# Busca por similaridade em índice invertido utilizando o framework Hadoop Map-Reduce

**Paulo V. M. Cardoso, Sergio L. S. Mergen**

Curso de Ciência da Computação – Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS – Brazil

{pcardoso, mergen}@inf.ufsm.br

***Abstract.*** *In the Information Retrieval area, inverted indexes are commonly used in systems that do object search through keyword search. In this context, the similarity search is a powerful strategy, once it allows the retrieval of objects that would not be found if exact match search was used instead. However, the amount of objects stored, the way the similarity is computed and the amount of terms used in a query may lead to a prohibitive execution time if the processing is not distributed. Therefore, in this paper we explore the possibility of using Hadoop Map-Reduce as a way to distribute the processing involved in this type of search. Besides proposing an Map-Reduce based algorithm, we discuss keyword-based search experiments over a repository of digital objects. The results show when the distributed processing overcomes the centralized processing.*

***Resumo.*** *Na área de Recuperação de Informação, é comum o uso de índices invertidos em sistemas que realizam a busca de objetos armazenados através de consultas compostas por palavras chave (termos). Nesse contexto, a busca por similaridade representa uma estratégia poderosa, uma vez que permite a recuperação de objetos que não seriam recuperados se fosse empregada a busca por equivalência. Contudo, a quantidade de objetos armazenados, a maneira como a similaridade é calculada e o número de termos da consulta podem levar a tempos de execução proibitivos se o processamento não for distribuído. Assim, esse artigo explora a possibilidade de uso do Hadoop Map-Reduce como forma de distribuir o processamento envolvido nesse tipo de busca. Além de propor um algoritmo de mapeamento, o artigo discute experimentos de busca realizados sobre um repositório digital de objetos. Os resultados mostram os cenários de teste em que o processamento distribuído se sobressai em relação ao processamento centralizado.*

## 1. Introdução

A Recuperação de Informação é uma área de pesquisa que visa a busca de objetos armazenados a partir de consultas, onde em uma consulta são informadas palavras chave (termos) que caracterizam os objetos que se pretende acessar [Korfhage 2008]. Os objetos podem estar disponíveis 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. Os objetos podem ser representados de diversas formas, como documentos hierárquicos em formato XML, tuplas de relações, arquivos de mídia, entre outros.

A eficiência na busca de objetos depende de muitos aspectos, que compreendem desde a definição e configuração da plataforma de *hardware* que suportará a computação até o uso de estruturas de dados adequadas responsáveis pela indexação dos objetos. Uma estrutura de dados bastante usada no problema de busca é chamada de índice invertido, onde uma entrada(chave) do índice leva à lista de objetos(valores) que estão associados à entrada. Essa estrutura é particularmente atraente quando as consultas são formuladas como uma lista de palavras chave. Um exemplo de aplicação que recorre à consultas por palavra chave são os motores de busca WEB, que recuperam páginas HTML a partir de termos informados pelo usuário.

Consultas compostas por termos podem ser usadas para a realização de buscas por equivalência ou por similaridade. O segundo caso representa uma estratégia poderosa, uma vez que permite a recuperação de objetos que não seriam recuperados se fosse empregada a busca por equivalência. Por outro lado, o custo do processamento é maior. 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 de processamento distribuído e ferramentas de gestão de Big Data.

Uma das tecnologias mais utilizadas no contexto de programação distribuída é o APACHE HADOOP, proposto para atender à demandas de processamento em grandes volumes de dados. O *framework* APACHE HADOOP faz uma implementação do modelo de programação MAP-REDUCE, através das funções *map* e *reduce*.

Existe semelhança na forma como o *framework* realiza a distribuição de processamento e a estrutura de um índice invertido: na etapa de redução, o *framework* repassa aos nós de processamento os dados compostos por uma chave e uma lista de valores associados. Por sua vez, um índice invertido também tem na sua composição a figura da chave associada a uma lista de valores.

Neste contexto, o objetivo desse trabalho é analisar de que forma uma ferramenta de distribuição de processamento que utilize o paradigma MAP-REDUCE pode ser usada para distribuir a carga de processamento em buscas por similaridade sobre índices invertidos. O trabalho consiste em propor um algoritmo de mapeamento para esse problema e demonstrar através de experimentos, situações em que o uso do processamento distribuído resulta em um tempo de execução inferior na comparação com o processamento centralizado.

O trabalho está estruturado da seguinte forma: a seção 2 apresenta os conceitos e funcionamento do HADOOP MAP-REDUCE, além de sua estrutura de distribuição de carga em diferentes nós computacionais. A seção 3 apresenta os algoritmos propostos e os conceitos teóricos usados no trabalho. A seção 4 apresenta os experimentos realizados em uma aplicação que acessa objetos de um repositório digital.A seção 5 apresenta os trabalhos relacionados. A seção 6 apresenta as considerações finais.

## 2. Apache Hadoop

O APACHE 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 [White 2012]. O HADOOP é inspirado no paradigma MAP-REDUCE

[Dean and Ghemawat 2008] 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), os quais estão inclusos no topo do Hadoop Ecosystem. Abaixo desses dois elementos, diversas ferramentas e tecnologias são descritas para vários tipos de aplicações.

Uma aplicação no HADOOP MAPREDUCE é chamada de *job* e, como especifica o paradigma, tem como composição essencial 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*. As saídas dos *mappers* são agrupadas para que a etapa de redução seja executada e, finalmente, gerar a saída da aplicação.

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. A composição do HDFS é dividida pelo papel de cada nó em relação à manutenção dos dados, através de 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 modelo MAP-REDUCE, 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*. O *ResourceManager* deve ter uma comunicação com o cliente da aplicação, uma vez que o nó mestre deve atender às requisições de aplicações. Já o *NodeManager* gerencia e oferece seus recursos ao *ResourceManager*, para que este decida como o trabalho será executado.

Um importante fator de configuração dos *jobs* no APACHE HADOOP é o método de escalonamento das tarefas. O APACHE HADOOP 2.7.2, utilizado neste trabalho, usa como escalonador *default* o *CapacityScheduler*, que tem como princípio básico uma alocação de recursos para ambientes com concorrência entre mais de um usuário (organização). Desta forma, uma organização recebe uma garantia mínima de recursos. Em alternativa, outro escalonador que pode ser utilizado no HADOOP é o *FairScheduler*, que realiza 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.

## 3. Apresentação do Problema e Proposta de Mapeamento

Esta seção descreve a metodologia de busca utilizada neste trabalho, bem como noções sobre busca por similaridade em índices invertidos que serão úteis no transcorrer do artigo.

Os índices invertidos são estruturas de dados comumente usadas em problemas de recuperação de informação para responder a consultas formuladas como uma coleção de

termos (palavras chave). Dentro desse contexto, índices invertidos podem ser definidos como apresentado na 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 artigo, 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 $OI_l^i$.

A localização de um objeto depende do tipo de objeto armazenado, e como ele foi armazenado. Exemplos de possíveis valores comprendem IDs de tuplas e o caminho absoluto em sistemas de arquivos.

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 definida por um escore medido pela função $q\_escore(Q, O_x)$ apresentada na equação 1.

$$q\_escore(Q, O_x) = \sum_{i=0}^{n} \sum_{j=0}^{m} escore(k_i, P_j^x) \tag{1}$$

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 $|PI|$.

A Figura 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.

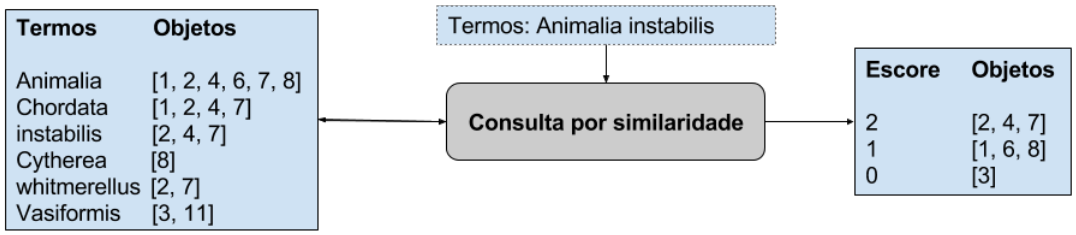



**Figura 1. Exemplo de um caso de busca por similaridade usando índice invertido**

O uso desta função de escore faz com que nenhum objeto seja descartado da resposta, mesmo que o nível de similaridade seja considerado irrelevante, como é o caso, por exemplo, do objeto 3. Para contornar esse problema, o método prevê o uso de um limiar de corte (*threshold*), que define um limite mínimo de escore a ser considerado para a soma.

Para a aplicação deste cenário de busca foram criadas duas abordagens de consulta: centralizada e distribuída. Na abordagem centralizada o processamento é realizado sem auxílio de ferramentas de distribuição ou paralelização. Neste caso, o índice invertido $I$ é lido e o produto cartesiano entre os $k$ termos da consulta $Q$ e as entradas $E$ acontece de forma totalmente sequencial.

O processo possui duas partes principais. Em um primeiro momento, o produto cartesiano é realizado e cada função de escore é calculada e atrelada ao objeto à que se refere. Uma vez finalizados os cálculos, cada objeto possuirá uma lista de escores que é somada, na segunda parte do processamento, para definir a soma total de similaridade. A saída, então, é composta pelos níveis de similaridade e dos objetos que possuem esse valor.

A abordagem distribuída foi concebida de modo que o processamento pudesse ser dividido entre mais de um recurso computacional. Para tal, a distribuição foi realizada utilizando o paradigma MAP-REDUCE, com o auxílio da ferramenta APACHE HADOOP (v2.7.2). A entrada do mapeamento distribuído se dá pelo índice invertido $I$, fazendo com que o índice seja configurado para ser dividido entre os nós de acordo com o número de *splits* definido. Cada *split* gerado deverá conter uma lista de entradas $E$ do índice, podendo cada divisão ter seu cálculo feito de forma independente.

Para aplicar o paradigma MAP-REDUCE, as funções *map* e *reduce* foram desenvolvidas. Todo *mapper* recebe uma lista de entradas $E$ do índice $I$ e, então, executa o procedimento de mapeamento para cada *key-value* contido em $E$. Um único par chave-valor é processado por vez. Cada *key-value* processado pode emitir 0 ou mais saídas, as quais consistem em novos 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. As entradas e saídas das etapas *map* e *reduce* são exibidas na Tabela 1.

| 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 1. Entradas e saídas das etapas do mapeamento de busca distribuído**

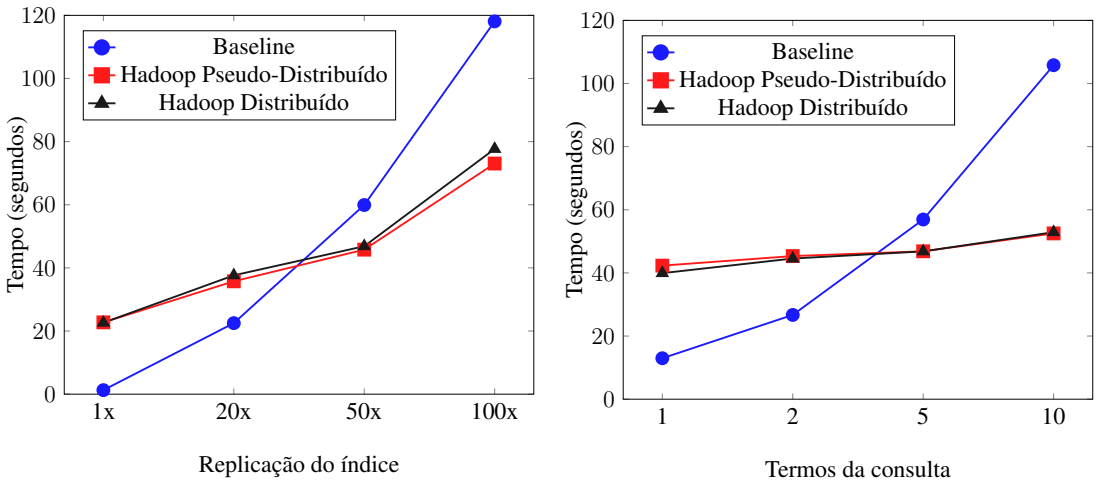
## 4. Experimentos

Nos experimentos realizados, foi usado um *cluster* configurado através da ferramenta *Google Compute Engine*. O *cluster* usado possui 3 nós, sendo 1 *master* e 2 *slaves*, com configurações idênticas: o processador conta com 2 núcleos 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 *slaves*.

O índice invertido foi criado sobre a coleção de objetos do sistema BioID, que é um ambiente colaborativo para a manutenção e catalogação de dados de espécies biológicas. Os objetos do BioID são as espécies e as propriedades indexáveis são atributos e valores. Por exemplo, um objeto $O_i$ pode conter como propriedades indexáveis os atributos $Reino$ e $Classe$, e os valores $Animalia$ e $Mammalia$. Em média, um objeto possui 4 propriedades indexáveis. O repositório de objetos do BioID é composto por 222.120 espécies. Para testar a escalabilidade do algoritmo MAP-REDUCE, foram criadas diferentes amostragens dessa base, criando subconjuntos do total. 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 and Yianilos 1998].

Inicialmente, três estratégias de busca foram comparadas: uma estratégia *baseline*, em que todo o processamento é centralizado, e outras duas utilizando o HADOOP MAP-REDUCE, sendo um cenário Pseudo-Distribuído, em que apenas um nó executa o *job*, e outro Distribuído. O APACHE HADOOP foi definido com suas configurações *default*, ou seja, nenhum atributo de configuração foi alterado.

Os primeiros testes contaram com a variação do tamanho da base utilizados, comparando entre replicações de 1x (base original), 20x, 50x e 100x. A Figura 2(a) mostra o desempenho das soluções centralizada e usando o APACHE HADOOP Pseudo-Distribuído e Distribuído com 3 nós e configurações *default*. Foi utilizado um número de 5 termos de consulta e nível de *threshold* em 50%. Pode-se notar que a abordagem centralizada é mais eficiente quando tem-se uma base pequena, porém perde desempenho de forma significativa a partir da expansão dos dados, alcançando um tempo superior de busca quando se atinge o índice replicado em 50x. Já a Figura 2(b) mostra os testes feitos com uma variação do número de termos da consulta. Nesse quesito, as soluções usando o APACHE HADOOP apresentaram uma eficiente escalabilidade, enquanto o processo centralizado perde bastante desempenho à medida em que a quantidade de palavras aumenta.




(a) Relação entre o desempenho dos mapeamentos e o tamanho do índice utilizado

(b) Desempenho com variação do número de termos da consulta

**Figura 2. Desempenho dos cenários de teste centralizado e usando o Apache Hadoop com configurações *default***

O próximo teste focou na aplicação de diferentes níveis de *threshold* para verificar o impacto do limiar nos mapeamentos propostos, conforme exibe a Figura 3. Foram utilizados níveis em 0% (sem *threshold*), 25%, 50% e 90%. Os resultados mostram que a aplicação de um limite, em todos os casos, possibilta 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 *threshold* de 25%, o erro não acontece e a busca retorna um resultado final.

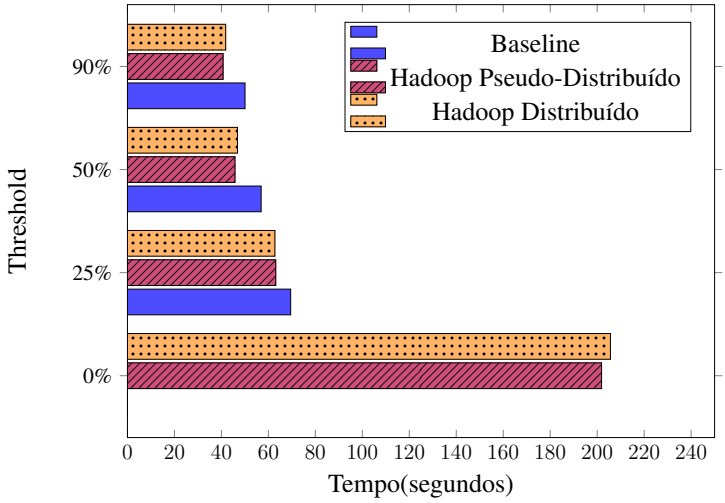
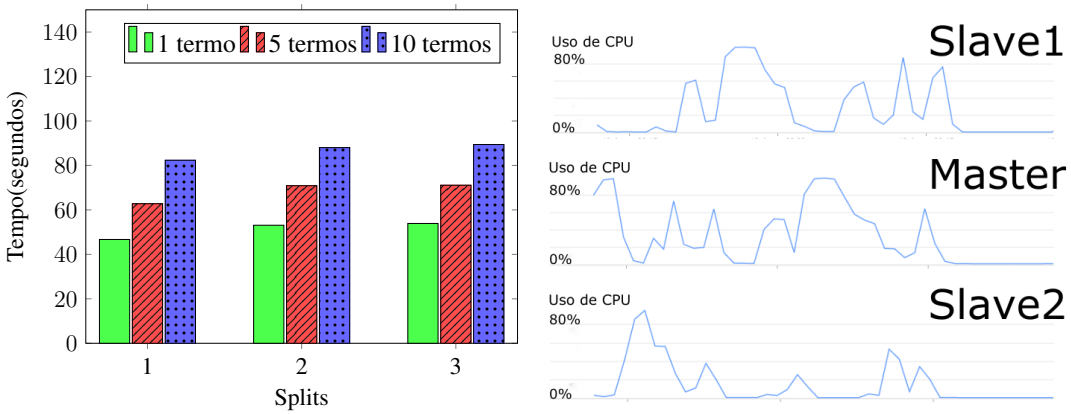



**Figura 3. Impacto dos diferentes níveis de *threshold* nas aplicações de busca**

Os resultados alcançados pelas abordagens Distribuída e Pseudo-Distribuída do HADOOP permaneceram próximos em todos os testes feitos. Uma vez que o *framework* 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. Sem nenhuma divisão, o processos Pseudo-Distribuído e Distribuído executam em apenas um

nó trabalhador. Por isso, foram definidos testes com variações na configuração do APACHE HADOOP a fim de forçar 2 e 3 *splits* para cada caso de replicação do índice. Como mostra a Figura 4(a), o desempenho variando-se o número de *splits* pouco muda, chegando a piorar conforme se aumenta esse nível. Esse comportamento, como mostra a Figura 4(b), ocorreu pelo fato de ainda haver apenas um nó com atividade intensa em cada *job*.




(a) Tempo de execução para casos de 1, 2 e 3 *splits* (b) Uso de CPU durante testes em um cenário com 2 e 3 *splits*

**Figura 4. Desempenho e uso de CPU para a busca com mais de 1 *split***

Um dos possíveis motivos é a forma como o escalonador estaria distribuindo as tarefas através do *cluster*, que poderia não estar utilizando todo o poder computacional do ambiente para a execução da aplicação de busca. Sendo assim, modificando a configuração *default* do HADOOP, foi configurado o uso do escalonador *FairScheduler*, que divide o *cluster* de forma igualitária entre os *jobs*. Os testes foram feitos comparando-se o desempenho dos escalonadores com 2 e 3 *splits* em uma busca no índice replicado em 50x, 5 termos de consulta e 50% de *threshold*. O ganho de desempenho do *FairScheduler* em relação ao *baseline* e o APACHE HADOOP *default* com 2 e 3 *splits* é visto na Figura 5.

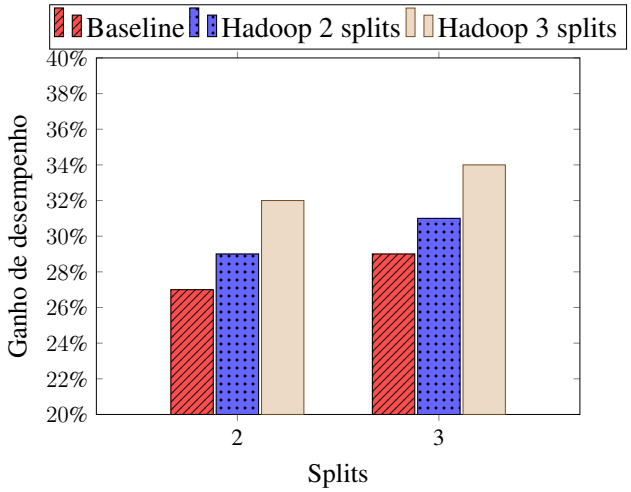



**Figura 5. Ganho de desempenho do escalonador *FairScheduler* em relação ao escalonador *default* do Hadoop**

Pode-se verificar que o escalonador justo chega a proporcionar um desempenho 30% mais eficiente. Essa característica expõe uma lacuna do escalonador Capacity Scheduler em relação à utilização de todo o potencial computacional disponível. A correção desse problema foi possível com o escalonador Fair Scheduler, o qual utilizou de forma mais proveitosa o cluster para a alocação das tarefas MAP-REDUCE.

## 5. Trabalhos Relacionados

A área de recuperação de informação é um terreno fértil para o uso do MAP-REDUCE. Exemplos incluem o uso de termos para a pesquisa de documentos hierárquicos [Zhou et al. 2010] e o uso de termos com conotação espacial para a localização dos objetos mais próximos [Li et al. 2012]. A própria construção de índice invertidos pode se valer do MAP-REDUCE, conforme descrito em [Dean and Ghemawat 2008]. Para tanto, um histograma contendo todos termos é inicialmente criado. O *mapper* emite pares compostos pelo id do documento e cada um dos termos que ele possui. Por sua vez, para cada termo, o *reducer* agrupa os ids dos documentos que o possuem.

Trabalhos que usam índices invertidos em algoritmos MAP-REDUCE são normalmente associados à computação da similaridade entre documentos. A similaridade entre documentos é um recurso útil para ambientes onde a navegação é orientada a documentos, ou seja, quando um documento acessado leva a outros documentos relevantes. Nesse contexto, o trabalho de [Elsayed et al. 2008] utiliza MAP-REDUCE para resolver o problema do cálculo da similaridade entre todos pares de documentos de uma base composta por artigos jornalísticos. Índices invertidos são usados para indexar os documentos, e esse índice serve de entrada para a primeira camada de mapeamento. O resultado final é uma lista composta por pares de documentos associados a um valor de escore. A similaridade entre pares de documentos também foi explorada pelo trabalho de [Lin 2009], que testou diferentes algoritmos usando o *framework Hadoop*.

Um aspecto em comum com a nossa proposta é o uso do produto cartesiano no cálculo da similaridade entre termos do índice invertido e os termos de cada documento. No entanto, a correspondência entre termos é feita por equivalência. Um tópico relacionado é a pesquisa de algoritmos MAP-REDUCE para resolver o problema de junções por similaridade [Metwally and Faloutsos 2012, Vernica et al. 2010]. Apesar desses trabalhos utilizarem funções de similaridade, o objetivo é ortogonal ao nosso, já que o resultado são conjuntos de objetos similares e não objetos relevantes à termos de uma consulta.

De modo geral, trabalhos científicos normalmente exploram o uso de algoritmos MAP-REDUCE para processamentos *offline*. Já a busca por termos, seja ela por equivalência ou por similaridade, requer processamento em tempo real. Ou seja, esse tipo de aplicação é um nicho ainda pouco explorado. Apesar de o processamento distribuído de consultas baseadas em termos ser em muitos casos mais custoso do que o processamento isolado, existem situações em que a distribuição compensa, como foi descrito na seção anterior.

## 6. Considerações finais

Este trabalho mostrou uma perspectiva sobre o problema de busca por similaridade usando índices invertidos com uma ferramenta de distribuição de processamento. Em relação ao tempo de execução da consulta, percebe-se que o desempenho da busca utilizada neste trabalho não é eficiente quando se utiliza uma base de dados pequena, em comparação à

uma abordagem centralizada. Como esse tipo de busca prioriza o tempo de resposta, o uso da técnica acaba não sendo uma boa opção quando se utiliza pouca informação.

Porém, destaca-se a grande esacalabilidade do *framework* APACHE HADOOP, consolidando um melhor tempo de execução em relação ao *baseline* a partir de uma replicação em 50x do índice invertido utilizado. Outro fator que deve ser levado em consideração é o impacto da aplicação de um *threshold* no processamento da busca. Como o APACHE HADOOP utiliza o disco para realizar a comunicação entre as etapas de *map* e *reduce*, a limitação da quantidade de informação que é emitida pelos mappers acaba gerando arquivos intermediários menores e, consequentemente, um ganho de desempenho significativo.

Além disso, pode-se verificar uma diferença de desempenho notável entre os tipos de escalonador utilizados. Enquanto o escalonador *default* do APACHE HADOOP (v2.7.2) concentra-se em uma divisão mínima de recursos por organização, o *FairScheduler* consegue dividir o *cluster* com uma noção igualitária de recursos por utilizador, permitindo um melhor uso do ambiente para a aplicação deste trabalho.

## Referências


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

Elsayed, T., Lin, J., and Oard, D. W. (2008). Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, pages 265–268, Stroudsburg, PA, USA. Association for Computational Linguistics.

Korfhage, R. R. (2008). *Information storage and retrieval*. Wiley.

Li, W., Wang, W., and Jin, T. (2012). Evaluating spatial keyword queries under the mapreduce framework. In *International Conference on Database Systems for Advanced Applications*, pages 251–261. Springer.

Lin, J. (2009). Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 155–162, New York, NY, USA. ACM.

Metwally, A. and Faloutsos, C. (2012). V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715.

Ristad, E. S. and Yianilos, P. N. (1998). Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532.

Vernica, R., Carey, M. J., and Li, C. (2010). Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM.

White, T. (2012). *Hadoop: The definitive guide*. "O'Reilly Media, Inc.".

Zhou, M., Hu, H., and Zhou, M. (2010). Searching xml data by slca on a mapreduce cluster. In *Universal Communication Symposium (IUCS), 2010 4th International*, pages 84–89. IEEE.