T. In case the head of

the hlist passed to Selector is equivalent to the requested element, which is in the second position, this function matches. The function itself returns the element of the requested type.

```
object Selector {
  implicit def select[H, T <: HList]: Selector[H :: T, H] =
    new Selector[H :: T, H] {
      def apply(l : H :: T) = l.head
    }
```

In case the type of the head of the hlist is not equivalent to the type of the requested element, this function matches instead of the one described just above. It looks for the next Selector function, which may recurse or find the element. It then passes the element up the call stack.

```
implicit def recurse[H, T <: HList, U]
  (implicit st : Selector[T, U]): Selector[H :: T, U] =
    new Selector[H :: T, U] {
      def apply(l : H :: T) = st(l.tail)
    }
```

What is missing is equally important. There is no implicit with type signature Selector[HNil, U], which indicates the compiler will throw an error because it cannot construct a recursive function that matches the current hlist. The only function that ends the recursion is the select, which needs the head to be of equivalent type to the selected element.

```
}
```

The method is attached to HLists using an implicit class. The class is used by the compiler to lookup further methods that may be called on an hlist. The type is passed to the select statement.

```
object ImplicitOps {
  implicit class Ops[L <: HList](l : L) {
    def sel[U](implicit selector : Selector[L, U]) : U = selector(l)
```

```
  }
}
```

The implicits have to be imported to apply to the current scope.

```
import ImplicitOps._
("foo" :: 1 :: HNil).sel[Int] // => 1
```

Without passing an argument, the first element in the hlist is selected. When the type argument is unspecified, it is simply Any, so it may match any type.

```
("foo" :: 1 :: HNil).sel // => "foo"
```

The two functions ‘head‘ and ‘tail‘ are defined in shapeless in a typeclass called ‘IsHCons‘. This typeclass provides both the head and the tail function. The types of head and tail are described as ‘H‘ and ‘T‘ and are not yet linked to the type of the list itself. The type of the tail is bound to be an HList.

```
trait IsHCons[L <: HList] {
  type H
  type T <: HList

  def head(l : L) : H
  def tail(l : L) : T
}
```

To construct a relation between ‘L’, ‘H’, and ‘T’, an ‘Aux’ type is required. The ‘Aux’ type alias makes the types ‘H’ and ‘T’ available to the generics system. The apply method connects ‘Aux’ and the original ‘IsHCons’. To finally connect ‘H’ and ‘T’ with ‘L’, the implicit method defines ‘IsHCons[H0, T0]’ to be ‘Aux[H0 :: T0, H0, T0]’ via return type. The ‘Aux’ defines the second and third argument to be the type of the head and the tail.

```
object IsHCons {
  type Aux[L <: HList, H0, T0 <: HList] =
    IsHCons[L] { type H = H0; type T = T0 }
  def apply[L <: HList](implicit isHCons: IsHCons[L]):
```

```
    Aux[L, isHCons.H, isHCons.T] = isHCons
  implicit def hlistIsHCons[H0, T0 <: HList]: Aux[H0 :: T0, H0, T0] =
    new IsHCons[H0 :: T0] {
      type H = H0
      type T = T0

      def head(l : H0 :: T0) : H = l.head
      def tail(l : H0 :: T0) : T = l.tail
    }
}
```

The 'Aux' pattern is a helpful shorthand for all operations where the compiler has to infer the return type while passing another type as argument. An example for such an operation is 'Remove', where the first element of the requested type is removed and the rest of the hlist is returned. To have the type available in the generics system, it is necessary to specify the Aux version, because otherwise the return type is not available for further processing. An hlist without type information can not have any operations invoked on it, which makes it unusable.

The API is not complete yet, as it does not define graph-based annotations. Because the annotations can be arbitrary Scala objects, it is already possible to store graph-based data. The API would define a fast access to the graph-based data. However, the serializer used with Spark runs into performance issues because there are too many references within the annotations. This problem is known, and an patch has been proposed to solve it in the past. [1] The standoff annotations themselves could be serialized into JSON (JavaScript Object Notation), although this functionality is not implemented yet.

The Slab API is not as written into stone as e.g. UIMA analyzers. There is no common interface which all analyzers implement, the only requirement is that it has the Slab to be processed as the only argument and the Slab as output. The reasoning behind the decision to not define an API is a technical one. The interface would also have to include the typeclasses, which are different depending on the analyzer. It still needs to be researched if it is possible to create an interface which includes all possible cases.

[1] https://github.com/EsotericSoftware/kryo/issues/103

4.3 The Slab

The main logic of storing the different annotations in the Slab is delegated to hlists, so the Slab itself is nothing more but about 20 lines of code which define the methods to create Slabs, store new annotations and retrieve them.

There is a custom operation for adding more annotations to the hlist, because of the requirement where more elements of an already existing annotation type may be added to the Slab. The method is defined within the typeclass 'Adder' and recurses through the hlist until it either finds the element of the same type or reaches the end of the list. Then either the elements are appended to the existing element or to the hlist. While leaving the functions, the hlist is reconstructed because each recursion returns the current head and the tail returned from the function one call deeper.

Because of limitations of the scala compiler, the decision was made to only allow 'List's to be used to store annotation information. A data type such as the 'Vector', which guarantees O(1) random access is not required because the elements of the list are not guaranteed to be in a certain order.

The Slab itself does not hold any information on how to access elements efficiently with an operation such as "all elements covered by the span X to Y". This functionality is implemented using external indexes which can create fast access paths to the elements. The indexes can currently not be passed along with Slabs and have to be generated for each annotator.

4.4 Indexing

A common pattern in an annotator is to iterate over all sentences and the corresponding word boundaries, also known as tokens. In a tree structure which stores the tokens as leafs from the sentence accessing each sentence with the corresponding tokens would be similar to a tree traversal. This approach is not possible, because the Slab uses flat standoff annotation to stay general. In the specific case of sentences and tokens, there is the implicit constraint that tokens do not overlap and that no tokens cut across sentence boundaries. To solve the problem on a more general basis for any two kind of annotations, those constraints are dropped.

The simplest possible approach to iterate over sentences and tokens is to iterate over all possible tokens and select the ones which are within the sentence boundaries. This

approach is inefficient. With 100 sentences and 10 tokens each sentence, $100 * 100 * 10$ comparisons would be needed, because each sentence scans the whole list of tokens, which is $100 * 10$. So the complexity is $O(s^2t)$, where s is the number of sentences and t the average number of tokens per sentence. Any solution to improve the speed should have lower complexity.

One possible solution to this problem would be to copy code from UIMA, namely the AnnotationIndex, which solves exactly this problem. However, the code has too many references to the UIMA structures, which makes it hard to reuse.

The Scala standard library offers a structure called SortedMap, which is perfect for our means, because it offers methods to access elements from a specific point. The specific point is given by the starting index of the sentence. Then all elements that start within the sentence are accessed and those that are within the boundary set by the end of the sentence are filtered out. The concept of "sentence" can be replaced with a more general concept of Span, which indicates a span over characters from character offset to character offset. The concept of "token" can be replaced by any type of annotation which is defined over a span. This approach does however not support indexing different types of annotations at the same time.

UIMA also has specific type priorities, for when an index returns more than one type. (Ide and Suderman, 2009) The SortedMap uses an Ordering, which can only be defined on a single type. However, it is possible to define a new type and converters from all annotation types to that single type. The converters add another field which can then be used to sort the collection. Naturally, this strategy comes at a loss of the type information of the single annotations, and this information may be relevant.

Chapter 5

Case Study: OpenNLP

This case study implements a bridge for the OpenNLP library (Baldridge, 2005) and Slabs to illustrate the differences in the approach between UIMA and Slabs. OpenNLP is chosen because it also serves as an example pipeline in UIMA. (Wilcock, 2009) The maintenance of the implementation has been transferred to the OpenNLP project in the past, but was originally built by UIMA developers, so it should serve as a good reference implementation of an UIMA bridge.

For parallelization, the UIMA community provides DUCC (Distributed UIMA Cluster Computing), which is a cluster management software. (Brown et al., 2013) It has more features than just scaling out, such as user management, whereas the tools for Slabs are only specific to the scale out. DUCC also does runtime cost analysis, where it prevents spawning more workers if creating a new one is too expensive compared to simply reusing the existing pipelines.

The final code implementing the OpenNLP pipeline is available online via https://github.com/reactormonk/epic-opennlp.

5.1 Reading the Documents

The first step for analyzing a document with OpenNLP is reading the documents. The stock code from DUCC reads all documents from a single directory on the node which submits the job. This part is called a CollectionReader in UIMA, as it reads a collection and creates an iterator which produces CAS. Each time a worker is free to process a CAS, the next one

is ordered from the iterator, serialized, and shipped to the worker.

With Slabs and Spark, an RDD is created from the documents, which is a distributed collection. The distributing is done via file names, so the files have to be available to all machines. This can be done either via manually copying the documents to all nodes or having them in a generally accessible storage. The second solution is more practical, because it does not go up in time required when the cluster size increases.

5.2 The Annotators

The annotators encapsulate the code generating the actual annotations. They convert from the data structures in the analysis pipeline to the types required by the backend code. They bridge between the two interfaces, and are a good example how the annotator interface works.

The OpenNLP library isn't completely threadsafe. Each parser object in itself is threadsafe, but the calls to that object are not. So a new object needs to be generated for each document, because it cannot be reused without additional pooling logic. This pattern is common when encapsulating non-functional libraries into functions, but it comes at a performance cost. The optimization to reuse objects is possible, although it comes at the cost of abstraction. The Spark API offers access to each partition of documents, so it would be possible to create the object only once per partition, without pooling logic. The cost of this solution is that the code will be specific to Spark and does not translate to any new library which emerges in the functional world.

5.2.1 Loading the Models

The OpenNLP library is based on machine-learning, so it requires its models to operate. The models have to be provided to the library at startup. The models are available via a resource from the JAR. The JVM offers the functionality to store arbitrary binary data within a package and exposes them as a resource. The UIMA framework provides functionality to encode the path to the models in the annotator description.

```
<analysisEngineDescription>
  <externalResourceDependencies>
    <externalResourceDependency>
      <key>opennlp.uima.ModelName</key>
```

```
      <interfaceName>opennlp.uima.sentdetect.SentenceModelResource</interfaceName>
    </externalResourceDependency>
  </externalResourceDependencies>
</analysisEngineDescription>

  <resourceManagerConfiguration>
    <externalResources>
      <externalResource>
        <name>SentenceModel</name>
        <fileResourceSpecifier>
          <fileUrl>file:en-sent.bin</fileUrl>
        </fileResourceSpecifier>
<implementationName>opennlp.uima.sentdetect.SentenceModelResourceImpl</implementationName>
      </externalResource>
    </externalResources>
    <externalResourceBindings>
      <externalResourceBinding>
        <key>SentenceDetector/opennlp.uima.ModelName</key>
        <resourceName>SentenceModel</resourceName>
      </externalResourceBinding>
    </externalResourceBindings>
  </resourceManagerConfiguration>
```

The model is then retrieved in the initialization of the annotator.

```
public void initialize(UimaContext context) {
  SentenceModel model;
  SentenceModelResource modelResource = (SentenceModelResource) context
      .getResourceObject(UimaUtil.MODEL_PARAMETER);
  model = modelResource.getModel();
  sentenceDetector = new SentenceDetectorME(model);
}
```

With Slabs, the model is passed to the constructor which creates the annotator. The process can also be viewed as partial function application of the annotator which has

the signature Model -> SlabIn -> SlabOut. The pipeline expects functions of the signature SlabIn -> SlabOut. The model is applied to the annotator to receive a function of the signature SlabIn -> SlabOut, which fits in a pipeline.

```
object SentenceDetector {
  def apply(model: SentenceModel): SentenceDetector =
    new SentenceDetector(model)
}

object Pipeline {
  val sentence = SentenceDetector(
    new SentenceModel(getClass.getResource("/epic/opennlp/en-sent.bin")))
}
```

5.2.2 Types

Types are essential in the pipeline, because they describe how the data can be accessed. The two systems are very similar in how they construct their types, although the actual representation is very different. UIMA represents their types in XML format, while Slabs are Scala code.

Both type annotations are derived from a type that describes annotations over spans of input. In the UIMA case, that type is uima.tcas.Annotation, in the Slab case, it is Sentence. The Sentence type is more specialized, because it is already defined in the epic library [1]. The ProbabilityAnnotation exists because other parts of the OpenNLP library also provide the probability of annotations.

```
<analysisEngineDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <analysisEngineMetaData>
    <typeSystemDescription>
      <types>
        <typeDescription>
          <name>opennlp.uima.Sentence</name>
          <supertypeName>uima.tcas.Annotation</supertypeName>
        </typeDescription>
```

[1] https://github.com/dlwh/epic

```
    </types>
  </typeSystemDescription>
 </analysisEngineMetaData>
</analysisEngineDescription>

trait ProbabilityAnnotation { def probability: Double }
class PSentence(override val span: Span, val probability: Double)
 extends Sentence(span) with ProbabilityAnnotation
```

With the UIMA annotation, it is possible to define the type parameters, so the sentence type can be swapped. There also exists a parameter to encode which languages are supported by the analyzer. The XML parameters only exist to swap out the sentence type. There is no annotation which type the annotator actually produces.

```
<analysisEngineDescription xmlns=”http://uima.apache.org/resourceSpecifier”>
  <analysisEngineMetaData>
    <configurationParameters>
      <configurationParameter>
        <name>opennlp.uima.SentenceType</name>
        <type>String</type>
        <multiValued>false</multiValued>
        <mandatory>true</mandatory>
      </configurationParameter>
      <configurationParameter>
        <name>opennlp.uima.ContainerType</name>
        <type>String</type>
        <multiValued>false</multiValued>
        <mandatory>false</mandatory>
      </configurationParameter>
    </configurationParameters>

    <configurationParameterSettings>
      <nameValuePair>
        <name>opennlp.uima.SentenceType</name>
```

```
            <value>
              <string>opennlp.uima.Sentence</string>
            </value>
          </nameValuePair>
        </configurationParameterSettings>
      <capabilities>
        <capability>
          <inputs />
          <outputs />
          <languagesSupported>
            <language>en</language>
          </languagesSupported>
        </capability>
      </capabilities>
    </analysisEngineMetaData>
</analysisEngineDescription>
```

5.2.3 Storing Annotations

In UIMA, the annotations are stored into the CAS with a createAnnotation method, which takes the type as a string argument. The annotation is then added to the index, which provides access to the annotation.

```
public void process(CAS cas) throws AnalysisEngineProcessException {
    for (int i = 0; i < sentPositions.length; i++) {
      sentences[i] = cas.createAnnotation(sentenceType,
          startPos, endPos));
      cas.getIndexRepository().addFS(sentences[i]);
    }
  }
```

With Slabs, there is a trait which already provides storing and retrieving. So it is possible to reuse this trait and define the annotator as a function which takes the document as string and returns the list of sentence annotations.

```
class SentenceDetector(val model: SentenceModel)
  extends AnalysisFunction01[String, PSentence] {
  def apply(document: String): List[PSentence] = {
    // Implementation
  }
}
```

The trait connects the function to the Slab. It does not retrieve any data other than the content of the Slab, but it adds annotations. To add annotations, an Adder typeclass is required. The typeclass is retrieved via an implicit, which is constructed by the compiler. The Adder typeclass is passed to Slab.add, which combines the Slab and the new annotations to a new Slab and returns it. Because of the required implicit, it is to possible to implement the Function interface provided by the Scala language.

```
trait AnalysisFunction01[C, O] {
  def apply[In <: HList, Out <: HList](slab: Slab[C, In])
    (implicit adder: Adder.Aux[In, O, List[O], Out]): Slab[C, Out] = {
    slab.add(apply(slab.content))(adder)
  }
  def apply(content: C): List[O]
}
```

5.2.4 Retrieving Data

The annotation data needs to be retrieved in the next stage, the tokenizer. The tokenizer requires the sentence annotations, because the algorithm operates on a per sentence basis. With the UIMA framework, the sentences can be accessed via a call to the annotation index. For the actual sentence body, another call is available.

```
FSIndex<AnnotationFS> sentences = cas.getAnnotationIndex(sentenceType);
for (AnnotationFS sentence : sentences) {
  String sentence = sentence.getCoveredText()
}
```

With Slabs, an instance of the Selector typeclass is required to retrieve the annotation data. The annotation data does not have any order and is not indexed yet.

However, in this use case that is not relevant, because an iteration over all sentences is all that is required to generate the token annotations.

```
trait AnalysisFunction10[C, I, O] {
  def apply[In <: HList](slab: Slab[C, In])
    (implicit sel: Selector[In, List[I]],
      adder: Adder.Aux[In, O, List[O], Out]
    ): Slab[C, Out] = {
    // Implementation here
  }
}
```

The sentence body can be accessed by passing using the sentence annotation on the full document.

```
val sentence = document.substring(sentence.begin, sentence.end)
```

5.2.5 Intersecting Annotations

The next element in the pipeline, a PoS tagger, requires token lists, one list per sentence. So tokens and sentences are required, and it should be possible to iterate over sentences and the corresponding token annotations.

UIMA provides an AnnotationComboIterator, which provides two nested iterators, where the outer iterates over the sentences and the inner over the corresponding tokens.

```
for (AnnotationIteratorPair annotationIteratorPair : comboIterator) {
  for (AnnotationFS tokenAnnotation : annotationIteratorPair.getSubIterator()) {
      // Implementation
  }
}
```

With Slabs, an index on the tokens has to be generated, then the sentence can be used to retrieve the corresponding tokens.

```
val index = SpanIndex(tokens)
sentences.map({ sentence =>
  val tokens = index(sentence.span)
})
```

5.3 Connecting Annotators

The last step is to connect the annotators to form a pipeline. In UIMA, this part is solved by XML configuration.

```
<flowConstraints>
  <fixedFlow>
    <node>SentenceDetector</node>
    <node>Tokenizer</node>
    <node>PosTagger</node>
    <node>Chunker</node>
    <node>Parser</node>
  </fixedFlow>
</flowConstraints>
```

With Slabs, the annotators are joined together using collection transformations. The same collection transformations can be used inside Spark.

```
documents
  .map(sentence.slabFrom(_))
  .map(tokenizer(_))
  .map(pos(_))
  .map(chunker(_))
  .map(parser(_))
```

The type safety ensures that each annotator has the correct inputs. The following pipeline is invalid, because the tokenizer is after the PoS tagger instead of in front.

```
documents
  .map(sentence.slabFrom(_))
  .map(pos(_))
  .map(tokenizer(_))
```

The compiler can not find a way to extract the sentences and the tokens, because there are no tokens available for the tagger.

```
could not find implicit value for parameter sel:
```

SelectMany.Aux[List[PSentence] :: HNil],
List[Sentence] :: List[Token] :: HNil,
List[Sentence] :: List[Token] :: HNil]

5.4 Performance

Performance is as important as a good API when processing large datasets. Because the NLP approaches grow to be more and more data-dependent, it is not uncommon to process corpus with one billion words for a project. To complete an analysis in a reasonable time frame, the code needs to be fast. The speed also depends on the parallelism framework it is built on, which can help increase the performance by an order of magnitude.

To measure the performance between the UIMA DUCC implementation and the OpenNLP implementation of this project based on Spark, 2000 documents from the OANC corpus (Fillmore et al., 1998) were used. The cluster consists of 20 machines from Amazon EC2, specification r3.xlarge. Each node has 4 cores of 2.5 GHz and about 27 GB RAM available for computation. The Slabs do not implement proper serialization into files yet, which accounts for about 5% of the overall run time.

Spark partitions the data into sets of data, where DUCC distributes the documents one by one. The partitioning gives a raw, but blunt speed advantage. As soon as one partition is done, there is no way to redistribute the documents still to be analyzed. The detection how many partitions are needed does not perform well, so this number is manually passed and is equivalent to the number of cores. DUCC creates statistics and does not create new processes if creating new ones would not lead to an increase of speed. Also, an additional worker is started on the master to compensate for DUCC, which also has a worker on the master.

The experiment is run on a varying cluster size. The relationship between more machines and processing time is not linear and gives a good indication of the overhead of distributing work over multiple machines. The amount of documents is also varied to see how well the two clusters deal with a different amount of documents. The full amount of documents only includes the two written parts of OANC, without the spoken one. The full collection is about 6.5k documents.

Spark processes the same documents in about half the speed of DUCC. DUCC does not use the full cluster with 15 machines, only about 10 of them run tasks. Following that,

the scaling between 10 and 15 isn't as great as with spark. From the mailing list, DUCC is designed for a small cluster with a few high-end machines rather than a lot of cheaper ones.

On 15 machines, DUCC finishes in 2359 seconds. The web interface lists only half of the machines as used. However, when increasing the amount of shares used by upping the thread limit, the job does not complete anymore. So the actual performance could be interpolated to that of a 7-8 cluster system. This would still be slower compared to spark on 5 machines.

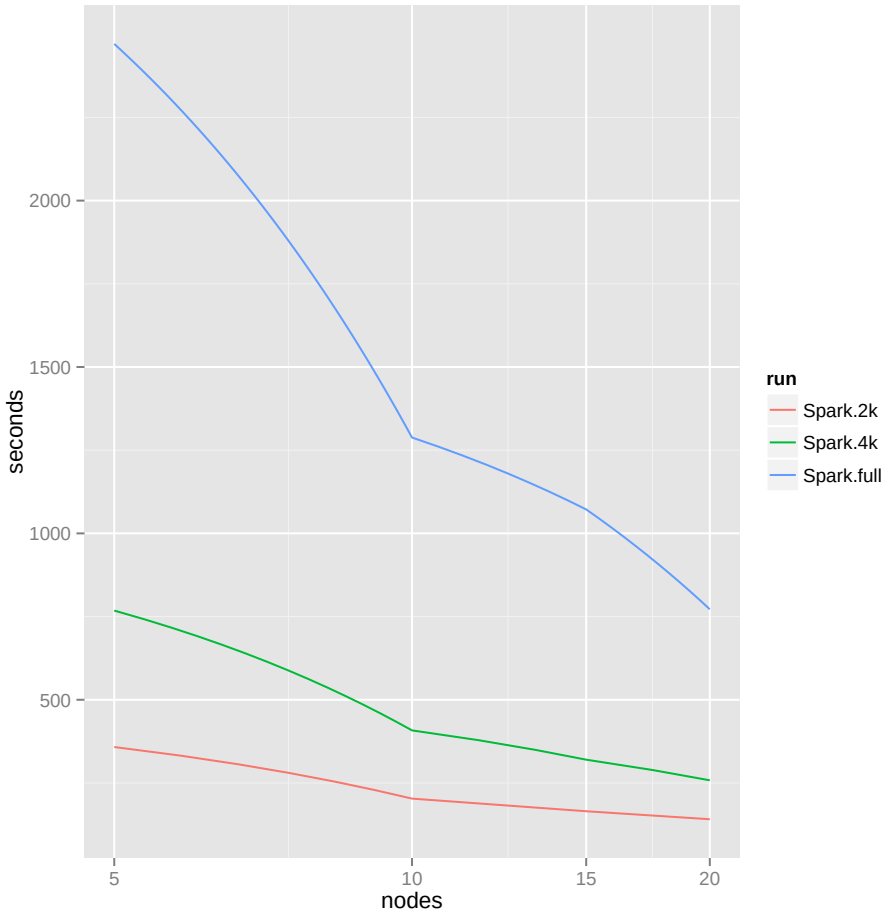

Figure 5.1: Time to process subsets of the OANC.

With parallelization by partition, the disadvantage of a set amount of tasks is that there is variance in the time to finish a partition. The less partitions, the more variance,

which leads to longer overall time because the program has to wait for the last one to finish.

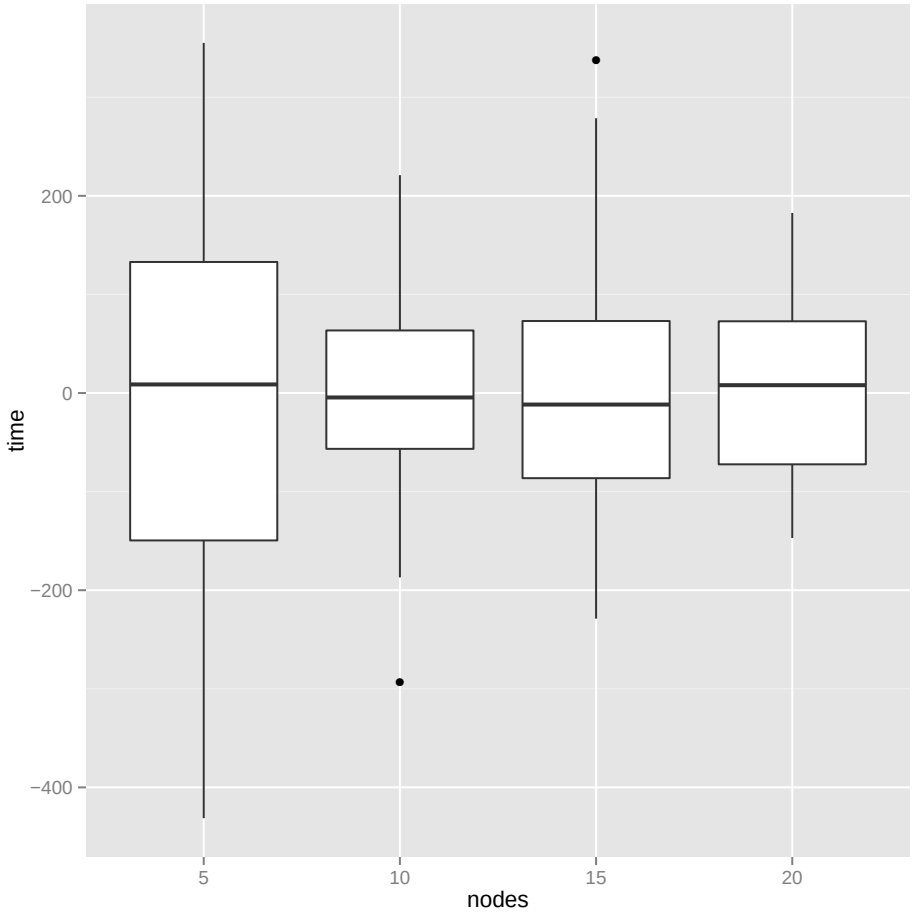

Figure 5.2: Variance in Spark partitions duration.

Chapter 6

Conclusion

This project focused on creating a new data representation for natural language annotations in documents, named Slabs. This data structure is important to connect NLP processing pipelines, because it defines a common interface for all components.

Functional concepts were used, which makes it possible to use existing libraries for parallelization. Slabs offer an expressive API without the need for XML configuration. The API allows arbitrary Scala constructs, which enables other developers to use any objects as annotation without additional restrictions.

The Slabs are built on the concepts of functional programming and type safety. Functional programming makes it easier to reuse components to build new pipelines from existing components.

Because the Slab API implements the principles of functional programming with NLP pipelines, it allows the usage of tools which rely on functional principles to parallelize code. Although functional programming may be slower than a solution based on imperative code, the abstraction allows the developer to reuse the components in a parallelized setting without any additional work.

In conclusion this project will help developers to construct pipelines form existing components more easily. Future work will focus on implementing non-binary serialization and graph annotations.

The final code is available online via https://github.com/reactormonk/epic, the OpenNLP pipeline via https://github.com/reactormonk/epic-opennlp.

Chapter 7

References

Baldridge, J. (2005). The opennlp project. URL: http://opennlp. apache. org/index. html,(accessed 2 February 2012).

Bird, S. and Liberman, M. (2001). A formal framework for linguistic annotation. Speech communication, 33(1):23–60.

Brown, E., Epstein, E., Murdock, J. W., and Fin, T.-H. (2013). Tools and methods for building watson.

Curran, J., Clark, S., and Bos, J. (2007). Linguistically motivated large-scale NLP with c&c and boxer. In Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions, pages 33–36, Prague, Czech Republic. Association for Computational Linguistics.

Ferrucci, D. and Lally, A. (2004). UIMA: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering, 10(3-4):327–348.

Fillmore, C., Ide, N., Jurafsky, D., and Macleod, C. (1998). An american national corpus: A proposal. In Proc. First International Conference on Language Resources and Evaluation (LREC 1998), pages 965–970.

Goldberg, A. and Robson, D. (1983). Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc.

Gupta, M. (2012). Akka Essentials. Packt Publishing Ltd.

Ide, N. and Suderman, K. (2007). GrAF: A graph-based format for linguistic annotations. In Proceedings of the Linguistic Annotation Workshop, pages 1–8. Association for Computational Linguistics.

Ide, N. and Suderman, K. (2009). Bridging the gaps: interoperability for GrAF, GATE, and UIMA. In Proceedings of the Third Linguistic Annotation Workshop, pages 27–34. Association for Computational Linguistics.

Kiselyov, O., Lämmel, R., and Schupke, K. (2004). Strongly typed heterogeneous collections. In Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, pages 96–107. ACM.

Lin, J. and Dyer, C. (2010). Data-intensive text processing with MapReduce. Synthesis Lectures on Human Language Technologies, 3(1):1–177.

Miller, H., Haller, P., and Odersky, M. (2011). Tools and frameworks for big learning in scala: Leveraging the language for high productivity and performance. In NIPS 2011 Workshop on Parallel and Large-Scale Machine Learning (BigLearn).

Nioche, J. (2010). Large scale document processing with hadoop.

Odersky, M. (2006). Poor man's type classes. In Presentation at the meeting of IFIP WG, volume 2.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the scala programming language. Technical report.

Oliveira, B. C., Moors, A., and Odersky, M. (2010). Type classes as objects and implicits. In ACM Sigplan Notices, volume 45, pages 341–360. ACM.

Sinha, A., Paradkar, A., Takeuchi, H., and Nakamura, T. (2010). Extending automated analysis of natural language use cases to other languages. pages 364–369. IEEE.

Verspoor, K., Baumgartner Jr, W., Roeder, C., and Hunter, L. (2009). Abstracting the types away from a UIMA type system. From Form to Meaning: Processing Texts Automatically. C. Chiarcos, Eckhart de Castilho, Stede, M, pages 249–256.

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). A note on distributed computing. Springer.

White, T. (2009). Hadoop: the definitive guide: the definitive guide. " O'Reilly Media, Inc.".

Wilcock, G. (2009). Text annotation with OpenNLP and uima.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, pages 10–10.