UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DO DESEMPENHO DA BUSCA POR SIMILARIDADE UTILIZANDO O PARADIGMA MAPREDUCE

TRABALHO DE GRADUAÇÃO

Paulo Vinicius Mendonça Cardoso

Santa Maria, RS, Brasil

2016

ANÁLISE DO DESEMPENHO DA BUSCA POR SIMILARIDADE UTILIZANDO O PARADIGMA MAPREDUCE

Paulo Vinicius Mendonça Cardoso

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Sergio Luis Sardi Mergen

429
Santa Maria, RS, Brasil

2016

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Bacharelado em Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

ANÁLISE DO DESEMPENHO DA BUSCA POR SIMILARIDADE UTILIZANDO O PARADIGMA MAPREDUCE

elaborado por
Paulo Vinicius Mendonça Cardoso

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Sergio Luis Sardi Mergen, Dr.
(Presidente/Orientador)

Ana Trindade Winck, Drª. (UFSM)

Patrícia Pitthan de Araújo Barcelos, Drª. (UFSM)

Santa Maria, 16 de Dezembro de 2016.

AGRADECIMENTOS

Em primeiro lugar, agradeço às duas pessoas mais importantes da minha vida, que fizeram o possível e o impossível para que eu pudesse realizar esse sonho: Paulo e Terezinha, meus pais. Obrigado por nunca terem deixado de acreditar em mim. Jamais vou poder expressar o quanto sou agradecido por tudo. Agradeço também à minha irmã, Patrícia, e ao Dudu por toda motivação e apoio que sempre me ofereceram.

Um agradecimento especial à Bruna, minha namorada e grande companheira, que sempre esteve ao meu lado, me apoiando e dando força tanto nos momentos alegres como nos difíceis, de estresse e fim de semestre. Obrigado por me fazer tão bem e por tudo o que tu significa para mim. Te amo muito.

À minha família, por todo o apoio. Aos meus padrinhos e madrinhas, Dagna, Valdo, Rose e Silvio, que considero meus pais de coração. À todos meus tios(as), primos(as), aos meus avós Jodi, Darci e em especial à vó Santa e vô Cardoso que infelizmente não estão conosco, mas tenho a absoluta certeza que, onde quer que estejam, sempre permaneceram ao meu lado.

Ao meu orientador Sérgio, pelo enorme suporte desde o início do BioID até o TG, por todos os ensinamentos, conselhos e pela grande pessoa que é. Às professoras Ana e Patrícia, que participaram da avaliação do meu trabalho. E aos demais professores que muito me ensinaram e ajudaram nesta trajetória, desde a pré-escola até a graduação.

Aos grandes amigos que a UFSM e o curso de Ciência da Computação me deram, principalmente ao pessoal da turma: Leonardo, Flavio, Jhillian, Mateus, Vinicius e Peripolli. Valeu pelas risadas, partidas de COD e pelo companheirismo, mesmo com alguns imprevistos. Ao pessoal do PET-CC e todos os colegas que tive a oportunidade de estar junto. Aos amigos de Rosário e Santa Maria pela grande amizade, muitos desde os tempos de Manolos FC, outros tantos desde o ensino médio e fundamental.

Enfim, à todos que contribuíram de alguma forma nesta etapa tão importante, fica aqui o meu sincero muito obrigado.

“Sometimes it is the very people who no one can imagine anything of who do the things no one can imagine.”
— THE IMITATION GAME (ALAN TURING)

RESUMO

Trabalho de Graduação
Curso de Bacharelado em Ciência da Computação
Universidade Federal de Santa Maria

ANÁLISE DO DESEMPENHO DA BUSCA POR SIMILARIDADE UTILIZANDO O PARADIGMA MAPREDUCE

AUTOR: PAULO VINICIUS MENDONÇA CARDOSO
ORIENTADOR: SERGIO LUIS SARDI MERGEN
Local da Defesa e Data: Santa Maria, 16 de Dezembro de 2016.

A Recuperação de Informação (RI) é uma área de pesquisa envolvida na criação de soluções para buscas em repositórios de dados, a fim de se atender à uma necessidade de informação do usuário. Uma estrutura bastante utilizada para consultas em RI é o índice invertido, onde uma entrada do índice leva à lista de objetos associados. Nesse contexto, buscas por objetos podem ser definidas por equivalência ou por similaridade. Buscas por similaridade apresentam uma perspectiva mais poderosa, já que permitem a recuperação dos objetos mais similares às consultas. Porém, o cálculo de similaridade pode tornar o processo complexo e custoso, podendo ser necessário recorrer à técnicas alternativas de processamento. A computação distribuída foi criada para atender a esse tipo de problema, oferecendo soluções como ferramentas, modelos e arquiteturas distribuídas. Um paradigma de computação distribuída que pode-se aplicar em buscas com índice invertido é o MapReduce, proposto para o processamento de grandes quantidades de dados em ambientes de *cluster*. Desta forma, o objetivo deste trabalho é analisar o funcionamento de ferramentas de processamento distribuído que implementam o MapReduce em um problema de busca com índices invertidos. Os resultados mostram as diferenças de desempenho dos *frameworks* através de diversos cenários de teste.

Palavras-chave: Desempenho. Recuperação de Informação. Índice Invertido. MapReduce. Hadoop. Spark.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

PERFORMANCE ANALYSIS OF SIMILARITY SEARCH USING THE MAPREDUCE PARADIGM

AUTHOR: PAULO VINICIUS MENDONÇA CARDOSO
ADVISOR: SERGIO LUIS SARDI MERGEN
Defense Place and Date: Santa Maria, December 16th, 2016.

The Information Retrieval (RI) is a research area involved in creating solutions for databases search. The aim of RI is to answer a user information needed. A common data structure used to assist the search process is the inverted index, composed by entries that lead to a related object list. In this context, an object search can be done by equivalence or similarity. Similarity search is a powerful method, since it can retrieve the most similar objects according to the request. However, the complexity involved in computing the similarity can harm the performance, making it necessary to resort to alternative processing techniques. The distributed computing was created to help finding solutions for this type of problem, with tools, paradigms and distributed architectures. The MapReduce paradigm is an example of distributed model with the purpose of processing a big amount of data on cluster environments. This model fits into the inverted index search context because of its key-value architecture. Thus, the aim of this work is to analyses the distributed processing tools that implement the MapReduce concept over a similarity search problem that relies on an inverted index. The results shows how different frameworks behave under several test scenarios.

Keywords: Performance, Information Retrieval, Inverted Index, MapReduce, Hadoop, Spark.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

RI	Recuperação de Informação
HPC	*High Performance Computing*
HPCC	*High Performance Computing Cluster*
IaaS	*Infrastructure as service*
SaaS	*Software as service*
Paas	*Platform as service*
IoT	Internet das Coisas
HDFS	*Hadoop Distributed File System*
YARN	*Yet Another Resource Navigator*
RDD	*Resilient Distributed Dataset*

SUMÁRIO

1 INTRODUÇÃO

A Recuperação de Informação é uma área de pesquisa que visa a busca por objetos armazenados a partir de consultas. Em uma consulta, são definidos um ou mais termos que caracterizam os objetos que se pretendem acessar (KORFHAGE, 2008). Esses objetos definem a informação de interesse da consulta e devem ser acessados de forma que o usuário tenha suas expectativas cumpridas ao realizar a busca. Isto é, a eficiência de uma aplicação de RI preza pela satisfação de busca do usuário, em que os objetos mais relevantes à consulta devem ser priorizados.

Em um sistema de busca, os objetos podem ser armazenados em memória primária, quando há espaço suficiente, em memória secundária ou em ambos os meios de armazenamento. Um fator importante para a manutenção da base de dados consiste na sua indexação, através de estruturas que auxiliem no acesso aos dados. Uma estrutura bastante usada em problemas de busca é chamada de índice invertido, onde uma entrada (chave) do índice leva à lista de objetos (valores) que estão associados à essa entrada.

No contexto da RI, consultas compostas por termos podem ser usadas para a realização de buscas por equivalência ou por similaridade. No primeiro caso, o resultado final deve considerar apenas objetos que possuem total semelhança com os termos solicitados. Por outro lado, buscas por similaridade apresentam-se como opções mais abrangentes, uma vez que os objetos são classificados de acordo com um nível de proximidade ao que foi requisitado.

A abordagem de consultas usando métricas de similaridade oferecem uma perspectiva de busca mais poderosa, já que permite a recuperação dos objetos mais similares possíveis à uma dada consulta, os quais não seriam recuperados em uma abordagem de equivalência. Em contrapartida, buscas por similaridade possuem uma grande complexidade envolvida, podendo tornar o processo de Recuperação de Informação custoso. Dependendo de como a similaridade for computada, da quantidade de objetos indexados e a quantidade de termos da consulta, pode ser necessário recorrer a técnicas alternativas de processamento.

Para resolver esse tipo de impasse tem-se o conceito de computação distribuída, que busca resolver problemas através da distribuição e paralelização de tarefas em ambientes com mais de uma máquina. Um ambiente distribuído comumente usado é o *cluster*, composto por máquinas (nós) interligadas entre si através de uma interface de rede, com o propósito de realizarem um trabalho de forma cooperativa.

Uma opção de modelo de computação distribuída que pode ser interessante para o caso de busca com índice invertido é o paradigma MapReduce. Proposto por (DEAN; GHEMAWAT, 2008), o MapReduce é um modelo voltado para o processamento de grandes quantidades de dados e amplamente utilizado em contextos de processamento distribuído e paralelo. Esse modelo é constituído de duas etapas essenciais: *map* e *reduce*. Aplicações do MapReduce foram projetadas para serem executadas em ambientes de *cluster*, a partir da definição de um nó mestre (que realiza o controle do processo) e diversos nós trabalhadores (estes realizam o processo propriamente dito).

O uso do MapReduce neste trabalho justifica-se pela semelhança do seu funcionamento com o conceito de índice invertido. Isto é, o paradigma de processamento distribuído faz o uso de listas contendo pares chave-valor para a comunicação entre cada etapa de sua aplicação, bem como é estruturada a indexação invertida.

Desta forma, o objetivo geral deste trabalho é realizar uma análise de desempenho e usabilidade de ferramentas que implementam o paradigma MapReduce no contexto da Recuperação de Informação. Para isso, serão propostos mapeamentos de busca por similaridade que utilizem a estrutura de índice invertido para execução em ambientes centralizados e distribuídos, a fim de determinar uma comparação entre essas abordagens.

Além disso, será feito um estudo do comportamento dessas ferramentas em um ambiente de *cluster*, possibilitando a análise e a identificação de casos em que o cenário distribuído passa a ser mais vantajoso em relação aos métodos tradicionais. Para um melhor estudo, os mapeamentos serão testados levando em consideração diversos fatores, como a variação tamanho da base de dados e do número de termos, além de configurações das ferramentas utilizadas.

O trabalho está dividido da seguinte forma: no capítulo 2 é descrita uma fundamentação teórica sobre os principais conceitos utilizados. Já no capítulo 3 o enfoque é a revisão das ferramentas utilizadas. O capítulo 4 conta com a definição dos mapeamentos e algoritmos criados. O capítulo 5 expõe os resultados alcançados diante dos diversos cenários criados, tendo como estudo de caso o BioID: um sistema de indexação colaborativa de seres vivos que contém uma base de dados sobre espécies catalogadas de forma taxonômica. Por fim, o capítulo 6 apresenta as considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve uma fundamentação dos conceitos teóricos utilizados neste trabalho. Serão discutidas as definições de *Big Data*, Sistemas Distribuídos, o modelo de computação MapReduce, além dos desafios e aplicações de Recuperação de Informação.

2.1 Big Data

A quantidade de informação processada e armazenada sofreu uma grande ascensão, chegando a grandes níveis de crescimento. Segundo (SAGIROGLU; SINANC, 2013), estima-se que o volume de dados gerados até o ano de 2003 tenha alcançado cerca de 5 exabytes (10^{18} bytes), chegando à marca de 2.72 zettabytes (10^{21} bytes) em 2012. Até 2020, a estimativa é de que o tamanho do universo digital duplique a cada ano (GANTZ; REINSEL, 2012). Com esse grande avanço tecnológico, aliado à noções como IoT e computação em nuvem, a análise de dados tem incorporado-se nas mais diversas áreas de aplicação, servindo como um importante fator de produção na indústria (MANYIKA et al., 2011).

O conceito de *Big Data*, apesar de não possuir uma definição concreta, foi criado para atender a uma requisição cada vez mais frequente e abrangente de processamentos sobre bases de dados que são caracterizados por uma grande complexidade e, principalmente, por grandes dimensões. A essência dessa concepção se propõe a oferecer um auxílio para esse tipo de problema através de ferramentas, como *softwares* e serviços, além de paradigmas e modelos de computação.

No entanto, *Big Data* pode envolver, além do $Volume$, outros fatores que aumentam a complexidade envolvida. A $Velocidade$ se refere ao tempo que determinado problema deve ser processado, visto que muitas requisições possuem urgências nesse quesito; outro elemento é a $Variedade$, atribuído à diversidade de informações e fontes de dados com que se pode trabalhar; por outro lado, a análise de dados não confiáveis pode gerar inconsistência no estudo, fazendo com que o elemento $Veracidade$ se faça necessário; finalmente, uma solução de *Big Data* deve trazer benefícios e compensar o investimento do trabalho realizado, definido pelo $Valor$ da operação. Os fundamentos citados fazem parte dos 5 V's que definem a *Big Data* (TAURION, 2013).

A partir disso, a análise de informações que se encaixam no contexto da *Big Data* tem

criado uma demanda cada vez maior de recursos computacionais de alto desempenho. Por outro lado, essa exigência pode se tornar um problema, uma vez que a aquisição de infraestruturas que sejam capazes de atender aos requisitos esperados acaba, muitas vezes, tornando-se inviável. Além disso, as soluções para esses problemas tornam-se ineficientes à medida que o volume de dados cresce, principalmente quando utiliza-se os métodos tradicionais de processamento em que nada é paralelizado ou distribuído.

2.2 Processamento em Sistemas Distribuídos

O conceito de sistemas distribuídos surgiu para que a soma de recursos computacionais independentes seja capaz de cooperar para a realização de um processamento mais eficiente, aparecendo para o usuário final como um sistema único (TANENBAUM; VAN STEEN, 2007). Desta forma, é possível paralelizar parte da demanda em um ambiente distribuído, além de fazer com que recursos (como *hardware*, *software* e até mesmo informações) possam ser compartilhados entre vários computadores (COULOURIS et al., 2013). A comunicação entre várias máquinas a fim de oferecer serviços e compartilhamento de recursos é visto de forma globalmente difundida na Internet, principalmente através da *World Wide Web* (CHOO; DETLOR; TURNBULL, 2013).

A técnica de distribuição entre sistemas é importante quando tem-se obstáculos no que concerne ao desempenho de problemas que envolvam *Big Data* e a aquisição de computadores voltados à computação de alto desempenho (HPC) não é viável. Nesse âmbito, pode-se definir modelos de disposição e modelagem para atender à todo tipo de necessidade. Nesta seção serão abordadas duas concepções de sistemas distribuídos utilizadas para este trabalho: os *clusters* e a computação em nuvem (*cloud computing*).

2.2.1 Clusters

Um dos ambientes mais comuns de sistemas distribuídos são os *clusters*. Este modelo de distribuição consiste em um conjunto de máquinas (nós), interligadas através de uma interface de rede, que realizam um trabalho cooperativo como se fosse um único sistema computacional de alto desempenho.

O uso de um *cluster* geralmente se dá em programações paralelas, em que um único programa pode ser executado em paralelo através da distribuição em vários nós (PITANGA,

2004). Essa distribuição é controlada por um dos nós (*master node*), que define e faz a disposição da carga de tarefas para os nós escravos (*slave nodes*), conforme é mostrado na Figura 2.1. Neste caso, não é necessário que os nós possuam configurações de *hardware* e *software* idênticas entre si, mas a homogeneidade pode facilitar a comunicação entre nós e o controle de recursos do sistema.

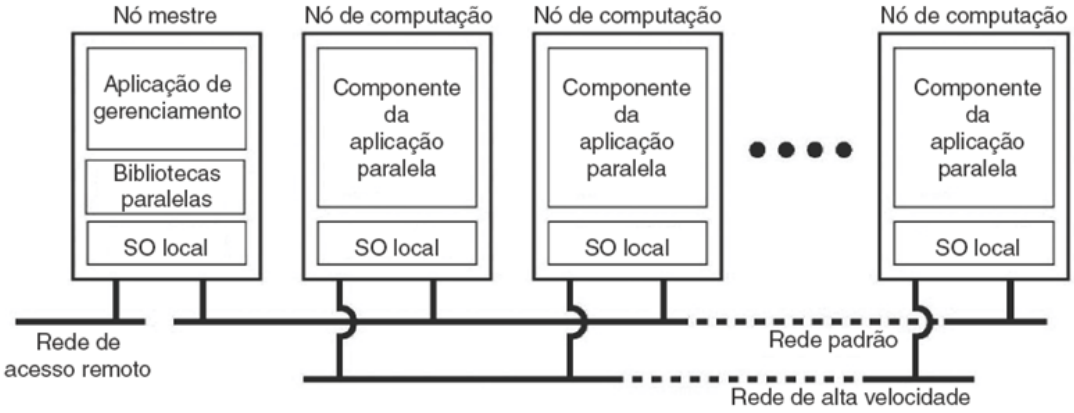

Figura 2.1 – Exemplo de um sistema de *cluster* (TANENBAUM; VAN STEEN, 2007)

Uma das principais vantagens de um *cluster* é a capacidade de processamento de um supercomputador, podendo-se usar recursos computacionais que não possuem configurações tão potentes. O projeto Beowulf (BECKER et al., 1995) descreve um *cluster* econômico configurado por computadores pessoais que continham *hardwares* de níveis convencionais. Outra característica importante desse ambiente é a sua alta escalabilidade. Isto é, um *cluster* pode ter controle da quantidade de nós, bem como inserção e remoção de máquinas, sem que exista uma quebra de transparência para o usuário final.

Os *clusters* podem ser categorizados para três propósitos principais: alto desempenho, alta disponibilidade e balanceamento de carga. O *Cluster* de Alto Desempenho (*High Performance Computing Cluster*) é voltado para tarefas de processamento que demandam eficiência. Os benefícios deste tipo de *cluster* se encontram, principalmente, na análise de *Big Data* e para que os resultados dessa análise seja gerado de forma rápida. Por sua vez, um *cluster* voltado para alta disponibilidade visa a manutenção do funcionamento de uma determinada aplicação. Já um *cluster* para balanceamento de carga tem como objetivo a distribuição uniforme de requisições entre os nós, ao invés de haver uma tarefa em comum a ser paralelizada.

2.2.2 Cloud computing

O crescente avanço do uso da Internet, aliado à velocidade de comunicação, possibilitou que a computação em nuvem (*cloud computing*) surgisse como um novo paradigma que busca realizar a entrega *online* de serviços e recursos computacionais. Essa demanda ocorre por meio de aplicações e *softwares* disponibilizados em conjunto com um ambiente (*cloud*) definido por *softwares* e *hardwares* controlados pelo provedor da nuvem (ARMBRUST et al., 2009).

Em termos gerais, a *cloud computing* provê um padrão de acesso sob demanda à uma série de serviços e recursos computacionais configuráveis, possibilitando o uso dos mesmos em um ambiente adaptável e facilmente escalável. A popularização deste modelo se deu pela expressiva redução do investimento em infraestrutura necessário para o desenvolvimento de soluções computacionais, que pode ser substituído pela contratação de serviços em nuvem. Desta forma, o usuário do serviço tem a possibilidade de aumentar sua produtividade economizando o tempo de configuração e manutenção dos recursos.

As ofertas apresentadas pela arquitetura *cloud computing* são divididas em três categorias essenciais, como mostra a Figura 2.2. Um serviço do tipo *Software as Service* (SaaS) oferece o uso de softwares que estejam rodando em uma infraestrutura de nuvem, porém restringe a configuração do ambiente e do *software* de forma que o usuário possa modificar poucos detalhes. Com a *Platform as Service* (PaaS), o utilizador pode realizar o *deploy* de suas próprias aplicações na nuvem, limitando-se a desenvolver de acordo com o que o ambiente suporta.

No nível mais abrangente de serviços encontra-se a *Infrastructure as Service* (IaaS), que delega ao usuário a configuração de recursos de processamento, rede e armazenamento, concedendo liberdade para escolha de parâmetros mais específicos de configuração, como o sistema operacional e as ferramentas utilizadas. Utilizando um serviço de infraestrutura o usuário consegue usufruir melhor dos recursos contratados, podendo formar *clusters* de baixo custo através da virtualização de sistemas.

2.3 MapReduce

O MapReduce foi proposto pela *Google* como um novo paradigma de programação para processamento de grandes volumes de dados de forma distribuída (DEAN; GHEMAWAT, 2008). O principal objetivo do MapReduce é oferecer um modelo de processamento em que um conjunto de dados possa ser distribuído em ambientes distribuídos de forma simplificada.

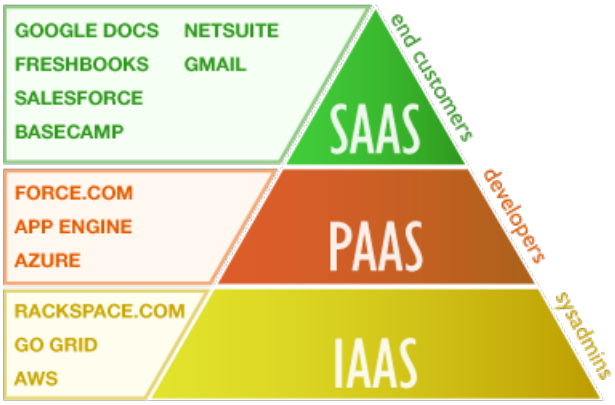

Figura 2.2 – Hierarquia dos serviços de computação em nuvem (FERNANDEZ et al., 2012).

Essa distribuição implica na divisão da carga do processamento entre os nós a fim de aumentar o desempenho da aplicação, podendo-se chegar a níveis satisfatórios mesmo usando máquinas de baixo custo. Além disso, o paradigma oferece outras vantagens como o controle do balanceamento de carga entre as máquinas envolvidas, mecanismos de tolerância a falhas e uma grande escalabilidade. Assim, boa parte do processo de distribuição é abstraído e o trabalho de análise de dados passa a ser mais produtivo.

O ambiente mais indicado para a execução de algoritmos que utilizam o modelo MapReduce são os *clusters*, em que nós são definidos por diversas máquinas que interagem entre si para a execução de uma tarefa (*job*). Os nós de um *cluster* possuem diferentes papéis na execução de uma tarefa MapReduce. O nó *master* é responsável por atender requisições de execução e gerenciar as tarefas, enviando-as aos outros nós, denominados *slaves*. A transferência de dados entre os elementos do *cluster* é facilitada pelo uso de um sistema de arquivos distribuído (*DFS*).

O MapReduce é composto por duas funções essenciais, baseadas em métodos de linguagens funcionais: o mapeamento (*map*) e a redução (*reduce*). A função *map* foi pensada para que o processamento pudesse ser mapeada no ambiente distribuído, dividindo-se o(s) arquivo(s) de entrada entre os nós. O *reducer*, por sua vez, encarrega-se de agrupar as saídas pelos mapeamentos e, se necessário, realizar algum tipo de processamento antes de gerar a saída final da aplicação.

O mapeamento é responsável por realizar uma divisão da entrada entre os nós do ambiente, com base em algum fator, como o tamanho do arquivo, a quantidade de linhas ou qualquer outra condição definida pelo programador. Os fragmentos criados são chamados de *splits*. Desta forma, cada nó fica responsável por parte da entrada e a análise do problema como um

todo passa a ser feita em partes, de forma paralela. Uma vez feita a distribuição, os nós realizam uma ou mais funções *map* para a entrada a qual estão alocados, em que a quantidade de mapeamentos também é definida por um fator de divisão. Com os dados definidos e divididos, o *mapper* pode impor um processamento sobre a entrada e, então, transmitir sua saída como um ou mais pares chave-valor.

Ao final dos *mappers*, suas saídas são agrupadas de acordo com a chave dos pares chave-valor. Novos pares são gerados contendo todos os valores associados às chaves correspondentes. Logo após, o *reducer* é executado recebendo como entrada os pares chave-valor já agrupados. Para cada entrada, é realizado um processamento e, novamente, a transmissão de saída ocorre por meio de pares chave-valor, que juntos configuram a saída do *job*.

A Figura 2.3 mostra as principais etapas do MapReduce para o exemplo de um contador de palavras em uma entrada de texto. Os dados de entrada são divididos em diferentes *splits*, que por sua vez executam um *mapper* específico. Os *mappers* são alocados através dos nós escravos para que cada elemento trabalhe de forma paralela. Após o processamento, a saída de cada *mapper* é agrupada em uma saída intermediária armazenada no *DFS* e a etapa de redução é iniciada. A saída também é armazenada no sistema de arquivos distribuído.

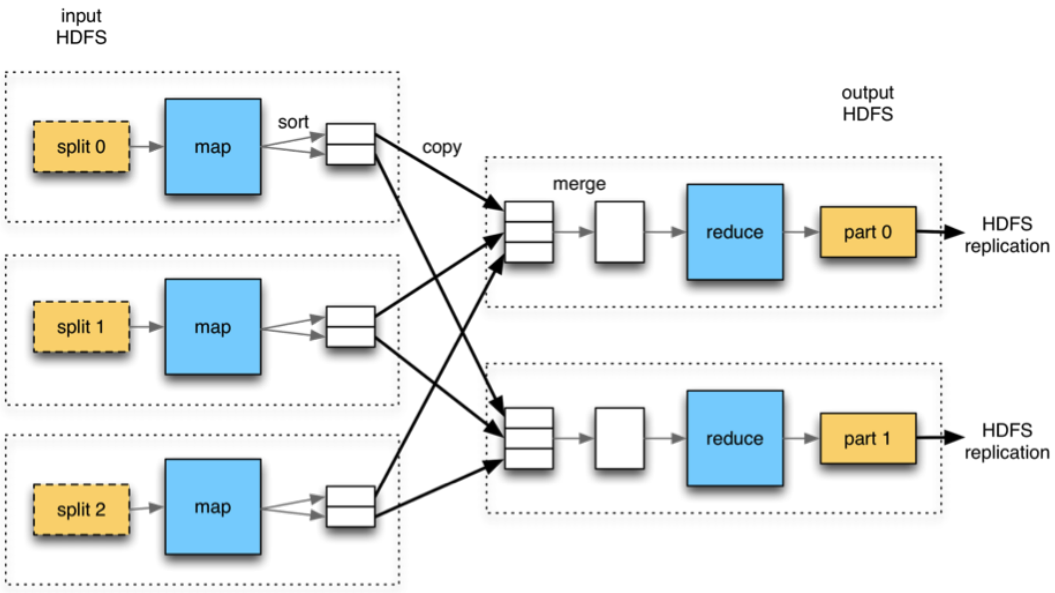

Figura 2.3 – Principais etapas descritas pelo paradigma MapReduce (WHITE, 2012)

A função de agrupamento pode ser modificada, através da função *combiner*, que é utilizada para se conseguir resultados diferentes na aplicação. Essa função atua em uma camada intermediária, entre o *mapper* e o *reducer*. O objetivo da implementação do *combiner* é realizar

um pré-processamento com a saída do *mapper*, de forma a modificar os dados a serem enviados para a função *reduce*.

2.4 Recuperação de Informação

A Recuperação de Informação (RI) é uma área de pesquisa que visa o desenvolvimento de ferramentas para a representação, armazenamento e busca de informações a partir de consultas. Em uma expressão de consulta são informadas palavras chave (termos) que caracterizam os objetos que o usuário deseja ter acesso (KORFHAGE, 2008). Esses objetos definem a informação de interesse da consulta e devem ser acessados de forma fácil e eficiente, de forma que o usuário tenha suas expectativas cumpridas ao realizar a busca.

Um sistema de RI é composto por três camadas essenciais: identificação da necessidade de informação (requisição), representação dos objetos e especificação da função de comparação para definições de relevância. A Figura 2.4 mostra o processo de recuperação com as etapas citadas. A arquitetura é adaptada do modelo exibido em (BAEZA-YATES; RIBEIRO-NETO, 2013, p. 7).

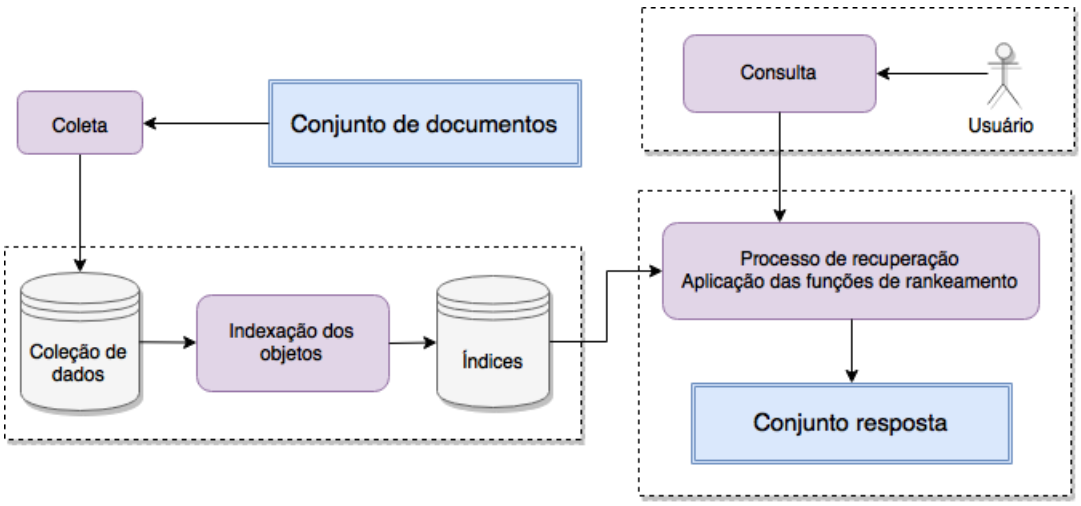

Figura 2.4 – Processo de Recuperação de Informação

Os elementos que contém informações podem ser caracterizados de diversas formas, como documentos hierárquicos em formato XML, tuplas de relações, arquivos de mídia, entre outros. Isto é, os objetos tratados pela RI não assumem apenas dados bem estruturados, podendo-se trabalhar com textos de linguagem natural ou documentos não totalmente estruturados. Para facilitar e acelerar o processo de recuperação, é viável a utilização de técnicas de indexação em que os objetos passam a ser representados através de estruturas de dados. Dentre

as diferentes estruturas existentes, destaca-se o conceito de índice invertido, que será mostrado mais adiante. O armazenamento é feito principalmente em memória primária, quando existe espaço suficiente nesse meio para armazenar toda a coleção de objetos existentes, ou em memória secundária, caso essa memória de acesso mais rápido seja insuficiente.

Após a indexação, o usuário pode utilizar o sistema para submeter consultas. O papel do usuário é informar, através da consulta, a sua necessidade de informação, muitas vezes expressa por um conjunto de palavras-chave ou texto livre. A consulta, que pode ser modificada pelo sistema a fim de corrigir e adaptar os termos, é então processada pela etapa de Recuperação para que os documentos do conjunto resposta sejam criados.

Uma vez que os objetos recuperados tenham sido reunidos no documento final, o sistema pode iniciar a etapa de ranqueamento para que os objetos com maior probabilidade de satisfazer a necessidade do usuário tenham preferência. Essa etapa é essencial para definir a qualidade da recuperação.

Diante disso, pode-se observar que a principal diferença da área de RI para a Recuperação de Dados (RD) é a forma como o resultado final é interpretado. Na RD, um único objeto incorreto entre os recuperados faz com que todo o processo seja considerado falho. Por outro lado, a Recuperação de Informação possibilita o uso de técnicas de relevância para buscar, além de elementos exatamente relacionados à consulta, objetos que possuam um certo nível de afinidade (BAEZA-YATES; RIBEIRO-NETO, 2013). Em outras palavras, a recuperação de informação não se preocupa em responder a uma requisição de forma precisa, mas identificar objetos que satisfaçam a necessidade de informação do usuário (CHOWDHURY, 2010).

Uma das estruturas de dados comumente usadas em problemas de recuperação de informação são os índices invertidos. Essa estrutura é útil para responder a consultas formuladas como uma coleção de termos (palavras chave). Um índice invertido é formado por uma chave (vocabulário) e suas ocorrências. Um vocabulário consiste em todas as diferentes palavras encontradas na base de dados, e as ocorrências são referências aos documentos que possuem aquela entrada. O nome índice invertido explica-se pelo fato de podermos reconstruir todos os documentos através da estrutura de indexação (BAEZA-YATES; RIBEIRO-NETO, 2013, p. 342).

A Figura 2.4 exemplifica o processo de indexação utilizando índice invertido. As ocorrências normalmente são armazenadas de forma ordenada, semelhante à estrutura de um dicionário. A ordenação pode ser útil para facilitar o processo de busca no índice.

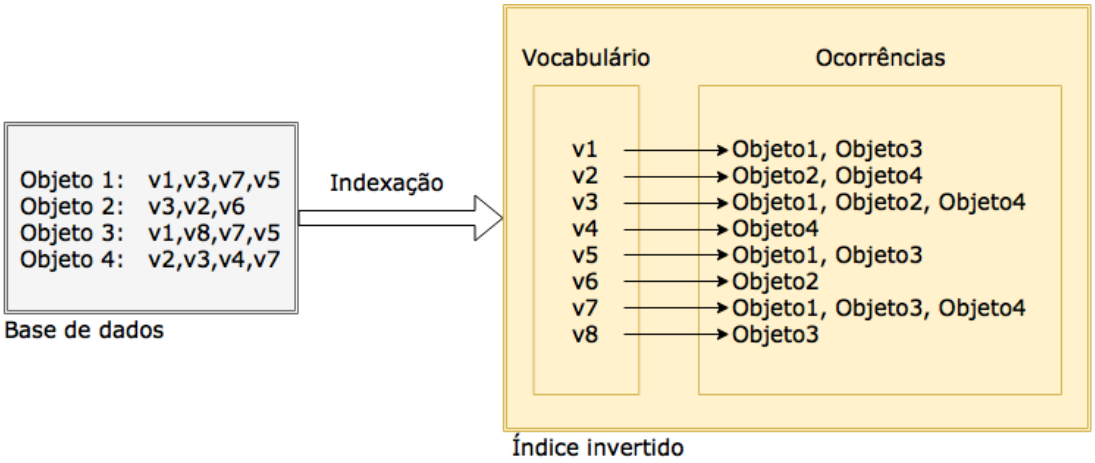

Figura 2.5 – Exemplo de uma estrutura de índice invertido

O processo de busca em um índice invertido é feito tomando o termo pesquisado e retornando a lista invertida associada. Esse processo é atraente quando se trabalha com um sistema de consultas em que os termos de busca são definidos por uma lista de palavras chave. Desta forma, o índice auxilia a procura pelos objetos requisitados oferecendo um acesso direto à localização dos mesmos. A busca em um índice invertido pode ser feita através de várias estruturas de dados, como árvores ou tabelas *hash*.

3 FERRAMENTAS UTILIZADAS

Neste capítulos serão abordadas as definições e especificações das ferramentas utilizadas no trabalho: Apache Hadoop e Apache Spark.

3.1 Apache Hadoop

O Hadoop é um projeto de código aberto originalmente proposto pela Yahoo!, junto ao projeto Nutch, e mantido pela Apache com o objetivo de fornecer um *framework* para o desenvolvimento de aplicações distribuídas e o armazenamento de grandes quantidades de dados. O Hadoop é inspirado no paradigma MapReduce e oferece uma das implementações mais utilizadas desse modelo de programação.

A arquitetura do Apache Hadoop consiste de duas camadas principais: um sistema de arquivos distribuídos (HDFS) e de um gerenciador de recursos computacionais (YARN). Abaixo desses dois elementos, diversas ferramentas e tecnologias são descritas para vários tipos de aplicações. Por isso, além do MapReduce, existem projetos como o Pig, Hive e o HBase que também são oferecidos pelo Hadoop. Além disso, também existe o módulo Hadoop Commom que define um conjunto de bibliotecas e arquivos necessários para a manutenção dos elementos do *framework*.

O *Hadoop Distributed File System* (HDFS) é um sistema de arquivos distribuído utilizado pelo Apache Hadoop a fim de armazenar e replicar dados com grandes volumes através do ambiente distribuído utilizado. Os arquivos são armazenados com o conceito de bloco, assim como sistemas de arquivos convencionais. O tamanho padrão de um bloco pode ser de 64MB ou 128MB, dependendo da versão do *framework* utilizada.

A composição do HDFS é divida pelo papel de cada nó em relação à manutenção dos dados, através duas categorias: *DataNodes* e *NameNodes*. O nó mestre implementa o *NameNode* e tem a responsabilidade de gerenciar informações e metadados sobre os arquivos armazenados no HDFS, mas sem trabalhar com os dados propriamente ditos. Por sua vez, o *DataNode* é implementado pelos nós escravos e tem como objetivo o real armazenamento dos dados no *cluster*, além de fazer realizar operações CRUD sobre os arquivos.

Para o gerenciamento de todos os recursos disponíveis e para a alocação de tarefas, o Hadoop utiliza a ferramenta *Yet Another Resource Navigator* (YARN). Assim como o HDFS e o MapReduce, o YARN é dividido em uma arquitetura mestre-escravo, onde um nó mestre deve

implementar a função de *ResourceManager*, enquanto os nós escravos executam o *NodeManager*, conforme mostra a Figura 3.1.

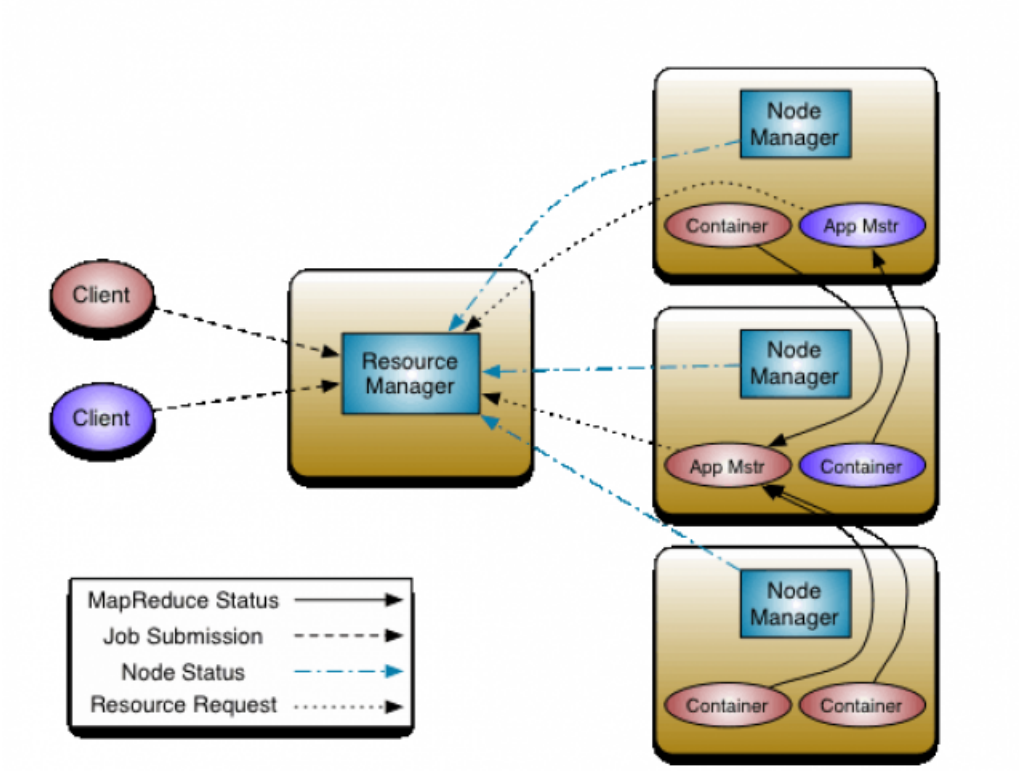

Figura 3.1 – Estrutura do gerenciador de recursos usado pelo Hadoop (WHITE, 2012)

O *ResourceManager* deve ter uma comunicação com o cliente da aplicação, uma vez que o nó mestre deve atender às requisições de aplicações. Ademais, essa função deve realizar o gerenciamento dos recursos do *cluster* e o escalonamento de tarefas para as aplicações. Já o *NodeManager* gerencia e oferece seus recursos ao *ResourceManager*, para que este decida como o trabalho será executado. As tarefas são executadas usando a concepção de *container*, o qual define a parcela de recurso que cada *NodeManager* pode usar para determinada tarefa. Nessa arquitetura também existe um módulo *ApplicationMaster* para cada aplicação, que gerencia as chamadas de tarefas e as remete aos *containers* disponíveis.

Uma aplicação no Hadoop Madurece é chamada de *Job* e, como especifica o paradigma, tem como composição as funções *map* e *reduce*. A entrada de dados é dividia em *splits* de acordo com uma métrica definida pelo tamanho do arquivo e o tamanho de um bloco no HDFS. Cada *split* criado origina uma tarefa (*task*), que por sua vez executa um *mapper*. A saída dos *mappers* são agrupadas para que a etapa de redução seja executada e, finalmente, gerar a saída da aplicação.

Por consequência, o fluxo da submissão de uma aplicação MapReduce no Hadoop é composto pelas seguintes etapas: submissão de uma aplicação pelo cliente ao nó mestre do YARN, escalonamento de recursos (*containers*) pelo *ResourceManager*, realização do trabalho nos *containers* e execução do *ApplicationMaster* para monitoramento da aplicação. A Figura 3.2 resume o fluxo de trabalho do Hadoop MapReduce.

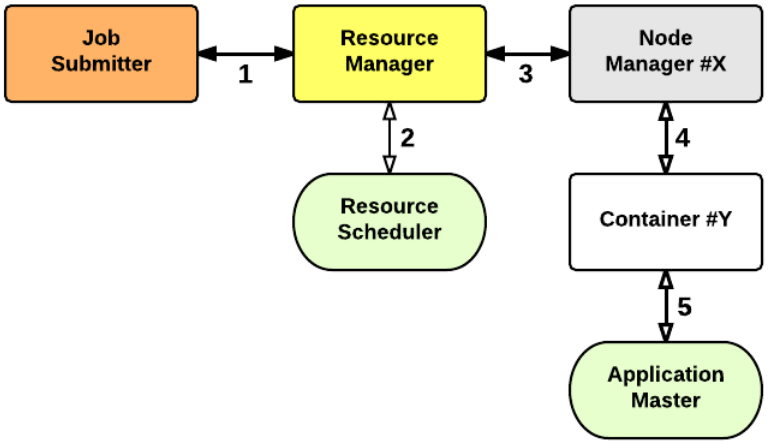

Figura 3.2 – Etapas básicas para a execução de um *job* no Hadoop MapReduce

Existem diversos tipos de algoritmos de escalonamento que o Hadoop pode suportar. Dois dos principais são: *CapacityScheduler* e *FairScheduler*.

- *CapacityScheduler*: é utilizado como escalonador padrão na versão 2.7.2 do Apache Hadoop e tem como princípio básico uma alocação de recursos para ambientes que tem concorrência de mais de um usuário (organização). Cada organização possui suas próprias finalidades e, por isso, esse *scheduler* dá a possibilidade dos usuários simularem um *cluster* MapReduce separado para cada organização, através de filas (*queues*) de requisições específicas, garantindo um mínimo de recursos. Assim, um *job* de uma organização é alocado para a sua respectiva *queue* e, dentro dessa fila, o *job* irá competir recursos através de outro processo de escalonamento, em que é usado a abordagem FIFO. Essa abordagem é interessante quando há mais de um usuário concorrendo pelo uso de um mesmo *cluster*, pelo fato do escalonador assegurar que as organizações sempre tenham, ao menos, uma parte do *cluster* disponível.

- *FairScheduler*: ao contrário do *CapacityScheduler*, o escalonador do tipo *FairScheduler* tem como objetivo a distribuição dos recursos do *cluster* de forma justa. Neste caso também há o conceito de organizações, sendo que um usuário requisita uma execução dentro de sua entidade específica. Contudo, a alocação dos recursos vê o *cluster* de modo geral,

oferecendo uma parcela igualitária para cada organização que esteja executando. Essa abordagem de distribuição justa suporta preempção, isto é, se um usuário não recebe uma quantia justa de recursos, outros *Jobs* que estejam usando mais recursos poderão ter tarefas interrompidas ou suspensas a fim de liberar recursos. Uma vantagem do *FairScheduler* em relação ao *CapacityScheduler* é a garantia que, quando existirem requisições de recursos suficientes para utilizar a capacidade máxima do *cluster*, o mesmo irá disponibilizar todo o seu poder computacional.

3.2 Apache Spark

O Apache Spark é um *framework* de código aberto voltado para processamento de soluções em *BigData*. Esse *framework* oferece uma alternativa ao uso do Apache Hadoop para o desenvolvimento de aplicações MapReduce, além da especificação de uma maior variedade de funções e de classes de aplicações, sem perder a característica de tolerância a falhas presente no Hadoop (KARAU et al., 2015).

A generalização dos estágios do MapReduce é uma das principais vantagens do Spark. A implementação pura do MapReduce, inclusive através do Apache Hadoop, limita o desenvolvedor a seguir os passos de mapeamento, agrupamento e redução, não havendo a possibilidade de realizar mais de um tipo de operação no mesmo *Job*. Por sua vez, o Apache Spark utiliza o conceito de Grafo Dirigido Acíclico (DAG) para a especificação de várias etapas complexas em uma mesma tarefa, em que há o compartilhamento de dados da memória entre os nós do grafo. Desta forma, o *framework* utiliza a memória para realizar suas operações, evitando ao máximo a leitura e escrita de dados em disco.

Essa arquitetura dá a liberdade de desenvolvimento de aplicações iterativas com operações não triviais. Porém, o Spark não deve ser considerado um substituto do MapReduce, mas uma alternativa mais abrangente que pode expressar de forma mais clara e eficiente diversos problemas de *BigData*.

A arquitetura do Apache Spark é composta por três camadas principais: a API, o armazenamento dos dados e o gerenciador de recursos, conforme mostra a Figura 3.3. A API oferece suporte para o desenvolvimento de aplicações baseadas no Spark de acordo com ferramentas para as linguagens Scala, Python e Java. O armazenamento do *framework* se dá pelo HDFS, ainda que exista suporte para uma série de serviços de armazenamento compatíveis com o Apache Hadoop. Já o gerenciador de recursos define a ferramenta que faz a manutenção dos

recursos do ambiente, sendo suportado um gerenciador *Standalone* do Spark, além do YARN e a ferramenta Mesos.

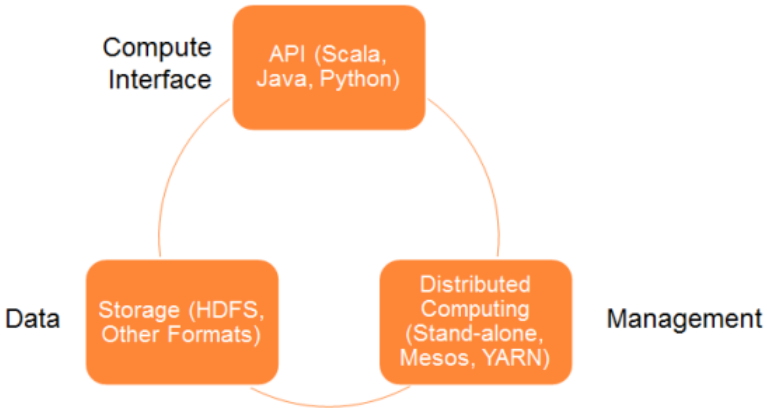

Figura 3.3 – Arquitetura do *framework* Apache Spark (PENCHIKALA, 2015)

A especificação do Spark ainda define uma abstração de dados chamada *Resilient Distributed Dataset* (RDD), que consiste em uma estrutura de dados que é armazenada de forma distribuída entre os nós do ambiente utilizado (ZAHARIA et al., 2012). Essa estrutura possui característica imutável, não sendo possível modificá-la. Se for necessário uma mudança, um novo RDD é criado com as modificações realizadas. Existem tipos de operações possíveis em RDD's: as transformações, que retornam novos *datasets* de acordo com a função (como *map*, *filter* e *reduce*), e as ações (*count*, *first* e *take*) que retornam informações avaliadas. O principal benefício de um RDD se dá pelo fato de não necessitar de uma replicação de dados para apresentar tolerância a falhas. Isso acontece pois o Spark armazena o método de criação (isto é, o histórico de funções empregadas) do RDD, de forma que uma falha pode ser totalmente recuperada ao reconstruir-se os *datasets* perdidos.

Além disso, um grande benefício proposto pelo Spark é a sua integração com outras ferramentas. Dentre as ferramentas, estão o Spark Streaming que permite o processamento de *streams* de informações de tempo real. Já o SparkSQL oferece um modulo para o uso de dados estruturados, além de uma engine SQL. Existe também um módulo de aprendizado de máquina, chamado MLib, que oferece implementações de algoritmos de *machine learning* e inteligência artificial. A arquitetura do Spark, com os módulos disponíveis, é mostrada na Figura 3.4.

O funcionamento do Spark segue um conceito semelhante ao visto no Apache Hadoop, com um nó *master* e vários nós trabalhadores (*workers*). Porém, conforme é exibido na Figura 3.5, o nó mestre implementa o *DriverProgram* e define o *SparkContext*, que por sua vez deve interagir com o gerenciador de recursos para alocar as tarefas. Assim que a conexão entre os

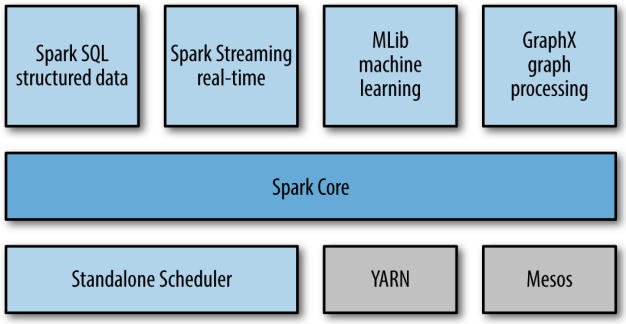

Figura 3.4 – Módulos disponíveis no Spark (KARAU et al., 2015)

nós é definida, o *framework* aloca processos (*executors*) de computação e armazenamento de dados nos nós trabalhadores, onde a computação é realizada por meio de *tasks*.

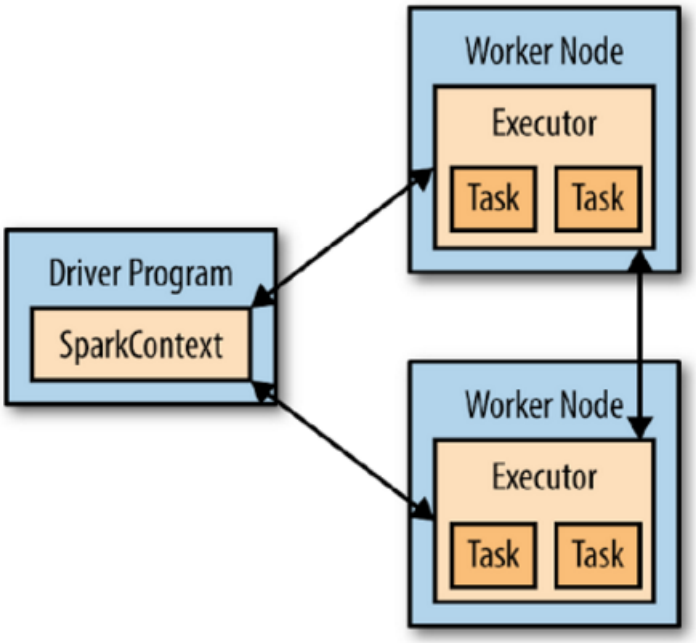

Figura 3.5 – Fluxo de trabalho no Spark (KARAU et al., 2015)

4 USO DE MAPREDUCE PARA PROBLEMAS DE BUSCA POR SIMILARIDADE

Esta seção descreve o desenvolvimento do trabalho e das propostas criadas para solução de problemas de Recuperação de Informação, através de buscas por similaridade com requisições *real-time* usando uma estrutura de índice invertido.

4.1 Representação dos dados

A representação dos dados usados utilizou o conceito de índice invertido, a fim de agilizar o processo de consulta. Os objetos são definidos de acordo com a Definição 1.

Definição 1 *(Objeto) Um objeto O^i é uma tupla $\{B^i, PI^i\}$, onde B^i é o corpo do objeto O^i, e PI^i é a sua coleção de propriedades indexáveis $\{P_1^i, P_2^i, P_3^i, ..., P_n^i\}$. Por sua vez, uma propriedade indexável P_j^i é um valor literal que complementa a descrição do objeto O^i.*

Para os objetivos do trabalho, é irrelevante definir o o conteúdo do corpo do objeto. Basta a noção de que um objeto possui propriedades indexáveis, que são informações que podem ser usadas como chaves em um índice invertido, como explicado na Definição 2.

Definição 2 *(Indice Invertido) Um índice invertido I é uma lista ordenada de entradas $\{E_1, E_2, E_3, ..., E_n\}$, onde uma entrada E_i é uma tupla $\{C_i, V_i\}$ associando uma chave C_i a um valor V_i. Por sua vez, um valor V_i pode ser definido como uma lista de objetos indexados $\{OI_1^i, OI_2^i, OI_3^i, ..., OI_m^i\}$, sendo que um objeto indexado corresponde à localização onde um objeto possa ser encontrado.*

Para que o índice seja válido, é necessário que, para cada propriedade indexável $\{P_j^i\}$, exista uma entrada E_k de modo que sua chave C_k seja igual a $\{P_j^i\}$ e algum dos objetos indexados OI_l^k contenha a localização de O^i. Ainda, é necessário que, para qualquer entrada E_i, sua chave C_i seja igual a alguma propriedade indexável P_k^j, de tal modo que O^j seja localizado por algum OI_l^i.

4.2 Metodologia de busca

O mapeamento proposto realiza a varredura do índice como um todo n vezes, sendo n o número de termos da consulta. A cada varredura, o nível de similaridade (escore) é calculado

entre os termos pesquisados e as chaves C_i do índice através de uma função de similaridade, de forma que seja feito um produto cartesiano entre as entradas do índice e cada elemento da consulta do usuário.

A Figura 4.1 ilustra um exemplo em que é calculado o escore de similaridade dos objetos a partir de uma consulta composta pelos termos 'Animalia instabilis'. Para simplificar, considere que seja usada uma função de equivalência, e não de similaridade, em que o valor 1 representa igualdade e o valor 0 representa desigualdade. Nesse exemplo didático, percebe-se que os documentos mais relevantes são aqueles que possuem o maior número de casamentos com termos da consulta.

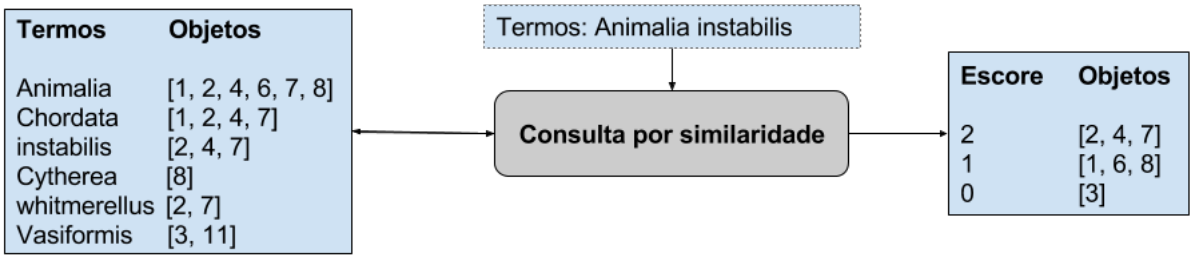

Figura 4.1 – Exemplo de uma consulta com o mapeamento proposto

A Definição 3 apresenta o que é uma consulta em um ambiente que permita acesso a objetos indexados.

Definição 3 *(Consulta por termos) Uma consulta por termos Q é uma coleção de termos $\{K_1, K_2, K_3, ..., K_n\}$, onde um termo K_i é um valor textual.*

A proposta de solução para o problema de busca foi criada com o objetivo de encontrar os objetos cujas propriedades indexáveis correspondam aos termos da frase de busca (composta por uma ou mais palavras). Para que isso aconteça, os objetos com maior nível de similaridade devem ter prioridade, já que possuem uma maior relação com a busca e são considerados mais relevantes. Dada uma consulta Q, a relevância de um objeto O_x é definido por um escore medido pela função $q\_escore(Q, O_x)$ apresentada na equação 4.1.

$$q\_escore(Q, O_x) = \sum_{i=0}^{n} \sum_{j=0}^{m} escore(k_i, P_j^x) \tag{4.1}$$

A função q_escore é a soma dos escores de similaridade entre cada uma das m propriedades indexáveis de um objeto e cada um dos n termos da consulta. O escore é normalizado de

0 a 1, onde 0 representa a ausência de similaridade e 1 representa a completa equivalência. Já o escore entre uma consulta e um objeto varia de 0 a $|PI|$.

O uso da função de escore faz com que nenhum objeto seja descartado da reposta, mesmo que a relação de similaridade calculada seja relativamente pequena, ou mesmo nula. Além disso, o produto cartesiano possibilita que cada uma das propriedades indexáveis contribua n vezes com o nível de similaridade final de um objeto, sendo n o número de termos da consulta. Assim, objetos com maiores números de propriedades tendem a ser considerados mais relevantes, mesmo que existam objetos mais semelhantes com a consulta mas que possuem poucas propriedades.

Uma técnica para contornar este problema é a aplicação de um *threshold*, que define um limite mínimo de escore a ser considerado para a soma. Desta forma, escores considerados irrelevantes são descartados da soma final, tornando a consulta mais específica sem a necessidade de eliminar objetos. A função de escore pode ser modificada para a definição do *threshold*, como mostra a Equação 4.2, a qual conta com a imposição da Equação 4.3.

$$q\_escore(Q, O_x, threshold) = \sum_{i=0}^{n} \sum_{j=0}^{m} th(escore(k_i, P_j^x)) \tag{4.2}$$

$$th(x) = \begin{cases} 0, & \text{se } x < threshold \\ x, & \text{se } x \geq threshold \end{cases} \tag{4.3}$$

Para a aplicação deste cenário de busca foram criadas duas abordagens de consulta. Inicialmente desenvolveu-se uma abordagem totalmente centralizada, de forma que o processo completo de busca seja realizado sem auxílio de ferramentas de distribuição ou paralelização. Em seguida, pensou-se um novo procedimento de busca distribuído, a fim de encontrar um melhor uso de recursos com o propósito de chegar a melhores desempenhos em relação à velocidade de processamento. As duas estratégias são discutidas a seguir.

4.3 Consulta centralizada

O objetivo do desenvolvimento de um módulo centralizado é a análise do problema de busca quando não há nada distribuído e todo o processo (incluindo a leitura do índice e os cálculos de escore) faz-se de forma sequencial. Este cenário é importante para que futuras comparações com abordagens distribuídas possam ser realizadas, tornando a centralização uma base (*Baseline*) para novos resultados a serem alcançados.

O processo de busca centralizado recebe um arquivo contendo o índice I, além da consulta Q realizada e um valor de *threshold*, que pode variar de 0% à 100%. A saída gerada cria um novo arquivo com os escores e cada objeto atrelado à esse valor. O Algoritmo 1 mostra as principais etapas do processo criado, onde todos os cálculos são feitos sequencialmente.

Algoritmo 1: BUSCA POR SIMILARIDADE CENTRALIZADA

```
Entrada: I, Q, threshold
Saída: Arquivo com similaridades calculadas
1  início
2      para cada k ∈ Q faça
3          para cada i ∈ I faça
4              s ← SIMILARIDADE(C_i, k, threshold)
5              se s > 0 então
6                  para cada objeto ∈ V_i faça
7                      F_objeto ← F_objeto + s
8                  fim
9              fim
10         fim
11     fim
12     para cada objeto ∈ F faça
13         soma ← F_objeto
14         S_soma.put(objeto)
15     fim
16 fim
17 retorna S
```

Através do Algoritmo 1, pode-se notar que o produto cartesiano é feito através da iteração nos termos k da consulta Q dada como entrada e as entradas do índice invertido. O escore encontrado é associado à cada um dos objetos correspondentes à entrada. Porém, caso o valor de *threshold* limite o escore calculado, este é ignorado. A associação do escore com o objeto é feita somando-se o escore anterior com o novo valor de similaridade. Ao final dos loops anteriores, mais uma iteração ocorre e os escore finais de cada objeto são armazenados em uma nova estrutura do tipo chave-valor, em que a chave é a similaridade e o valor são os objetos associados.

4.4 Consulta distribuída

Após o desenvolvimento do método centralizado, buscou-se implementar uma nova abordagem de consulta em que o processamento pudesse ser dividido entre mais de um recurso computacional, distribuindo e paralelizando o processo. Para tal, a distribuição do problema foi

desenvolvida utilizando o paradigma MapReduce, com o auxílio da ferramenta Apache Hadoop (v2.7.2). Os dados do índice foram armazenados no sistema de arquivos distribuídos oferecido pelo Hadoop (HDFS) e o controle de recursos ficou à cargo da ferramenta YARN.

A proposta de mapeamento para o problema de busca em índice invertido usando o modelo MapReduce tem como configuração de entrada o próprio arquivo do índice I. Neste mapeamento o intuito é fazer com que o índice invertido seja configurado como a entrada do processo para que o *framework* de distribuição faça a quebra da estrutura em uma ou mais divisões (*splits*), a fim de criar um *mapper* (*map task*) para cada divisão.

Fazendo com que o índice seja dividido em *splits*, a carga de processamento pode ser distribuída entre os nós do ambiente utilizado. A divisão da entrada é feita com base no tamanho do arquivo e em atributos de configuração do Hadoop. Os parâmetros usados para a definição dos *splits* são os valores $minimumSize$ e $maximumSize$, os quais descrevem, respectivamente, os valores mínimo (*default* 1 *byte*) e máximo (*default* 819 PB) possíveis para uma divisão. Além disso, também é considerado o tamanho de um bloco do HDFS ($blocksize$), definido pelo parâmetro $dfs.blocksize$. Assim, o tamanho de divisão para os *splits* segue a fórmula da Equação 4.4, descrita em (WHITE, 2012).

$$max(minimumSize, min(maximumSize, blockSize)) \tag{4.4}$$

Cada *split* criado deverá conter uma lista de entradas E do índice, podendo-se calcular os escores de forma independente entre as divisões. A divisão nem sempre se faz igualitária, já que o último *split* pode não completar o tamanho total definido. Essa abordagem de divisão foi escolhida por oferecer uma perspectiva escalável, uma vez que a inserção de mais nós no *cluster* e o crescimento do índice possibilitam uma maior distribuição naturalmente.

O Algoritmo 2 descreve o pseudo-código das funções *map* e *reduce* desenvolvidas. Todo *mapper* recebe uma lista de entradas E do índice I e, então, executa o procedimento de mapeamento para cada par chave-valor contido em E. Um único par é processado por vez, contendo um termo C_i (chave) e uma lista de objetos O_i^j associados (valor).

Cada entrada E processada pode emitir 0 ou mais saídas, as quais consistem em pares chave-valor configurados com os objetos do campo valor de E e o escore calculado.Na etapa de redução os escores são somados e levados à saída do algoritmo como um par chave-valor constituído do somatório das similaridades de cada objeto e os próprios objetos relacionados. A Tabela 4.1 resume a relação de entrada e saída das etapas *map* e *reduce*.

Algoritmo 2: FUNÇÕES MAP E REDUCE PARA A BUSCA POR SIMILARIDADE

```
1  Function map(key, value)
2      para cada k ∈ Q faça
3          s ← SIMILARIDADE(key, k, threshold)
4          se s > 0 então
5              para cada objeto ∈ value faça
6                  emit(objeto, s)
7              fim
8          fim
9      fim
10 end
11 Function reduce(key, value)
12     soma ← 0
13     para cada escore ∈ value faça
14         soma ← soma + escore
15     fim
16     emit(soma, key)
17 end
```

Etapa	Entrada	Saída
mapper	Lista de E	[{O^1, $Score_i$}, {O^2, $Score_i$}, ..., {O^n, $Score_i$}]
reducer	$list$({O, $Lista\ de\ scores$})	$list$({$sum(scores)$, $Lista\ de\ O$})

Tabela 4.1 – Entradas e saídas das etapas do mapeamento de busca distribuído

Para este caso, o *threshold* também pode ser aplicado para definir quais informações os *mappers* devem considerar no seu *output*. A aplicação do limite implica em um arquivo de saída menor ou igual à situação em que o *threshold* não é definido, diminuindo o volume de dados enviados ao *reducer*.

5 EXPERIMENTOS

Nos experimentos realizados, foi usado um *cluster* configurado através da ferramenta Google Cloud, que oferece serviços na nuvem de diversos tipos, inclusive a utilização de uma infraestrutura (IaaS). O *cluster* usado possui 3 nós, sendo 1 mestre e 2 escravos, com configurações idênticas: o processador conta com 2 núcleos virtuais, rodando em 2.4GHz, 4GB de memória RAM e 100GB de disco disponíveis por nó. Para a execução do mapeamento centralizado, os testes foram executados através do nó mestre. Já na consulta distribuída, o nó mestre ficou responsável por rodar o *ResourceManager*, para controle de recursos, mas também trabalhou como escravo atendendo à requisições de tarefas *map* e *reduce*, totalizando 3 nós trabalhadores.

O índice invertido foi criado sobre a coleção de objetos do sistema BioID (CARDOSO et al., 2015), que representa as espécies de seres vivos catalogadas junto aos seus atributos taxonômicos. Esses atributos, bem como seus nomes e valores, são definidos como as propriedades indexáveis dos objetos. Por exemplo, um objeto O_i indexado pode conter como PI os atributos $Reino$ e $Classe$ e seus respectivos valores $Animalia$ e $Mammalia$, também definidos como propriedades indexáveis. Assim, o índice deve ser composto por pares <PI_j, $Lista\ de\ O$> em que a lista de objetos contém todos seres vivos indexados que possuem a propriedade PI indexada.

Quanto à função de similaridade, foi implementada uma versão do algoritmo de distância entre termos de Levenshtein, que calcula o número de edições necessária para transformar um elemento textual em outro (RISTAD; YIANILOS, 1998). A distância é normalizada para que o valor de escore fique no intervalo de zero (sem nenhuma similaridade) a um (similaridade total).

Para testar a escalabilidade dos algoritmos, foram criadas novas versões do índice em que as entradas foram replicadas a fim de aumentar o número de propriedades indexáveis e objetos. O tamanho da base de dados foi aumentada 10, 20, 50 e 100 vezes em relação ao tamanho original. O número de entradas para cada tamanho da estrutura é descrita na Tabela 5.1. Além disso, o cenário de testes contou com uma variação do número de termos da consulta e do limite de *threshold*.

O resultado detalhado de todos os testes, incluindo as soluções centralizadas e distribuídas, além das diferentes ferramentas utilizadas, estão expostos no Apêndice A. O tempo de

Replicação	Número de entradas	Tamanho do arquivo
x1	27.000	1,9 MB
x10	270.000	25,4 MB
x20	540.000	50,9 MB
x50	1.350.000	127,5 MB
x100	2.700.000	255,2 MB

Tabela 5.1 – Número de entradas e tamanho dos arquivos para os índices utilizados

execução é mostrado em tabelas, levando em conta os diferentes tamanhos do índice, número de termos da consulta (1, 2, 5 e 10 palavras chave) e o níveis de *threshold* (0%, 25%, 50%, 80% e 90%).

5.1 Resultados com o Apache Hadoop

Inicialmente, três estratégias de busca foram comparadas: uma estratégia *Baseline*, que realiza o processamento centralizado, e outras duas utilizando o Apache Hadoop (v2.7.2), sendo um cenário Pseudo-Distribuído, em que apenas um nó executa o *job*, e outro Distribuído. O Hadoop foi definido com sua configuração padrão, ou seja, nenhum atributo ou elemento de configuração foi alterado.

O primeiro cenário para a comparação entre os resultados obtidos contou com uma variação no tamanho do índice. Foram usados o índice original e as versões replicadas em 20x, 50x e 100x, a fim de verificar em qual dos cenários a distribuição é vantajosa em relação ao *Baseline*. Neste caso, foi usado o cenário com 5 termos de consulta pois se aproxima da média de 4 atributos por objeto indexado no sistema BioID (CARDOSO; FRANZIN; MERGEN, 2016). Além disso, foi definido o limite de *threshold* em 50% com o Hadoop rodando suas configurações padrão de escalonador e tamanho de *split*.

Conforme o gráfico da Figura 5.1 exibe, a solução centralizada se mostra significativamente mais eficiente que as soluções com o Hadoop quando a base de dados é pequena. Porém, o uso do *framework* de distribuição passa a ser vantajoso conforme o volume de dados vai tornando-se maior.

Os resultados da Figura 5.1 mostram que a aplicação do Hadoop para o caso de busca por similaridade não se justifica quando se tem uma base de dados pequena. O algoritmo centralizado leva vantagem de desempenho em relação às abordagens distribuídas até o caso de replicação em 20 vezes, obtendo um tempo de resposta mais longo quando replica-se em 50 e 100 vezes a base.

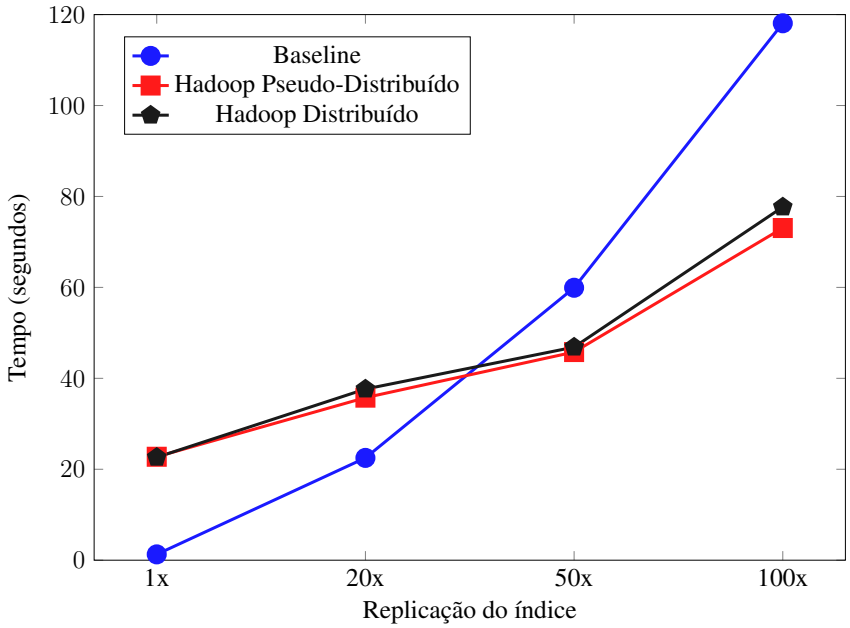

Figura 5.1 – Re1ação entre o desempenho dos mapeamentos e o tamanho do índice utilizado

A partir disso, a escalabilidade do número de termos foi posta em comparação, de acordo com a Figura 5.2. Neste cenário, a consulta variou a quantidade de palavras chave em 1, 2, 5 e 10 valores. O nível de *threshold* usado foi de 50% e a base permaneceu aumentada em 50 vezes. Os resultados mostram que o uso do Hadoop indica uma alta escalabilidade quando varia-se os termos da consulta, acontecimento que não se apresenta no *Baseline*, em que as diferenças entre os tempos de execução são maiores.

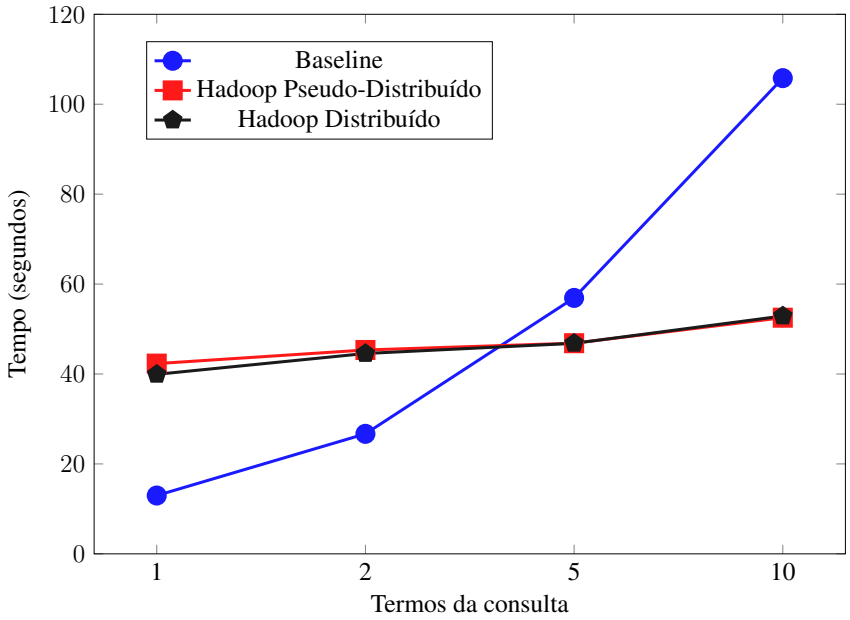

Figura 5.2 – Desempenho com variação do número de termos da consulta

Para comparar o impacto dos diferentes níveis de *threshold* usados, o gráfico da Figura 5.3 mostra os testes feitos para o ambiente centralizado e os algoritmos do Hadoop em uma consulta feita com 5 palavras chave através do índice aumentado 50 vezes. O *threshold* varia entre 0%, 25%, 50% e 90% e, conforme é aumentado, restringe o número de objetos utilizados para os cálculos de escore. Os resultados mostram que a aplicação de um limite, em todos os casos, possibilita um ganho real de tempo de execução. O *Baseline*, inclusive, gera um erro de execução ao executar com o *threshold* zerado, causado por um estouro de pilha. Mas, ao aplicar-se um limite de de 25%, o erro não acontece e a busca retorna um resultado final.

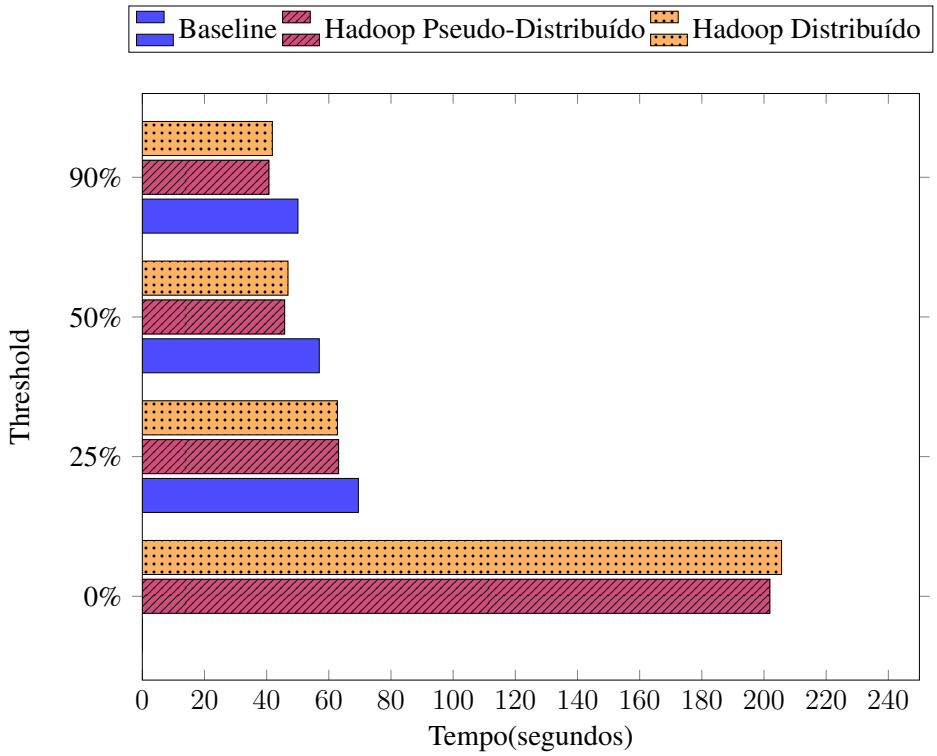

Figura 5.3 – Desempenho dos cenários com 5 termos variando o nível de *threshold*

Os resultados alcançados pelas abordagens Distribuída e Pseudo-Distribuída do Hadoop permaneceram próximos em todos os testes feitos, com uma pequena vantagem para o modo sem distribuição completa. Uma vez que o Hadoop estava rodando com a configuração padrão, apenas o caso de replicação em 100x dividiu o índice em 2 *splits*, enquanto os outros casos não ofereciam divisão alguma. Por isso, foram definidos os testes com diferentes números de *splits* a fim de verificar o desempenho do Hadoop com mais divisões.

Para controlar o número de *splits* para cada caso, o arquivo de configuração do Hadoop $mapred-site.xml$ foi modificado para incluir a propriedade que define o tamanho máximo de uma divisão. A Tabela 5.2 mostra o tamanho escolhido para criar 2 e 3 *splits* com cada tamanho

de índice. A escolha dessas quantidades se dá pelo fato de se poder testar uma divisão com distribuição mas com menor quantidade de *splits* em relação ao número de nós trabalhadores, além de uma opção com o mesmo número de *splits* e *slaves*.

Replicação do índice	2 splits	3 splits
x1	0,95 MB	0,64 MB
x10	12,8 MB	8,5 MB
x20	25,5 MB	16,98 MB
x50	64 MB	42,6 MB
x100	127,8 MB	86,1 MB

Tabela 5.2 – Tamanho máximo de um *split* para a geração de 2 e 3 divisões nos diferentes índices usados

A partir das definições para cada split, foram submetidos os testes com 2 e 3 splits com o Hadoop distribuído. Para comparação, o gráfico da Figura 5.4 mostra os resultados obtidos com o índice replicado em 50x, usando um nível de similaridade de 25% a fim de verificar as mudanças no tempo de execução sem o gargalo do threshold nulo.

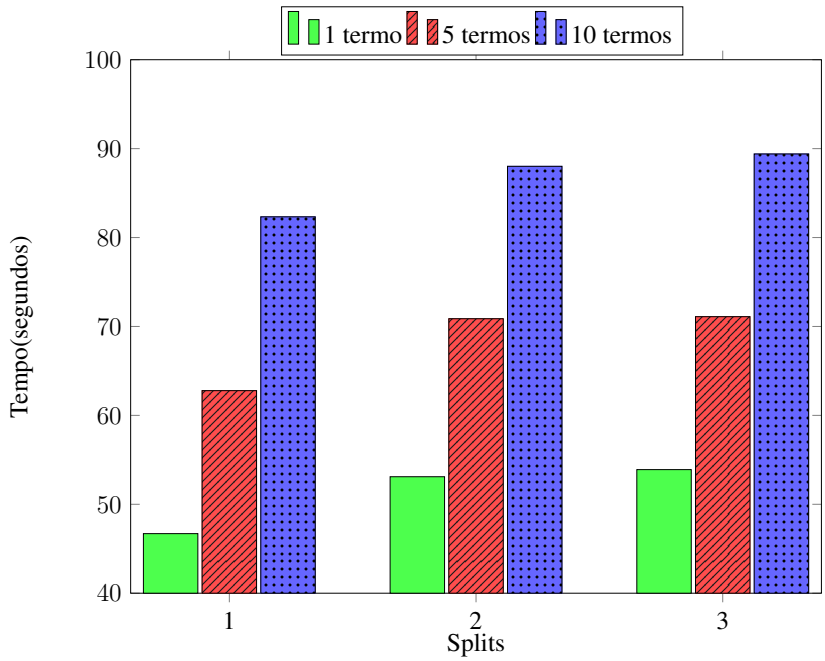

Figura 5.4 – Desempenho dos cenários com diferentes *splits*

Pode-se notar que o desempenho da aplicação piora ao aumentarmos o número de *splits*. Para analisar essa queda de eficiência, a Figura 5.5 mostra o uso do processador para o processo de busca com 2 *splits*. Pode-se notar que a CPU total do *cluster* é utilizada, na sua maior parte do tempo, por apenas uma máquina em atividade intensa. Ou seja, quando um nó está no seu pico de atividade, os outros nós tendem a diminuir a intensidade.

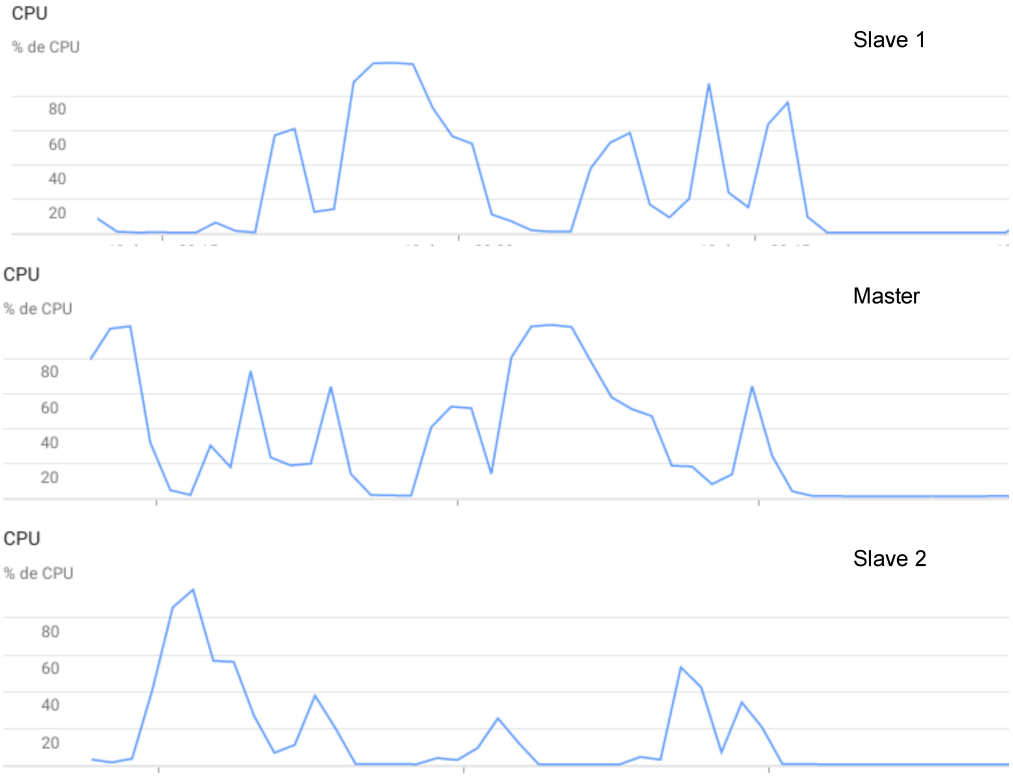

Figura 5.5 – Uso da CPU nos nós do cluster com processamento do Hadoop com 2 *splits*

Um os possíveis motivos que justifica essa relação é a forma como o escalonador estaria distribuindo as tarefas através do *cluster*, não utilizando todo o poder computacional do o ambiente para a execução da aplicação de busca.

Visto que o escalonador padrão da versão do Hadoop utilizada é o *CapacityScheduler*, foi configurado o uso do escalonador *FairScheduler*, que divide o *cluster* de forma igualitária para os *jobs*. Os testes com o escalonador modificado foram feitos com 2 e 3 *splits*, também utilizando o índice com replicação em 50 vezes e um nível de *threshold* em 50%. A Figura 5.6 mostra o ganho de desempenho alcançado com a definição do *FairScheduler* como escalonador para as tarefas de busca. A comparação se dá com o *Baseline* e o Hadoop com o escalonador *CapacityScheduler* em 2 e 3 *splits*.

Em todos os casos o ganho de desempenho ao utilizar o escalonador *FairScheduler* é positivo, chegando a ter um nível de 29% de melhora em comparação ao algoritmo *Baseline* no mesmo cenário. Além disso, em relação ao Hadoop com o seu escalonador padrão, os níveis de melhora chegam a valores ainda maiores, mostrando que o poder computacional do cluster não havia sido totalmente usado nos testes anteriores. Com o escalonamento justo, a distribuição ocorreu através dos nós trabalhadores e o tempo de execução foi menor na maioria dos casos.

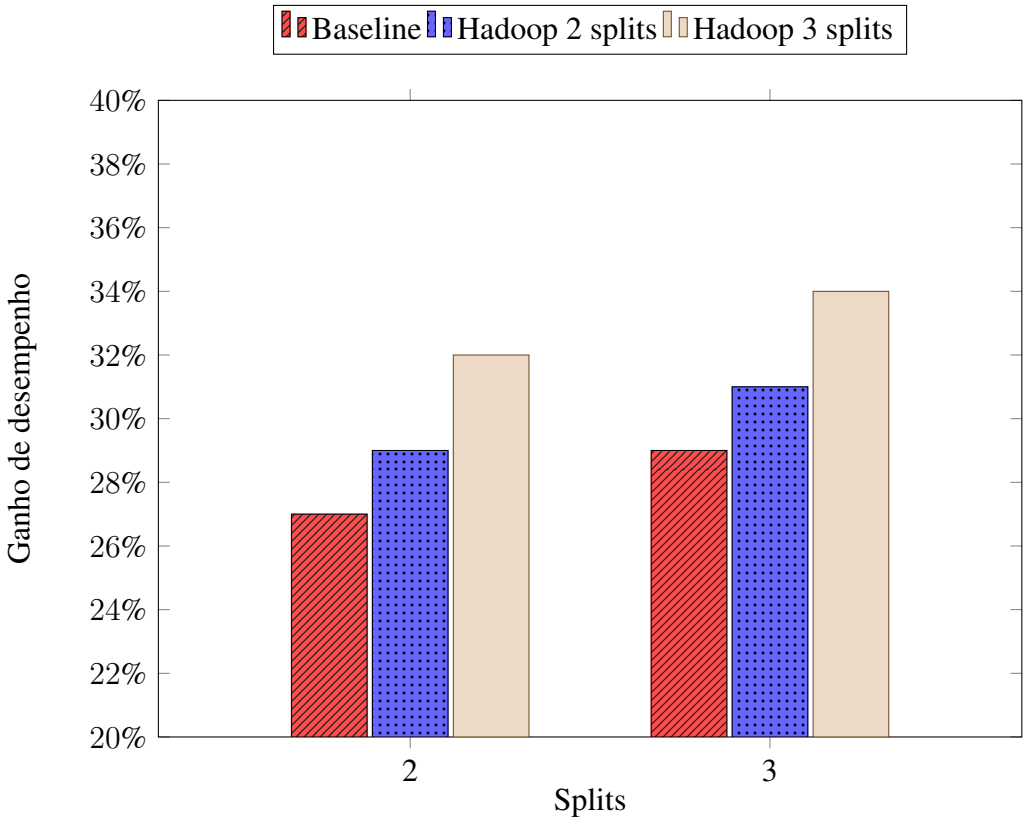

Figura 5.6 – Ganho de desempenho do problema de busca utilizando o escalonador *FairScheduler* em relação ao *Baseline* e à configuração padrão do Hadoop

A aplicação do *Baseline* para os casos em que a base é pequena ainda se mostra mais eficiente do que a distribuição com o MapReduce, porém também sofre com a queda de desempenho conforme são utilizadas bases mais volumosas, mostrando-se pouco escalável nesse sentido.

5.2 Resultados com o Apache Spark

Para realizar os testes com a ferramenta Apache Spark (v2.2.2), foram criados cenários semelhantes ao apresentado na seção anterior: a abordagem centralizada como base para comparações, além de uma versão Pseudo-Distribuída e outra Distribuída do Spark, através de um plano de teste com 20 amostras e o resultado final representado pela média dos tempos de execução. Nos testes iniciais, foram mantidas as configurações padrão da ferramenta.

Ao iniciar os testes, constatou-se que, das 20 amostras de cada cenário, a primeira execução leva um tempo superior ao restante da amostragem. Esse comportamento se dá pelo fato do Apache Spark ter uma abordagem *lazy* de processamento, em que as transformações são computadas somente quando a aplicação requisita uma resposta. Assim, na primeiro amostra o

Spark processa o índice invertido, guardando-o em memória principal e consumindo um tempo extra neste processo, diferente do restante das execuções em que o *dataset* já está disponível na memória. Para uma melhor apresentação dos testes, a média dos resultados alcançados não leva em consideração a primeira execução.

O primeiro cenário de comparação do Apache Spark avaliou o desempenho da aplicação diante das diferentes bases de dados criadas, a fim de analisar-se a escalabilidade da ferramenta. Neste caso, foi utilizado uma busca por 5 termos com de similaridade em 25%. Conforme o gráfico da Figura 5.7 exibe, o aumento da quantidade de informação que o Spark trabalha acaba influenciando de forma impactante no tempo de resposta.

Nos casos que a base é aumentada em mais de 20 vezes, o desempenho da aplicação é pior em relação ao *Baseline* e ao Hadoop Distribuído com suas configurações padrão. Por outro lado, com a base pequena o Spark acaba obtendo resultados mais eficientes que os cenários testados com o Hadoop, permanecendo próximos, inclusive, do *Baseline*.

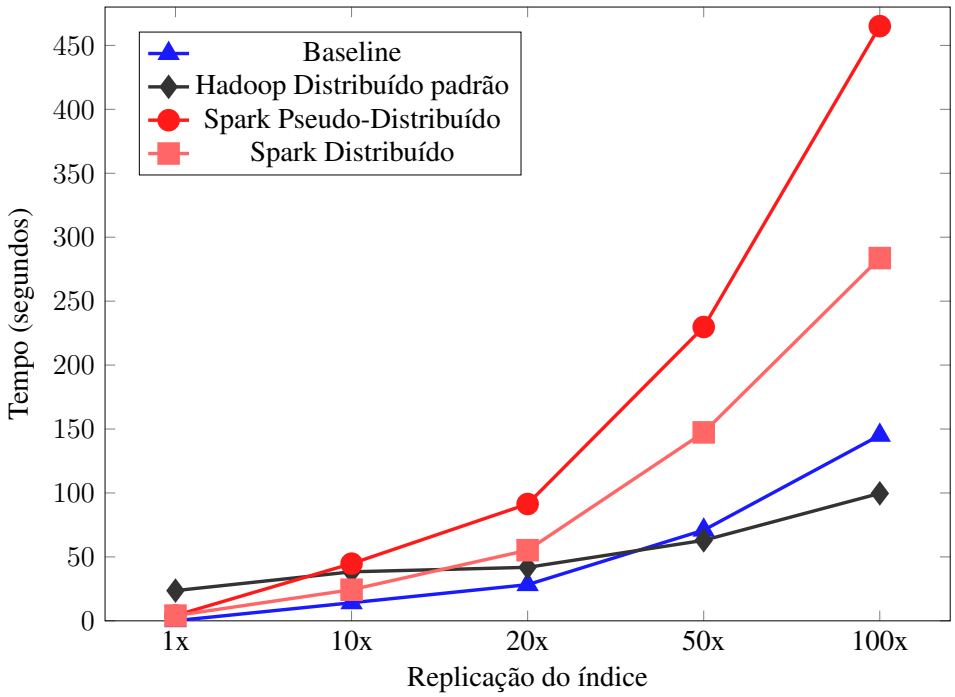

Figura 5.7 – Desempenho do Spark aumentando-se o tamanho do índice utilizado

A partir disso, analisou-se a escalabilidade do Spark em relação à quantidade de termos utilizados na consulta. No gráfico da Figura 5.8, os cenários foram variados entre 1, 2, 5 e 10 palavras chave usando uma base aumentada em 50 vezes e *threshold* de 50%. Assim como o teste anterior, o Apache Spark se mostrou menos eficiente à medida que a quantidade de palavras aumenta.

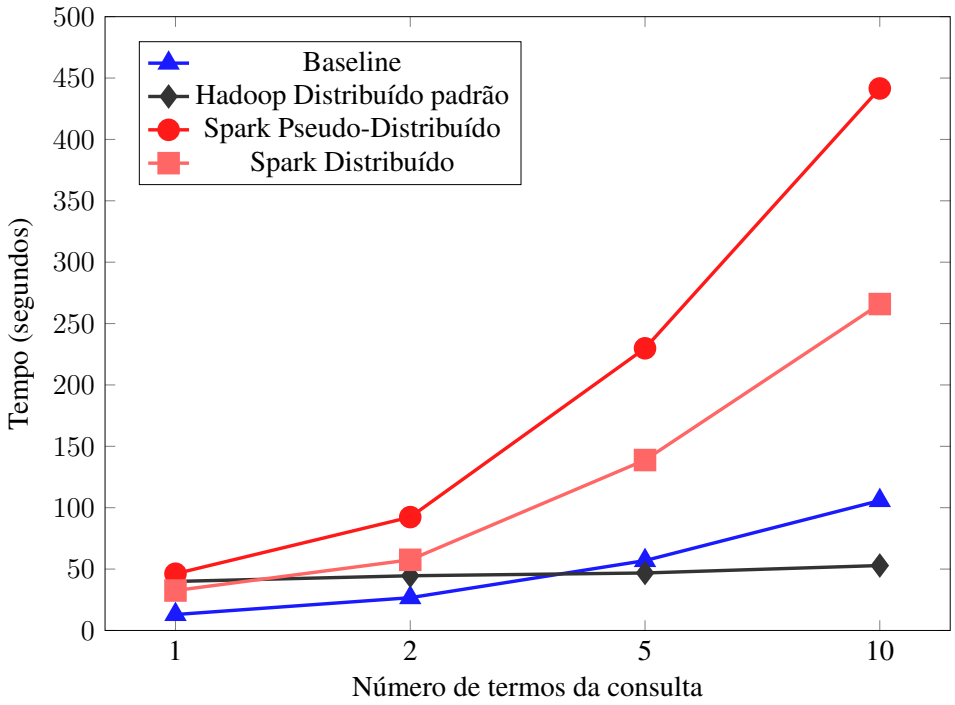

Figura 5.8 – Variação do número de termos para a busca no Spark

Como os testes realizados apresentaram uma baixa escalabilidade, foi verificado a quantidade de recursos que o *cluster* poderia estar utilizando para executar as tarefas. O escalonador *Standalone* do Spark inicia um *executor* para cada núcleo do processador, sendo que cada *executor* processa uma divisão das transformações, levando à conclusão de que o *dataset* trabalhado não estava sendo bem dividido de maneira a utilizar completamente o *cluster*.

A fim de proporcionar uma melhor utilização do ambiente distribuindo, foram submetidos novos testes da aplicação de busca com o Spark, em que o número de divisões da base de dados foi alterado manualmente para exigir que mais *executors* pudessem trabalhar em conjunto. Ao total, criou-se cenários com 3, 6 e 12 divisões no arquivo de entrada e novos testes foram submetidos. A Figura 5.9 exibe o tempo de execução dos testes com o Spark em uma busca por 5 termos, 50% de *threshold* e bases aumentadas em 20 e 50 vezes, levando em consideração os novos cenários e os testes realizados anteriormente.

É notável o ganho de desempenho no tempo de execução ao aumentar-se o número de divisões, com exceção do caso de 3 *executors*. A partir de 6 divisões, ao menos 2 *executors* submetidos em cada nó, sendo um para cada núcleo do processador, possibilitando a participação de todo o *cluster* na execução da aplicação.

Por fim, foi comparado o impacto do limite de *threshold* no desempenho das aplicações de busca no Spark. A figura 5.10 mostra o tempo de execução de consultas com base aumentada

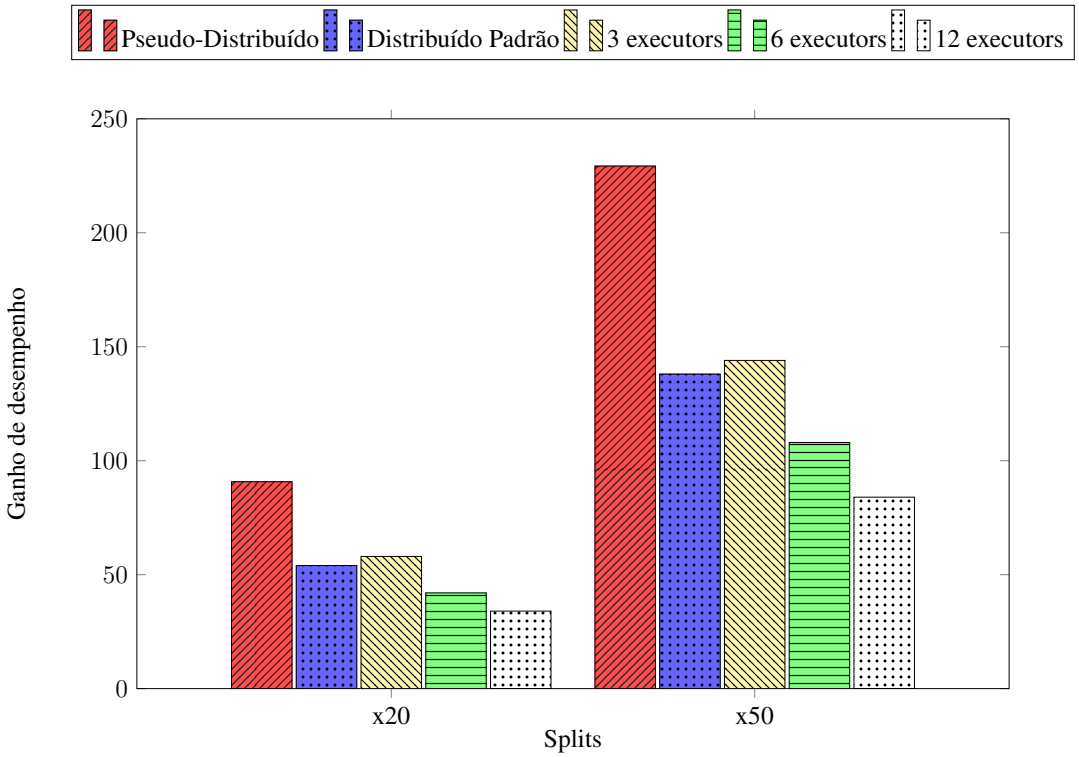

Figura 5.9 – Desempenho do Spark variando-se o número de *executors*

em 50 vezes e 5 termos, variando o limiar de similaridade em 0%, 25%, 50% e 80%. Os resultados apresentados mostram que a aplicação do *threshold* não ofereceu um impacto direto no desempenho da busca no Apache Spark, mostrando que o custo do emprego de um limite pode não ser uma boa alternativa.

5.3 Discussão

Após a finalização de todos os testes, pode-se analisar um panorama geral do funcionamento das diferentes ferramentas de processamento distribuído, através dos diversos cenários definidos.

O Apache Hadoop alcançou um desempenho inferior ao *Baseline* em casos onde a base de dados não possuía uma quantidade expressiva de entradas. Porém, mostrando a grande escalabilidade do *framework* de distribuição, os testes com bases maiores sobressaíram-se em relação ao cenário centralizado.

Ainda assim, o desempenho do Hadoop sofreu gargalos referente à distribuição das tarefas, visto que os métodos Pseudo-Distribuído e Distribuído executaram, de forma geral, na mesma faixa de tempo. O ganho de desempenho desejado foi alcançado ao mudar-se o escalo-

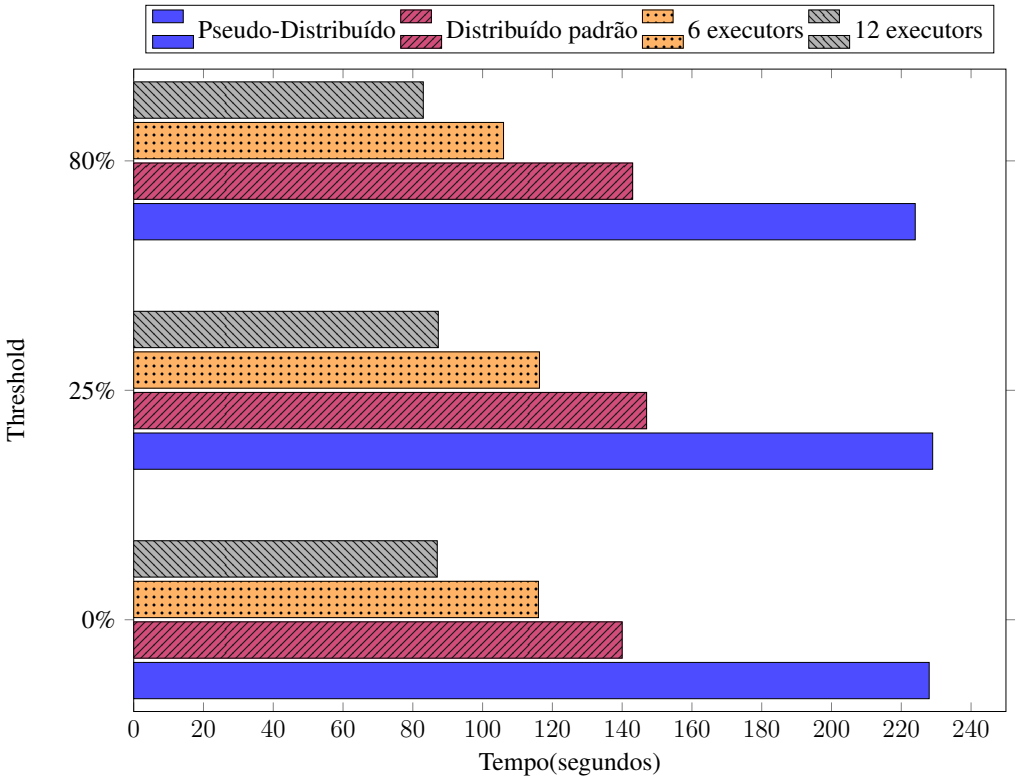

Figura 5.10 – Impacto do *threshold* na busca com Spark

nador padrão pelo escalonador *FairScheduler*. Este fato se dá pela diferença na característica de delegação de tarefas dos escalonadores, uma vez que a divisão do *cluster* para cada *Job* é feita de forma justa no *FairScheduler*. Neste caso, ao definir-se um número de divisões igual ao número de nós trabalhadores, foi identificado o melhor cenário testado neste trabalho usando o Hadoop.

Outro interessante comportamento apresentado no Apache Hadoop, assim como no *Baseline*, foi o impacto, no tempo de resposta da aplicação, de um limiar de corte de similaridade (*threshold*). A diferença de desempenho quando define-se o *threshold* com os valores 0% e 25% é significativamente grande para todos os cenários testados. No Hadoop, esse comportamento é justificado pela limitação do tamanho de arquivos intermediários usado pelo *framework*, reduzindo os custos de leitura e escrita no disco. Já no *Baseline*, a justificativa é semelhante, já que o *threshold* diminui o número de entradas em estruturas de dados auxiliares.

Já o uso da ferramenta Apache Spark apresentou um desempenho satisfatório quando se trabalhou com bases de dados pequenas, visto que, nesses cenários com poucos dados, o tempo de busca aproximou-se do *Baseline*. A partir do armazenamento dos *datasets* distribuídos (RDD's) em memória, aplicações que façam um reuso de dados já processados são beneficiadas por essa característica, tendo um acesso mais rápido em relação às outras abordagens.

Por outro lado, o Spark não teve um comportamento aceitável no quesito escalabilidade, apresentando um desempenho cada vez pior à medida em que foi aumentado o tamanho da base de dados ou o número de termos da consulta. Como esse *framework* faz uso da memória principal, esperava-se que o tempo de execução da aplicação de busca alcançasse valores mais adequados em relação ao Apache Hadoop, o que não se confirmou nos testes. Sendo assim, uma investigação mais profunda de técnicas de otimização do Apache Spark podem ser relevantes para que a característica de armazenamento em memória traga reais benefícios para as aplicações desenvolvidas.

Após ser verificado que o *cluster* não estava sendo usado em sua totalidade, sugeriu-se a divisão manual da base de dados a fim de gerar uma maior distribuição no Spark. De fato, com maiores divisões (e mais *executors* trabalhando), chegou-se a um ganho de desempenho significativo. Ainda assim, os valores mantiveram-se acima das outras abordagens de processamento nos casos com bases grandes e muitas palavras chave na consulta.

Outro fator importante nos testes do Spark é o impacto irrelevante do *threshold* nos tempos de busca. Ao contrário do Apache Hadoop, que aplica o *threshold* e corta valores tão logo o valor de similaridade é calculado, no Spark é gerado um RDD com todos os valores calculados e, através de uma operação de filtro, outro RDD é criado com o *threshold* aplicado. Logo, de modo geral, a operação de filtro não se mostra tão eficiente neste contexto.

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma perspectiva sobre um problema comum na área de Recuperação de Informação, através de problemas de busca por similaridade com o auxílio de uma estrutura de índice invertido. Foi possível verificar o comportamento dessa aplicação implementada em ferramentas de distribuição de processamento, em comparação com uma abordagem centralizada.

A visão que permanece diante dos resultados alcançados mostra que o uso das ferramentas de processamento distribuído, Apache Hadoop e Apache Spark, não se mostraram eficientes para o tipo de requisição com que se trabalhou. Buscas por similaridade configuram um problema de tempo real (*online*) e, por isso, prezam por respostas com tempos criticamente curtos.

A ferramenta Apache Spark se mostrou eficiente para um problema *online* envolvendo uma base pequena, mas sofreu com a falta de escalabilidade. Já o Apache Hadoop se mostrou altamente escalável, mas a limitação do desenvolvimento em relação à abrangência de métodos disponíveis, além do grande uso do disco para as operações e armazenamento intermediário, faz com que o *framework* não seja aplicável em situações de tempo real.

Portanto, é importante realizar estudos de ferramentas voltadas para problemas que envolvem grandes quantidades dados em requisições do tipo *offline*, que são voltadas para o tipo de ferramenta utilizada neste trabalho. Nessas requisições, o tempo de execução é importante mas deixa de ser um aspecto crítico para a operação. Desta forma, o tempo que as ferramentas utilizam para configurar o processamento, ou mesmo o custo de operações em disco, podem ser fatores irrelevantes para o desempenho final.

Existem diversos problemas *offline* que beneficiam-se do uso de um processamento distribuído e podem ser encarados como trabalhos futuros. Um exemplo que pode ser investigado é a análise e a comparação de *datasets* mais robustos, como uma busca por similaridade envolvendo imagens. Neste caso, conforme a qualidade das imagens é mais refinada, o custo de processamento centralizado pode ser um impasse a ser resolvido por algoritmos distribuídos, inclusive podendo-se utilizar a ideia do MapReduce e dos mapeamentos vistos neste trabalho.

REFERÊNCIAS

ARMBRUST, M. et al. Above the clouds: a berkeley view of cloud computing. , [S.l.], 2009.

BAEZA-YATES, R.; RIBEIRO-NETO, B. **Recuperação de Informação-**: conceitos e tecnologia das máquinas de busca. [S.l.: s.n.], 2013. 6-7p.

BECKER, D. J. et al. BEOWULF: a parallel workstation for scientific computation. **Proceedings, International Conference on Parallel Processing**, [S.l.], v.95, 1995.

CARDOSO, P. et al. Ambiente Colaborativo para Identificação de Espécies. **Anais do XIII Simpósio de Informática (SIRC)**, [S.l.], p.68–73, 2015.

CARDOSO, P. V.; FRANZIN, F. F.; MERGEN, S. L. Using Active Mediators and Passive Extractors Inside Materialized Data Integration Systems. **Congresso da Saciedade Brasileira de Computação - CTIC**, [S.l.], v.35, p.501–510, 2016.

CHOO, C. W.; DETLOR, B.; TURNBULL, D. **Web work**: information seeking and knowledge work on the world wide web. [S.l.]: Springer Science & Business Media, 2013. v.1.

CHOWDHURY, G. **Introduction to modern information retrieval**. [S.l.]: Facet publishing, 2010. 14–18p.

COULOURIS, G. et al. **Sistemas Distribuídos-**: conceitos e projeto. [S.l.]: Bookman Editora, 2013.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Communications of the ACM**, [S.l.], v.51, n.1, p.107–113, 2008.

FERNANDEZ, A. et al. An overview of e-learning in cloud computing. **Workshop on Learning Technology for Education in Cloud (LTEC'12)**, [S.l.], p.35–46, 2012.

GANTZ, J.; REINSEL, D. The digital universe in 2020: big data, bigger digital shadows, and biggest growth in the far east. **IDC iView: IDC Analyze the future**, [S.l.], v.2007, p.1–16, 2012.

KARAU, H. et al. **Learning spark**: lightning-fast big data analysis. [S.l.]: "O'Reilly Media, Inc.", 2015.

KORFHAGE, R. R. **Information storage and retrieval**. [S.l.: s.n.], 2008. 1–16p. v.2008.

MANYIKA, J. et al. Big data: the next frontier for innovation, competition, and productivity. **IDC iView: IDC Analyze the future**, [S.l.], v.2007, p.1–16, 2011.

PENCHIKALA, S. **Big Data com Apache Spark**. 2015.

PITANGA, M. **Construindo supercomputadores com Linux**. [S.l.]: Brasport, 2004.

RISTAD, E. S.; YIANILOS, P. N. Learning string-edit distance. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, [S.l.], v.20, n.5, p.522–532, 1998.

SAGIROGLU, S.; SINANC, D. Big data: a review. **Collaboration Technologies and Systems (CTS), 2013 International Conference on**, [S.l.], p.42–47, 2013.

TANENBAUM, A. S.; VAN STEEN, M. **Distributed systems**. [S.l.]: Prentice-Hall, 2007.

TAURION, C. **Big Data**. [S.l.]: Brasport, 2013.

WHITE, T. **Hadoop**: the definitive guide. [S.l.]: "O'Reilly Media, Inc.", 2012. 202-203p.

ZAHARIA, M. et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. **Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation**, [S.l.], p.2–2, 2012.

APÊNDICES

APÊNDICE A – Resultados detalhados dos experimentos para o problema de busca

Neste Apêndice são mostrados os resultados do desempenho dos testes feitos no Capítulo 5 para o problema de busca por similaridade em índices invertidos. Os resultados são exibidos por meio de tabelas, as quais informam o tempo de execução em relação aos cenários criados de threshold, número de termos da busca e replicação do índice.

Indice	Termos	Tempo (segundos)				
		th: 0%	th: 25%	th: 50%	th: 80%	th: 90%
1x	1	0,03	0,03	0,03	0,04	0,04
	2	0,04	0,04	0,04	0,04	0,05
	5	0,06	0,11	0,07	0,07	0,06
	10	1,09	0,08	0,08	0,09	0,08
10x	1	3,48	3,05	2,90	3,10	2,59
	2	9,45	6,12	5,24	5,18	5,19
	5	23,83	14,29	12,48	12,52	11,60
	10	53,39	29,99	22,62	21,78	21,22
20x	1	7,76	5,73	5,06	5,29	5,05
	2	17,20	11,67	10,27	10,29	9,73
	5	53,86	28,09	24,05	23,82	22,64
	10	120,94	57,30	45,20	43,93	43,54
50x	1	24,43	13,86	12,15	12,85	11,93
	2	57,36	27,68	24,80	23,77	22,36
	5	$erro$	70,50	57,88	58,53	54,81
	10	$erro$	145,06	114,80	108,96	105,70
100x	1	$erro$	26,79	24,22	24,06	23,00
	2	$erro$	59,26	50,52	50,01	43,55
	5	$erro$	146,21	119,26	116,27	108,02
	10	$erro$	301,61	219,98	209,51	204,07

Tabela A.1 – Tempo de execução do baseline

Indice	Termos	Tempo (segundos)				
		th: 0%	th: 25%	th: 50%	th: 80%	th: 90%
1x	1	30,03	22,73	21,70	21,80	22,62
	2	25,76	23,68	23,55	22,67	22,68
	5	26,59	24,23	23,31	22,58	23,71
	10	31,84	25,10	23,18	23,13	21,65
10x	1	38,24	34,73	29,22	29,75	28,23
	2	51,00	36,28	31,82	32,73	28,21
	5	74,06	39,23	33,73	34,73	28,72
	10	134,93	45,90	31,79	32,84	30,26
20x	1	44,95	35,33	33,83	33,19	32,26
	2	79,59	39,31	35,31	34,22	32,30
	5	117,35	43,40	35,75	36,85	33,76
	10	159,14	52,41	37,29	37,48	35,80
50x	1	71,53	46,49	42,31	41,85	40,33
	2	100,81	51,96	45,34	45,31	39,82
	5	201,89	64,10	46,85	45,49	41,30
	10	509,18	82,84	52,52	47,84	43,82
100x	1	130,65	63,52	59,50	61,51	62,06
	2	169,65	82,69	68,64	74,15	63,97
	5	518,87	98,42	73,03	77,56	70,58
	10	733,66	136,22	93,23	69,06	71,53

Tabela A.2 – Tempo de execução do Hadoop Pseudo-Distribuído

Indice	Termos	Tempo (segundos)				
		th: 0%	th: 25%	th: 50%	th: 80%	th: 90%
1x	1	21,99	22,69	21,90	21,69	21,68
	2	24,68	23,46	22,50	21,52	21,57
	5	27,04	23,64	23,20	21,52	21,86
	10	26,19	24,06	22,13	20,61	21,58
10x	1	39,97	33,68	29,68	28,36	25,63
	2	49,71	33,67	30,96	29,98	28,13
	5	85,30	38,64	32,35	33,50	29,98
	10	117,15	46,10	34,08	31,15	28,64
20x	1	45,20	34,57	32,94	31,73	30,19
	2	75,85	37,86	34,48	33,13	31,01
	5	106,98	41,11	37,57	36,64	32,63
	10	153,79	48,66	36,67	36,63	36,40
50x	1	73,25	46,55	39,93	39,66	40,59
	2	101,24	53,99	44,56	44,10	38,78
	5	205,66	62,83	46,82	46,16	41,13
	10	644,01	83,23	52,90	46,60	43,78
100x	1	113,03	65,42	57,41	57,88	59,26
	2	176,40	91,04	71,24	61,85	54,90
	5	372,18	99,26	79,42	66,91	70,92
	10	702,36	141,41	97,08	68,48	78,03

Tabela A.3 – Tempo de execução do Hadoop Distribuído com configurações padrão

Indice	Termos	Tempo (segundos)				
		th: 0%	th: 25%	th: 50%	th: 80%	th: 90%
1x	1	32,84	27,62	22,63	24,69	25,57
	2	29,97	27,71	24,73	24,90	23,72
	5	33,61	26,60	26,78	23,75	22,75
	10	33,88	27,76	23,69	24,71	24,50
10x	1	67,90	37,02	33,92	31,90	29,69
	2	69,93	43,88	33,87	32,79	29,67
	5	78,96	47,77	37,82	35,93	32,65
	10	91,20	59,04	40,85	34,95	34,85
20x	1	77,88	40,72	37,76	39,61	38,98
	2	73,15	47,03	40,73	41,90	34,77
	5	149,25	48,95	47,06	43,85	38,74
	10	167,52	60,87	46,74	44,84	38,76
50x	1	103,15	53,20	48,86	46,10	42,76
	2	105,43	52,56	49,12	49,78	43,27
	5	183,81	70,85	54,63	50,98	47,66
	10	489,89	88,02	55,95	53,09	52,38
100x	1	168,56	67,10	56,99	55,81	49,87
	2	168,69	88,14	69,87	65,04	51,92
	5	383,71	92,45	71,85	77,36	56,93
	10	718,01	138,77	93,08	65,91	66,08

Tabela A.4 – Tempo de execução do Hadoop Distribuído com Capacity Scheduler e 2 splits

Indice	Termos	Tempo (segundos)				
		th: 0%	th: 25%	th: 50%	th: 80%	th: 90%
1x	1	38,27	29,82	29,65	29,75	29,68
	2	30,75	29,76	29,65	26,69	27,46
	5	32,58	28,70	27,70	28,75	28,85
	10	37,60	34,70	27,89	26,75	27,64
10x	1	41,90	37,81	35,99	34,68	33,87
	2	54,94	44,75	35,82	36,58	34,66
	5	63,88	44,66	40,57	42,60	35,68
	10	110,97	51,86	42,85	37,42	38,87
20x	1	76,88	39,93	39,63	40,92	35,75
	2	64,63	51,57	38,59	41,04	37,09
	5	149,33	57,68	46,86	47,52	42,88
	10	152,76	54,60	49,48	45,94	42,89
50x	1	93,78	52,46	48,32	46,72	45,86
	2	92,56	53,11	47,94	46,51	46,06
	5	181,37	71,35	59,55	58,98	46,93
	10	410,85	89,39	62,07	61,54	52,16
100x	1	210,30	63,02	71,14	62,87	54,89
	2	172,73	91,25	81,13	67,25	70,11
	5	359,77	98,15	72,99	87,38	81,20
	10	654,97	143,24	100,29	73,11	81,01

Tabela A.5 – Tempo de execução do Hadoop Distribuído com Capacity Scheduler e 3 splits

Indice	Termos	Tempo (segundos)				
		th: 0%	th: 25%	th: 50%	th: 80%	th: 90%
1x	1	22,14	22,10	21,15	21,35	21,55
	2	25,53	22,81	23,42	22,48	22,34
	5	25,09	24,42	23,03	21,40	22,97
	10	25,31	25,88	22,43	23,07	22,95
10x	1	39,88	31,53	27,48	28,73	24,59
	2	39,22	34,07	27,93	26,75	22,51
	5	52,11	35,79	28,38	27,89	23,44
	10	65,46	43,60	33,01	28,62	27,79
20x	1	43,77	32,14	28,70	33,71	25,63
	2	60,29	36,99	30,72	33,94	27,24
	5	101,24	42,01	33,43	40,16	27,35
	10	103,87	53,47	36,88	34,22	27,01
50x	1	71,68	39,77	41,09	34,20	35,43
	2	86,62	44,96	37,05	38,78	36,83
	5	135,69	49,84	38,92	39,55	34,39
	10	293,49	112,92	42,66	40,46	39,03
100x	1	94,40	47,76	52,57	48,21	47,01
	2	177,98	61,12	46,24	45,22	44,45
	5	380,57	71,13	53,08	47,82	41,96
	10	485,71	97,11	59,05	54,78	56,55

Tabela A.6 – Tempo de execução do Hadoop Distribuído com Fair Scheduler e 2 splits

Indice	Termos	Tempo (segundos)				
		th: 0%	th: 25%	th: 50%	th: 80%	th: 90%
1x	1	24,89	23,29	22,43	22,75	22,27
	2	24,69	22,02	21,61	22,72	22,29
	5	28,05	21,50	25,21	22,03	22,38
	10	27,26	23,16	23,42	21,67	21,68
10x	1	32,59	26,82	23,14	24,32	23,78
	2	39,01	30,41	27,66	28,72	24,04
	5	45,12	33,17	25,98	27,25	21,77
	10	60,48	40,67	24,86	26,62	24,75
20x	1	41,19	29,30	27,84	29,71	26,89
	2	46,84	35,91	29,72	28,83	23,60
	5	88,89	36,58	33,69	31,46	26,71
	10	125,25	42,67	33,48	27,28	26,94
50x	1	50,30	38,90	35,97	33,60	32,33
	2	78,92	40,63	35,29	36,17	34,66
	5	172,21	48,65	40,42	37,57	31,01
	10	231,75	71,16	40,41	36,51	36,38
100x	1	97,20	40,70	39,08	39,56	37,11
	2	127,91	53,45	42,93	46,46	36,77
	5	187,40	60,26	50,09	52,11	36,25
	10	362,12	109,30	58,27	43,21	40,39

Tabela A.7 – Tempo de execução do Hadoop Distribuído com Fair Scheduler e 3 splits

Indice	Termos	Tempo (segundos)			
		th: 0%	th: 25%	th: 50%	th: 80%
1x	1	1,31	1,17	1,08	1,09
	2	1,99	1,95	1,86	1,80
	5	4,14	4,21	4,20	4,09
	10	7,77	7,75	7,71	7,74
10x	1	11,54	10,04	9,06	9,13
	2	19,32	18,68	17,83	17,72
	5	43,95	44,41	43,55	42,93
	10	83,53	83,86	84,02	83,74
20x	1	23,45	20,26	18,48	18,51
	2	39,85	38,30	36,98	36,40
	5	91,74	91,65	90,77	87,99
	10	174,24	173,93	172,71	171,40
50x	1	58,55	50,65	46,28	46,25
	2	99,23	95,43	92,42	91,12
	5	228,63	229,02	229,27	224,44
	10	441,23	441,16	441,88	440,84
100x	1	118,97	102,17	93,66	93,35
	2	199,71	190,79	186,90	186,06
	5	463,42	465,28	456,01	451,16
	10	894,82	894,76	889,49	883,69

Tabela A.8 – Tempo de execução do Apache Spark Pseudo-Distribuído

Indice	Termos	Tempo (segundos)			
		th: 0%	th: 25%	th: 50%	th: 80%
1x	1	0,75	0,69	0,59	0,95
	2	1,87	1,78	1,67	1,73
	5	3,90	4,10	3,90	4,06
	10	7,40	7,59	7,46	7,54
10x	1	6,28	5,07	4,78	4,83
	2	10,46	10,33	10,29	10,00
	5	24,21	24,16	25,21	24,45
	10	48,67	45,89	45,88	45,86
20x	1	14,09	12,70	11,71	10,60
	2	23,57	22,27	22,48	22,70
	5	54,89	55,50	54,09	54,87
	10	110,17	106,78	108,02	106,87
50x	1	40,81	33,42	32,59	30,85
	2	64,67	63,31	57,06	58,05
	5	140,45	147,95	138,46	143,04
	10	268,67	267,00	265,40	261,25
100x	1	70,21	60,07	57,49	56,04
	2	120,25	120,02	119,41	193,03
	5	278,13	283,93	482,41	268,16
	10	513,58	517,47	496,04	493,75

Tabela A.9 – Tempo de execução do Apache Spark com configurações padrão

Indice	Termos	Tempo (segundos)			
		th: 0%	th: 25%	th: 50%	th: 80%
1x	1	0,91	0,77	0,75	0,69
	2	1,26	1,30	1,22	1,20
	5	2,71	2,68	2,68	2,87
	10	5,35	5,21	5,01	5,12
10x	1	7,37	6,38	5,59	5,49
	2	12,56	11,61	11,57	10,80
	5	27,82	27,66	27,18	27,08
	10	56,35	55,03	54,95	53,12
20x	1	15,83	12,19	11,29	11,44
	2	24,75	14,69	25,71	22,65
	5	58,43	58,56	58,36	57,23
	10	115,85	109,64	114,92	114,41
50x	1	35,76	30,16	28,14	28,19
	2	61,59	64,05	57,94	58,37
	5	149,48	148,40	144,20	148,62
	10	273,12	275,51	275,44	272,19
100x	1	74,12	70,03	69,56	69,04
	2	128,16	128,13	127,40	124,07
	5	309,45	302,77	312,40	308,15
	10	551,34	547,10	546,93	596,31

Tabela A.10 – Tempo de execução do Apache Spark com 3 executors

Indice	Termos	Tempo (segundos)			
		th: 0%	th: 25%	th: 50%	th: 80%
1x	1	0,97	0,72	0,76	0,61
	2	1,16	1,20	1,00	0,89
	5	1,87	1,84	1,89	1,84
	10	3,08	3,19	3,05	3,17
10x	1	5,95	4,81	4,20	4,51
	2	9,79	9,36	8,22	8,65
	5	22,61	20,91	21,99	20,95
	10	40,94	40,56	41,14	40,33
20x	1	10,91	8,89	8,42	9,04
	2	18,50	19,21	18,07	16,28
	5	43,41	43,67	42,36	41,93
	10	88,32	83,31	90,59	83,29
50x	1	26,62	24,13	21,01	21,26
	2	49,28	45,11	45,38	44,01
	5	116,63	116,35	108,62	106,71
	10	224,91	228,92	226,47	225,14
100x	1	53,62	45,40	42,72	45,06
	2	100,72	90,42	86,16	85,38
	5	227,52	236,83	225,02	228,05
	10	440,17	452,22	449,64	452,11

Tabela A.11 – Tempo de execução do Apache Spark com 6 executors

Indice	Termos	Tempo (segundos)			
		th: 0%	th: 25%	th: 50%	th: 80%
1x	1	1,22	0,96	0,92	0,75
	2	1,35	1,52	1,37	1,39
	5	2,93	2,41	2,63	2,54
	10	4,56	4,24	3,86	3,86
10x	1	6,20	4,81	4,38	4,04
	2	9,76	8,27	7,27	7,05
	5	18,17	16,24	17,82	16,96
	10	31,04	30,34	30,71	30,26
20x	1	10,53	7,71	7,84	6,69
	2	16,38	13,92	13,44	13,14
	5	36,16	32,99	34,20	30,71
	10	63,75	61,03	61,88	61,74
50x	1	28,83	18,01	16,69	16,22
	2	40,38	37,68	33,92	32,90
	5	87,89	87,90	84,32	83,43
	10	163,52	155,84	165,93	154,55
100x	1	51,26	35,12	33,14	33,72
	2	77,39	68,02	61,65	63,61
	5	170,95	172,46	168,95	163,13
	10	333,18	324,57	322,44	318,40

Tabela A.12 – Tempo de execução do Apache Spark com 12 executors