João Manuel Pinto Simões

Licenciado em Engenharia Informática

Estudo de implementações eficientes em correlações estatísticas de expressões relevantes em documentos de linguagem natural

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Joaquim Francisco Ferreira da Silva, Professor Auxiliar,
FCT, Universidade Nova de Lisboa

Co-orientador: Vítor Manuel Alves Duarte, Professor Auxiliar,
FCT, Universidade Nova de Lisboa

Júri

Presidente: Ana Maria Diniz Moreira
Arguente: Carlos Jorge de Sousa Gonçalves
Vogal: Joaquim Francisco Ferreira da Silva

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2019

Estudo de implementações eficientes em correlações estatísticas de expressões relevantes em documentos de linguagem natural

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Agradecimentos

A realização desta Dissertação só foi possível graças ao contributo, de forma direta ou indereta, de várias pessoas e instituição, às quais gostaria de dedicar algumas palavras de agradecimento.

À **Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa** em nome do *Prof. Doutor Virgílio Machado*, Diretor da FCT.

Ao **Departamento de Informática da FCT UNL**, por me ter consigo proporcionar um excelente espaço de trabalho, mais propriamente, à sala **default** e à **neo-default**.

Ao **Professor Doutor Joaquim Silva** e **Professor Doutor Vítor Duarte**, o meu orientador de dissertação e co-orientador, pelo tempo que dedicaram a transmitir-me os vossos conhecimentos, teóricos e práticos, por todo o apoio e, essencialmente, por toda a paciência que tiveram comigo.

A todos **os meus colegas e colaboradores do DI**, por terem sido muito hospitaleiros desde o primeiro dia, por estarem sempre disponíveis a ajudar e por terem contribuído para que esta etapa fosse tão enriquecedora.

À **Dona Maria**, do bar do DI, que me aturou nas fases boas e menos boas no decorrer deste trabalho e que tinha sempre uma palavra amiga para me consolar.

Aos meus **amigos**, pela amizade e pela força que me deram durante este trabalho. Quero agradecer especialmente aos companheiros de luta que me acompanharam durante esta última fase académica, ao **Miguel (Sixty)**, **Oliveira**, **Boto** e **Mário** que estiveram sempre comigo a apoiar-me.

À **Inês da Silva**, por estar sempre presente e por ser quem é.

À minha **família**, em especial ao meu **pai**, à minha **mãe**, ao meu **irmão** e à minha **avó Brigída**, por todo o apoio incondicional que sempre me deram, toda a força, por nunca me deixarem desistir, por terem sempre uma palavra amiga, por estarem sempre presentes e por acreditarem sempre em mim. Muito obrigado por tudo, sem vocês não conseguia ter chegado até aqui.

Mais uma vez, muito obrigado a todos.

"If we knew what it was we were doing, it would not be called research, would it?" Albert Einstein

Resumo

Aferiu-se que 90% dos dados que existem na Internet foram criados nos últimos dois anos. Tendo em vista este crescimento de dados, o número de padrões/relações neles contida é também muito grande. Com o objetivo de obter meta-dados que descrevam fenómenos linguísticos, na linguagem natural, reúnem-se conjuntos de documentos (*corpus* linguístico), a fim de obter robustez estatística. Num *corpus* existem vários *n*-gramas que podem, ou não, estar fortemente ligados entre si. Os *n*-gramas mais informativos têm a propriedade de refletir fortemente o conteúdo "core" dos documentos onde ocorrem. Formam por isso, expressões relevantes (*multi-word expression*). Uma vez que as ERs são extraíveis diretamente do *corpus*, é possível medir quão semanticamente próximas estão umas das outras. Tomando como exemplo as ERs, "crise financeira" e "desemprego na Zona Euro", é de esperar que exista uma proximidade semântica forte entre elas. Esta proximidade pode ser calculada através de métricas de correlação estatística. Também, o conteúdo "core" dum documento pode estar semanticamente ligado a um conjunto de ERs mesmo que estas não estejam presentes no documento; por exemplo, num documento de texto curto que trate de questões relativas ao ambiente e contenha a ER "global warming" mas não contenha a ER "Ice melting", à qual está semanticamente próxima, como facilmente se compreende. Seria útil que em ambiente de pesquisa, um motor de busca pudesse recuperar este documento após a pesquisa sobre "Ice melting", mesmo que o documento não contivesse explicitamente esta ER. De modo a conseguir a construção automática de tais descritores de documentos, é necessário dispor da capacidade de cálculo da correlação entre pares de ERs. Considerando que o número de pares cresce com o quadrado do número de ERs dos *corpora*, este processamento requer um ambiente paralelo e distribuído sendo, Hadoop e Spark abordagens a ter em conta. O desafio desta dissertação inclui a implementação dum protótipo que consiga de forma automática, em tempo útil, construir descritores de documentos a partir de *corpora* linguísticos. Este protótipo pode vir a ser útil em diversas áreas, como é o caso de *query expansion*, entre outros.

Palavras-chave: *Big Data*, Expressões Relevantes (ERs), Extratores de ERs, *N*-grama, Correlação de Pearson, Sistemas Paralelos e Distribuídos, Hadoop, Spark.

Abstract

It was estimated that 90% of the data that currently exist on the Internet was created in the last 2 years. Taking the data growth into account, the number of relations/patterns in them increases rapidly. Together, sets of documents can be processed to obtain meta-data that describes these linguistic phenomena in natural language. Among the *n*-gram are those which are associated with a more specific semantics. In a *corpus* there are several *n*-grams which may, or may not, be semantically associated. These *n*-grams have the property to strongly reflect the "core" content of documents where they occur. This kind of *n*-grams form, therefore, multi-word expression, also known as relevant expressions. It is possible to measure the semantic association between MWEs. Taking as an example these two MWEs, "financial crises" and "unemployment in the Euro Zone", we can say that there is a strong semantic relation between them. This relation can be calculated by statistic correlation metrics. Other MWEs may be semantically associated to the core content of a document, even if these are not present in the document. For example, in a short text document about environment and contains the MWE "global warming", but does not contain the MWE "Ice melting", which is semantically close to "global warming". In a search environment would be really useful if a search engine could retrieve that text document after the query "Ice melting", even knowing that the document does not contain explicitly that MWE. In order to achieve the automatic construction of such descriptors of documents, it is necessary to possess the ability to calculate the correlation between pairs of MWEs. Considering that the number of these pairs grows with the square of the total MWEs of the *corpora*, the implementation needs a parallel and distributed approach; Hadoop and Spark are possible frameworks to solve this problem. The challenge of this dissertation is to build a prototype that can automatically build descriptors of documents from *corpus*, in useful time. This prototype may be useful in various areas, as is the case of query expansion, as has already been previously mentioned.

Keywords: Big Data, Multi-word Expression (MWE), Extractors of MWEs, *N*-gram, Pearson Correlation, Distributed and Parallel Systems, Hadoop, Spark.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos dias que correm existe uma grande quantidade de dados disponíveis nas plataformas a que temos acesso. Foi estimado que 90% dos dados que existem na Internet foram criados nos últimos dois anos. Estima-se que em 2010 cerca de 29% da população mundial utilizava a Internet. Esse número cresceu para 40% em 2014, enquanto que em 2016 a percentagem de população mundial a utilizar a Internet já tinha atingido 47% [1]. Isto perfaz cerca de 3.7 mil milhões de utilizadores na Internet [2].

Tendo isto em conta, podemos facilmente concluir que os utilizadores, negócios e todos os aparelhos tecnológicos que podemos ter ligados à Internet contribuem para o crescimento enorme de informação que é gerada todos os dias. Com este crescimento da quantidade de dados, o número de padrões e relações neles contida é também muito grande, sendo a sua deteção uma tarefa que exige a criação de mecanismos automáticos para o conseguir.

Com o objetivo de obter meta-dados que caracterizem os fenómenos linguísticos, relacionados com cada idioma, na língua natural, reúnem-se conjuntos de documentos a fim de obter robustez estatística que garanta fiabilidade a esses meta-dados. Nestes meta-dados podemos incluir a frequência relativa dos diferentes n-gramas (sequências de uma ou mais palavras na mesma língua) e outras métricas que podemos considerar de informação cruzada entre n-gramas. De entre os n-gramas existem aqueles aos quais está associada uma semântica mais dirigida/específica. A título de exemplo, o n-grama "Universidade Nova de Lisboa" tem associado um significado muito preciso. Ao contrário deste, o n-grama "Em que" é semanticamente muito vago ou mesmo despido de semântica, em termos práticos. O primeiro n-grama faz parte do conjunto a que chamamos Expressões Relevantes (ERs), expressões que concentram significados muito fortes capazes de num texto quase resumi-lo.

Como a deteção deste tipo de expressões relevantes é uma temática importante para

esta tese, há que ter em conta que existe todo um conjunto de métodos que a pode aplicar. Estes métodos analisam grandes quantidades de dados porque incidem num conjunto de documentos de texto (*corpus* linguístico) de grandes dimensões. Por exemplo, métodos baseados em métricas de correlação, que têm como objetivo medir a dependência/associação entre duas variáveis, (neste caso em particular, com variáveis queremo-nos referir às expressões relevantes) que por surgirem com frequências diferentes em cada documento, podem ser tomadas como variáveis.

Como veremos no capítulo 2 secção 2.1, as Expressoões Relevantes podem ser extraídas/mineradas a partir quer de métodos NLP (*Natural Language Processing*), onde é usada informação morfo-sintática, quer a partir de métodos estatísticos. Qualquer das duas famílias de abordagens usam *corpora* linguísticos.

Uma das tarefas que queremos realizar nesta dissertação, para além de outras, consiste no cálculo da correlação entre todos os pares de expressões relevantes. Assim, tendo em conta que a partir dum *corpus* de, por exemplo, 10×10^6 palavras se podem extraír 10^6 expressões relevantes, ou um número dessa ordem de grandeza, teremos $10^6 \times (10^6 - 1)/2 \simeq 4,99 \times 10^{11}$ correlações a calcular. O número de correlações cresce com o quadrado do número de expressões relevantes do *corpus*, tarefa que se torna num problema de Big Data. Com esta realidade em mente, facilmente percebemos que este tipo de processamento requer o recurso a técnicas de paralelismo. Computação paralela é um tipo de computação que permite distribuir a carga (por exemplo: distribuição do *input* a processar, distribuição das várias tarefas a cumprir) por várias *threads*/processos, de forma equilibrada. Desta forma é possível implementar projetos, que seriam muito pesados computacionalmente, de uma maneira muito mais eficiente [3].

1.1 Motivação

A motivação por detrás deste trabalho assenta principalmente em dois pontos. O primeiro decorre da necessidade de resolver com eficiência um dos problemas criados pelo crescimento do volume de dados: o cálculo da correlação entre grandes quantidades de pares de expressões relevantes; cálculo que permitirá a ligação semântica entre documentos e a construção automática de descritores de documentos através das suas *keywords* também automaticamente extraídas.

O segundo ponto reside na área do paralelismo. Pretende-se que a abordagem a desenvolver seja implementada em protótipo e que este possa ser utilizado para processar *corpora* de média e grande dimensão com eficiência. Um terceiro motivo, passa pela elaboração de um artigo que mereça a consideração da comunidade é também uma motivação.

A elaboração de técnicas como as que vão ser usadas no âmbito desta dissertação pode reequacionar o peso a investir nas componentes de hardware e software, por parte das organizações/empresas.

Relacionado com o primeiro ponto de motivação já referido, está a capacidade que se pretende adquirir de modo a obter relações de proximidade entre documentos através do

seu conteúdo (os seus tópicos), bem como a construção automática de descritores desses documentos.

1.2 Identificação do Problema

Um *corpus* linguístico é um conjunto de documentos de texto escritos normalmente numa determinada língua, que serve como base de análise para diferentes propósitos. Como já foi referido, algumas sequências de palavras (*n*-gramas) estão fortemente ligadas e formam, por isso, expressões relevantes (ERs), também designada por *Multi-word Expression* (MWEs). Estas expressões têm a propriedade de refletir fortemente o conteúdo *core* dos documentos onde ocorrem. São elas por exemplo, "crise financeira mundial", "aquecimento global", "direitos humanos", entre outras. As ERs podem ser extraídas por extratores próprios.

Uma vez extraídas, é possível medir quão semanticamente próximas as ERs estão umas das outras. É de esperar que exista uma proximidade semântica forte entre as ERs "crise financeira mundial" e "desemprego na Zona Euro" mas, não é de esperar qualquer proximidade semântica entre as ERs "seca na Península Ibérica" e "Liga NOS". A proximidade semântica pode ser calculada a partir de métricas de correlação estatística.

Por outro lado, através de métricas estatísticas, é também possível selecionar as ERs mais informativas/relevantes e que, por isso mesmo, possam vir a ser usadas como *keywords* descritoras do conteúdo dos documentos. No entanto, tendo em conta os tópicos de que trata um documento, este pode não conter explicitamente algumas ERs/*keywords* com as quais está semanticamente relacionado. Por exemplo, um documento muito pequeno pode tratar de questões relacionadas com o ambiente e conter ERs como *"global warming"* mas não conter a ER *"Ice melting"*, da qual está semanticamente próximo como facilmente se compreende; seria útil que, num ambiente de pesquisa, um motor de busca pudesse recuperar também este documento após uma pesquisa sobre *"Ice melting"*, apesar do documento não conter explicitamente esta ER.

Para conseguir a construção automática de tais descritores de documentos, é necessário dispôr da capacidade de cálculo da correlação entre pares de ERs, sendo que, o número de pares cresce com o quadrado do tamanho do número de ERs no *corpus*, como já foi referido. Um ambiente de processamento não paralelo é pois ineficiente para resolver este problema. Por outro lado, seria desejável que estes mecanismos pudessem ser usados para várias línguas sem ter que conhecer a sua estrutura morfo-sintática.

1.3 Objetivos

A construção automática de descritores de documentos a partir de *corpora*, não dependente da língua, impõe dois objetivos principais:

Um primeiro objetivo reside na elaboração de técnicas que selecionem as ERs mais semanticamente informativas explicitamente escritas em cada documento. A estas ERs

chamaremos as *keywords* explícitas. Esta seleção deverá usar métodos estatísticos, de forma a não dependerem da estrutura de qualquer língua. Para este trabalho foi previamente fornecido o conjunto de ERs para todos os documentos do *corpus* através de um extrator apropriado. Este objetivo envolverá a necessidade de processamento paralelo e distribuído com vista ao mapeamento das ERs nos respetivos documentos, processo que envolve uma combinatória computacionalmente pesada.

Um segundo objetivo, assenta na necessidade de dotar cada descritor com "pontes semânticas" para outros documentos. Este mecanismo será obtido através da identificação daquilo a que chamamos *keywords* implícitas. Estas *keywords* correspondem a ERs muito informativas que, não residindo no documento, estão no entanto fortemente correlacionadas com as *keywords* explícitas do documento. Esta correlação será calculada estatisticamente. Para conseguir este segundo objetivo, impõe-se a necessidade de usar técnicas de processamento paralelo e distribuído, dado o elevado número de pares de correlações a calcular.

1.4 Estrutura do documento

A figura 1.1 mostra a organização deste documento por 5 capítulos. O capítulo 1, **Introdução**, é composto por esta secção e mais 3, onde é contextualizado o problema e a identificação do mesmo. O capítulo 2, **Estado de Arte**, é dividido em 4 secções principais que descrevem o trabalho relacionado. Capítulo 3, **Solução Proposta**, é constituído por 3 secções onde são descritos detalhadamente os 3 protótipos produzidos para este trabalho, bem como uma última secção no qual é demonstrado como os descritores de documentos são construídos. Capítulo 4, **Avaliação**, que é composto por 5 secções onde começamos por apresentar uma análise entre as expressões relevantes e a correlação de Pearson, posteriormente a uma descrição do ambiente de execução experimental. Seguido da avaliação dos resultados obtidos dos protótipos Hadoop MapReduce e Spark, e uma última secção onde podemos visualizar os resultados da construção de descritores de documentos. Por fim, o capítulo 5, **Conclusões e Trabalho Futuro**, onde estão as conclusões retiradas deste trabalho, bem como possíveis direcções para um trabalho futuro.

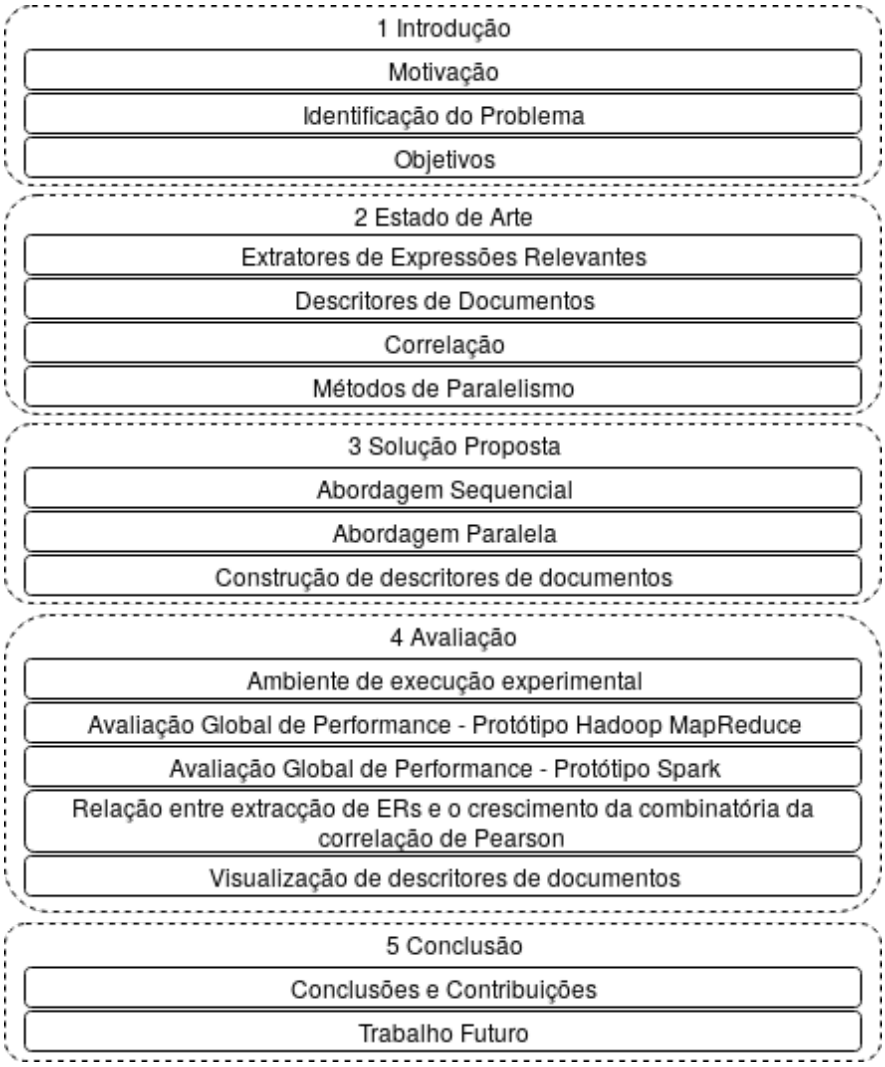


Figura 1.1: Organização do documento

Capítulo 2

Estado de Arte

Embora a dissertação se centre em trabalho a desenvolver, digamos, a jusante da extração de ERs, não deixa de fazer sentido analisar as diferentes abordagens para extração destes termos.

2.1 Extratores de expressões relevantes

Existe uma quantidade considerável de métodos de extração de ERs. Considerando estas expressões como as mais relevantes de um determinado *corpus*, elas contribuem fortemente para a caracterização do conteúdo de um ou mais documentos. Com o aparecimento dos métodos de extração de ERs houve desde logo a distinção entre dois tipos de extratores: os extratores simbólicos e os estatísticos.

Os extratores simbólicos, baseiam a sua extração em informação morfo-sintática. O *Parser* é um dos mecanismos fulcrais na extração de ERs. O termo *parsing* provem das palavras em Latim *pars orationis*, que quer dizer "parte do discurso", isto é, separar uma frase das palavras que são verbos, números, nomes compostos, entre outros. Este mecanismo tem como propósito ajudar a identificar nomes, verbos, adjetivos, entre outros. Tendo em consideração a construção frásica produzida pelo ser humano, o processo de *parsing* pode ser complicado, pois por vezes a semântica ou o verdadeiro significado estão escondidos dentro de um intervalo de possibilidades que pode vir a ser quase ilimitado. Com o intuito de fazer *parsing* em língua natural é necessário recorrer à sua estrutura gramatical. O *Parser* é um componente de software que, por norma, tem como *input* um documento texto e a partir dele constrói uma estrutura de dados, tipicamente em árvore ou alguma estrutura hierárquica.

Outro dos mecanismos usados pelos extratores simbólicos é o *part-of-speech tagging* (POS -tagging) aplicado num *corpus*. Este processo está relacionado com o mecanismo

de *parsing* e consiste em atribuir uma etiqueta morfo-sintática a uma sequência de uma ou mais palavras. Possíveis *tags* são Vb, Adj, Nm, SN, SV, SAdv, entre outros, para designar verbo, adjetivo, nome, sintagma nominal, sintagma verbal e sintagma adverbial, respectivamente.

Embora os *pos-taggers* sejam na sua maioria baseados exclusivamente em informação morfo-sintática, sendo por isso *rule-based*, como por exemplo o E. Brill's tagger [4], existem abordagens que recorrem também a métodos estocásticos (como por exemplo em [5]).

Por outro lado, os extratores estatísticos baseiam-se em probabilidades estatísticas que refletem dependendências e relações que são transversais à grande maioria das línguas. Estas propriedades são, por exemplo, dadas pela probabilidade condicional que pode medir a coesão entre as palavras constituintes de um n-grama. Métricas de correlação entre palavras também podem refletir essa coesão. Sendo estas propriedades comuns às diferentes línguas, deixa de ser necessário utilizar técnicas como por exemplo, *POS-Tagger*, que são ferramentas dependentes da língua. A título de exemplo, admitindo que a grande parte das ERs está nos sintagmas nominais, em Inglês, estes têm a sequência morfológica (Adj,Nm), ou seja, adjectivo seguido de nome; em Português a sequência seria (Nm, Adj).

Em geral, este tipo de extratores tem no entanto a desvantagem de tender a apresentar valores de precisão mais baixos. Por outras palavras, estes extratores cometem mais erros de seleção ao dizer que um certo termo é uma ER quando na verdade não o é.

Foram avaliados alguns extratores, simbólicos e estatísticos, a fim de escolher o mais indicado para o contexto desta dissertação. Avaliação esta que teve como base os seguintes critérios:

- **Suporte Multi-língua** - se o extrator tem suporte para mais de uma língua, por exemplo: Conseguir extrair expressões relevantes de textos em Português, Inglês, e Francês;

- **Utilização de *POS-tagging*** - se o extrator precisa de recorrer a algoritmos de *POS-tagging*;

- **Taxa de sucesso** - se o extrator tem resultados de precisão suficientemente bons para a sua posterior utilização.

Por aplicação de um *parser* [6], apresentado em [7], para a elaboração de terminologias em 18 línguas diferentes, foi obtido um conjunto de 5 milhões de palavras multi-língua com as devidas anotações e correspondências do *corpus*. Um extrator de *multi-word expressions*, apresentado no Workshop [7], usou o referido *parser* para extrair apenas *verbal multi-word expressions*. Como na temática desta dissertação pretendemos identificar mais do que apenas expressões relevantes verbais, acabou por se descartar o uso deste extrator.

Foi encarada a hipótese da utilização do *Fips parser* e *collocations identification mechanism*, por ser um parser que tem suporte para 8 línguas, sendo elas Francês, Inglês,

Alemão, Italiano, Espanhol, Grego Moderno, Romeno e Português. Este baseia-se em conceitos de gramática genérica para que possa fazer *parsing* de uma maneira modelar e que consiga ser versátil para se adaptar a uma língua em particular. Porém, a ocorrência de qualquer *collocation* (frase idiomática) constitui sempre um desafio às estruturas gramaticais. E, tendo em conta os resultados que foram apresentados por esta proposta, decidiu-se não aplicar este método de extração de ERs, pois mesmo tendo mecanismos que são bastante promissores, a precisão era na ordem do 58%, o que fez com que fosse descartada esta opção.

Em [8], os autores utilizaram um sistema com base em redes neuronais para identificar expressões relevantes verbais (VMWEs). Este sistema foi treinado com um *data set* contendo já VMWEs e também informação morfo-sintática. Apesar do seu suporte para 15 línguas diferentes, um dos sistemas que consegue categorizar quase todas as línguas, decidiu-se não fazer uso dele pois não queremos apenas extrair expressões relevantes verbais.

Em [9] é feita a extração de ERs, limitado a 2-gramas, em Lituano e Letão. Não se considerou o uso deste mecanismo, pois não queremos um extrator com suporte apenas para estas duas línguas e só para 2-gramas, apesar de a identificação de ERs nestas línguas ser, segundo os autores, muito complexa.

Os autores em [10] apresentam um método de extração de ERs com suporte para Inglês e Alemão, focando-se apenas naqueles que são substantivos compostos. Este tipo de substantivo é formado pela junção de duas ou mais palavras, como por exemplo Cachorro-quente. A aplicação deste método na língua alemã foi adequado, devido à possibilidade de criação de substantivos compostos com um grande número de palavras, por exemplo *Herzblut* que é um substantivo composto por *Herz* (tradução para coração) com *blut* (tradução para sangue). Pelo facto desta abordagem se limitar à extração de ERs apenas em duas línguas e só para esta classe morfológica, acabou por se excluir a sua utilização.

Em [11] é apresentado um método eficiente capaz de identificar ERs bastante ambíguas. Este método baseia-se num tipo de *POS-tagging*, *Conditional Random Fields*(CRFs), que suporta mecanismos como *strutured perceptron*, o que permite desambiguar ERs. Por só ter suporte para a língua francesa, este extrator foi descartado das opções viáveis.

Foi considerado outro tipo da abordagens à problemática de extração de expressões relevantes, tais como [12] e [13]. A solução descrita em [12] consiste não só em identificar ERs (n-gramas), mas também em identificar unigramas relevantes. Segundo esta abordagem, numa ER multi-palavra, as palavras tendem a ocupar distâncias preferenciais entre elas. Este fenómeno não depende dos idiomas. Para o caso dos unigramas relevantes, estes apresentam também um conjunto relativamente limitado de palavras com as quais se combinam preferencialmente, na sua vizinhança. Apesar de necessitar de *thresholds*, este extrator apresenta resultados muito promissores e aliciantes à sua utilização. Em termos de extração de unigrama na língua Portuguesa, Inglesa e Alemã, consegue oferecer uma precisão de 0.92, 0.90 e 0.87, respetivamente. No que toca à extração de n-gramas na língua Portuguesa, Inglesa e Alemã oferece uma precisão de 0.87, 0.82, 0.87, respetivamente.

Apesar deste extrator oferecer os melhores resultados e com os métodos mais indicados para a temática desta tese, infelizmente não foi possível obter a sua implementação em tempo útil, por isso não foi utilizado.

Já o extrator [13], apresenta um método de extração de expressões relevantes, mas apenas as que são compostas por 2 ou mais palavras. *LocalMaxs* é baseado na ideia de que todos os *n*-gramas possuem aquilo a que se poderia chamar "*cola*" ou coesão entre as palavras constituintes do *n*-grama. Normalmente *n*-gramas diferentes têm valores de coesão diferentes. Podemos facilmente concluir que existe uma coesão forte entre "Donald Trump", isto é entre as palavras "Donald" e "Trump". Contudo não se pode concluir o mesmo de "de que" ou "em que". Tendo isto em conta, a métrica usada para medir a coesão entre não só de bigramas, mas também *n*-gramas $\geqslant 2$, foi $SCP\_f(.)$:

$$SCP\_f(w_1...w_n) = \frac{p(w_1...w_n)^2}{Avp} \tag{2.1}$$

Sendo:

$$Avp = \frac{1}{n-1}\sum_{i=1}^{n-1} p(w_1...w_i)\cdot p(w_{i+1}...w_n) \tag{2.2}$$

Onde $p(w_1...w_n)$ é a probabilidade do *n*-grama $w_1...w_n$ ocorrer no *corpus*. Por consequência, um *n*-grama de qualquer tamanho é *transformado* num pseudo-bigrama que reflete a média da coesão entre quaisquer dois sub-*n*-gramas adjacentes contíguos do *n*-grama original. Posto isto, é possível comparar a coesão entre quaisquer *n*-gramas de diferentes tamanhos. *LocalMaxs* é um extrator que é independente da língua, e produz como resultado, após seleção, *n*-gramas a partir de um *corpus*, podendo estes ser conjuntos de palavras, *tags*, ou caracteres. Não sendo sequer necessário atribuir um *treshold*. Segundo este algoritmo, sendo $len(W)$ o tamanho (número de elementos) do *n*-grama W, segue a definição de expressão relevante.

Definição Seja $W = w_1...w_n$ um *n*-grama e g(.) uma função genérica de coesão. E seja $\Omega_{n-1}(W)$ um conjunto de valores de $g(.)$ para todos os (*n-1*)-gramas contidos no *n*-grama W; e $\Omega_{n+1}(W)$ um conjunto de valores de $g(.)$ para todos os (*n+1*)-gramas contíguos que contêm o *n*-grama W. W é uma expressão relevante(*multi-word expression*(MWE)) se e só se,
$for(\forall x \epsilon \Omega_{n-1}(W), \forall y \epsilon \Omega_{n+1}(W))$
$(len(W) = 2 \wedge g(W) > y) \vee (len(W) > 2 \wedge g(W) > \frac{x+y}{2})$

Assim, para a escolha de *n*-gramas com $n \geq 2$, o algoritmo *LocalMaxs* elege todos os *n*-gramas cujo valor da sua coesão é maior do que a média de dois máximos: o maior valor de coesão encontrado num (*n-1*)-grama contíguo que está contido no *n*-grama(W), e o maior valor de coesão encontrado num (*n+1*)-grama contiguo que está contido no *n*-grama(W). Na temática desta tese usamos o algoritmo *LocalMaxs* como extrator de expressões relevantes. A razão pela qual se escolheu este extrator, foi não só por ser

independente da língua, mas também por apresentar resultados em termos de *precision* e *recall*, na língua Inglesa, Portuguesa, Francesa, Alemã, entre outras, de 0.72%.

Para eleger unigramas relevantes, será utilizado um método baseado na métrica TF-IDF.

2.2 Descritores de documentos

2.2.1 *Keywords* Explícitas

No conjunto das ERs de um documento, existem naturalmente as expressões que são mais informativas e reveladoras do conteúdo "core" do documento. Estas podem ser identificadas automaticamente usando, em alternativa ou em combinação, métricas como o *TF-IDF* ou a média/mediana dos comprimentos das palavras que constituem cada ER: as maiores médias/medianas correspondem às ERs mais informativas, provavelmente, *keywords* explícitas.

2.2.2 *Keywords* Implícitas

Existem ERs fortemente informativas que estão correlacionadas com cada documento, ou seja, com as suas *keywords* explícitas mas no entanto, não estão explicitamente escritas no documento. Estes termos implícitos podem ser identificados através dos valores de correlação mais altos entre as *keywords* explícitas do documento e as diferentes ERs de todo o *corpus*. Podemos calcular a correlação entre duas ERs de diferentes maneiras, estando estas explanadas na secção 2.3.

2.3 Correlação

Considerando a temática deste trabalho, foram estudadas algumas métricas de correlação estatística. A correlação é uma medida de associação/relação estatística entre duas variáveis aleatórias. No contexto desta tese vai ser usada a correlação entre duas variáveis, tendo como objetivo medir quão semanticamente ligadas estão duas ERs.

2.3.1 Correlação de Kendall (*Kendall rank correlation coefficient*)

O coeficiente τ de Kendall, apresentado em [14] é uma medida de relação estatística entre duas variáveis. Este coeficiente é dado por:

$$\tau = \frac{n_c - n_d}{\frac{n(n-1)}{2}} \tag{2.3}$$

Onde n_c indica o número de pares concordantes, n_d o número de pares discordantes, e n o número de elementos assumidos pela variável. O denominador corresponde ao número total de pares dado um conjunto de n elementos.

Considerando o par de observações x_i, y_i e x_j, y_j, das variáveis conjuntas de X e Y, com $i \neq j$, diz-se que estamos em presença de um par concordante se se verificarem as seguintes condições: $x_i > x_j$ e $y_i > y_j$ ou $x_i < x_j$ e $y_i < y_j$. É par discordante no caso contrário, ou seja, quando se verifique: $x_i > x_j$ e $y_i <= y_j$ ou $x_i <= x_j$ e $y_i > y_j$. O intervalo de resultados do coeficiente de Kendall compreende valores entre -1 e 1, sendo que os valores próximos de 1 indicam uma forte correlação. Valores próximos de -1 correspondem, claramente, a correlações negativas. Valores próximos de 0 refletem correlações fracas.

X	Y
5	50
20	30
30	15
90	80

Tabela 2.1: Exemplo de uso de τ

Tendo em conta o exemplo da tabela 2.1, $n_c = 3$ e $n_d = 3$, logo $\tau = 0$, pelo que os dois conjuntos de valores, X e Y, não estão correlacionados de acordo com o coeficiente de Kendall. Este coeficiente exibe uma desvantagem levando em conta a temática desta tese. Apenas tem em conta a posição relativa(*rank*) dos valores, logo não avalia com rigor as variações dos valores assumidos pelas variáveis.

2.3.2 Correlação de Spearman (*Spearman's rank correlation coefficient*)

O coeficiente de correlação de Spearman, ρ, foi proposto em [15], mede a dependência entre o *rank* assumido por duas variáveis. Este coeficiente é dado por:

$$\rho = \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{2.4}$$

No qual d_i é a diferença entre os *ranks* das duas variáveis. O intervalo de resultados deste coeficiente é, [-1 ; 1]. Intuitivamente o coeficiente de Spearman entre duas variáveis vai ser mais alto (próximo do 1) quando as variáveis se encontram em *ranks* semelhantes; mais baixo (próximo do 0) quando as variáveis têm *ranks* díspares; e quando estas têm *ranks* fortemente "opostos" o resultado vai ser próximo de -1.

Considerando o exemplo dado pela tabela 2.2, calculando as distâncias entre os *ranks* das variáveis: empregando a fórmula do cálculo da correlação de Spearman, obtemos um valor de ρ=0.2. Com este valor conseguimos perceber que existe uma correlação relativamente fraca entre as variáveis X e Y.

Este coeficiente é apropriado para o uso de variáveis sejam elas contínuas ou discretas. A razão pela qual se descartou o uso desta fórmula foi análoga à razão pela qual não utilizamos a de Kendall, apesar do coeficiente de Spearman já ter em conta a distância entre os *ranks*, não pondera suficientemente bem os desvios de valores que as variáveis

X	Y	Rank(X)	Rank(Y)	d_i	d_i^2
5	50	1	3	-2	4
20	30	2	2	0	0
30	15	3	1	2	4
90	80	4	4	0	0

Tabela 2.2: Exemplo de uso de ρ

podem tomar.

2.3.3 Correlação de Pearson (*Pearson correlation coefficient*)

Numa situação em que o contexto fale de cálculo de correlações entre duas variáveis, a fórmula mais frequentemente utilizada é o coeficiente de correlação de Pearson, também conhecido como *bivariate correlation*. Este coeficiente mede a correlação entre duas variáveis, X e Y. Tendo sido inicialmente apresentado em [16, 17], este coeficiente foi posteriormente aplicado em diversas áreas de estudo. Por exemplo, na área da medicina foi aplicado em dois estudos feitos em eletrofisiologia [18], e na interpretação biológica de associações entre genomas [19]. Este coeficiente é vulgarmente representado por ρ:

$$\rho_{(X,Y)} = \frac{cov(X,Y)}{\sqrt{cov(X,X)} \cdot \sqrt{cov(Y,Y)}} \tag{2.5}$$

ou ainda por:

$$\rho_{(X,Y)} = \frac{cov(X,Y)}{\sigma_x \cdot \sigma_y} \tag{2.6}$$

sendo:

$$cov(X,Y) = \frac{1}{|Docs|-1} \sum_{d \epsilon Docs} (f_r(X,d) - f_{rm}(X,\cdot)) \times (f_r(Y,d) - f_{rm}(Y,\cdot)) \tag{2.7}$$

$$f_r(X,d) = \frac{f(X,d)}{TotalPalavras(d)} \tag{2.8}$$

$$f_{rm}(X,\cdot) = \frac{1}{|Docs|} \sum_{d \epsilon Docs} f_r(X,d) \tag{2.9}$$

Considerando que $cov(X,Y)$ representa a co-variância entre as variáveis X e Y, no contexto desta tese e tendo como base as ERs, X e Y representam quaisquer duas ERs, digamos ER_1 e ER_2. Em que *Docs* representa o conjuntos dos documentos no *corpus*, e $|Docs|$ a sua cardinalidade; $f_r(X,d)$ significa a frequência relativa da ER X num documento d, no qual $f(X,d)$ é a frequência absoluta da ER X em d; sendo $TotalPalavras(d)$ o número total de palavras contidas em d. Por fim, $f_{rm}(X,\cdot)$ designa a frequência relativa média da ER X nos documentos do *corpus*.

Isto é, para a obtenção do coeficiente de Pearson calcula-se o quociente da co-variância das duas variáveis *(X,Y)* pelo produto dos seus desvios padrão. Da mesma maneira que

nas correlações anteriores, o seu intervalo de resultados situa-se entre -1 e 1. No qual -1 indica a existência de, digamos, uma semântica "oposta" entre as duas ERs; valores próximos de 1 mostram proximidades semânticas fortes entre as duas ERs em causa, resultado da sua provável co-ocorrência nos mesmos documentos.

Colocando em perspetiva o tema em estudo, este coeficiente demonstra ser uma ótima medida de correlação. Trabalha com variáveis contínuas, tomando os valores diretos das frequências das expressões relevantes e não dos seus respetivos *ranks*, sendo uma vantagem em relação às métricas de Kendall e Spearman. Esta métrica permitirá uma medida mais precisa da correlação entre as duas variáveis. Além do mais, este coeficiente é dos três o mais leve computacionalmente, visto que não é necessário fazer uma ordenação prévia dos valores para obter os respetivos *ranks*.

2.3.4 Word2vec

O Word2vec é um projecto que foi criado por uma equipa de investigação liderada por Tomás Mikolov na Google. Esta técnica é feita com base em redes neuronais, redes que, por isso mesmo, têm de ser previamente treinadas. Tipicamente esta abordagem toma como *input* corpora linguísticos de médias/grandes dimensões e produz como resultado vetores com muitas dimensões, em que cada palavra do *corpus* é representada por um vetor nesse espaço vetorial[20].

Tendo isto em conta, o Word2vec consegue posicionar palavras num espaço vetorial atendendo ao contexto onde elas surgem no texto. No entanto, não contempla o conceito de expressão relevante e a sua potencialidade na captura do valor semântico do grupo de palavras relevantes, semântica essa que não é capturada com a simples agregação semântica das palavras individuais que constituem a expressão relevante. Por exemplo a expressão "raining cats and dogs", não significa que chovem cães e gatos, mas sim que está a chover imenso.

O nosso protótipo pretende construir descritores de documentos, baseados em *keywords* automaticamente extraídas dos documentos. As *keywords* correspondem às expressões relevantes mais informativas. Nesta medida, o Word2vec não oferece potencialidade para ser uma solução para o problema, pois a noção de expressão relevante não é contemplada nessa técnica. Quanto às *keywords* implícitas, o Word2vec poderia ser uma solução, no sentido em que consegue determinar a "distância semântica" entre palavras, mas ainda assim, não determina essa distância aplicada a expressões relevantes (tipicamente, conjuntos de 2 ou mais keywords, n-grams).

Quanto ao paralelismo, julgamos não haver à partida nenhuma impossibilidade do cálculo relativo do posicionamento das palavras pelo Word2vec ser feito em paralelo, depois de construída a rede neuronal. No entanto, põe-se sempre a questão de não satisfazer a condição das expressões relevantes.

2.4 Métodos de Paralelismo

Como já foi referido anteriormente, nos últimos anos registou-se um crescimento massivo de dados (*data*). As aplicações permitem às organizações ter cada vez mais informação e o desenvolvimento tecnológico na informática e no seu hardware tem permitido o armazenamento de cada vez maiores volumes de dados, assim como o seu processamento. Tecnologias como *World Wide Web*, serviços empresariais, entre outras aplicações de engenharia, têm contribuído para este crescimento. Deste modo, não é tarefa fácil organizar e analisar todos estes dados. Tais tecnologias têm sido acompanhadas pelo desenvolvimento das infraestruturas (em particular sistemas paralelos e distribuídos oferecidos em serviço de *cloud*) e ferramentas/*frameworks* para facilitar o seu processamento. Tendo em conta estas necessidades, resultaram duas áreas, *data science* e *data mining*. Estas são muito relevantes nos dias que correm (Era da Informação) [21].

O volume de dados processado pelos sistemas de hoje em dia já ultrapassou o poder de processamento dos sistemas tradicionais. O crescimento de novas tecnologias e novos serviços (por exemplo, Serviços *cloud*) têm levado ao crescimento contínuo de informação na Internet. Este fenómeno representa um grande desafio à comunidade de analistas de dados (*data analytics*). E este crescimento é a razão pela qual surgiu e se tornou importante a área de *Big Data*, que tem vindo a crescer. Esta área sugere a procura de mecanismos de alto desempenho de processamento, para tratar de uma vasta e variada quantidade de dados [21].

A computação paralela e distribuída já era usada antes do advento de *Big Data*. Muitos dos algoritmos standard que consumiam muito tempo de execução, foram substituídos pela sua versão paralela. Tornou-se por assim obrigatório a existência de abordagens paralelas e distribuídas para os problemas que envolvam tratamento de grandes volumes de dados [21].

Enquadrando no tratamento de grandes volumes de dados a problemática que queremos resolver, se executado numa só máquina, esta solução irá estar limitada às suas peças constituintes de *hardware*, no entanto, se aumentarmos a quantidade de máquinas aumentamos a quantidade de trabalho que se pode fazer em paralelo. O problema a tratar envolve mapear expressões relevantes nos respetivos documentos e a posterior combinatória computacionalmente pesada das mesmas. Como o mapeamento das ERs e alguns dos passos que envolvem a combinatórias das ERs podem ser feitos de forma independente, revelou-se assim que este é um problema que sugere a execução em paralelo.

Muitas das plataformas para processamento de larga escala têm tentado atacar a problemática de *Big Data* nos últimos anos [22]. Este tipo de plataformas tenta aproximar as tecnologias paralelas e distribuídas aos seus utilizadores regulares (engenheiros, ou *data scientists*), omitindo algumas das nuances técnicas. Esta dissertação concentra-se em plataformas capazes de executar em ambiente *cloud*, pelas suas características de escalabilidade a pedido, em particular duas plataformas bastante comuns, Hadoop MapReduce e Spark.

Para que estas plataformas consigam tratar de um grande volumes de dados, estas precisam de ter alguns componentes interligados entre si. Componentes estes que visam resolver diferentes problemas na computação paralela e distribuída. Estes são os 3 problemas fulcrais: armazenamento, processamento, e coordenação.

2.4.1 Computação *Cloud*

O conceito de computação *cloud* trouxe muitas vantagens no que toca ao desenvolvimento de máquinas configuradas em *cluster*, que podem facilmente ser requisitadas a pedido e à medida das necessidades, fazendo com que a carga de processamento possa ser distribuída de uma maneira equilibrada. Um provedor de serviços *cloud* pode oferecer 3 tipos de serviços (todos por via da Internet) [23]:

IaaS : *Infrastructure as a service*, tem como objetivo fornecer aos utilizadores/organizações instâncias de máquinas virtuais, entre outros recursos físicos. Contudo este é um nível que costuma ser muito baixo para grande parte dos utilizadores. Este tipo de serviço oferece uma abstração sobre o hardware, pois muitas das organizações podem não ter o capital ou as competências para construir um centro de dados (*datacenter*).

PaaS : *Platform as a service*, oferece tanto hardware como software. Estas plataformas são um conjunto bem definido de serviços na qual, permitem aos utilizadores/organizações desenvolver uma aplicação "por cima" deste serviço. Um dos exemplos da utilização deste tipo de serviços é a *Google App Engine*, que fornece armazenamento de *backend* e uma API para desenvolver aplicações web altamente escaláveis. Com este serviço os utilizadores/organizações ficam livres de instalar, tanto hardware como software, para desenvolver uma aplicação.

SaaS : *Software as a service* que está no nível mais alto de abstração, faz *host* de aplicações e deixa-as disponíveis aos seus utilizadores. Um exemplo deste serviço é a empresa Salesforce, que é líder na gestão de relacionamento com o cliente (*Customer Relationship Management*, *CRM*). Este serviço remove a necessidade das organizações instalarem e correrem as aplicações nas suas próprias máquinas ou centros de dados.

Estes serviços oferecem vantagens no desenvolvimento e gestão das infraestruturas (máquinas e restantes recursos) e sua oferta aos utilizadores/organizações. Também procuram facilitar ao utilizador/programador no desenvolvimento de soluções de software que usam essas infraestruturas e serviços *cloud*. Uma das vantagens mais relevantes é o facto de se poder alugar recursos computacionais que de outro modo lhes seria impossível adquirir, como por exemplo, *clusters* e armazenamento. Consequentemente, o custo do processamento destes dados só vai ser gasto no momento em que os dados estiverem prontos para serem processados. Ou seja, dá a vantagem ao utilizador de pagar apenas o

que usa. Devemos estar conscientes de que, embora o montante global de dados nestes casos realmente exceda os limites dos atuais computadores físicos, a frequência tanto da obtenção como do processamento de dados pode ser variável, enfatizando assim o quão útil a computação *cloud* é.

As aplicações *web* vistas hoje em dia podem ser chamadas também de "aplicações *cloud*". De facto, os utilizadores ao acederem a sites web com o seu *browser*, sem saberem, podem estar a usar aplicações que estão a coligir informação sobre a sua utilização e comportamento dos utilizadores. Muitas destas aplicações são suportadas por servidores (possivelmente em *cloud*) com a interface via *browser*. Como por exemplo, serviços de redes sociais como Facebook, sites de partilha de vídeo como Youtube, sites baseados em serviços de email como Gmail, ou aplicações como o Google Docs. Neste contexto o conceito de computação *cloud* é um modelo que permite de forma ubíqua, aceder a uma rede com uma *pool* partilhada de recursos computacionais configuráveis (p. e. servidores, armazenamento, aplicações, e serviços) que podem ser rapidamente provisionados e lançados, não requerendo muito esforço em geri-la [24]. Claro que a vasta quantidade de dados gerada pelos utilizadores cria grandes problemas para processar esses dados em tempo útil.

2.4.1.1 Armazenamento de Dados

Para suportar o volume de dados necessário e para a execução paralela eficiente, houve necessidade de distribuir os dados pelo *cluster* num sistema de ficheiros distribuído [25] agregando as capacidades de todos os recursos. Tal é também assim no caso do Hadoop com o seu HDFS—Hadoop Distributed File System— [26]. O HDFS, é um sistema de ficheiros distribuídos que armazena grandes quantidades de dados. Este sistema não deve ser usado em aplicações que necessitem de acesso rápido a um determinado registo, mas sim aplicações que precisem de ler uma quantia grande de dados. Um facto importante a manter em mente é que não deve ser utilizado para ler ficheiros muito pequenos (menores que 64 Mega Bytes), tendo em conta a sobrecarga na memória envolvida.

HDFS baseia-se no conceito de blocos, tal como nos sistemas Unix, o tamanho dos seus blocos normalmente varia entre 64 a 128 Mega Bytes. Um ficheiro muito grande pode ser particionado em vários blocos armazenados em mais do que um nó. Cada bloco é replicado em 3 servidores diferentes para garantir tolerância a falhas. Este sistema possui dois tipos de nós: *Master* (ou NameNode) e *Worker* (*Datanode*), como indicado na figura 2.1, retirada de [23]. O *Master* armazena informações da distribuição de arquivos e metadados. Já o *Worker* armazena os dados propriamente ditos, como podemos ver na imagem descrita 2.1. Logo o *Namenode* necessita sempre de estar disponível.

2.4.1.2 MapReduce

Uma abordagem que se revela simple e eficaz para atacar a problemática de volumes grandes de dados é dividir e conquistar (*divide-and-conquer*), um conceito fundamental

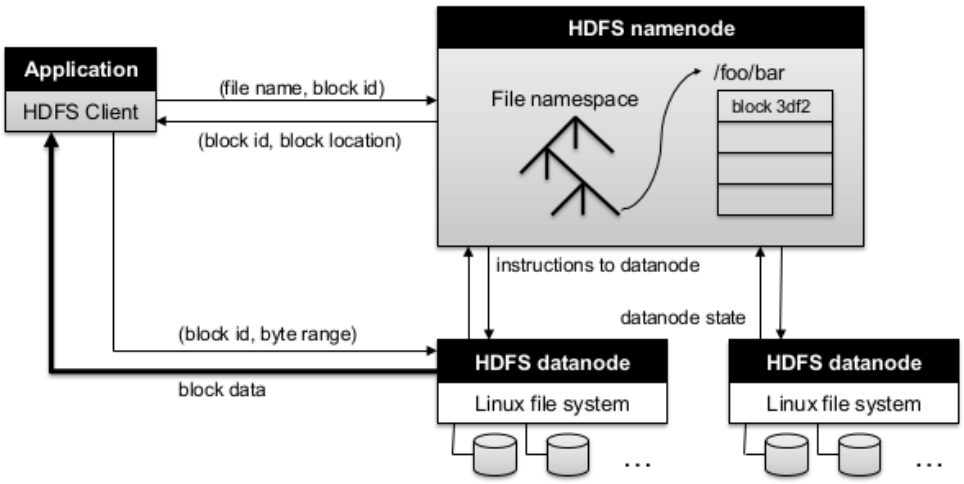


Figura 2.1: Arquitetura de HDFS [26]

à ciência da computação. A ideia base é particionar um grande problema em pequenos sub-problemas. Sendo estes sub-problemas independentes, podem ser processados em paralelo por diferentes trabalhadores (*threads* de um processador, muitos processadores numa máquina, ou muitas máquinas em *cluster*). Resultados intermédios dos vários trabalhadores (máquinas) são combinados, resultando no *output* final [23].

Os princípios por detrás de algoritmos *divide-and-conquer* aplicam-se a um grande leque de problemas em diferentes ramos. Contudo os detalhes das suas implementações são variados e complexos. Aqui estão algumas das questões que devem ser respondidas para desenvolver uma aplicação paralela:

- Como partir o problema grande em pequenos sub-problemas?
- Como espalhar as tarefas por trabalhadores que estão distribuídos num grande número de máquinas?
- Como garantir que os trabalhadores recebem os dados de que precisam?
- Como coordenar a sincronização entre os diferentes trabalhadores?
- Como partilhamos resultados de um trabalhador que são precisos para outro?
- Como conseguimos realizar todas as questões acima face a erros de software e falhas no hardware?

Na programação paralela tradicional o programador pode ter de responder e implementar uma solução, na maior parte dos casos a todas as questões acima. Se se programa numa infraestrutura de memória partilhada é necessário coordenar explicitamente os acessos às estruturas de dados partilhada, pelo uso de primitivas de sincronização tais

como, *mutexes*, coordenar a sincronização entre as diferentes máquinas, e ainda manter-se atento a possíveis problemas como *deadlocks*. Um sistema como o OpenMP [27], providencia uma API com abstrações lógicas que escondem os detalhes de sincronização de sistemas e primitivas de comunicação. Contudo, mesmo com estas extensões o programador tem de continuar a preocupar-se em manter disponível os diversos recursos de que cada trabalhador precisa. Além disso, estas *frameworks* são maioritariamente desenhadas para problemas de processamento intensivo e têm um suporte fraco para lidar com quantidades significativas de dados [23]. Ficando tal a cargo do programador.

Com o objetivo de colmatar esta problemática, a primeira *framework* de sucesso em ambiente *cloud* que permitiu processar grande quantidade de dados em larga escala foi o MapReduce [28]. Esta *framework* pode ser utilizada em serviços de email, para analisar mensagens e o comportamento do utilizador com o objetivo de otimizar a seleção de publicidades adequadas. Ou para analisar grafos enormes que representam as amizades de um utilizador numa rede social (p. e. Facebook), para lhe sugerir novas amizades. Estes são os grandes problemas de dados que o MapReduce tem atacado [23].

Uma das grandes vantagens que o MapReduce [28] tem é o facto de disponibilizar uma abstração de mais alto nível, escondendo muitos detalhes de diferentes níveis do sistema. Isto permite que o *developer* se foque mais no que é que os processamentos precisam para ser executados, em vez de se preocupar como é que essas computações são realmente realizadas ou de como é que os processos obtêm os dados que requerem. Tal como o OpenMP, o MapReduce concede os meios de computação paralela e sua distribuição sem sobrecarregar o programador. Organizar e gerir grandes volumes de processamento é só uma parte do desafio. Processar largas quantias de dados requer juntar grandes *datasets*[1] a código para ocorrer a computação. A *framework* MapReduce resolve este problema de maneira a fornecer um modelo de programação simples ao programador, tratando de forma transparente os detalhes da execução paralela numa maneira escalável, robusta, e eficiente. Por razões de eficiência, em vez de mover grandes volumes de dados, se possível, mover o código pelos nós que possuem os dados. Esta operação consiste em espalhar os dados em discos locais nos nós do *cluster* e correr os processos nos nós que possuem os dados. A tarefa de gerir o armazenamento nos processos, normalmente é tratada por um sistema de ficheiros distribuído que assenta por baixo do MapReduce [23].

MapReduce é um modelo de programação que tem as suas raízes em programação funcional. Uma das vantagens de linguagens funcionais é que existe o conceito de hierarquia de funções, ou funções que aceitam outras funções como argumento. Para além disto, também existe a diferença de que neste modelo baseamos o programa na avaliação de funções sem necessidade de estado global partilhado entre diferentes contextos de avaliação. Todo o estado é pesado/recebido como parâmetros da função ou resultados da mesma. Como consequência, não existem problemas de concorrência ou operações explícitas que o programador tenha de usar para a partilha de dados entre diferentes

[1] Coleção de dados.

threads.

Esta *framework* possui duas fases: a fase do *Map* que corresponde à operação *map*[2] em programação funcional; E a fase de *Reduce* que corresponde à operação *fold*[3] em programação funcional. Cada computação recebe uma coleção de pares *key/value*, e produz como *output* pares de *key/value*. O utilizador da biblioteca MapReduce expressa esta computação com duas funções: *map* e *reduce* [23].

Map, escrita pelo utilizador, recebe um par como *input* e produz uma coleção(*set*) de pares intermédios. Esta biblioteca trata de agrupar todos os valores intermédios associados a uma chave X intermédia, e envia-os para a função *reduce*.

A função *reduce*, igualmente escrita pelo utilizador, aceita a chave intermédia e uma coleção de valores associados a essa chave. Junta todos esses valores para formar um *set* pequeno de valores. Estes valores intermédios são fornecidos ao utilizador da função reduce por via de um iterador. Isto permite que possamos trabalhar com grandes listas de valores que são demasiado grandes para serem suportadas pela memória [23].

E como é que este modelo pode ser vantajoso para atacar este tipo de problemas? Como exemplo, podemos considerar o exemplo de contar o número de ocorrências de cada palavra em grandes coleções de documentos. Um exemplo de pseudo-código, retirado de [28]:

```
1  map(String key, String value):
2   // key: document name
3   // value: document contents
4   for each word w in value:
5    EmitIntermediate(w, "1");
6
7  reduce(String key, Iterator values):
8   // key: a word
9   // values: a list of counts
10  int result = 0;
11  for each v in values:
12   result += ParseInt(v);
13  Emit(AsString(result));
```

A função *map* emite cada palavra com um contador de ocorrências associado (neste exemplo é só '1'). Já o *reduce* soma todos os contadores associados a uma palavra em particular [23].

Esta implementação foca-se apenas no processamento de dados, deixando de fora um grande desafio, a distribuição de dados, que as *frameworks* paralelas oferecem implicitamente, como é o caso da implementação Google de MapReduce, ou do sistema Hadoop que é similar ao da Google. Recentemente começaram a aparecer outras implementações

[2]Dada uma lista, a operação *map* recebe como unico argumento uma função f que aplica a todos os elementos da lista.

[3]Dada uma lista, a operação fold recebe como argumetos uma função g (que tem dois argumentos) e um valor inicial: g é primeiro aplicado ao valor inicial e no primeiro elemento da lista, depois o resultado é guardado em variáveis intermédias.

com modelos de programação mais flexíveis, como Spark. Para a temática desta tese foram escolhidas como objeto de estudo, as implementações Hadoop e Spark, pois são duas das *frameworks* mais utilizadas e bem documentadas.

2.4.2 Hadoop

O Hadoop [29] é um projeto de *open-source* desenvolvido pela Apache Software Foundation. É uma plataforma de computação distribuída, que oferece escalabilidade, confiabilidade e tolerância a falhas. Esta implementação permite processar grandes quantias de dados através de *clusters* de computadores usando o modelo de programação MapReduce. Em vez de se confiar no hardware para proporcionar maior disponibilidade, a própria *framework* foi concebida para detetar e tratar falhas na camada da aplicação, de modo a fornecer um serviço com alta disponibilidade. Para além da biblioteca base escrita em Java (Hadoop Common), tem também a biblioteca HDFS—Hadoop Distributed File System—, o MapReduce [23] e o gestor de recursos Yarn.

2.4.2.1 Coordenação e gestão: Yarn

O Yarn [30] é um projeto *open-source* desenvolvido pela Apache Software Foundation. A ideia fundamental da tecnologia Yarn é baseada em dois conceitos chave: controlar recursos no *cluster*; e escalonar as tarefas a executar nos diferentes nós do *cluster*.

Tal como explicito na figura 2.2, existem dois componentes principais, o *Resource Manager* (RM) e o *Node Manager*, que formam a *framework* de computação de dados. Para controlar todos os recursos o *Resource Manager* desempenha um papel crucial e só existe uma instância dele pelo *cluster*. Esta é a autoridade máxima no que toca a alocar recursos entre todas as aplicações no sistema. O *Node Manager* é responsável por monitorizar os recursos em uso dos *containers*, sendo estes *containers* os componentes que são alocados pelo *Resource Manager*. Estes são um conjunto de recursos (RAM, numero de CPUs, Network usage, entre outros) que têm como finalidade executar tarefas, como por exemplo, executar uma tarefa de *Reduce* ou *Map*.

No que toca a escalonar os diferentes tipos de tarefas a executar, o componente *Application Master* (AM) é o responsável. Por norma existe uma instância de AM para cada aplicação existente no sistema. A sua função é negociar com o *Resource Manager* por recursos para a execução de determinada tarefa e, também, trabalhar com o *Node Manager* que executa e monitoriza as diferentes tarefas.

2.4.2.2 Modelo de execução MapReduce

A biblioteca MapReduce é análoga ao que foi explicado em cima. O fluxo de execução desta biblioteca consiste em (exemplificado na figura 2.3): Distribuir as tarefas *Map* e *Reduce* pelo *cluster* (indicado na figura 2.3 como (2)); Segue-se a leitura dos dados distribuídos, isto é, cada nó que tem a responsabilidade de executar a tarefa *Map*, vai executá-la

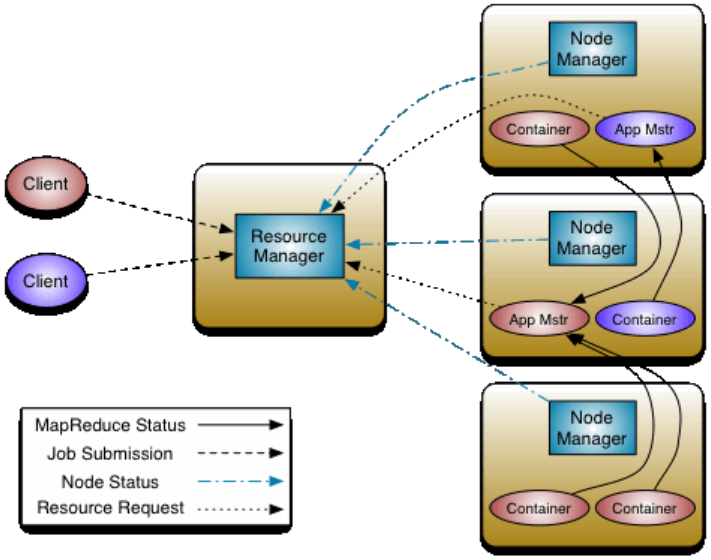


Figura 2.2: Arquitetura de YARN [30]

sobre os dados que possuí (indicado na figura 2.3 como (3)); A função *map* produz resultados intermédios, resultados estes que vão ser guardados no disco local e ordenados (indicado na figura 2.3 como (4)); de seguida procede-se à execução concorrente dos nós que foram designados para a execução da tarefa *Reduce*; estes vão receber uma chave juntamente com um iterador de todos os valores que lhe correspondem e vai agregá-los num só resultado (indicado na figura 2.3 como (5)); A fase final consiste na obtenção dos dados produzidos pelos *reducers* (indicado na figura 2.3 como (6)).

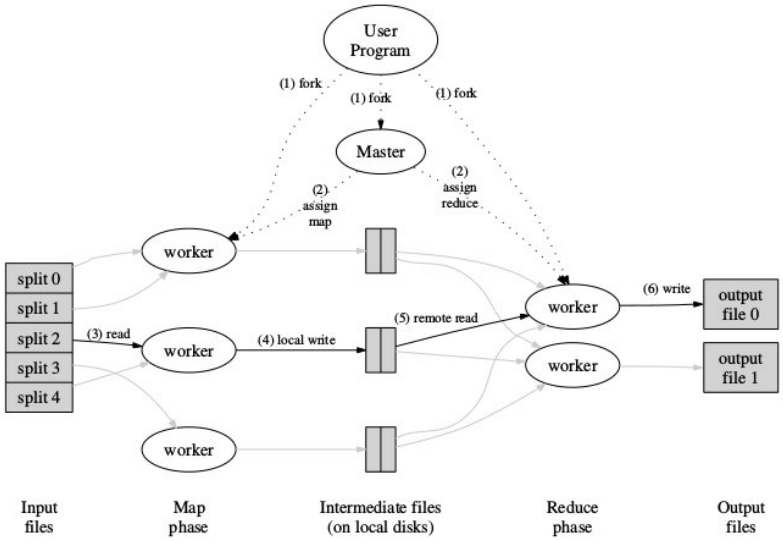


Figura 2.3: Fluxo de execução Hadoop [28]

2.4.3 Spark

Tanto o MapReduce, como as suas variantes que foram criadas ao longo do tempo, têm tido muito sucesso na implementação de aplicações escaláveis e de grandes montantes de dados em *clusters*. No entanto, elas pecam por serem construídas num modelo de fluxo de dados acíclico. Isto é, aplicações que não reutilizam coleções de dados para múltiplas operações em paralelo. A *framework Spark* propõe suportar este tipo de aplicações, mantendo ainda assim a escalabilidade e tolerância a falhas do MapReduce. Para atingir esta finalidade uma das abstrações oferecidas é chamada *resilient distributed datasets* (RDDs). RDD é uma coleção de objetos particionados, só de leitura, por várias máquinas [31].

A utilização de uma abordagem que reutiliza coleções de dados, tal como o Spark, torna possível resolver dois casos para os quais a utilização do MapReduce se revelou ineficiente. Sendo esses os seguintes:

- Execuções iterativas: Muitos dos algoritmos de aprendizagem automática aplicam repetidas vezes a mesma função à mesma coleção de dados para otimizar parâmetros. Sendo cada iteração expressada num processo MapReduce, cada processo vai ter de carregar dados do disco, fazendo com que haja um declínio na performance.
- Análise interativa: O ambiente Hadoop é regularmente usado para pesquisas *ad-hoc* em grandes coleções de dados, por via SQL, com interfaces como *Pig* e *Hive*. O ideal seria o utilizador ser capaz de carregar uma porção da coleção de dados do seu interesse para memória, através de várias máquinas, e pesquisar sobre essa coleção as vezes que bem entender. No entanto, no Hadoop, cada pesquisa ocorre com alguma latência porque executa como um processo MapReduce e lê dados do disco.

Para usar Spark, os utilizadores escrevem um programa principal (*driver program*) que implementa o controlo de alto nível do fluxo da aplicação, e lança as várias operações em paralelo. Este ambiente fornece duas abstrações principais: *resilient distributed datasets* e operações paralelas. Os elementos de um *resilient distributed datasets* (RDD) não necessitam de existir em armazenamento persistente, porque um identificador de RDD contém informação suficiente para calcular o RDD a partir de dados em armazenamento confiável. Isto quer dizer que o RDD pode sempre ser reconstruído se um nó que o continha falhou. Existem dois tipos de operações paralelas que podem executar sobre os RDDs:

- *transformações*, que vão criar um novo *dataset* (RDD) de um já existente. Exemplo: *map(func)*, *mapPartitions(func)*, *reduceByKey(func, [numTasks])*;
- *ações*, que retornam os valores ao *driver program* depois da computação ser efectuada no RDD. Exemplo: *reduce(func)*, *collect()*, *saveAsTextFile()*

Este ambiente não suporta uma operação *reduce* análoga à *framework* MapReduce, pois todos os resultados são recolhidos no programa principal. Apesar de existirem pequenas reduções locais nos nós [31]. Este modelo pode ser vantajoso em vários aspetos.

Considerando um exemplo que conta o número de ocorrências de cada palavra em grandes coleções de documentos (retirado de [32]):

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split("␣"))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

A função *map* vai criar um par (String, Integer) para cada palavra que lê, enquanto que a função *reduceByKey* faz uma redução local, somando todos os valores de chaves iguais.

2.4.4 Métricas de Desempenho Paralelo

As seguintes métricas podem ser aplicadas às diferentes execuções das fases do algoritmo (correlação de Pearson 2.5), ou à execução completa do mesmo para avaliar o ganho ao optar por uma execução em paralelo. Tipicamente, o *speedup* da execução de uma implementação paralela é comparado com a execução da melhor implementação sequencial conhecida. No entanto, para processar *corpora* de grandes dimensões juntamente com expressões relevantes, torna complicada a execução sequencial do algoritmo em tempo útil. Para isso recorremos ao uso das métricas de desempenho relativo, definidas com base num número mínimo de vCPUs (CPUs virtuais) capazes de processar o dado *corpus* com as devidas expressões relevantes [33].

2.4.4.1 Métricas de Desempenho Absoluto

Tendo em conta as métricas de desempenho absoluto no contexto deste tema, consideramos o *speedup* e a eficiência da seguinte maneira:

***Speepup* Absoluto:** Definimos o *speedup* absoluto ($Sp_0(K)$) de uma implementação paralela com K vCPUs, relativo a uma implementação ideal sequencial com tempos de acesso nulos de *input* e *output*, da seguinte maneira:

$$Sp_0(K) = \frac{T_0}{T_{paralelo}(K)} \tag{2.10}$$

Onde T_0 representa tempo de execução da implementação sequencial com uma máquina ideal, incluíndo apenas o tempo de computação do algoritmo para um problema de tamanho fixo, sem custos adicionais associados (*overheads*). E o $T_{paralelo}(K)$ é o tempo total de execução com K vCPUs. Caso o $Sp_0(K) = K$ então o *speedup* absoluto diz-se linear.

Eficiência Absoluta: Definimos a eficiência absoluta ($E_0(K)$) nas mesmas condições de cima, com a seguinte equação:

$$E_0(K) = \frac{1}{K} \cdot Sp_0(K) = \frac{1}{K} \cdot \frac{T_0}{T_{paralelo}(K)} \tag{2.11}$$

2.4.4.2 Métricas de Desempenho Relativo

Devido às limitações do uso da implementação sequencial, as métricas que se seguem comparam a implementação paralela quando variamos o número de vCPUs a partir de um certo mínimo. As métricas usadas foram:

***Speedup* Relativo:** Definimos o *speedup* relativo de uma implementação paralela quando passamos de K_1 para K_2 vCPUs ($Sp_{K_1 \to K_2}$) para um problema de tamanho fixo, com a seguinte equação:

$$Sp_{K_1 \to K_2} = \frac{T_{paralelo}(K_1)}{T_{paralelo}(K_2)} \tag{2.12}$$

Onde $T_{paralelo}(K_1)$ representa o tempo total de execução com K_1 vCPUs.

Eficiência Relativa: Definimos a eficiência relativa de uma implementação paralela quando passamos de K_1 para K_2 vCPUs ($E_{K_1 \to K_2}$) para um problema de tamanho fixo, com sendo:

$$E_{K_1 \to K_2} = \frac{K_1}{K_2} * \frac{T_{paralelo}(K_1)}{T_{paralelo}(K_2)} \tag{2.13}$$

***Sizeup* Relativo:** Definimos o *sizeup* relativo de uma implementação paralela quando aumentamos o tamanho do *input*, isto é, para um *corpus* fixo quando aumentamos o número de expressões relevantes de Ers_1 (tamanho $|Ers_1|$) com K_1 vCPUs para um número de ERs Ers_2 (tamanho $|Ers_2|$) com K_2 vCPUs, com a seguinte equação:

$$Szp_{Ers_1 K_1 \to Ers_2 K_2} = \frac{NErs_2(K_2, T)}{NErs_1(K_1, T)} \tag{2.14}$$

Onde $NErs(K, T)$ representa o número total de ERs a correlacionar que foram processados num dado tempo T, usando um número de vCPUs K. Quando $\frac{|NErs_2|}{|NErs_1|} = \frac{K_2}{K_1}$, o do *sizeup* relativo diz-se crescer linearmente.

Capítulo 3

Solução Proposta

O objetivo principal deste tema consistiu em desenvolver dois protótipos que fossem capazes de construir automaticamente descritores de documentos. Para além de *keywords* explícitas, estes descritores deverão conter *keywords* implícitas que possam estabelecer pontes semânticas entre os documentos. A identificação destes elementos implícitos envolve o cálculo de correlações entre pares de expressões relevantes. Considerando que o número destes pares cresce quadraticamente com o número de expressões relevantes do *corpus*, esta computação exige processamento paralelo e distribuído. Como foi referido anteriormente, optou-se pelo coeficiente de correlação de Pearson (secção 2.5).

Este capítulo destina-se primeiramente a apresentar uma solução sequencial, que irá servir para provar necessidade da implementação do coeficiente de Pearson num ambiente paralelo e distribuído. Seguidamente apresentamos duas soluções produzidas em duas *frameworks* distintas com vista a uma posterior comparação de performance.

3.1 Abordagem sequencial

Com vista ao desenvolvimento de dois protótipos finais, foi criada uma implementação sequencial da fórmula de correlação de Pearson. Versão sequencial esta que, para além de nos ajudar a decompor a fórmula desta correlação em sub-funções que podem ser otimizadas, também ajuda a perceber até que ponto e onde é que a fórmula pode ser paralelizável. Para além disso, esta versão sequencial vai servir como base para mostrar a correlação entre as várias expressões relevantes e com vista a uma futura comparação de resultados com os protótipos finais paralelos.

A versão sequencial já foi construída com recurso à linguagem de programação Java. Optou-se por esta linguagem porque oferece uma quantidade considerável de bibliotecas

úteis para ler documentos de texto e extrair/comprimir documentos no decorrer da execução do programa e poder ser usada com os *frameworks* em estudo. O ambiente de teste desta versão é constituído por um *corpus* extraído do Wikipédia [34] que contém 23218 documentos de texto (todos eles comprimidos em modo *.zip*), perfazendo um total de aproximadamente 50 milhões de palavras e com 222 mil ERs. Estas ERs foram extraídas do *corpus* utilizando o algoritmo *LocalMaxs*, como referido acima (final de secção 2.1).

3.1.1 Decomposição da correlação de Pearson

Após a obtenção das ERs, o passo que se seguiu foi decompor a fórmula do coeficiente de correlação de Pearson em sub-funções apresentadas pela equação 2.5. Pretendemos partir a equação em problemas menores, por forma a encontrar possíveis optimizações à execução da fórmula e pontos de paralelismo. Esta fórmula tem como numerador a covariância entre duas ERs (ER_1 e ER_2), e como denominador sobre o produto entre as raízes quadradas das covariâncias de cada ER com elas próprias. Achámos por bem começar por decompor a função $cov(ER_1, ER_2)$ (equação 2.7). O resultado da decomposição está descrito abaixo:

$$cov(ER_1, ER_2) = [A \quad -B + C] \tag{3.1}$$

Onde **A** representa a soma do produto das frequências relativas de ER_1 e ER_2 no mesmo documento; **B**, a subtração entre a soma das frequências relativas de ER_1 nos documentos onde aparece, vezes a frequência relativa média de ER_2, e a soma das frequências relativas de ER_2, vezes a frequência relativa média de ER_1; e **C** representa o produto entre a frequência relativa média de ER_1, a frequência relativa média ER_2 e o número total de documentos no *corpus*.

Podemos observar que na expressão 3.1 falta o primeiro fator relativamente à fórmula 2.7 (o inverso do número total de documentos menos um), pois este fator anula-se na fórmula da correlação (equação 2.5). Assim, não é necessário incluí-lo. Podemos assim, identificar três parcelas distintas para a implementação desta expressão:

A:

$$\sum_{d \epsilon Docs} (f_r(ER_1, d) \times f_r(ER_2, d)) \tag{3.2}$$

B:

$$(f_{rm}(ER_2, \cdot) \times \sum_{d \epsilon Docs} f_r(ER_1, d)) \quad - (f_{rm}(ER_1, \cdot) \times \sum_{d \epsilon Docs} f_r(ER_2, d)) \tag{3.3}$$

C:

$$\sum_{d \epsilon Docs} f_{rm}(ER_1, \cdot) \times f_{rm}(ER_2, \cdot) \tag{3.4}$$

Após a separação da $cov(ER_1, ER_2)$ em várias parcelas, procedeu-se à decomposição do denominador da fórmula de correlação de Pearson. Decomposição esta que resultou em:

$$\sqrt{cov(ER_1, ER_1)} = \sqrt{D - E + F} \tag{3.5}$$

Analogamente à primeira decomposição, também foi retirado o fator correspondente ao inverso do número total de documentos menos um. Podemos também identificar três parcelas em $\sqrt{cov(ER_1, ER_1)}$:

D: soma do quadrado da frequência relativa de ER_1 em cada documento;

$$\sum_{d \epsilon Docs} f_r(ER_1, d)^2 \tag{3.6}$$

E: multiplicação de -2 pela frequência relativa média de ER_1, vezes o número total de documentos;

$$2 \times f_{rm}(ER_1, \cdot)^2 \times |Docs| \tag{3.7}$$

F: a terceira parcela, representa o produto entre o quadrado da frequência relativa média de ER_1, e o número total de documentos.

$$f_{rm}(ER_1, \cdot)^2 \times |Docs| \tag{3.8}$$

3.1.2 Modelo de Programa

A estrutura deste programa vai ser descrita desde a tarefa que só se responsabiliza por contar o número de ocorrências de uma ER no *corpus*, até à operação principal que controla o fluxo de execução das diferentes tarefas, tal como está exibido na figura 3.1. De notar que todos os resultados produzidos pelas diferentes tarefas nesta versão, são escritos para ficheiros de texto.

Tendo em conta as parcelas que foram identificadas a partir da fórmula de Pearson, a versão sequencial dividi-se nas seguintes tarefas:

ERFileCounter: conta o número de ocorrências de uma dada ER para um dado documento;

DocsWordCounter: conta o número total de palavras que um documento contém;

RelativeFreq_ER: calcula a frequência relativa para todas as ERs do *corpus*;

AverageRelativeFreq_ER: calcula a frequência relativa média para todas as ERs do *corpus*;

CovarianceER1_ER1: calcula a covariância para todas os pares de ERs do *corpus*, sendo que o par é formado por duas ERs idênticas;

CovarianceER1_ER2: calcula a covariância para todos os pares de ERs distintas do *corpus*;

CorrelationER1_ER2: calcula o resultado final da correlação de Pearson, para todos os pares de ERs distintas do *corpus*.

Ao longo da construção deste protótipo sequencial, constatou-se que existem precedências entre as várias tarefas, ou seja, existem as que só podem ser executadas após a execução de outras. Como podemos observar na figura 3.1, temos as duas primeiras tarefas (*ERFileCounter* e *DocsWordCounter*) que dependem diretamente do *corpus*, podendo ser executadas em paralelo, pois não dependem dos resultados uma da outra. Segue-se uma parte sequencial: onde a tarefa *RelativeFreq_ER* depende diretamente das duas primeiras; e *AverageRelativeFreq_ER* que depende apenas de *RelativeFreq_ER*. De seguida temos as *CovarianceER1_ER1* e *CovarianceER1_ER2* que dependem dos resultados produzidos pelas tarefas *RelativeFreq_ER* e *AverageRelativeFreq_ER*. Contudo como executam cálculos independentes um do outro pode ser executadas em paralelo. Por fim temos a tarefa *CorrelationER1_ER2*, que necessita dos resultados produzidos por *CovarianceER1_ER1* e *CovarianceER1_ER2*, para poder calcular o resultado final.

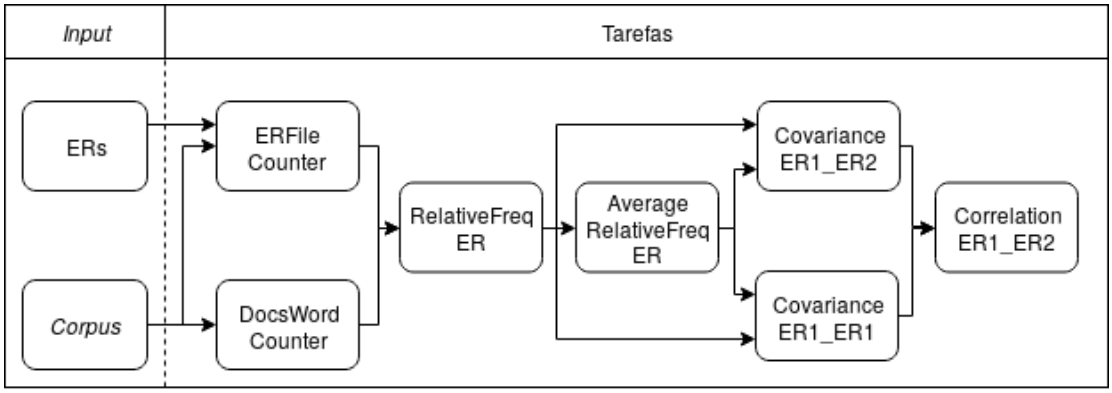


Figura 3.1: Fluxo de execução da versão sequencial

3.1.3 Resultados - Versão Sequencial

Os resultados obtidos da correlação entre duas ERs foram entre -1 e 1, como se esperava. Relembrando que, 1 significa fortemente correlacionado e -1 ERs de assuntos tendencialmente "opostos". Para além da versão sequencial servir como termo de comparação de resultados, também mostra a necessidade de um ambiente de computação paralelo e distribuído, neste caso Hadoop e Spark, para estabelecer o cálculo da correlação de Pearson aplicado a um grande número de pares de ERs.

A versão sequencial foi executada num *desktop* com a versão de 64 bits do Windows 10 Pro, um processador Intel Core i5-4690 3.50GHz e com memória RAM de 8GB DDR4. No decorrer da execução das várias tarefas deste programa, algumas revelaram ser computacionalmente pesadas. Para suportar este facto, registaram-se tempos de execução das várias tarefas. Como podemos observar no gráfico 3.2, os tempos de execução das

tarefas *DocsWordCounter*, *RelativeFreq_ER*, *AverageRelativeFreq_ER*, e *CovarianceER1_ER1* revelam que são as mais leves.

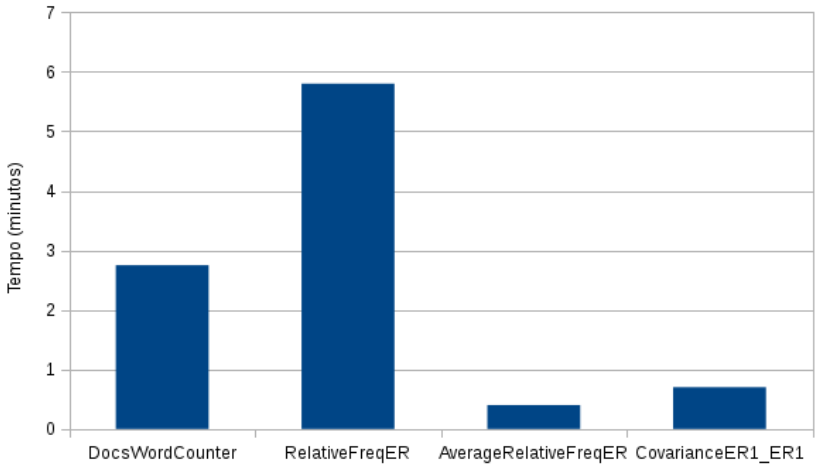


Figura 3.2: Tarefas com os menores tempo de execução

As tarefas que iram ser descritas a seguir, não foram colocadas no gráfico 3.2 porque a escala faria desaparecer as restantes. A tarefa *ERFile_Counter* registou um tempo de execução de aproximadamente 2 dias e meio, pois para cada documento é necessário calcular a frequência de cada ER. Na tarefa *CovarianceER1_ER2* foi estimado 14 segundos como o tempo de cálculo da covariância entre cada par de ERs. Para a tarefa *CorrelationER1_ER2*, estimou-se que o cálculo de uma correlação entre duas ERs demorasse cerca de 27 segundos; É de notar que, os tempos referidos de 14 e 27 segundos resultam de um programa ainda não optimizado. Torna-se claro que uma implementação sequencial como esta não conseguirá resultados em tempo útil quando o número de documentos ou ERs aumentar.

3.1.4 Optimizações encontradas - fórmula da Correlação de Pearson

Após verificarmos que a versão sequencial não optimizada apresenta tempos de execução fora do espectro do tempo de execução útil, voltámos a rever a fórmula da correlação de Pearson, de modo a encontrar possíveis optimizações matemáticas. Todos os atalhos encontrados visam reduzir ao máximo a computação da mesma. Conseguimos então optimizar duas sub-fórmulas da correlação, $cov(ER_1, ER_2)$ e $cov(ER_1, ER_1)$.

Vendo a covariância entre duas expressões relevantes distintas, referida em 3.1, apresentada a seguinte disposição:

$$\begin{aligned} cov(ER_1, ER_2) = [& \sum_{d \epsilon Docs} (f_r(ER_1, d) \times f_r(ER_2, d)) \\ & - (f_{rm}(ER_2, \cdot) \times \sum_{d \epsilon Docs} f_r(ER_1, d)) \quad - (f_{rm}(ER_1, \cdot) \times \sum_{d \epsilon Docs} f_r(ER_2, d)) \\ & + (\sum_{d \epsilon Docs} f_{rm}(ER_1, \cdot) \times f_{rm}(ER_2, \cdot))] \end{aligned} \tag{3.9}$$

Após terem sido aplicadas técnicas de factorização, foram encontradas optimizações para $cov(ER_1, ER_2)$, esta é apresentada de maneira diferente 3.10.

$$cov(ER_1, ER_2) = \frac{1}{|Docs|} \cdot \sum_{d \epsilon Docs} (f_r(ER_1, d) \times f_r(ER_2, d)) - (f_{rm}(ER_1, \cdot) \times f_{rm}(ER_2, \cdot)) \tag{3.10}$$

Já a fórmula da covariância entre a mesma expressão relevante, referida em 3.5, tem a seguinte disposição:

$$\sqrt{cov(ER_1, ER_1)} = \sqrt{(\sum_{d \epsilon Docs} f_r(ER_1, d)^2 - (2 \times f_{rm}(ER_1, \cdot)^2 \times |Docs|) + (f_{rm}(ER_i, \cdot)^2 \times |Docs|)} \tag{3.11}$$

Na covariância demonstrada pela equação 3.11, posteriormente a serem aplicadas técnicas de factorização, apresenta-se da seguinte maneira:

$$cov(ER_1, ER_1) = \frac{1}{|Docs|} \times [\sum_{d \epsilon Docs} f_r(ER_i, d)^2] - f_{rm}(ER_i, \cdot)^2 \tag{3.12}$$

3.1.5 Complexidade Temporal - Correlação de Pearson com optimizações

É apresentada nesta secção uma análise preliminar da complexidade temporal da fórmula da correlação de Pearson já com as optimizações aplicadas. Esta análise tem como objetivo mostrar a influência temporal das diferentes fases da fórmula numa execução completa, com um problema de tamanho fixo. Isto é, verificar a complexidade temporal das várias funções implícitas ao cálculo do coeficiente de Pearson, para um *corpus* de tamanho fixo e um determinado número de expressões relevantes a correlacionar. Esta análise não tem em conta custos adicionais associados (*overheads* de *Java virtual machine JVM*, sistema operativo, escritas e leituras do disco, entre outros).

Os conceitos base são explicados na secção 3.1.5.1. A complexidade temporal de cada uma das fases que implicam o cálculo do coeficiente de Pearson são descritas na secção 3.1.5.2.

3.1.5.1 Conceitos base

Cada fase da implementação foi analisada em separado. Sabendo que o fluxo deste programa é sequencial, o tempo total de execução (T_{exec}) do mesmo, sem *overheads*, é dado pela seguinte expressão 3.13:

$$T_{exec} = T_{allRelFreq} + T_{allAVGRelFreq} + T_{AllCov} + T_{allPairsCorrelation} \tag{3.13}$$

Os tempos computacionais são representados por 4 termos: $T_{allRelFreq}$ que é o cálculo da frequência relativa para todas as expressões relevantes; $T_{allAVGRelFreq}$, cálculo da frequência relativa média para todas as ERs; T_{AllCov} representa tanto o cálculo da

$cov(ER_1, ER_1)$ (3.12) como da $cov(ER_1, ER_2)$ (3.10); Por fim $T_{allPairsCorrelation}$ que é o cálculo efetivo do coeficiente da correlação de Pearson para todos os pares (C_2^{Ner}) de ERs. A expressão (3.13) é aplicada a uma máquina sequencial ideal, $K = 1$.

3.1.5.2 Tempo de computação

O tamanho do problema que pretendemos resolver cresce com o quadrado do número pares de ERs a correlacionar. O seu limite máximo que pode ser calculado (*upper bound* do problema) é dado por: $C_2^{Ner} = \frac{Ner(Ner-1)}{2}$, em que Ner é o número total de expressões relevantes. De notar que todas as operações básicas apenas executadas uma vez são desprezadas no tempo total de execução ($T_{divisao}$, $T_{multiplicacao}$, $T_{subtracao}$, e T_{soma}).

$T_{allRelFreq}$, sendo o tempo total da contagem de todas as ERs em todos os documentos, e sendo $T_{fr(ER_1,d))}$ o tempo de cálculo da frequência relativa de uma ER num documento de tamanho médio, temos que:

$$T_{allRelFreq} = T_{fr(ER_1,d)} \times Nd \times Ner \tag{3.14}$$

sendo Nd o número total de documentos e Ner o número total de expressões relevantes.

$T_{allAVGRelFreq}$ é modulada com o produto do número total de ERs envolvidas no cálculo e o número total de documentos de texto do *corpus* a processar. A complexidade temporal envolvida é a seguinte, considerando uma abordagem não optimizada:

$$T_{allAVGRelFreq} = Ner \times Nd \times T_{fr(ER_1,d)} + t_{divisao} \tag{3.15}$$

T_{AllCov} é modulada com o produto do número total de combinações de pares de ERs. Em que $T_{cov_{1par}}$ apresenta a complexidade temporal para calcular a covariância para um par de ERs da seguinte maneira:

$$T_{cov_{1par}} = Nd \times T_{multi+soma} + T_{divisao} + T_{multi} + T_{subtracao} \approx Nd \times T_{multi+soma} \tag{3.16}$$

$T_{allPairsCorrelation}$ é modulada com o produto de C_2^{Ner}. Apresenta a seguinte complexidade temporal:

$$T_{allPairsCorrelation} = C_2^{Ner} \times (3 \times T_{cov_{1par}} + T_{divisao} + T_{multiplicao}) \approx C_2^{Ner} \times (3 \times T_{cov_{1par}}) \tag{3.17}$$

Após discriminar todos os componentes da expressão 3.13, a complexidade temporal que esta apresenta perante um *corpus* de tamanho fixo, com um número total de documentos (Nd), e um número total de expressões relevantes (Ner), é a seguinte:

$$\begin{aligned} T_{exec} &= Ner \cdot Nd \cdot (T_{fr(ER_1,d)}) + Ner \cdot Nd \cdot (T_{fr(ER_1,d)}) + C_2^{Ner} \cdot (3 \times T_{cov_{1par}}) \\ T_{exec} &= Ner \cdot Nd \cdot (T_{fr(ER_1,d)}) + Ner \cdot Nd \cdot (T_{fr(ER_1,d)}) + \frac{3Ner^2}{2} \times Nd \times T_{multi+soma} \end{aligned} \tag{3.18}$$

Assumindo que $Ner^2 \times Nd \times T_{multi+soma} >> Ner \cdot Nd \cdot (T_{fr(ER_1,d)})$, a expressão 3.18 apresenta uma complexidade temporal de $O(Ner^2)$, o tempo total de uma execução do coeficiente de Pearson, numa máquina ideal, sem custos adicionais associados (*overheads*).

3.2 Abordagem Paralela

Para resolver a problemática da correlação estatística, assumindo sempre *datasets* (*corpora* linguísticos) de grandes dimensões, elaborámos uma solução baseada em duas das *frameworks* de processamento paralelo mais usadas e também bem documentadas. Spark e Hadoop MapReduce.

A meta que estas duas implementações pretendem atingir consiste em resolver com eficiência um dos problemas criados pelo crescimento do volume de dados. No âmbito deste problema, esta resolução assenta no cálculo em paralelo das frequências das ERs nos documentos e das correlações entre grandes quantidades de pares de expressões relevantes.

3.2.1 Protótipo Hadoop MapReduce

A solução produzida foca-se na implementação Hadoop que segue o paradigma MapReduce, preocupando-nos apenas em como processar *corpora* linguísticos de grandes dimensões de maneira eficiente.

MapReduce implementa um padrão paralelo popular tendo como umas das suas principais características, *key partitioning*. Este paradigma é versátil ao ponto de permitir que cada chave seja processada ou transformada por qualquer tipo de função. Fornecendo assim a noção de independência de chaves a um nível de abstração alto. Isto pode ser visto na fase de *Map*, onde a cada *input* que recebe aplica-lhe uma função escrita pelo utilizador e emite um par chave valor, (K,V), escrevendo-o para disco. Este mecanismo permite que os dados sejam manipulados livremente pelo *cluster*. Uma das razões pela qual MapReduce é tolerante a falhas (*fault-tolerant*), deve-se ao facto de os resultados serem escritos para disco. Visto que a problemática a resolver tem inerente a quantidade associada à combinatória que é gerada entre expressões relevantes, o facto de os resultados serem escritos para disco à medida que são processados no momento é mais vantajoso do que os manter em RAM. Para além disto, esta *framework* permite que cada transformação (*map*) seja independente, podendo ser recalculada caso falhe ou venha a perder os resultados. Como por exemplo, o Hadoop faz isso se detectar falhas de máquinas.

Para calcular a correlação estatística, esta implementação divide-se em dois trabalhos MapReduce:

- **1º trabalho:** (descrito na figura 3.4) conta as ocorrências de todas as ERs em cada documento; calcula a frequência relativa de cada ER no respetivo documento; calcula a frequência relativa média para cada ER no *corpus*; calcula a covariância para um par de ERs iguais, $Cov(ER_1, ER_1)$; valores intermédios ao cálculo da $Cov(ER_1, ER_2)$.

- **2º trabalho:** (descrito na figura 3.5) Lê o *output* produzido pela 1ª tarefa, calcula valores intermédios do cálculo da $Cov(ER_1, ER_2)$, calcula os valores finais para as duas covariâncias. E por fim calcula o resultado da correlação entre duas expressões relevantes.

3.2.1.1 Descrição Detalhada - 1º Trabalho MapReduce

Na fase de *Mapping*, a cada instância da classe Map implementada, existem mais duas funções para além da principal map(). Estas ocorrem sempre numa ordem específica: setup() -> map() -> cleanup(); O setup() é chamado apenas uma vez antes do início da tarefa; o cleanup() é chamado uma vez antes do final da tarefa. Tipicamente estas duas funções são usadas para carregar dados para a *cache* das tarefa *Map/Reduce*, ou fazer ligações a bases de dados, entre outras funções. Já o cleanup() é usado para o "inverso" da função setup(), fechar conexões a bases de dados, fechar escritas de ficheiros, entre outros. Tendo como base o que foi descrito acima, utilizou-se o setup() para que cada tarefa de *Map* vá ter carregado para *cache* um ficheiro que contém todas as expressões relevantes a correlacionar no *corpus*.

Depois do setup() a função que se segue é a map(). Tipicamente o *input default* desta função está no formato *TextInputFormat (key, value)*, em que a *key* corresponde ao *offset* do início de cada linha e o *Value* ao conteúdo da linha em si. Este trata cada linha de cada ficheiro de *input* de forma separada. Ou seja, este tipo de formato é útil para dados desformatados ou ficheiros de *log*. Primeiramente começou-se por usar este formato, mas rapidamente concluímos que seria melhor construir um formato próprio para o contexto do nosso problema. Para isso criou-se o *WholeFileInputFormat(key, value)*, em que a Key corresponde ao nome do documento a processar e o Value ao texto em si. Este formato permite que cada ficheiro que é dado como *input* não seja dividido em várias linhas pelas várias tarefas de *Mapper*. Assim sendo cada tarefa *Map* irá processar unicamente um ficheiro por completo. Ou seja, irá ser criado o número de instâncias/tarefas de *Mappers* igual ao número total de documentos de texto no *corpus*. O processamento é feito desta maneira, pois o objetivo é extrair o máximo de informação de cada documento e expressões relevantes, processando cada ficheiro uma só vez.

Tal como descrito na figura 3.3, após da função map() receber o devido *input*: conta as ocorrências de todas as ERs no documento; calcula a frequência relativa de todas as ERs que foram mapeadas nesse documento, e por fim calcula resultados intermédios para o cálculo da frequência relativa média das ERs, para a $Cov(ER_1, ER_2)$, e $Cov(ER_1, ER_1)$. O map() emite dois tipos de par (K,V) intermédios: ("SUM_ER1,ER1", DoubleWritable) e ("MULTI_ER1,ER2", DoubleWritable). Segue-se a fase do *Reduce*.

Tal como na fase de *Mapping*, numa tarefa de *Reduce*, também existem as funções setup(), cleanup(), e são executadas exatamente na mesma ordem (setup() -> reduce() -> cleanup()). Tendo isto em conta, nesta tarefa a função setup() é utilizada para abrir a escrita de dois ficheiro do tipo *SequenceFile*. Num deles irá ser escrito o resultado final da

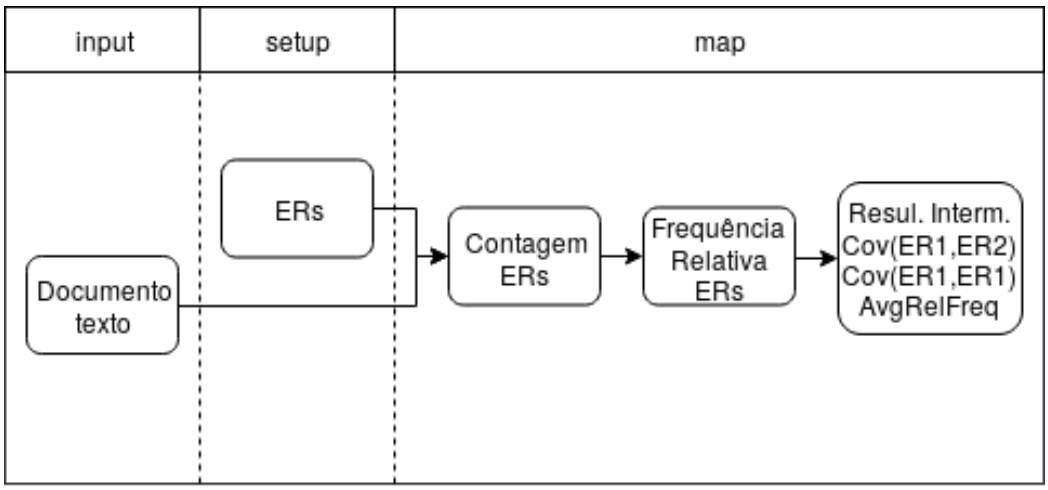


Figura 3.3: Protótipo MapReduce - Primeiro Trabalho, Fase de *Map*

frequência relativa média de todas as ERs mapeadas pela fase de *Mapping*; O outro vai servir para escrever os resultados finais da covariância entre a mesma ER ($Cov(ER_1, ER_1)$).

Este trabalho MapReduce (descrito na figura 3.4) foi implementado com apenas uma tarefa *Reduce*. No entanto na secção 5.1 indicamos um possível melhoramento acerca deste aspecto.

Assim sendo, o *Reduce* vai receber todos os pares (K,V) intermédios produzidos pelas tarefas de *Mapping*. Quando a função reduce() recebe uma chave do tipo "SUM_ER1,ER1", calcula o resultado final tanto da frequência relativa média, como da $Cov(ER_1, ER_1)$, e escreve-os para os *SequenceFile's* respectivos. Ao receber uma chave do tipo "MULTI_ER1,ER2" a função reduce() irá simplesmente agregar todos os valores correspondentes à respetiva chave e escrever para disco na respetiva forma, ("SUM_COV(ER1,ER2)", DoubleWritable). O formato do *output* escrito pelo reduce() é o *default*, *TextOutputFormat*. Este escreve um par *(key, value)* em cada linha para um ficheiro de texto, em que a chave é escrita separada do valor, pela expressão regular "Tab". Por fim a função cleanup() irá apenas fechar a escrita dos dois *SequenceFile's*.

3.2.1.2 Descrição Detalhada - 2º Trabalho MapReduce

A fase de *Map*, inicialmente na função setup() irá ser carregado para memória o *SequenceFile* correspondente à frequência relativa média de cada expressão relevante.

Como o *output* da função reduce() do 1º trabalho é do tipo *TextOutputFormat*, consequentemente o *input* da função map() deste trabalho é *KeyValueTextInputFormat(K,V)*. Este formato é similar ao *TextInputFormat*, também trata de cada linha do ficheiro de forma separada. No entanto o *KeyValueTextInputFormat(K,V)* difere num aspeto. Este, vai partir a linha dada como *input* pela expressão regular "Tab". À primeira parte corresponde a chave e ao restante corresponde o valor. Exemplo: (SUM_COV(ER1,ER2) DoubleWritable).

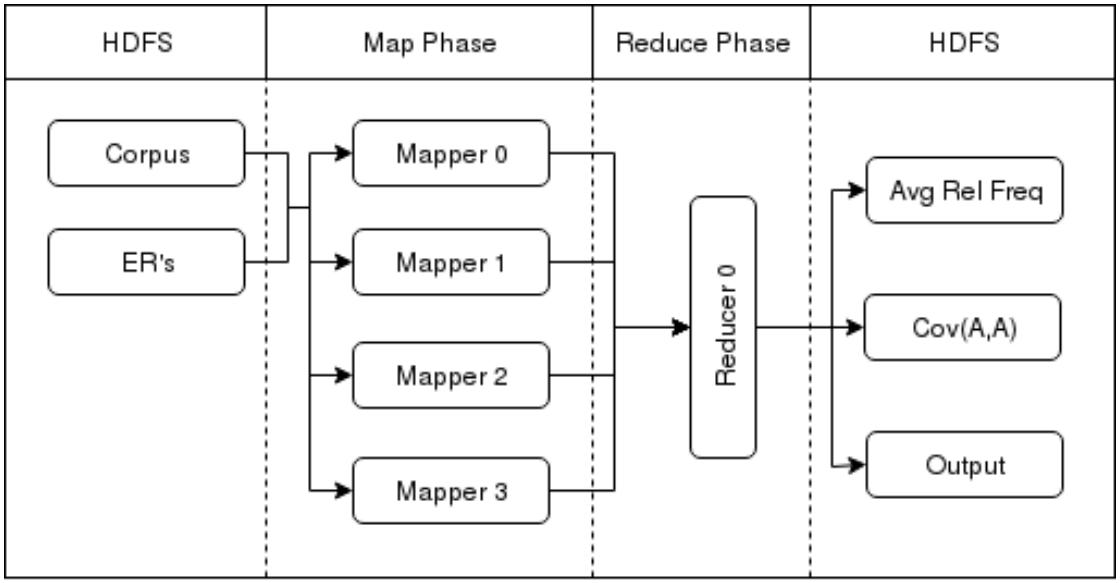


Figura 3.4: Protótipo MapReduce - Primeiro Trabalho

Posto isto, a função map() calcula a covariância entre duas ERs distintas, para cada par *(Key,Value)* que recebe como *input*. Este cálculo é realizado com o auxilio de valores extraídos do ficheiro que está em *cache*. Após o cálculo da covariância é emitido um par (K,V) intermédio, ("COV_ER1_ER2", DoubleWritable).

Na fase de *Reduce*, o primeiro método a ocorrer é o setup(). Nele é carregado para memória o *SequenceFile* correspondente à covariância entre expressão relevante e ela própria, ou seja a variância, $Cov(ER_1, ER_1)$. A função reduce(), a cada chave que recebe, vai agregar os seus valores, e calcular o resultado final da fórmula da correlação de Pearson entre duas expressões relevantes. Escreve então estes resultados para disco com o formato *TextOutputFormat*.

3.2.2 Protótipo Spark

A solução desenhada para Spark, produzida em Scala, privilegia a utilização de RAM como modo de acelerar o processamento, em vez de emitir escritas intermédias para disco como modo de precaver tolerância a falhas (*fault tolerance*). Para tal utiliza o modelo de dados Resiliant Distributed Datasets(RDD's).

O RDD não executa as transformações que lhe são dadas, até ao momento em que recebe uma ação, tendo a característica de execução retardada (*Lazy Evaluation*). Todas as dependências associadas ao RDD vão ser guardadas num grafo acíclico direcionado, independentemente dos dados [35]. Aquando a criação de um RDD novo (aplicação de transformação ou acção) a partir de um RDD existente, agrega-se a este um apontador para o RDD anterior. Linhagem de um RDD é no fundo, um grafo com todas as dependências dos RDDs pais para ele. Tal como descrito na figura 3.6.

Privilegiando o uso da RAM (*Random-Access-Memory*) à utilização do disco, contribuí

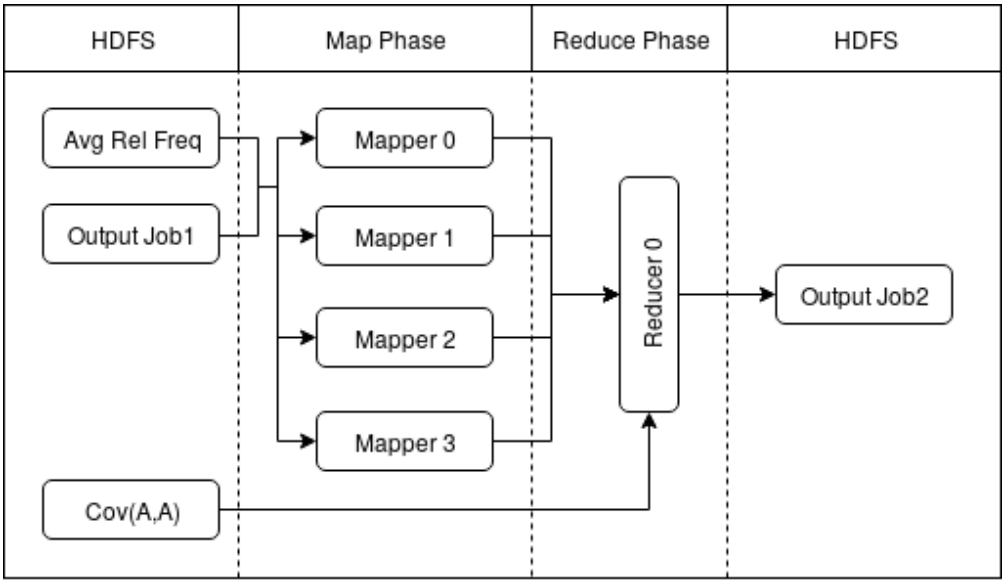


Figura 3.5: Protótipo MapReduce - Segundo Trabalho

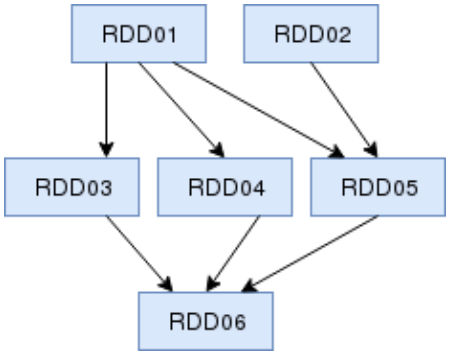


Figura 3.6: Grafo de linhagem dos RDDs

para uma solução mais eficiente, visto que os acessos à RAM são muito mais rápidos. No entanto, a RAM torna-se um recurso escasso, devido a ser mais escasso do que as outras alternativas de armazenamento. Portanto guardar conjuntos de dados *in-memory* levanta uma restrição relativamente ao espaço. Esta restrição torna-se mais evidente quando desejamos resolver problemas que crescem com o quadrado do número de ERs num *corpus*, que geram um grande volume de dados.

3.2.2.1 Operações em *Resilient Distributed Datasets*

Tal como referido na secção 2.4.3, há 2 tipos de operações que se podem aplicar num RDD. Vamos apenas dar ênfase nos métodos utilizados para esta solução e compará-los.

Existem quatro variantes que traduzem a operação de *map* sobe um dado RDD, recebendo como parâmetro uma função que rege a transformação, e executando iterativamente sobre cada elemento:

Map é aplicado ao escopo do RDD, produzindo apenas um resultado para cada elemento;

FlatMap similar ao Map, mas pode produzir múltiplos resultados para cada elemento;

MapPartition é aplicado a cada partição de um RDD, produzindo uma sequência de resultados por partição;

MapPartitionsWithIndex Proporciona o mesmo que o MapPartition, mas inclui juntamente com o resultado de cada partição, o seu respetivo índice no RDD.

Para o processamento de grandes quantidades de dados, o método de *mapPartitons* revelou ser o mais indicado, pois consegue aplicar a função ao nível de cada partição. Enquanto que os métodos *map* e *flatMap* aplicam a função no nível de cada elemento. Em termos práticos, isto quer dizer que se existir um RDD com 10000 elementos numa das suas partições, o *map* irá chamar a função utilizada pela transformação 10000 vezes. Por outro lado, se usarmos o *mapPartitions*, chamamos apenas uma vez a função de transformação. Mas no entanto passamos como *input* os 10000 elementos dessa partição e recebemos todas as transformação feitas em apenas uma chamada da função. Como isto causa impacto na performance do nosso problema, optámos então por utilizar o *mapPartitions*. Para corroborar a utilização deste, foi testada a diferença de performance entre *map* e *mapPartitions*. Reparamos que ao executar a mesma transformação num dado RDD, utilizando os dois métodos descritos a cima, o *mapPartitions* revelou ser sempre mais rápido cerca de 30%.

São suportadas ainda transformações de agregação. *ReduceByKey(func)* foi o método utilizado para agregar os nossos dados nesta solução. Quando este método é chamado num *RDD(Key, Value)*, retorna outro *RDD(K,V)* onde os valores de cada chave são agregados segundo uma função *reduce func*. O *groupByKey()* por outro lado também consegue produzir exatamente o mesmo resultado, no entanto não oferece tão boa performance. A razão pela qual isto acontece deve-se ao facto do *reduceByKey* combinar os valores de cada chave para cada partição antes de enviar os dados para a rede, isto faz com que na fase de *shuffle* sejam enviados menos dados. *Shuffle* consiste numa operação de muitos-para-muitos (NxM), durante a qual vai ler a informação de todas as partições e encontrar todos os valores correspondentes a cada chave, agregando-os. Já o *groupByKey* manda todos os pares (K,V), para a fase de *shuffle*, sem os combinar primeiro dentro de cada partição (bloco) de dados, isto faz com que haja muita informação a ser enviada para a rede sem qualquer necessidade.

As ações [36] utilizadas neste protótipo foram:

Collect retorna todos os elementos de um RDD para um vetor alocado no *driver program*;

CollectAsMap retorna todos os elementos de um RDD para um mapa alocado no *driver program*;

Count conta o número total de elementos de um RDD para uma variavél no *driver program*;

SaveAsTextFile escreve todos os elementos pertencentes a um RDD para um ficheiro de texto, dada uma diretoria (sistema de ficheiros local, HDFS, entre outros).

3.2.2.2 Descrição detalhada - Protótipo Spark

Analogamente ao protótipo MapReduce, esta solução visa processar eficientemente *corpora* linguísticos de grandes dimensões. Com o objetivo de calcular a correlação estatística entre duas expressões relevantes, a implementação faz uso de vários RDD's interligados entre si. A versão do Spark utilizado foi a 2.3.0, recorrendo a Scala, na versão 2.11.8.

O Spark consegue criar dados de forma distribuída a partir de qualquer tipo de sistema armazenamento de dados suportado pela Apache Hadoop, incluindo o sistema de ficheiros local, HDFS, Cassandra, Amazon S3, entre outros. Como *input* o Spark suporta *SequenceFiles*, ficheiros de texto, ou outro *InputFormat* do Hadoop.

Tendo isso em conta, foram usados dois métodos diferente para conseguir carregar todos os dados necessários para fazer os cálculos da correlação. O *SparkContext.textFile(path)* que cria um RDD a partir de um *input* do tipo *URI*, como por exemplo: file:///, hdfs://, s3a://. Ao criar um RDD com este método ele irá retornar uma linha de cada vez por cada ficheiro, a framework irá criar uma partição por cada bloco do ficheiro. No HDFS os blocos têm de tamanho *default* 128MB, que foi o sistema de armazenamento de dados utilizado para suportar este protótipo de Spark. A outra função usada foi *SparkContext.wholeTextFiles(path)*, esta que lê uma diretoria com múltiplos ficheiros. Ao contrário do *textFile()* este retorna um par (nome do ficheiro, texto) para cada ficheiro.

Os RDDs criados para o processamento da correlação de Pearson foram:

AllERs: possuí todas as expressões relevantes a correlacionar;

AllDocs: guarda todos os documentos de textos pertencentes ao *corpus* linguístico;

CounterAllWordsOnDoc: conta o total de palavras em cada ficheiro de texto pertencente ao *corpus*;

ERCounter: conta o número total de ocorrências de todas as expressões relevantes encontradas para cada ficheiro de texto;

MultiRelFreq: calcula valores intermédios para a covariância entre duas ERs diferentes. Após esse cálculo, é-lhe aplicado um método de agregação, *reduceByKey*;

SumRelFreq: calcula valores intermédios para a obtenção da frequência relativa média de todas as ERs. Posteriormente aplicamos a transformação *reduceByKey*;

SumPowRelFreq: mapeia todos os valores para o seu quadrado e aplica o *reduceByKey*. Estes valores irão servir como à covariância entre uma expressão relevante e ela própria;

AVGRelFreq: produz o valor final para a frequência relativa média de todas as ERs;

Cov(A,A): computa o resultado final para a covariância entre uma expressão relevante e ela própria.

Correlation(A,B): calcula o valor final da covariância entre duas expressões relevantes diferentes. Em seguida produz o *output* final, sendo ele a correlação estatística de Pearson.

Todos os RDD's acima descritos recorrem ao uso do *mapPartitions*, à excessão de: AllERs; AllDocs; que usam *textFile* e *wholeTextFiles*, respectivamente.

A imagem 3.7 representa o grafo que contém todas as dependências dos RDDs descritos nesta secção. A descrição das diferentes dependências irá ser feita dos RDDs pai até ao RDD que efectua a última transformação no protótipo.

AllERs e AllDocs não dependem de outros porque são eles que guardam os dados ainda por processar, representando assim a posição de RDDs pai. Há 2 RDDs diretamente dependentes do AllDocs: CounterAllWordsOnDoc e ERCounter. Apesar do ERCounter depender também de AllERs.

Os próximos cálculos a serem efectuados são o cálculo da frequência relativa para cada expressão relevante encontrada em cada documento de texto, e a produção de valores intermédios para algumas das sub-fórmulas usadas para a correlação de Pearson. Estes cálculos irão ser produzidos a partir da criação de dois RDD's: MultiRelFreq e SumRelFreq. Estes possuem as mesmas dependências, sendo elas: ERCounter e CounterAllWordsOnDoc. Existe também a possibilidade de produzir os mesmos resultados de outra forma. Passaria pela criação de um RDD capaz de produzir os resultados intermédios que o MultiRelFreq, e o SumRelFreq produzem. Posteriormente a isso teria de ser aplicada uma transformação do tipo *filter(func)*, filtrando assim e separando as chaves para dois RDDs separados. Como o problema a resolver trata uma grande quantidade de dados, não seria viável estar a percorrer duas vezes de seguida o mesmo RDD para o separar em dois. Isto iria oferecer uma complexidade temporal ao protótipo desnecessária. A depender directamente de SumRelFreq, temos AVGRelFreq e SumPowRelFreq. Dependendo destes está o RDD Cov(A,A). Por fim o último RDD a ser formulado foi o Correlation(A,B). Este depende de Cov(A,A); AVGRelFreq; e MultiRelFreq.

3.3 Construção de descritores de documentos

Tal como descrito na secção 1.3 do capítulo 1, para a construção de descritores de documentos, são necessários dois tipos de *keywords* (expressões relevantes e unigramas relevantes): *keywords* explícitas e *keywords* implícitas. Para a obtenção das explícitas aplicamos a seguinte métrica:

$$Tf - idf(ER_i, d) \times Mediana(ER_i) \tag{3.19}$$

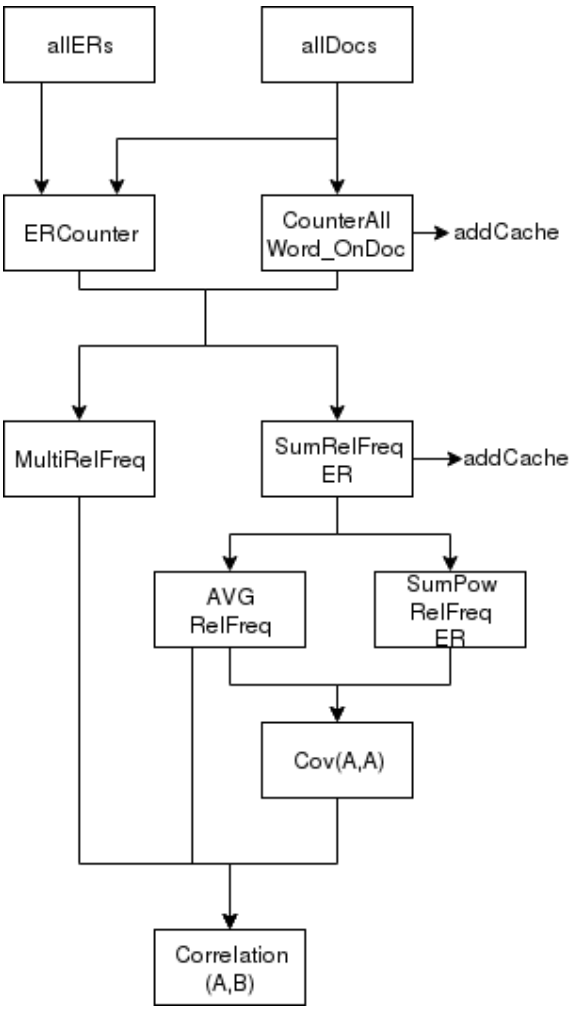


Figura 3.7: Grafo de linhagem - Protótipo Spark

Sendo que

$$Tf - idf(ER_i, d) = fr(ER_i, d) \times \log(\frac{|Docs|}{|d \epsilon Docs : f(ER_i, d) > 0|}) \tag{3.20}$$

O termo $Mediana(ER_i)$ corresponde à mediana dos comprimentos das palavras na expressão relevante, ER_i. A este propósito, sabemos que as palavras que ocorrem com elevada frequência nos documentos (artigos, preposições e pronomes) tendem a ser pouco informativas e consequentemente, por razões de economia de esforço de comunicação, tendem a ser de baixo comprimento. Assim, ao usar a mediana dos comprimentos das palavras - em vez da média que poderia ser uma alternativa -, a métrica definida em 3.19 torna-se mais robusta à ocorrência de *outliers*, o que se torna vantajoso, como veremos a seguir. O efeito desta componente ($Mediana(ER_i)$) é desvalorizar as ERs pouco informativas - parte delas resultantes da imperfeição do extrator de ERs - . Com efeito, a mediana dos comprimentos das palavras na ER pouco informativa, por exemplo, "no caso de" é 2. - Se fosse usada a média, este valor seria 2.66 - . Por outro lado a mediana aplicada à ER informativa, por exemplo, "Presidente da Republica" tem o valor de 9 - se fosse aplicada

a média o valor seria 7. Em resumo, a inclusão da mediana na expressão 3.19 valoriza as ERs mais informativas e desvaloriza as que têm uma semântica mais fraca, reforçando assim o efeito que já é parcialmente conseguido com o $TF - IDF$.

Quanto às *keywords* implícitas são obtidas através da correlação entre as várias *keywords* explícitas de cada documento com as diferentes ERs de todo o *corpus*, através da fórmula da correlação de Pearson (mostrada em 2.5).

Ou seja, para cada documento vão ser atribuídas *keywords* explicitas, correspondentes às ERs mais relevantes cada uma irá ser correlacionada com todas as outras ERs que não pertencem a esse documento para definir *keywords* implícitas. Para tal, escolhemos os pares de ERs que têm uma correlação forte entre elas (valores tipicamente acima de 0,3), perfazendo um descritor implícito de 8 *keywords* implícitas. Assim sendo, por cada documento iremos obter 16 *keywords*, 8 explícitos e 8 implícitos.

Todas as expressões relevantes que foram utilizadas para o cálculo da correlação de Pearson, a fim de obter as *keywords* implícitas, foram extraídas do algoritmo LocalMax (secção 2.1). Tendo em conta que o desejável passaria por fornecer como *input* para a seleção de *keywords* implícitas, apenas ERs com uma semântica forte, isso não foi possível, pois estamos limitados à precisão associada à extracção do algoritmo LocalMax.

Capítulo 4

Avaliação

Neste capítulo começamos por apresentar uma análise entre as expressões relevantes e a correlação de Pearson (secção 4.1). Seguido dos ambientes de teste em que as implementações paralelas foram executadas (secção 4.2). Resultante dos dados extraídos dos ambientes descritos, seguem duas secções que expõem a avaliação/análise feita aos dois protótipos, uma sobre o MapReduce (secção 4.3), e outra em relação ao Spark (secção 4.4). Por fim está a secção onde apresentamos resultados acerca da construção de descritores de documentos (secção 4.5).

4.1 Relação entre extracção de ERs e o crescimento da combinatória da correlação de Pearson

É sabido que o número de palavras distintas nos textos cresce de forma sub-linear com o tamanho dos *corpora*. Para além disso, tendo em conta que embora o vocabulário das línguas naturais evolua, ele é finito considerando um espaço de tempo de pelo menos alguns anos. Consequentemente, se o número de palavras distintas cresce sub-linearmente até a um limite (o tamanho do vocabulário), isto implica que o número de ERs extraídas dos *corpora* tende a ter um comportamento semelhante, já que estas são constituídas pelas palavras distintas. Este comportamento compensa favoravelmente a sub-linearidade do Speed-up e o Size-up da implementação devido aos custos adicionais associados (*overheads*).

Nos cenários de teste descritos em 4.2, não existe qualquer tipo de pre-processamento sobre as expressões relevantes usadas. Sabendo que algumas destas contêm caracteres especiais ($, -, #, entre outros), optamos por não os retirar das ERs. Por este motivo as correlações poderão levar-nos a expressões implícitas com estes caracteres. Tendo em conta que este tipo de pre-processamento é computacionalmente pesado, optamos por não o

fazer. Sendo assim todas as ERs são processadas em *run-time* nos protótipos, por razões de coerência. Ou seja, num caso onde exista no ficheiro que contêm todas as expressões relevantes, a ER "Global Warming" e também "global warming", apesar destas serem dois conjuntos de palavras idênticos para efeitos de pesquisa, têm de ser transformadas para letras minúsculas para que sejam contadas como a mesma ER. Este é um problema dos dados de *input* devido ao uso do LocalMax sem qualquer tipo de filtragem ou processamento para melhorar as ERs.

4.2 Ambiente de execução experimental

Efectuaram-se vários testes relativos às implementações MapReduce e Spark. Estes foram executados num ambiente *cloud* Microsoft Azure [37]. Avaliámos o cálculo da correlação de Pearson variando o número de expressões relevantes a combinar. Logicamente, as expressões relevantes analisadas foram extraídas do mesmo *corpus* linguístico que é fornecido como *input* tanto para a implementação MapReduce como para a de Spark. Estes testes foram conduzidos sobre *corpora* linguísticos, escrito em língua inglesa obtidos da Wikipedia [34].

O Microsoft Azure é um conjunto de serviços *cloud* que permite criar, gerir e implementar aplicações numa rede global em grande escala através das ferramentas e arquitecturas que tem disponíveis. Dentro dos vários serviços que esta plataforma oferece, o utilizado foi o Azure HDInsight, que é dedicado à análise de dados. Este disponibiliza algumas *frameworks* de processamento paralelo, tais como Hadoop, Apache Spark, Apache Hive, Apache Kafka, Apache Storm, entre outros. Usámos então este serviço para configurar/montar o conjunto de máquinas virtuais necessário de modo a conseguir processar os testes delineados em tempo útil.

De notar que todos os testes foram realizados sobre a subscrição "Avaliação Gratuita". Esta subscrição abre acesso a certo e determinado tipo de máquinas, isto é, com a subscrição utilizada temos acesso a menos recursos do que com uma subscrição "Pay-as-you-go".

Portanto para formar os *clusters* de Hadoop e Spark, sobre a alçada da subscrição usada, foram utilizados dois tipos de configurações. Tanto a configuração D3_v2 como a D14_v2 têm em comum a mesma gama de processador, 2.4 GHz Intel Xeon E5-2673 v3, com a tecnologia Intel Turbo Boost 2.0 que possibilita chegar de 2.4 a 3.1 GHz, possuíndo também HyperThreading. As máquinas virtuais tinham as seguintes configurações:

D3_v2: disponibiliza 4 cores, correspondendo a 8 vCPUs. 14 GB de RAM, com 200 GB de armazenamento local SSD.

D14_v2: disponibiliza 16 cores, correspondendo a 32 vCPUs. 112 GB de RAM, com 800 GB de armazenamento local SSD.

Para que seja possível executar o protótipo MapReduce num *cluster* Hadoop, as máquinas virtuais possuem os subsequentes componentes de software: (1) Yarn para coordenação; (2) HDFS para armazenameno; (3) Hadoop para processamento; (4) JVM com a versão $\geqslant$1.8.x; (5) Sistema operativo Ubuntu Linux com acesso SSH. Onde a configuração miníma exigida pelo HDInsight é composta por 2 nós (máquinas virtuais) *master* e 2 nós *worker*.

Para que seja possível executar o protótipo Spark num *cluster* Spark, as VMs (máquinas virtuais) são compostas por: (1) Yarn para coordenação; (2) HDFS para armazenamento; (3) Spark para processamento; (4) JVM com a versão $\geqslant$1.8.x; (5) Sistema operativo Ubuntu Linux com acesso SSH. Onde a configuração mínima exigida pelo HDInsight é composta por 2 nós *master* e 2 nós *worker*.

Todos os dados de avaliação obtidos dos testes realizados, isto é, tempos de execução, entre outros, foram extraídos de apenas uma execução de cada teste.

Foi delineado o seguinte cenário de teste para os protótipos MapReduce e Spark:

Cenário de avaliação: testado com a implementação MapReduce e Spark. Processando um *corpus* fixo de 50 milhões de palavras, com 23216 documentos de texto. Foram extraídas através do algoritmo LocalMax (secção 2.1), expressões relevantes. Estas eram n-gramas, em que $2 \leqslant n \geqslant 6$. Perfaziam cerca de um total de 1 milhões de expressões relevantes onde 222 mil ERs eram bigramas, 240 mil trigramas, 230 mil tetagramas, 160 mil pentagramas e 100 mil hexagramas. Os testes foram planeados, de modo a aumentar o número de vCPUs, e aumentar iterativamente o número total de ERs a correlacionar (222 mil -> 500 mil -> 750 mil -> 860 mil -> 960 mil). Para tal foram usadas as seguintes configurações (número total de máquinas, tipo de máquina, número de masters, número de workers): (6, D3_v2, 2, 4); (10, D3_v2, 2, 8); (14, D3_v2, 2, 12); (18, D3_v2, 2, 16); (34, D3_v2, 2, 32); (5, D14_v2, 2, 3); (6, D14_v2, 2, 4); (10, D14_v2, 2, 8).

À medida que o tamanho do *corpus* e o número de ERs a correlacionar crescem, torna-se ineficiente executar os protótipos implementados numa só máquina, pois as execuções não são realizadas em tempo útil. O termo "tempo útil" significa neste contexto tempo abaixo das 6 horas. Em adjuvante temos também o facto de existir a restrição de armazenamento físico, pois a quantidade de meta-dados gerada pelos protótipos nas fases intermédias é muito grande. Assim sendo, todos os resultados experimentais que foram extraídos, foram avaliados consoante um número mínimo de máquinas virtuais requeridas para computar cada teste.

4.3 Avaliação Global da Performance do Protótipo MapReduce

4.3.1 Análise dos resultados experimentais - Cenário de avaliação

4.3.1.1 Tempos de Execução - Protótipo MapReduce

A tabela 4.1 mostra os tempos de execução descriminados de cada trabalho, bem como o tempo total de execução desta implementação, para o Cenário de avaliação (secção 4.2).

		Corpus de 50 Milhões de Palavras			
	K	Expressões Relevantes			
		222k	500k	750k	860k
1º Trabalho	32	4.32			
	64	3.03	4.47		
	96	2.28		4.1	
	128	1.92	3.27	3.36	4.83
	256	1.33	2.72	3.02	3.55
2º Trabalho	32	0.35			
	64	0.22	0.61		
	96	0.23		0.9	
	128	0.25	0.58	0.87	1.02
	256	0.23	0.57	0.82	1.12
Total	32	4.66			
	64	3.25	5.08		
	96	2.51		5	
	128	2.16	3.85	4.46	5.85
	256	1.55	3.28	3.85	4.66

Tabela 4.1: Tempos de execução do Cenário de avaliação em horas, variando o número de ERs e o número de vCPUs (K)

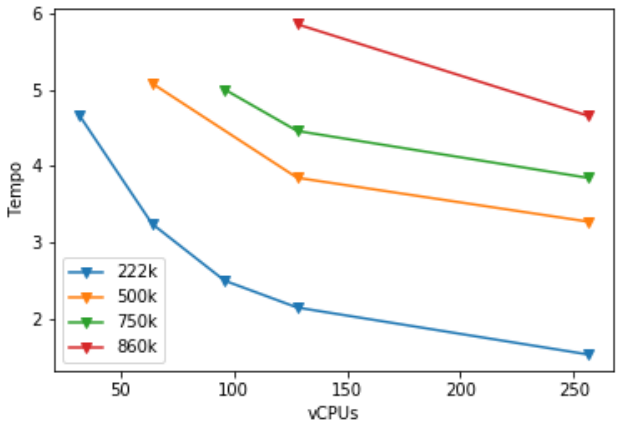
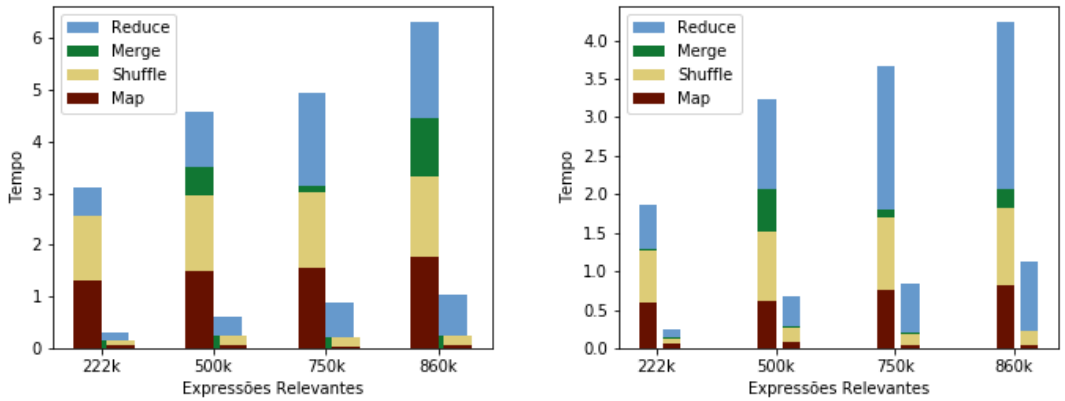


Figura 4.1: Tempo total de execução, Protótipo MapReduce - cenário de avaliação, em função do número de vCPUs.

O comportamento das execuções temporais do protótipo MapReduce para diferentes números de ERs e números de vCPUs (K), com o *corpus* fixo de 50 milhões de palavras, é ilustrado na figura 4.1. Este gráfico mostra o tempo total, em horas, de cada execução *versus* o número de máquinas usadas no cenário de avaliação. A partir desta conseguimos constatar que efectivamente os tempos globais de cada execução completa do protótipo MapReduce tem um comportamento sublinear.

Os gráficos apresentados na figura 4.2 representam a soma acumulada das diferentes fases de cada trabalho do protótipo MapReduce, para um número de vCPUs fixo (128 e 256). Para cada número de ERs a correlacionar, é apresentado um grupo de duas barras, em que a barra da esquerda representa os tempos de cada fase do 1º trabalho, e a barra da direita representa os tempos de cada fase do 2º Trabalho. A apresentação em separado põe em evidência que o 1º trabalho é o mais demorado. Decidimos apresentar os valores extraídos desta maneira, para que fosse possível comparar o crescimento temporal das várias fases em separado, à medida que aumentávamos o número total de expressões a correlacionar, mantendo um número de vCPUs fixo.



(a) Tempos de execução cumulativos - 128 vCPUs(b) Tempos de execução cumulativos - 256 vCPUs

Figura 4.2: Execução total temporal dos dois trabalhos MapReduce com cada fase descriminada, em função do número de ERs, para um *corpus* fixo de 50 milhões de palavras - cenário de avaliação. O eixo do Y representa o tempo de execução em horas e o eixo do X representa o número de ERs.

A partir da figura 4.2 conseguimos retirar as seguintes conclusões:

- Conseguimos constatar que tanto nas figuras 4.2a e 4.2b ao aumentar o número de pares de correlações a calcular, a fase de Map e Shuffle têm um crescimento pouco significativo;

- No entanto a fase de Reduce apresenta um crescimento temporal muito alto. Isto deve-se ao facto de só existir uma instância de Reduce para agregar todos os valores

que são emitidos, fazendo assim com que os dados sejam agregados sequencialmente. Verificamos isto, tanto no 1º trabalho como no 2º. Ou seja, o tempo de execução total de cada trabalho é extremamente influenciado pelo crescimento temporal da fase Reduce;

- Verificámos também que no 2º Trabalho o número de instâncias *mappers* é sempre menor que o numero total de vCPUs disponíveis. Quer dizer então que o último trabalho já não tira o proveito máximo do paralelismo que é oferecido pelo aumento do número total de vCPUs. Os tempos de execução do 2º trabalho não apresentam muito impacto aos tempos de cada execução completa do algoritmo.

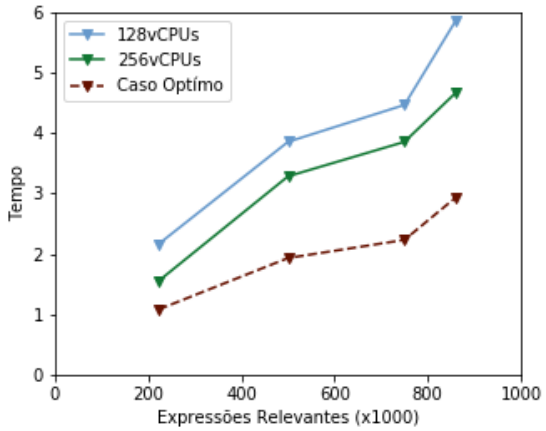


Figura 4.3: Tempo total de execução variando o número de ERs e usando 128 e 256 vCPUs, comparando com o tempo óptimo caso os 256 vCPUs passassem a metade do tempo. O eixo do Y representa o total de tempo de execução do algoritmo paralelo. O eixo do X representa o número total de expressões relevantes.

Analisando a figura 4.3 conseguimos justificar o que foi constatado acima. Podemos ver que o crescimento dos tempos de execução são extremamente afectados pelo tempo que a fase de Reduce demora a concluír. Tendo em conta que esta fase começa a partir de um certo limite de trabalho feito pela fase de Map, mesmo assim o tempo global de conclusão é afectado.

A partir da figura 4.2 conseguimos por em evidência que o 1º trabalho do protótipo MapReduce é mais demorado que o 2º. Se considerarmos apenas os tempos de execução das fases Map e Shuffle somados das várias execuções completas do protótipo MapReduce, podemos observar pela figura 4.4 que o crescimento temporal da execução que utiliza 256 vCPUs é linear nos primeiros dois testes (222 mil ERs e 500 mil ERs), e nos dois últimos podemos afirmar que é próximo da linearidade.

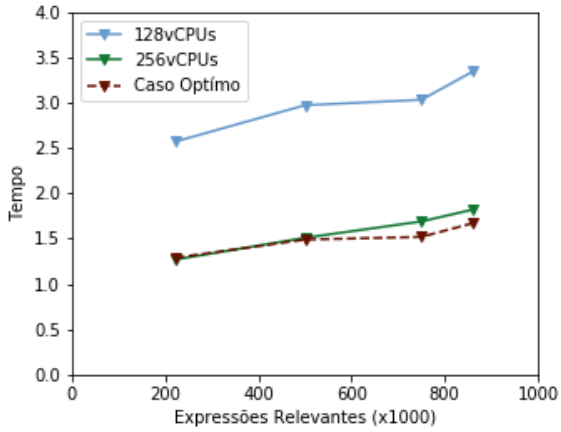


Figura 4.4: Soma de tempos de execução das fases *Map* e *Shuffle* variando o número de ERs e usando 128 e 256 vCPUs, comparando com o tempo óptimo caso os 256 vCPUs passassem a metade do tempo. O eixo do Y o tempo de execução em horas. O eixo do X representa o número total de expressões relevantes que vão ser correlacionadas.

4.3.1.2 *Speedup* Relativo

No cenário de avaliação, para cada número de ERs testadas, consideramos um valor de K_1 que corresponde ao mínimo do número de vCPUs requeridos para executar o prototipo MapReduce. O valor mínimo de K_1 é usado como referencia em relação ao *speedup* relativo: $Sp_{K_1 \to K} = \frac{T(K_1)}{T(K)}$ onde os valores correspondentes deste rácio são apresentados para todos os valores de K usados.

A figura 4.5 mostra o *speedup* relativo (definido em 2.12) do protótipo MapReduce *versus* o número de vCPUs (K) para o cenário de avaliação.

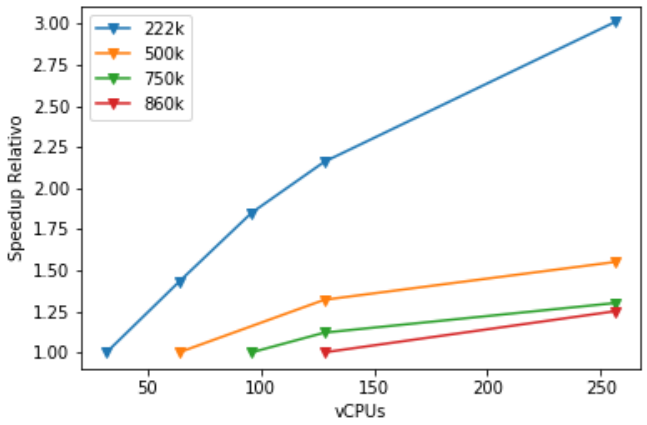
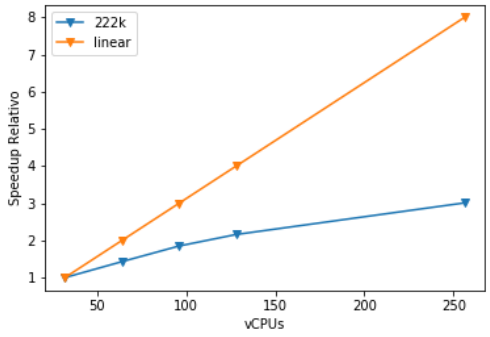


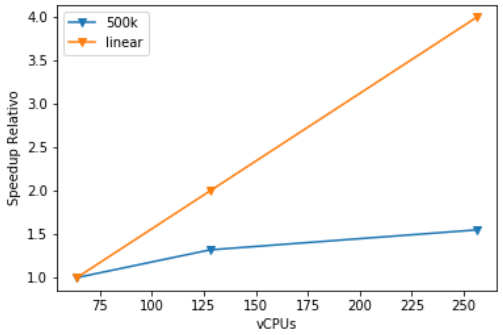
Figura 4.5: *Speedup* Relativo em função do número de vCPUs K, para diferentes números de ERs, no cenário de avaliação. O eixo do Y representa o *speedup* relativo e o eixo do X o número de vCPUs utilizados.

A figura 4.6 mostra com maior detalhe o crescimento dos dados apresentados na

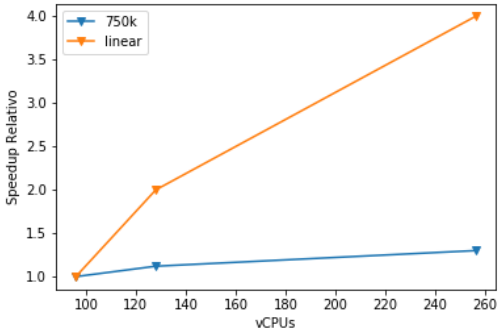
figura 4.5.



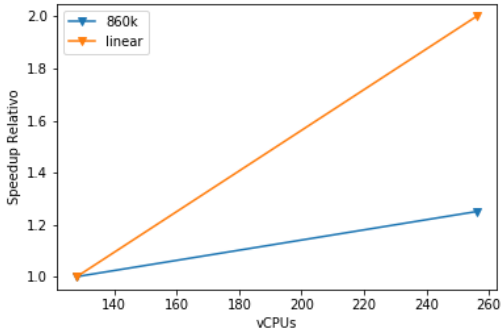
(a) *Speedup* relativo com 222 mil ERs



(b) *Speedup* relativo com 500 mil ERs



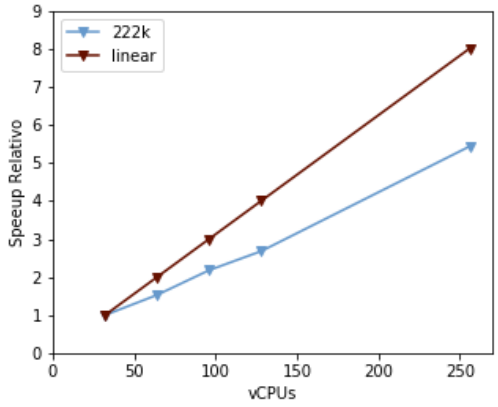
(c) *Speedup* relativo com 750 mil ERs



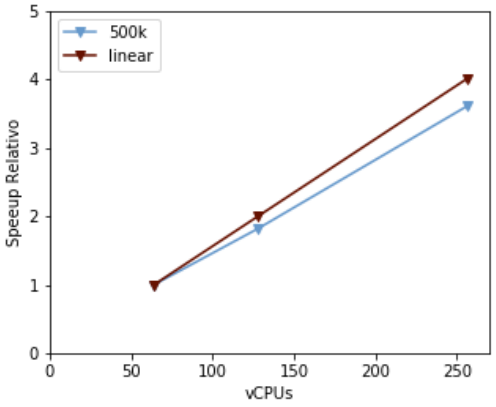
(d) *Speedup* relativo com 860 mil ERs

Figura 4.6: Figura detalhada de 4.5 em relação ao *speedup* relativo do protótipo MapReduce em função do número de vCPUs, para diferentes números de expressões relevantes, no cenário de avaliação. Onde o eixo do Y representa o *speedup* relativo e o eixo do X representa o número de vCPUs.

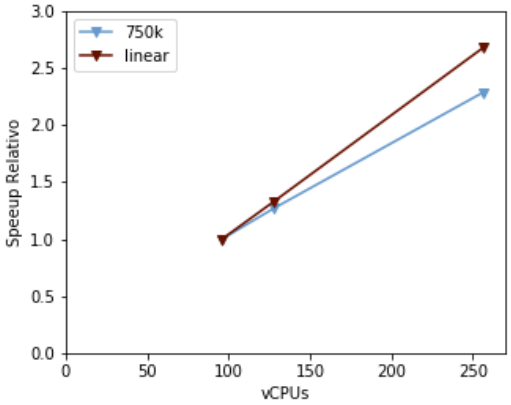
Aqui conseguimos observar que o *speedup* relativo é sublinear em todas as amostras testadas. No entanto se tivermos em conta apenas as fase de Map e Shuffle desta implementação pela figura 4.7: Conseguimos constatar que o *speedup* relativo está perto do linear, quando aumentamos o número total de expressões relevantes a correlacionar.



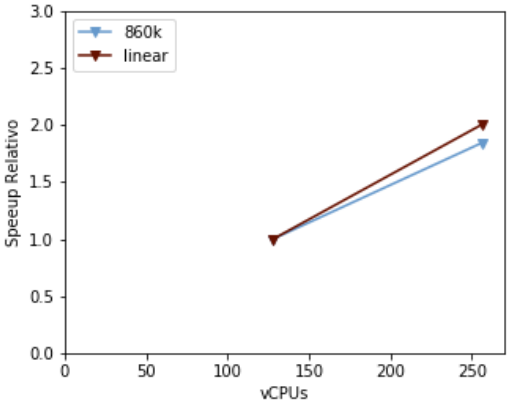
(a) *Speedup* relativo com 222 mil ERs



(b) *Speedup* relativo com 500 mil ERs



(c) *Speedup* relativo com 750 mil ERs



(d) *Speedup* relativo com 860 mil ERs

Figura 4.7: Figura detalhada de 4.6 em relação ao *speedup* relativo, das fases Map e Shuffle do protótipo MapReduce em função do número de vCPUs, para diferentes número de expressões relevantes no cenário de avaliação. Onde o eixo do Y representa o *speedup* relativo e o eixo do X representa o número de vCPUs.

4.3.1.3 Eficiência Relativa

A figura 4.8 mostra a eficiência relativa (definida em 2.13), para o cenário de avaliação.

Como resultado do *speedup* relativo descrito acima, a eficiência relativa mostrou ser cerca de 70%, diminuindo pouco significativamente em função do aumento dos vCPUs.

4.3.1.4 *Sizeup* Relativo

A figura 4.9 mostra o comportamento do protótipo MapReduce procurando demonstrar o *sizeup* relativo para o cenário de avaliação.

grafico demonstra a relaçao consideramos um referencial de 222k, 32 vCPUs

Através da figura 4.9a, quando aumentamos o número de vCPUs em conjunto com o número de ERs para um *corpus* de tamanho fixo, conseguimos constatar que o *sizeup* relativo do protótipo MapReduce demonstra alguma escalabilidade devido aos tempos de

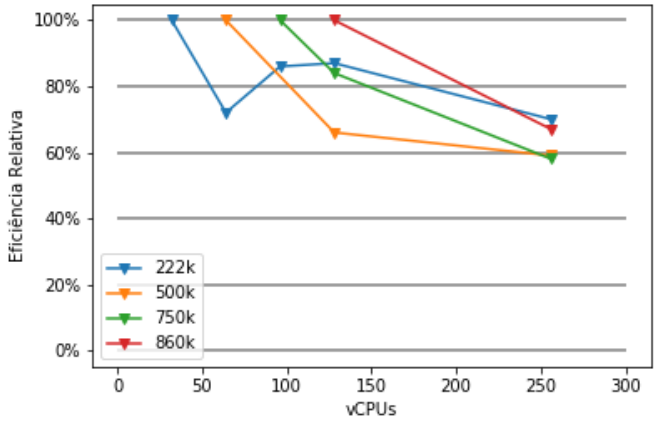
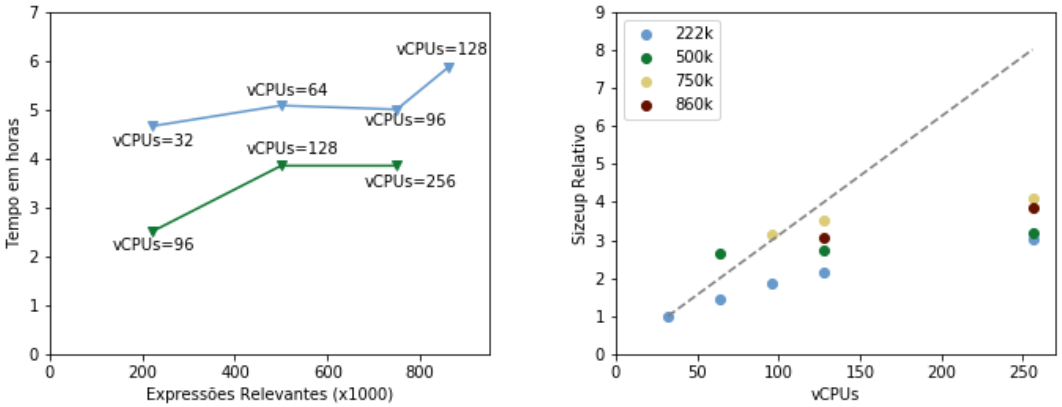


Figura 4.8: Eficiência relativa do protótipo MapReduce em função do número de vCPUs, para diferentes número de ERs a correlacionar. Em que o eixo do Y representa a eficiência relativa e o eixo do X representa o número de vCPUs.



(a) O eixo do Y representa o tempo de execução em horas e o eixo do X o número de vCPUs.

(b) O eixo do Y representa o *sizeup* relativo e o eixo do X o número de vCPUs.

Figura 4.9: *Sizeup* Relativo do protótipo MapReduce em função dos vCPUs, para diferentes números de ERs a correlacionar.

execução acompanharem o crescimento das ERs com o número de vCPUs. O gráfico 4.9b considera como referencial o caso de 222k com 32 vCPUs, sendo obtido o número de ERs processadas por hora. O eixo do Y dá a relação de *sizeup* para cada conjunto de expressões relevantes tratadas com o número de vCPUs no eixo do X. Esta relação seria linear se seguisse a linha a tracejada no gráfico (seria um crescimento direto entre o número de ERs tratas e o número de vCPUs). Este mostra um *sizeup* próximo do linear de início do crescimento do número de ERs processados com o número de vCPUs, mas depois degrada-se. Continua no entanto a conseguir tratar as ERs em tempo útil. Estes resultados poderão ser muito melhores quando o Reduce for feito em paralelo, como apresentado na secção 5.1.

4.3.1.5 Discussão dos Resultados

Tendo em conta todos os dados apresentados na secção 4.3, conseguimos retirar as seguintes conclusões acerca do protótipo MapReduce:

- A fase de **Map** consegue manter um crescimento temporal muito perto do linear, devido à maneira de como este está implementado (secção 3.2.1.1). Sabemos que o número de instâncias de *mappers* é igual ao número de documentos de texto no *corpus*. Se considerarmos um *corpus* de tamanho fixo e que o que varia é o número de ERs a correlacionar, a complexidade temporal de cada *mapper* aumenta consoante a quantidade de ERs a mapear pelo ficheiro. E posteriormente, pelo número de combinações que faz dessas ERs (C_2^{Ner}) para produzir valores intermédios para o cálculo da covariância entre duas expressões relevantes distintas. Como cada documento de texto não tende a abordar muitas categorias em simultâneo, podemos afirmar que a combinatória envolvida que é gerada para produzir os valores intermédios para a $Cov(ER_1, ER_2)$ não é significativamente maior, quando aumentamos o número total de ERs que queremos correlacionar.

- Uma das justificações para o **Shuffle** manter um crescimento linear, tem que ver com as redes em que os *clusters* foram configurados. Estas conseguem suportar eficientemente o crescimento de dados que é gerado nas duas fases de Map de cada trabalho. Tomando como exemplo a fase de Map do 1º Trabalho sobre o cenário de avaliação, onde é gerada a maior quantidade de valores intermédios, o crescimento em GB dos pares (chave, valor) é o seguinte: 14.5GB para 222 mil ERs -> 42.7GB para 500 mil ERs -> 73.02GB para 750 mil ERs -> 94.78GB para 860 mil ERs.

- A fase de **Reduce** apresenta um crescimento temporal muito alto. Isto deve-se ao facto de só existir uma instância de Reduce para agregar todos os valores que são emitidos. Fazendo assim com que os dados que são gerados por todos as instâncias de *mappers* sejam agregados sequencialmente. Verificamos isto, tanto no 1º trabalho como no 2º. Ou seja, o tempo de execução total de cada trabalho é fortemente influenciado pelo crescimento temporal da fase Reduce.

- Conseguimos também constatar que o tempo de execução do 2º trabalho (secção 3.2.1.2) não afecta significativamente os tempos globais de execução do protótipo, pois no 1º trabalho (secção 3.2.1.1) existe uma quantidade de trabalho muito maior do que no 2º.

- A componente paralela (fase de *mapping*) consegue tirar proveito máximo dos recursos disponíveis conseguindo obter um valor perto do óptimo, ou seja, apresenta uma redução temporal perto de linear. Enquanto que a parte sequencial, a fase de Reduce, aumenta de acordo com a combinatória envolvida (C_2^{Ner}) de ERs a correlacionar, mostrando assim um crescimento sublinear. Fazendo assim que exista uma perda nos ganhos temporais que a fase de Map consegue oferecer.

4.4 Avaliação Global da Performance do Protótipo Spark

De modo a verificar se os protótipos Hadoop e Spark estavam aptos para testar nos *clusters* criados pelo Microsoft Azure - HDInsight, para cada protótipo foi efectuado um teste num ambiente piloto. Neste ambiente foi processado um *corpus* fixo de 20 milhões de palavras, com 1000 ERs para correlacionar; A configuração dos *clusters* criados para cada protótipo era a mesma, sendo ela composta por 4 nós, em que 2 deles eram mestre e os outros *workers*.

Os tempos de execução que foram apresentados relativamente aos dois testes executados no ambiente piloto, para o protótipo Hadoop e Spark foram de 1 hora e de 30 minutos, respectivamente. Um resultado que à primeira vista, parece ser promissor para a implementação Spark. No entanto, não foi isso que se verificou quando tentamos executar os testes do cenário de avaliação. A avaliação da implementação Spark não pode ser efectuada pelos seguintes motivos:

- Uma vez que a *framework* Spark privilegia a utilização da RAM em vez do armazenamento local, nesta implementação guardam-se todos os valores em memória. Sabendo que o número de termos do somatório envolvido no coeficiente de Pearson cresce com o quadrado do número total de expressões relevantes, torna-se difícil lidar com a quantidade de dados que é gerada pelas fases intermédias do protótipo, tendo algumas execuções abortado por falta de memória;

- Os tempos de execução dos poucos testes experimentais relativamente ao cenário de avaliação que correram saíam fora do tempo útil. Assim sendo, estes acabaram por ser interrompidos.

Esta implementação carece de uma restruturação para usar menos memória, entre outras optimizações.

No capitulo 5, mais propriamente na secção 5.1 são indicadas algumas possíveis optimizações a fazer a este protótipo.

4.5 Visualização de descritores de documentos

Nesta secção iremos mostrar os resultados obtidos para 5 documentos exemplo através da tabela 4.2.

4.5.1 Problema da medição da qualidade dos descritores

Relativamente ao descritor explícito do documento pretende-se que ele espelhe o mais fielmente possível o conteúdo "core" do documento. Acontece que existe uma grande subjetividade na avaliação da correção desse conteúdo. Na verdade, quando são atribuídas manualmente *keywords* a um documento, pessoas diferentes poderão facilmente escolher dois conjuntos diferentes de *keywords* correspondendo cada conjunto a uma visão subjetiva. Por exemplo, um descritor dum documento reportagem sobre um jogo de futebol pode ser constituído pelas *keywords* "prolongamento", "grandes penalidades" e "final antecipada", resultantes da visão subjetiva dum certo jornalista. Porém, o mesmo jogo pode ser descrito com outras 3 *keywords* resultantes da visão de outro jornalista: "jogo equilibrado", "recurso a penaltis" e "futebol de alto nível". Apesar de diferentes, estes dois conjuntos podem retratar o "core"do documento sem que qualquer das 6 *keywords* seja desajustada e não capte parcialmente a semântica do documento; isto torna difícil a avaliação da correcção do conteúdo do descritor. Por outras palavras, não se pode usar a noção de Precisão clássica (*true positives* / (*true positives* + *false positives*)) uma vez que o apuramento dos *true positives* e *false positives* num documento, resultará sempre de uma avaliação humana subjetiva. Posto isto, torna-se necessário desenvolver um critério o menos subjectivo possível para avaliar a qualidade dos descritores, tarefa que requer um tempo que não está disponível no contexto desta tese, podendo enquadrar-se em trabalho futuro.

Quanto à avaliação da qualidade dos descritores implícitos, esta é ainda mais problemática pois seria necessário considerar todas as ERs existentes no *corpus* e escolher por avaliação humana, quais as mais fortemente ligadas às *keywords* explícitas de cada documento; tarefa naturalmente extremamente demorada e com resultados subjetivos. Apesar da ausência de um critério vocacionado para esta avaliação, os descritores apresentados na tabela 4.2 permitem reconhecer o conteúdo "core"dos documentos através das 8 *keywords* explícitas. E, apesar da imperfeição, podemos reconhecer a forte ligação semântica entre as *keywords* explícitas e implícitas de cada documento na tabela. Com efeito, tomando como exemplo o Documento 3, excepto "Albert" e de acordo com alguma subjetividade, todas as *keywords* explícitas são fortemente informativas, espelhando o conteúdo dum certo documento sobre física e as descobertas de Albert Einstein; quanto ao seu descritor implícito também podemos reconhecer que as suas *keywords* estão relacionadas com as *keywords* explícitas do mesmo documento, exceptuando, dir-se-ia, "There is hope for this" e "Which features of general".

	Keywords	
	Explicítas	Implicitas
Documento 1	Bill Clinton	White House
	Yasser Arafat	Palestine Liberation Organisation
	Saeb Erekat	Hagai Amir
	Yitzhak Rabin	Historic Compromise
	Oslo Accords	although hesitant at first
	Israel	Fifth Prime Minister of Israel
	Egypt	back them if heavy
	Palestine	Peace process
Documento 2	National Basketball Association	NCAA men's division
	NBA	an airplane pilot
	All-star Game mvp awards	Kobe Bryant
	Kareem	14.5 rpg
	Greatest Basketball Player	Michael Jordan
	James	NCAA women's division
	NCAA Championships	March Madness
	NBA Commissioner	David Stern
Documento 3	Albert Einstein's theory	Galilei and Isaac Newton
	Albert	Which features of general
	Albert Einstein's theory of relativity	Projectile motion
	Physics	There is hope for this
	Albert Einstein's special relativity	Particle-like behavior
	Theory of relativity	Max Planck
	Nobel	Atomic Bomb
	Isaac Newton	Newton's laws
Documento 4	Bush	September 11 attacks
	George W. Bush	Abolished slavery
	Abraham Lincoln	Republican Party
	President	$4.4 billion tariffs raise
	USA	Assassinated by Lee Harvey Oswald
	Donald Trump	First President of USA
	John F. Kennedy	Consistently ranks among the top
	George Washington	With the highest points in
Documento 5	North and South Korea	Era of no war
	Korea	United States and South Korea
	Korea's nuclear research	North Korean weapons of mass destruction
	Missiles	Agreed Framework
	Korean missile test	Kim Dynasty
	USA	But its formal implementation
	Korea's nuclear weapons program	Korean War
	North Korea	Avoid the large scale conventional

Tabela 4.2: Descritores de 5 documentos - *keywords* explicitas e implícitas

Tomando como outro exemplo, observando na mesma tabela o documento 1 conseguimos perceber que a ER "Bill Clinton" está fortemente correlacionada com a ER "White House", assim como "Yasser Arafat" está para "Palestine Liberation Organisation". No entanto mesmo reconhecendo valores de correlação forte, obtemos também aqueles pares correlacionados onde uma das ERs já de si têm uma semântica fraca. Como é o exemplo, de "Oslo Accords" com "although hesitant at first", ou de "Palestine" com "back them if heavy". Tal como referido anteriormente, as ERs foram fornecidas a partir do algoritmo LocalMax e, como nas nossas implementações não existiu qualquer tipo de preprocessamento, de modo a filtrar só as ERs com uma semântica forte, ficámos condicionados à precisão que o algoritmo LocalMax oferece.

Capítulo 5

Conclusões e Trabalho Futuro

O propósito principal desta tese consistiu em contribuir com métodos e ferramentas eficientes, de modo a conseguir construir de forma automática e independente da língua descritores de documentos a partir de expressões relevantes (n-gramas), para *corpora* de grandes dimensões. Os descritores são compostos por dois tipos de *keywords*: as explícitas e as implícitas. Sabendo que as implícitas são obtidas através do cálculo da correlação de Pearson, e que esta envolve um processamento pesado quando computado numa só máquina, foi feito um estudo que investigou como implementar de forma eficiente esta correlação (2.3.3). Estudo esse que recorreu à utilização da computação distribuída e paralela, tendo em vista obter execuções com tempos aceitáveis e comportamentos escaláveis em termos de tamanho do *corpus*, número total de ERs a correlacionar e o número de vCPUs utilizados. Como produto final deste trabalho podemos identificar as principais contribuições.

Descritores de documentos

Embora não haja ainda disponível um critério objetivo para a avaliar a qualidade dos descritores, como foi explicado na secção 4.5.1, pela amostra usada neste documento de dissertação é possível reconhecer a relevância dos descritores produzidos pela abordagem que seguimos. Esta relevância pode verificar-se quer no que respeita às *keywords* explicitas quer às implícitas.

Protótipo Versão Sequencial - Correlação de Pearson

Foi elaborada uma implementação sequencial da correlação de Pearson. Com esta pretendíamos: decompor a fórmula desta correlação em sub-funções de modo a optimizá-las; analisar a fórmula de Pearson ao pormenor por forma a encontrar todos os pontos que eram paralelizáveis; criar um conjunto de correlações entre as várias ERs, com vista à sua utilização para comparação dos resultados obtidos com os *outputs* produzidos pelos diferentes protótipos paralelos. Através da execução de um teste piloto com um *corpus*

de 50 milhões de palavras (23216 documento de texto) e 222 mil ERs, foi possível tirar as seguintes conclusões:

- Demonstrou-se que é possível obter *keywords* implícitas a partir desta implementação.
- Foram encontradas zonas de código passíveis de paralelização nas sub-funções da correlação de Pearson, bem como pontos sequenciais. Estes são descritos na secção 3.1.2.
- O tempo de execução global da versão sequencial para o teste piloto foi de 3 dias e meio. Este tempo não é aceitável perante um conjunto de ERs relativamente pequeno, ainda mais quando o mesmo teste executou em 4,66 horas no protótipo MapReduce com 32 vCPUs.

Implementações Paralelas e Distribuídas - Correlação de Pearson

Foi explorada a vertente da computação paralela e distribuída com a finalidade de conseguir obter execuções temporais e comportamentos escaláveis aceitáveis em função do tamanho dum *corpus*, variando o número total de ERs a correlacionar com o número de máquinas utilizadas para esse processamento. Para isso usamos duas *frameworks*, Hadoop e Spark.

Depois de feita a comparação de *outputs* finais entre a versão sequencial e os protótipos paralelos, conseguimos verificar que estes eram iguais. Ou seja, apresentam todos o mesmo número de pares correlacionados com o respectivo valor.

Observamos que, dada a estrutura da fórmula implementada, a geração de uma quantidade muito grande de dados intermédios, levanta uma restrição de armazenamento nos dois protótipos paralelos. Visto que o Hadoop escreve todos os seus dados para disco, este consegue lidar melhor com o armazenamento do que o Spark, que mantém todos os dados a processar em memória (RAM). Mais tarde viemos a apurar que o Spark também permite escrever os seus resultados intermédios para disco mas, dadas as restrições temporais, não conseguimos fazer as optimizações necessárias para obter execuções com tempos aceitáveis.

Sabendo que o desejável seria desenvolver uma análise de desempenho para cada um dos protótipos paralelos, esta foi feita apenas para o protótipo Hadoop. A razão pelo qual não foi possível à implementação Spark conseguir atingir tempos de execução aceitáveis, deve-se ao facto de termos restrições de tempo sobre a entrega deste trabalho. Com efeito, acabámos por dedicar mais tempo à implementação e posterior optimização do protótipo Hadoop, tendo assim menos tempo para optimizar o protótipo Spark.

Tendo em conta a análise feita à implementação Hadoop conseguimos verificar que esta apresenta um comportamento sub-linear em relação à eficiência e *speedup* relativos. No entanto, tal como descrito na secção 4.3.1, ignorarmos a fase de Reduce, conseguimos constatar que as restantes fases apresentam um crescimento próximo da linearidade. Este

comportamento por um lado deve-se ao resultado do *design* escalar da arquitetura distribuída que a *framework Hadoop* oferece, e por outro, aos pontos de paralelismo encontrados na fórmula da correlação de Pearson.

No entanto o protótipo Hadoop MapReduce implementado também demonstrou algumas limitações. Assim como retratado na secção 3.2.1.1, na fase de Map iremos obter o número de instâncias de *mappers* igual ao número total de documentos. Isto pode ser um factor limitativo, pois grande parte da computação do algoritmo reside no 1º trabalho, na fase de Map. Ou seja, a complexidade temporal irá aumentar consoante o tamanho dos documentos de texto analisados, o número total de ERs mapeadas por esse documento e o número total de combinações (C_2^{Ner}) de resultados intermédios à correlação de Pearson.

De notar que todos os testes experimentais executados para avaliar os protótipos paralelos (secção 4.3.1) foram realizados sobre a subscrição "Avaliação Gratuita" da Microsoft Azure. Esta subscrição abre acesso a determinado tipo de máquinas, ou seja, com a subscrição utilizada temos acesso a menos recursos do que com uma subscrição "Pay-as-you-go".

Todos os meta-dados obtidos a partir dos testes realizados, isto é, tempos de execução, entre outros, foram extraídos de apenas uma execução de cada teste. Tendo conhecimento que para extrair tempos de execução fiáveis, cada teste deveria ser realizado pelo menos 3 vezes de modo a amortizar o erro, esse procedimento não foi feito por não termos conseguido obter mais contas com subscrição "Avaliação Gratuita".

5.1 Trabalho Futuro

As amostras de descritores exemplo presentes neste documento reportam que há algumas expressões relevantes que não são, na realidade, *keywords*. Este facto resulta da imperfeição do extractor, ferramenta que tem limitações a nível da sua precisão. Este é um ponto a melhorar a montante no futuro.

Ficaram alguns caminhos desta investigação por explorar. Nomeadamente, os seguintes aspectos valem a pena ser investigados para as duas implementações paralelas realizadas:

- Sabendo que tanto na versão Spark como no Hadoop é gerado um grande volume de dados intermédios, seria importante tentar indexar estes dados, de modo a que eles não ocupassem tanto armazenamento. Isto seria vantajoso principalmente para o protótipo Spark, pois este privilegia a utilização da memória RAM. Um dos modos de indexar a informação tendo em conta o problema a resolver, poderia passar por usar algoritmos de compressão. Poderia ser implementada uma estrutura, Dicionário, que iria conter todas as palavras existentes de um *corpus*. No entanto este tipo de processos envolveriam um pre-processamento de todas as palavras do *corpus*, e o seu posterior processamento, que consiste em re-mapear todas as palavras de volta ao original.

- Depois de alguma pesquisa, aconselhamos a utilização do algoritmo de Boyer-Moore [38] na fase de mapeamento das expressões relevantes. Esta utilização iria oferecer eficiência ao *design* dos protótipos, sendo que é muito bem reconhecida pela comunidade informática, para *pattern matching*.

Relativamente ao protótipo Hadoop MapReduce, existem também alguns aspetos a melhorar. Verificou-se que a fase de Reduce do primeiro trabalho do MapReduce (3.2.1.1) não precisava efectivamente de ser executada em modo sequencial. Isto é, de momento, para agregar todos os dados que são gerados pelas instâncias de *mappers*, só existe neste momento uma instância de *reduce*. No entanto, através de uma reavaliação da fórmula de Pearson e da análise descrita em 4.3.1, conseguimos concluir que era muito mais vantajoso, se existissem mais do que uma instância de *reduce* a agregar dados, o que é possível. Tendo em conta os tempos de execução globais que este protótipo apresentou, o seu *speedup* e eficiência relativos poderiam vir a apresentar resultados próximos do linear.

Em relação ao protótipo Spark, existem vários aspetos a ter em conta, de modo a obter resultados eficientes.

- A gestão eficiente da memória RAM é crucial para que se possa tirar proveito da *framework*. Sabendo que esta permite fazer *cache* de RDDs na memória RAM, pode vir a ajudar imenso no auxílio ao processamento. No contexto do nosso trabalho como componente de coordenação no *cluster* utilizamos o YARN, que já por si ocupa alguma memória. Adjuvante a este, temos a JVM, deixando assim menos memória do que o esperado para mover dados e metê-los em *cache*. Finalmente temos a linguagem usada, Scala, que também introduz um *overhead* significativo nos RDDs. A título de exemplo, uma *String* em Java de 10 caracteres consome cerca de 60 bytes. O Spark por norma utiliza o *Java Serializer*, que foi o que nós utilizámos. No entanto também permite a utilização da *Kryo Serialization* que aparenta ser cerca de 10 vezes mais rápido que o *Java Serialization*, apesar de suportar menos tipos de classes. Seria interessante também dividir o processamento da implementação feita, em processos *batch* mais pequenos, escrevendo os resultados intermédios para disco, de modo a ter um novo processo limpo de todos os metadados e ficheiros de *cache* que foram utilizados no processo anterior.

- O uso excessivo de operações de *shuffle* em execuções de longa duração pode revelar-se caro, pois o *driver program* acumula bastante *metadata* sobre este tipo de operações. Evitar o uso deste tipo de operações e minimizar as transferências de dados ajudam também a que processamento seja mais rápido e confiável, mantendo sempre em mente o uso ocasional de *shuffles*, pois podem existir casos em que é iminente a necessidade da sua utilização.

- Como referido acima, movimentar dados pode sair caro, tanto pelo lado das operações *shuffle* como também pela utilização da operação *collect()* no *drive program*. Esta operação, para além de forçar a transferência dum RDD na sua totalidade,

guarda-o em memória, levantando novamente a problemática de gestão de memória. Uma das opções, que é totalmente dependente da memória de cada máquina, seria aumentar a memória alocada para o *driver program* (aumentando o parâmetro spark.driver.memory).

No futuro e como continuação deste trabalho, será importante executar e observar o crescimento temporal para um *corpus* de tamanho menor que o do cenário de avaliação descrito na secção 4.2. Esta experiência irá permitir avaliar os protótipos paralelos não só em termos do aumento do número total de ERs a correlacionar e o número de vCPUs, mas também tendo em conta a variação do tamanho do *corpus*, esperando assim, com um *corpus* menor, tempos de execução menores e com desempenhos escaláveis próximos da linearidade. Pelas mesmas razões anteriormente descritas, seria também interessante estudar o crescimento temporal para um ambiente de testes com um *corpus* de tamanho maior que o do cenário de avaliação.

Bibliografia

[1] I. T. Union. *Individuals using the Internet (% of population)*. 2010-2016. URL: data.worldbank.org/indicator/IT.NET.USER.ZS?end=2016&start=1960&view=chart0.

[2] I. T. Union. *ICT Facts and Figures 2010 - 2016*. 2010-2016. URL: www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2017.pdf.

[3] G. S. Almasi e A. Gottlieb. *Highly Parallel Computing*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN: 0-8053-0177-1.

[4] E. Brill. "A Simple Rule-based Part of Speech Tagger". Em: *Proceedings of the Third Conference on Applied Natural Language Processing*. ANLC '92. Trento, Italy: Association for Computational Linguistics, 1992, pp. 152–155. DOI: 10.3115/974499.974526. URL: https://doi.org/10.3115/974499.974526.

[5] K. W. Church. "A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text". Em: *Proceedings of the Second Conference on Applied Natural Language Processing*. ANLC '88. Austin, Texas: Association for Computational Linguistics, 1988, pp. 136–143. DOI: 10.3115/974235.974260. URL: https://doi.org/10.3115/974235.974260.

[6] A. Savary, C. Ramisch, S. Cordeiro, F. Sangati, V. Vincze, B. QasemiZadeh, M. Candito, F. Cap, V. Giouli, I. Stoyanova e A. Doucet. "The PARSEME Shared Task on Automatic Identification of Verbal Multiword Expressions". Em: *Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)*. Valencia, Spain: Association for Computational Linguistics, 2017, pp. 31–47. URL: http://aclweb.org/anthology/W17-1704.

[7] S. Markantonatou, C. Ramisch, A. Savary e V. Vincze. "Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)". Em: *Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)*. Valencia, Spain: Association for Computational Linguistics, 2017. URL: http://aclweb.org/anthology/W17-1700.

[8] N. Klyueva, A. Doucet e M. Straka. "Neural Networks for Multi-Word Expression Detection". Em: *Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)*. Valencia, Spain: Association for Computational Linguistics, 2017, pp. 60–65. URL: http://aclweb.org/anthology/W17-1707.

[9] J. Mandravickaite e T. Krilavičius. "Identification of Multiword Expressions for Latvian and Lithuanian: Hybrid Approach". Em: *Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)*. Valencia, Spain: Association for Computational Linguistics, 2017, pp. 97–101. URL: http://aclweb.org/anthology/W17-1712.

[10] F. Cap. "Show Me Your Variance and I Tell You Who You Are - Deriving Compound Compositionality from Word Alignments". Em: *Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)*. Valencia, Spain: Association for Computational Linguistics, 2017, pp. 102–107. URL: http://aclweb.org/anthology/W17-1713.

[11] M. Scholivet e C. Ramisch. "Identification of Ambiguous Multiword Expressions Using Sequence Models and Lexical Resources". Em: *Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)*. Valencia, Spain: Association for Computational Linguistics, 2017, pp. 167–175. URL: http://aclweb.org/anthology/W17-1723.

[12] J. Ventura e J. Silva. "Mining concepts from texts". Em: *Procedia Computer Science* 9 (2012), pp. 27–36.

[13] J. F. da Silva e G. P. Lopes. "A local maxima method and a fair dispersion normalization for extracting multi-word units from corpora". Em: *Sixth Meeting on Mathematics of Language*. 1999, pp. 369–381.

[14] M. G. KENDALL. "A NEW MEASURE OF RANK CORRELATION". Em: *Biometrika* 30.1-2 (1938), pp. 81–93. DOI: 10.1093/biomet/30.1-2.81. eprint: /oup/backfile/content_public/journal/biomet/30/1-2/10.1093/biomet/30.1-2.81/2/30-1-2-81.pdf. URL: +http://dx.doi.org/10.1093/biomet/30.1-2.81.

[15] C. Spearman. "The proof and measurement of association between two things". Em: *The American journal of psychology* 15.1 (1904), pp. 72–101.

[16] K. Pearson. "Mathematical contributions to the theory of evolution. III. Regression, heredity, and panmixia". Em: *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character* 187 (1896), pp. 253–318.

[17] K. Pearson. "Notes on the history of correlation". Em: *Biometrika* 13.1 (1920), pp. 25–45.

[18] T. K. Koerner e Y. Zhang. "Application of linear mixed-effects models in human neuroscience research: a comparison with Pearson correlation in two auditory electrophysiology studies". Em: *Brain sciences* 7.3 (2017), p. 26.

[19] T. H. Pers, J. M. Karjalainen, Y. Chan, H.-J. Westra, A. R. Wood, J. Yang, J. C. Lui, S. Vedantam, S. Gustafsson, T. Esko et al. "Biological interpretation of genome-wide association studies using predicted gene functions". Em: *Nature communications* 6 (2015), p. 5890.

[20] *Word2Vec*. Consultado em: 2018-10. URL: https://code.google.com/archive/p/word2vec/.

[21] S. García, S. Ramírez-Gallego, J. Luengo, J. M. Benítez e F. Herrera. "Big data preprocessing: methods and prospects". Em: *Big Data Analytics* 1.1 (2016), p. 9.

[22] A. Fernández, S. del Río, V. López, A. Bawakid, M. J. del Jesus, J. M. Benítez e F. Herrera. "Big Data with Cloud Computing: an insight on the computing environment, MapReduce, and programming frameworks". Em: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4.5 (2014), pp. 380–409.

[23] J. Lin e C. Dyer. "Data-intensive text processing with MapReduce". Em: *Synthesis Lectures on Human Language Technologies* 3.1 (2010), pp. 1–177.

[24] P. Mell, T. Grance et al. "The NIST definition of cloud computing". Em: (2011).

[25] S. Ghemawat, H. Gobioff e S.-T. Leung. "The Google File System". Em: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, 2003, pp. 20–43.

[26] *HDFS*. Consultado em: 2018-11. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[27] *OpenMP*. Consultado em: 2018-05. URL: https://www.openmp.org/.

[28] J. Dean e S. Ghemawat. "MapReduce: simplified data processing on large clusters". Em: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[29] *Hadoop*. Consultado em: 2018-08. URL: https://hadoop.apache.org/.

[30] *YARN*. Consultado em: 2018-09. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker e I. Stoica. "Spark: Cluster computing with working sets." Em: *HotCloud* 10.10-10 (2010), p. 95.

[32] *Spark Apache*. Consultado em: 2018-02. URL: https://spark.apache.org/examples.html.

[33] C. Jorge de Sousa Gonçalves, I. Orientador, J. Alberto Cardoso Cunha, P. Catedrático, J. Francisco Ferreira da Silva e P. Auxiliar. "Parallel and Distributed Statistical-based Extraction of Relevant Multiwords from Large Corpora Dissertação para obtenção do Grau de Doutor em". Em: (2017). URL: https://github.com/joaomlourenco/novathesis.

[34] *Wikipedia*. Consultado em: 2018-04. URL: http://dumps.wikimedia.org.

[35] *Spark Transformations*. Consultado em: 2018-10. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations.

[36] *Spark Actions*. Consultado em: 2018-10. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions.

[37] *MicrosofAzure*. Consultado em: 2019-01. URL: https://azure.microsoft.com.

[38] R. S. Boyer e J. S. Moore. "A fast string searching algorithm". Em: *Communications of the ACM* 20.10 (1977), pp. 762–772.