LUÍS HENRIQUE BORGES SOUSA

JUNÇÕES POR SIMILARIDADE EM MÚLTIPLOS CONJUNTOS UTILIZANDO MAPREDUCE

LAVRAS – MG

2013

LUÍS HENRIQUE BORGES SOUSA

JUNÇÕES POR SIMILARIDADE EM MÚLTIPLOS CONJUNTOS UTILIZANDO MAPREDUCE

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Orientador
Prof. Leonardo Andrade Ribeiro

LAVRAS – MG

2013

LUÍS HENRIQUE BORGES SOUSA

JUNÇÕES POR SIMILARIDADE EM MÚLTIPLOS CONJUNTOS UTILIZANDO MAPREDUCE

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

APROVADA em 29 de Agosto de 2013.

Profª. Marluce Rodrigues Pereira	UFLA
Prof. Denilson Alves Pereira	UFLA

Prof. Leonardo Andrade Ribeiro

(Orientador)

LAVRAS – MG

2013

Dedico este trabalho à minha família.

AGRADECIMENTOS

Agradeço ao Prof. Leonardo, meu orientador, pelo apoio e dedicação durante a realização deste projeto, e também aos demais membros do Grupo de Pesquisa em Banco de Dados do Departamento de Ciência da Computação da UFLA pelo companheirismo e aprendizado.

RESUMO

Operações de similaridade são operações fundamentais para busca, gerenciamento e análise de dados. Apesar dessas operações serem geralmente bastante onerosas computacionalmente, já foram desenvolvidos algoritmos eficientes. No entanto, para a realização de operações de similaridade sobre grandes volumes de dados, técnicas de programação paralela e distribuída são imprescindíveis. O objetivo deste trabalho é generalizar o algoritmo *mpjoin* para múltiplos conjuntos com pesos para execução em paralelo utilizando o *framework* MapReduce. Visto que até o presente momento apenas o algoritmo *ppjoin* (para um único conjunto e sem pesos) foi utilizado, este trabalho traz novidades para a área de pesquisa. Três estratégias para distribuição do algoritmo são propostas, sendo que, de acordo com os experimentos realizados, a terceira é a mais eficiente e escalável.

Palavras-Chave: Banco de Dados, Junções de Similaridade, MapReduce, Computação paralela, Computação Distribuída.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

A maioria dos Sistemas Gerenciadores de Bancos de Dados (SGBD) implementam a busca por informações idênticas, por meio de comparações *byte a byte* ou casamento de padrões entre cadeias de caracteres. Porém, nem sempre deseja-se encontrar registros idênticos, mas sim os que possuem alguma similaridade.

Uma das principais aplicações de operações por similaridade é a detecção de duplicatas. O objetivo é identificar informações cadastradas de formas distintas mas que condizem a uma mesma entidade. Erros de digitação e abreviações, por exemplo, podem gerar duplicatas no banco de dados. Dessa forma, "Antônio J. Alves" ou 'Antônio J. Alvse' provavelmente referem-se à mesma pessoa que "Antônio José Alves" se ambos os registros possuem um único endereço no cadastro. Outros exemplos são motores de busca que verificam páginas duplicadas ou plágios (VERNICA; CAREY; LI, 2010), servidores de e-mails que detectam *spams* verificando se eles seguem modelos utilizados por *spammers* (XIAO *et al.*, 2008) e também a remoção de erros de tipos ou de digitação (*data cleaning*) (CHAUDHURI; GANTI; KAUSHIK, 2006).

Além disso, a busca por entidades similares ainda está presente em diversas outras aplicações. Por exemplo, em técnicas de filtragem colaborativa (ou sistemas de recomendação) existem funcionalidades para sugestão de amigos em redes sociais, assim como de produtos em lojas virtuais, baseadas na rede de contatos do usuário ou nas últimas compras feitas pelo mesmo (SPERTUS; SAHAMI; BUYUKKOKTEN, 2005).

A verificação de registros similares não é trivial devido a diversos fatores e abordagens utilizadas. Entre elas estão as técnicas para geração de pares candidatos e os métodos para evitar comparações desnecessárias. No entanto, detectar informações similares é um desafio atualmente, pois a quantidade de dados é cada

vez maior – principalmente na *Web*, devido à expansão da computação nas nuvens (*Cloud computing*).

Por exemplo, o motor de busca *Bing*, da *Microsoft*, indexa cerca de 20 bilhões de documentos *web* (WANG *et al.*, 2010). Esses documentos e seus índices estão armazenados de forma distribuída em pelo menos 10.000 máquinas pelo mundo, totalizando mais de 300 TB (*terabytes*) de informação (WANG *et al.*, 2010). Como o tempo gasto no processo de indexação é proporcional à quantidade de dados, se feita a detectação e remoção de duplicatas, o custo do processo será substancialmente menor.

Dois algoritmos no estado da arte e demonstrados como os mais eficientes são o *ppjoin* e o *mpjoin*. O primeiro, proposto em (XIAO *et al.*, 2008) como uma melhoria do algoritmo *All-Pairs* (BAYARDO; MA; SRIKANT, 2007), é uma implementação que utiliza filtragens por posição e pelo prefixo. Por outro lado, o *mpjoin* utiliza uma generalização do conceito de filtragem por prefixo e propõe uma nova abordagem para diminuir o processamento na fase de geração de candidados e aumentar a carga de trabalho na etapa de verificação dos registros similares (RIBEIRO; HÄRDER, 2011).

Apesar das otimizações existentes, para que essas aplicações trabalhem com conjuntos massivos de dados e sejam eficientes, é preciso utilizar uma solução paralela e distribuída. Cada máquina deve realizar uma fração da computação total do problema para que, no fim, todos os resultados parciais sejam analisados e mesclados para constituir o resultado final.

Impulsionado pelo Google (DEAN; GHEMAWAT, 2004), o modelo de programação chamado *MapReduce* tem sido amplamente utilizado por pesquisadores e organizações para a computação em *clusters*. Além da simplicidade, um dos motivos em utilizar o *MapReduce* é sua capacidade de tolerância a falhas.

1.1 Definição do Problema

Para a paralelização das operações de similaridade é preciso particionar o processamento e também os dados para que a computação seja distribuída entre as máquinas. A dificuldade existente é que o particionamento feito simplesmente sobre toda cadeia de caracteres ou conjunto é insuficiente, mesmo se disponível um *cluster* com muitas máquinas. Dessa forma, a generalização de um algoritmo de operações de similaridade requer estratégias para o particionamento dos dados e distribuição do processamento, possibilitando que cada máquina execute tarefas independentemente das demais.

O algoritmo *mpjoin* (RIBEIRO; HÄRDER, 2011) foi demonstrado ser mais eficiente do que o *ppjoin* (XIAO *et al.*, 2008), mas até então apenas o segundo foi generalizado para execução utilizando o *MapReduce* (VERNICA; CAREY; LI, 2010).

Outro problema presente nos trabalhos existentes é a utilização de um único conjunto para representação dos registros. Isto é, registros que possuem mais de um atributo são modificados e todos os atributos são concatenados em um único conjunto. Como será descrito posteriormente, essa abordagem gera inconsistências nos resultados das junções por similaridade. Para evitar esses problemas, o algoritmo generalizado (*mpjoin*) utiliza o conceito de múltiplos conjuntos.

1.2 Objetivos Gerais e Específicos

O objetivo geral do trabalho é paralelizar o algoritmo *mpjoin* (RIBEIRO; HÄRDER, 2011) do estado da arte para identificação de registros similares baseando-se na semelhança de cadeias de caracteres pertencentes a um conjunto de atributos de cada registro. Executado de forma distribuída em um *cluster*, o algoritmo deve

explorar os benefícios da programação distribuída mantendo a exatidão dos resultados das operações de junção por similaridade.

Como objetivos específicos, ou seja, os objetivos intermediários até se atingir o objetivo geral, tem-se:

- Implementação do algoritmo *mpjoin* sobre o framework MapReduce, especificamente no *Apache Hadoop*;
- Distribuição do processamento das operações de similaridade por meio da adaptação da versão sequencial do algoritmo *mpjoin*;
- Criação de um *software* capaz de realizar junções por similaridade em múltiplos conjuntos de forma distribuída utilizando o modelo de programação *MapReduce*;
- Construção de um ambiente de testes (dados e *hardware*)
- Comparação dos tempos gastos pelo algoritmo sequencial e pelo algoritmo paralelo proposto, tal como cálculo de *speedup* e *scale-out*.

1.3 Contribuições

A contribuição deste trabalho com a área de pesquisa é essencialmente a generalização do algoritmo *mpjoin* para junções por similaridade para execução no framework *MapReduce*. Com a utilização desse algoritmo, este trabalho possui como diferencial a abordagem sobre múltiplos conjuntos e de conjuntos com pesos que, conforme será descrito adiante, implicam em maior acurácia dos resultados das junções por similaridade.

Três estratégias de funcionamento do algoritmo distribuído são propostas e diferentes experimentos foram feitos a fim de demonstrar o desempenho de cada abordagem.

Visto que a quantidade de dados é cada vez maior e uma variedade de aplicações necessita fazer detecção de duplicatas, pode-se dizer que os trabalhos nesta área de pesquisa terão utilidade em diversas soluções, sejam elas comerciais ou acadêmicas.

1.4 Estrutura do Trabalho

Este capítulo contextualizou a área de pesquisa, as dificuldades existentes e apresentou o conceito de operações por similaridade. No Capítulo 2 estão definidos detalhadamente os algoritmos, conceitos, ferramentas e tecnologias que constituem ou contribuem para o projeto final. O Capítulo 3 descreve a implementação e as estratégias utilizadas durante o desenvolvimento do trabalho. No Capítulo 4 os resultados obtidos estão ilustrados e discutidos com as devidas explanações sobre cada experimento. O Capítulo 5 encerra este trabalho com as conclusões encontradas e direciona trabalhos futuros nessa área de pesquisa.

2 REFERENCIAL TEÓRICO

2.1 Funções de Similaridade

No assunto de funções de similaridade existem diferentes abordagens, como os algoritmos baseados em distância de edição e em *tokens*. Este trabalho utiliza o segundo, baseado em conjuntos de *tokens*, que foi mostrado ser geralmente tão eficiente quanto o primeiro, porém mais flexível para modificações que visam evitar alguns cálculos de similaridade (RIBEIRO, 2010).

As funções de similaridade baseadas em *tokens* geralmente são compostas por três componentes (Figura 2.1): tokenização, ponderação e cálculo de intersecção. Cada componente está descrito a seguir.

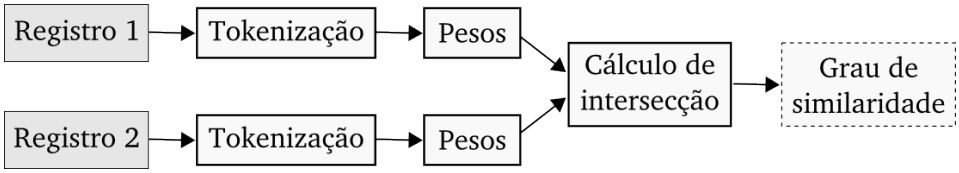

Figura 2.1: Funcionamento de uma função de similaridade baseada em *tokens*

2.1.1 Tokenização

As *strings* são representadas como conjuntos de *tokens*, isto é, são divididas em *substrings* menores para possibilitar a verificação do grau de similaridade.

A primeira forma de tokenização é pela divisão do texto pelos espaços e caracteres de pontuação. Por exemplo, as *strings* “Universidade Federal de Lavras” e “Universidade Federal de Minas Gerais” podem ser tokenizadas como [“Universidade”, “Federal”, “de”, “Lavras”] e [“Universidade”, “Federal”, “de”, “Minas”, “Gerais”].

Essa abordagem em que cada palavra corresponde a um *token* não é eficiente em muitos casos, como o de identificar erros de digitação. Portanto, introduz-se um conceito importante em funções de similaridade, que é o chamado *q-grams*. São *substrings* de tamanho q obtidas através do "deslizamento" de uma janela de tamanho q sobre determinada *string* (UKKONEN, 1992).

Por exemplo, sejam w_1 = "Universidade" e w_2 = "Univresidade" duas *strings*. Seus *2-grams* são:

$$q(w_1) = \{un, ni, iv, ve, er, rs, si, id, da, ad, de\}$$

$$q(w_2) = \{un, ni, iv, vr, re, es, si, id, da, ad, de\}$$

Após a tokenização, utiliza-se alguma função de similaridade para fazer o cálculo da similaridade, que a partir de então se torna uma operação sobre conjuntos.

2.1.2 Pesos

Uma abordagem comum é definir o peso de cada *token* de acordo com a sua frequência em um determinado texto. Uma abordagem conhecida é baseada no *IDF* (*Inverse Document Frequency*) (ROBERTSON; JONES, 1976), que basicamente utiliza a intuição de que uma palavra menos frequente, ou rara, é mais significante do que palavras com muitas ocorrências em um texto.

O cálculo do peso de um *token* baseado no IDF se dá pela seguinte fórmula:

$$IDF(t) = \log(1 + \frac{N}{freq(t)}) \tag{2.1}$$

Onde N é o número total de registros e $freq(t)$ é o número de registros em que t ocorre.

Na utilização de conjuntos com pesos, um método utilizado é fazer o somatório dos pesos de cada um dos seus *tokens* para definir o peso total do conjunto. O tamanho com peso de um conjunto x, denotado por $w(x)$, é a soma dos pesos de todos seus elementos, onde t é cada *token* pertencente ao conjunto x (Equação 2.2).

$$w(x) = \sum_{t \varepsilon x} w(t) \tag{2.2}$$

Como exemplo, considere as *strings* "Universidade Federal de Lavras" e "Universidade Federal de Minas Gerais". Se os *tokens* "Universidade", "Federal" e "de" são mais frequentes no texto, portanto, possuem pesos baixos. Por outro lado, os *tokens* "Lavras", "Minas" e "Gerais" são menos frequentes e, por isso, tem pesos maiores. Por exemplo, com os registros considerados na seção de tokenização, pode-se supor que os *tokens* "Universidade" e "Federal" possuem peso 2, e o *token* "de" tem peso 1, visto que é uma palavra bastante frequente na lingua portuguesa. Os outros *tokens* são supostos serem menos frequentes no texto, como os *tokens* "Minas" e "Gerais", com peso 5, e o *token* "Lavras", considerado o mais raro, com peso 8. Esses valores serão utilizados nas seções seguintes.

2.1.3 Cálculo da Intersecção

Uma função de similaridade opera sobre conjuntos de *tokens* para efetuar o cálculo de similaridade entre dois registros. O valor resultante está contido no intervalo $[0,1]$, sendo 1 quando os conjuntos são idênticos e 0 quando são totalmente diferentes. Na Tabela 2.1 estão as funções utilizadas na implementação deste trabalho e que são as mais comuns no estado da arte: Dice, Cosseno e Jaccard (RIJSBERGEN, 1979).

Tabela 2.1: Definição das funções de similaridade baseadas em *tokens*

Função	Definição
Dice	$2\frac{\lvert x_1 \cap x_2 \rvert}{\lvert x_1 \rvert + \lvert x_2 \rvert}$
Cosseno	$\frac{\lvert x_1 \cap x_2 \rvert}{\sqrt{\lvert x_1 \rvert \lvert x_2 \rvert}}$
Jaccard	$\frac{\lvert x_1 \cap x_2 \rvert}{\lvert x_1 \cup x_2 \rvert}$

Por exemplo, a função de similaridade de *Jaccard* (JS) é a razão entre o número de *tokens* em comum (intersecção) e o número total de *tokens* distintos (união). Utilizando essa função, as *strings* "Universidade Federal de Lavras" e "Universidade Federal de Minas Gerais", após a tokenização por palavras, possuem 3 *tokens* em comum ("Universidade", "Federal", "de") e 6 *tokens* distintos ("Universidade", "Federal", "de", "Lavras", "Minas", "Gerais"). Portanto, a função de similaridade resulta em $3/6 = 0,5$.

A função *Jaccard* para conjuntos com peso (*Weighted Jaccard Similarity*, Equação 2.3) é uma variação da função de *Jaccard* da tabela anterior que utiliza o peso de cada conjunto para efetuar o cálculo de similaridade.

$$WJS(x_1, x_2) = \frac{w(x_1 \cap x_2)}{w(x_1 \cup x_2)} \tag{2.3}$$

Considerando a abordagem com peso sobre as *strings* do exemplo anterior, tem-se:

$$WJS(x_1, x_2) = \frac{w(x_1 \cap x_2)}{w(x_1 \cup x_2)} = \frac{2+2+1}{2+2+1+5+5+8} = \frac{5}{23} \approx 0.217$$

Por isso, com uma função de similaridade que considera os pesos na verificação, o grau de similaridade entre as *strings* será mais consistente com os signi-

ficados reais dos registros, mesmo que eles possuam vários *tokens* (não raros, no caso) em comum.

É importante ressaltar que todas essas funções de similaridade operam em função da interseção (ou *overlap*) dos conjuntos. Utiliza-se uma função *minoverlap* (Equação 2.4) para definir o tamanho mínimo necessário para intersecção de dois conjuntos de *tokens* baseado em um *threshold* τ. Dessa forma, conjuntos distintos podem ser descartados sem verificar todos seus elementos, como será descrito posteriormente neste trabalho. Por exemplo, a função de *Jaccard* representada como uma função de *minoverlap* entre dois conjuntos x e y de *tokens* é da seguinte forma:

$$minoverlap(x,y) = \frac{\tau}{1+\tau}(|x|+|y|) \tag{2.4}$$

2.2 Junção por Similaridade

A definição de (RIBEIRO; HÄRDER, 2011) para junção por similaridade é a seguinte: dados uma coleção C registros, uma função de similaridade *sim*, e um *threshold* τ, define-se a junção por similaridade de C como a busca e combinação de todos os pares de registros de $x_1, x_2 \in C$ tal que $sim(x_1, x_2) \geq \tau$.

Geralmente os algoritmos de junção por similaridade são compostos de duas fases principais: geração de pares candidatos e verificação. Na primeira fase, descarta-se pares que não possuem chance de serem similares, baseando-se no *threshold* e tamanho do conjunto, por exemplo. Na segunda fase, faz-se a aplicação da função de similaridade e o cálculo do grau de similaridade sobre os pares filtrados na fase anterior.

2.3 Filtragem de Candidatos

Uma técnica existente no estado da arte é chamada de *Size Filtering*, ou filtragem por tamanho. O objetivo é descartar os registros com tamanhos menores ou maiores do que o intervalo criado pelo filtro. O princípio é considerar que registros similares possuem tamanhos similares (SARAWAGI; KIRPAL, 2004).

Outra técnica de filtragem é o *Prefix Filtering*, proposto para filtragem de candidatos utilizando a verificação de um prefixo de tamanho pré-determinado (SARAWAGI; KIRPAL, 2004). Esse filtro consiste em utilizar o *threshold* para calcular o tamanho de uma janela tal que os conjuntos que não possuírem algum *token* em comum dentro da mesma serão descartados. Os prefixos são obtidos após a ordenação de cada conjunto de *tokens* seguindo uma mesma ordenação global *O* – comumente usa-se a ordenação pela frequência de cada palavra no texto. Utiliza-se o prefixo do conjunto para que apenas os *tokens* iniciais sejam comparados.

Por exemplo, considerando dois conjuntos x e y de tamanhos iguais a 10, um *threshold* τ = 0,8 e a função de *minoverlap* baseada em *Jaccard* (Equação 2.4), tem-se que $minoverlap(x,y) = (0.8 * (10+10)/(1+0.8)) \approx 8,88$. Suponha que os conjuntos estão ordenados por alguma ordem *O* e que, por simplicidade, os *tokens* não possuem peso – onde uma função de tamanho do conjunto, *tam(x)*, é igual à cardinalidade de x, isto é, a quantidade de *tokens* pertencentes ao conjunto. Através da Equação 2.5, tem-se $prefsize(x) = 10 - 9 + 1 = 2$, ou seja, se não houver um *token* em comum dentro de um prefixo de tamanho 2, então o par de conjuntos candidatos pode ser descartado, pois não existe possibilidade de o grau de similaridade ser igual ou superior ao τ = 0,8.

$$prefsize(x) = tam(x) - \lceil minoverlap(x,y) \rceil + 1 \tag{2.5}$$

Os conjuntos de *tokens* precisam estar ordenados para garantir o funcionamento correto do algoritmo. Por exemplo, sejam r e s dois conjuntos de *tokens*, $r = \{e, a, *, *, *\}$ e $s = \{b, c, *, *, *\}$, onde o caractere * é alguma letra. Considerando a ordenação lexicográfica, os conjuntos de *tokens* não estão ordenados. Dessa forma, não é possível afirmar se a intersecção entre r e s possui, por exemplo, menos de 4 elementos. Se os conjuntos fossem $r = \{e, a, b, c, d\}$ e $s = \{b, c, a, e, d\}$, então eles possuem intersecção igual a 5. Por outro lado, se os conjuntos estão ordenados e são $r = \{a, e, *, *, *\}$ e $s = \{b, c, *, *, *\}$, então é possível afirmar que r e s não possuem uma intersecção de tamanho 4 em comum, mesmo sem acessar os demais elementos dos conjuntos (WANG; LI; FENG, 2012). Na abordagem por pesos, os *tokens* mais raros (ou com maior peso) são mantidos no prefixo para melhorar o desempenho da filtragem por prefixo, visto que menos candidatos terão tokens em comum no prefixo.

Ao verificar que a técnica de filtragem de prefixos possui um crescimento quadrádico do número de candidatos, novas técnicas foram propostas. Entre elas estão o *Position Filtering* e o *Suffix Filtering* – utilizados respectivamente nos algoritmos *ppjoin* e *ppjoin+* (XIAO *et al.*, 2008) – para desconsiderar ainda mais pares inviáveis. O primeiro utiliza a posição do *token* para definir um limite superior e comparar com o *threshold*, podendo descartar elementos em etapas iniciais. O segundo utiliza uma abordagem de verificar dinamicamente as posições de um registro em outro registro. Os três filtros (de prefixo, posição e sufixo) podem ser utilizados simultaneamente. (LAM *et al.*, 2010).

Por fim, tem-se a técnica do *Min-prefix*, uma generalização do *Prefix Filtering* aplicada a conjuntos indexados. Em suma, o *Min-prefix* permite manter dinamicamente o tamanho das listas invertidas reduzidas a um mínimo e, dessa forma, o tempo gasto na fase de geração de candidatos é reduzido drasticamente (RIBEIRO; HÄRDER, 2011).

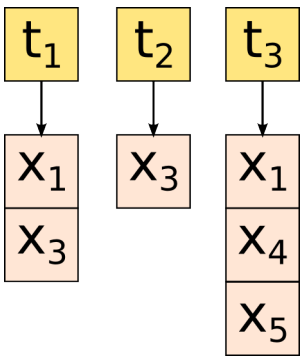

Figura 2.2: Exemplo de lista invertida gerada pela fase de indexação

2.4 Algoritmos de Junção por Similaridade

A maioria dos algoritmos de junção por similaridade são compostos por três fases:

- Indexação: nesta etapa os *tokens* são escaneados e indexados em um índice invertido utilizado para manter uma lista de todos os registros que possuem determinado *token*. Dessa forma, pode-se combinar os registros da lista para formar os pares de candidatos que possuem pelo menos um *token* em comum. A Figura 2.2 exemplifica a criação de uma lista invertida, onde t_i representa um *token*, e x_j um conjunto de *tokens*.

- Geração de candidados: Os primeiros pares obtidos da etapa de indexação são analisados e filtrados. Diferentes filtros podem ser aplicados para diminuir a quantidade de candidatos, como os filtros por prefixo e por tamanho descritos anteriormente.

- Verificação: etapa em que a função de similaridade é aplicada sobre os pares filtrados pela fase de geração de candidatos. O algoritmo então retorna apenas os pares que satisfazem o predicado da junção por similaridade.

Um dos primeiros algoritmos de junção por similaridade a utilizar o *Prefix Filtering* foi proposto em (SARAWAGI; KIRPAL, 2004), denominado de *All-Pairs*, e melhorado por (BAYARDO; MA; SRIKANT, 2007). Foi demonstrado por (XIAO *et al.*, 2008) que é um algoritmo eficiente e escalável, mas assim como outros algoritmos baseados em *Prefix Filtering*, geralmente gera uma grande quantidade de pares candidatos que precisam ser verificados na etapa seguinte.

Os algoritmos *ppjoin* e *ppjoin+* (XIAO *et al.*, 2008) são modificações do algoritmo *All-Pairs* que foram mostrados como de melhor desempenho do que o algoritmo base. O *ppjoin* inclui o *Position Filtering*, enquanto o *ppjoin+* explora o *Suffix Filtering*. Devido às filtragens aplicadas, existe a restrição de que os conjuntos devem estar ordenados.

A abordagem mais proposta por alguns pesquisadores é fazer uma filtragem mais intensa de registros na etapa de geração de pares candidatos para que a segunda etapa seja realizada sobre menos dados e, consequentemente, tenha menor tempo de execução. No entanto, pode acontecer de aumentar o tempo de execução gasto na primeira etapa e não haver diminuição considerável na segunda etapa, mesmo com menos registros.

Em (RIBEIRO; HÄRDER, 2011) propõe-se uma abordagem no caminho contrário dos algoritmos citados até o momento: diminuir o tempo de processamento para geração de pares candidatos e aumentar a carga de trabalho na fase de verificação de candidatos. A principal ideia do algoritmo *mpjoin* é explorar o conceito do *min-prefix* para reduzir dinamicamente o tamanho do prefixo a um mínimo. Dessa forma, vários conjuntos candidatos irrelevantes nunca serão acessados e então o custo de processamento das listas é reduzido consideravelmente. Também é apresentada a implementação do *mpjoin* para conjuntos com pesos e o algoritmo é denominado de *w-mpjoin* (*weighted-mpjoin*). Ele utiliza a soma dos pesos de todos registros de um conjunto para definir o peso do próprio conjunto. O artigo con-

cluiu que tanto o *mpjoin* quanto o *w-mpjoin* são mais eficientes do que os demais algoritmos anteriores.

2.5 Múltiplos Conjuntos

Nos algoritmos existentes no estado da arte, todos atributos são concatenados em uma única *string* antes de serem analisados. Como destacado na Tabela 2.2, a palavra "Lavras" é comum entre os registros, porém ela pertence a atributos diferentes. Isso pode ser um problema, pois, dependendo do objetivo da junção por similaridade, registros distintos ou pouco similares terão *tokens* em comum devido a essa concatenação dos atributos.

Tabela 2.2: Exemplo com dois registros e seus atributos

Nome	Cidade	Universidade
José da Silva	**Lavras**	Universidade Federal de Minas Gerais
Maria Aparecida Souza	Belo Horizonte	Universidade Federal de **Lavras**

Por exemplo, cada registro da Tabela 2.2 será tokenizado em um único conjunto, resultando nos dois conjuntos representados na Tabela 2.3.

Tabela 2.3: Exemplo de dois registros tokenizados e agrupados em um único conjunto

Nome ∪ Cidade ∪ Universidade
"José", "da", "Silva", **"Lavras"**, **"Universidade"**, **"Federal"**, **"de"**, "Minas", "Gerais"
"Maria", "Aparecida", "Souza", "Belo", "Horizonte", **"Universidade"**, **"Federal"**, **"de"**, **"Lavras"**

Se aplicada uma função de similaridade sobre esses dois registros, que não possuem semelhanças visíveis, o grau de similaridade será considerável, devido

aos quatro *tokens* em comum, mesmo com as ocorrências de “Lavras” em atributos diferentes (Cidade e Universidade).

Com esse problema, surge o conceito de múltiplos conjuntos, de forma a não considerar todos os atributos em uma única *string*. Algoritmos de junção por similaridade são adaptados para efetuar a junção por similaridade com um *threshold* específico para cada atributo, e depois calcular se os registros realmente são similares. Utilizando essa nova abordagem agora tem-se cada registro tokenizado em múltiplos conjuntos, resultando em duas *famílias de conjuntos* (ou *conjuntos de conjuntos*), como ilustrado na Tabela 2.4.

Tabela 2.4: Exemplo com dois registros e seus atributos na abordagem de múltiplos conjuntos

Nome	Cidade	Universidade
"José", "da", "Silva"	"Lavras"	"**Universidade**", "**Federal**", "**de**", "Minas", "Gerais"
"Maria", "Aparecida", "Souza"	"Ouro", "Preto"	"**Universidade**", "**Federal**", "**de**", "Lavras"

Portanto, se antes os registros tinham o *token* “Lavras” em comum, na abordagem de múltiplos conjuntos esse *token* não é considerado, pois pertence a conjuntos distintos. Dessa forma o resultado da junção por similaridade tende a ser mais condizente com o significado real desses registros.

Em suma, cada atributo é associado a um predicado composto por uma função de similaridade e um *threshold*. Os predicados serão conectados pelo operador booleano *and*, denotado por $\wedge$. Dessa forma, a condição da junção por similaridade será satisfeita somente quando todos predicados também forem. Poderia ser utilizado o operador *or*, $\vee$, fazendo que se um único predicado for verdadeiro então a condição será satisfeita.

Nos algoritmos *mpjoin* e *w-mpjoin* as otimizações são feitas em apenas um dos conjuntos. Se o predicado for satisfeito, então os demais conjuntos são indexados e verificados. A escolha do conjunto a ser otimizado não está no escopo deste trabalho.

2.6 *MapReduce*

MapReduce é um modelo de programação para processamento de grandes volumes de dados em um *cluster* (DEAN; GHEMAWAT, 2004). Além disso, a arquitetura implementada pelo Hadoop (WHITE, 2012) é baseada em um sistema de arquivos distribuído e no escalonamento e coordenação de tarefas executadas pelos nós integrantes do *cluster*.

2.6.1 Modelo de Programação

A computação no modelo *MapReduce* implementado pelo Hadoop é dividida em duas interfaces: *Map* e *Reduce*. Os dados de entrada do *Map* são representados como pares chave/valor *(k, v)*. O processo Map, também referenciado como *mapper*, aplica a função *map* sobre cada par (k_1, v_1) de entrada e produz zero ou mais pares intermediários *(lista(k_2, v_2))*, onde a chave k_2 não necessariamente é igual à chave k_1.

Após finalizar todas as execuções *map*, o *MapReduce* notifica as tarefas *Reduce* para elas iniciarem o processamento. Antes das execuções dos processos *Reduce*, ou *reducers*, o MapReduce executa uma fase de ordenação e agrupamento (*shuffle*). Essa fase recupera todas saídas dos *maps* e executa um *merge-sort* para combinar todos os pares que compartilham a mesma chave, gerando então novos pares *(k_2, lista(v_2))*. Cada conjunto de registros com chave k_2 é utilizado como entrada de um processo *Reduce*. Os processos *Reduce* são executados e seus re-

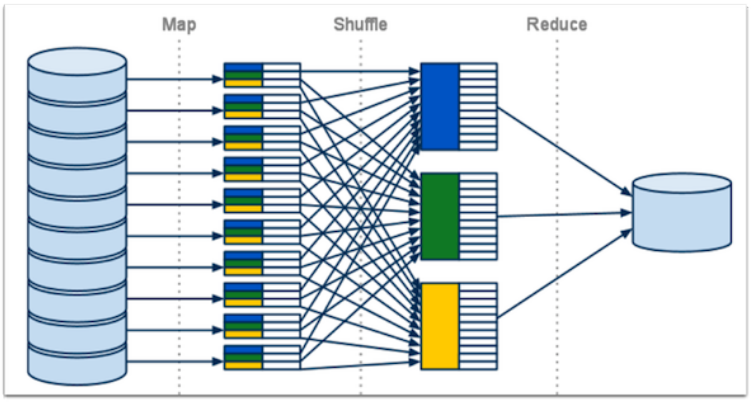

Figura 2.3: Modelo de programação *MapReduce* (GOOGLE, 2012)

sultados escritos no sistema de arquivos distribuído. A Figura 2.3 ilustra o funcionamento básico do MapReduce.

É possível, ainda, definir uma função *Combine*, intermediária entre a execução de um *Map* e de um *Reduce*, para realizar alguma operação sobre os dados antes que o *Reduce* seja executado (DEAN; GHEMAWAT, 2004). O *combiner* processa sobre os dados localmente antes de eles serem enviados aos *reducers*. Essa abordagem geralmente otimiza a execução, visto que será menor a quantidade de dados trafegados na rede e, consequentemente, menor será o trabalho do *shuffle* e do *reducer*. Porém, não são todas as aplicações que permitem o uso de um *combiner*, visto que os tipos de dados das chaves e valores precisam ser compatíveis com a saída do *mapper* e com a entrada do *reducer*.

2.6.2 Arquitetura

Como base na sua arquitetura, o *framework* MapReduce implementado pelo Hadoop possui um sistema de arquivos distribuído, ou DFS (*Distributed File System*). Ele é responsável por manter blocos dos arquivos em cada servidor e também pelas operações básicas de gerenciamento de arquivos. Para garantir segurança e de-

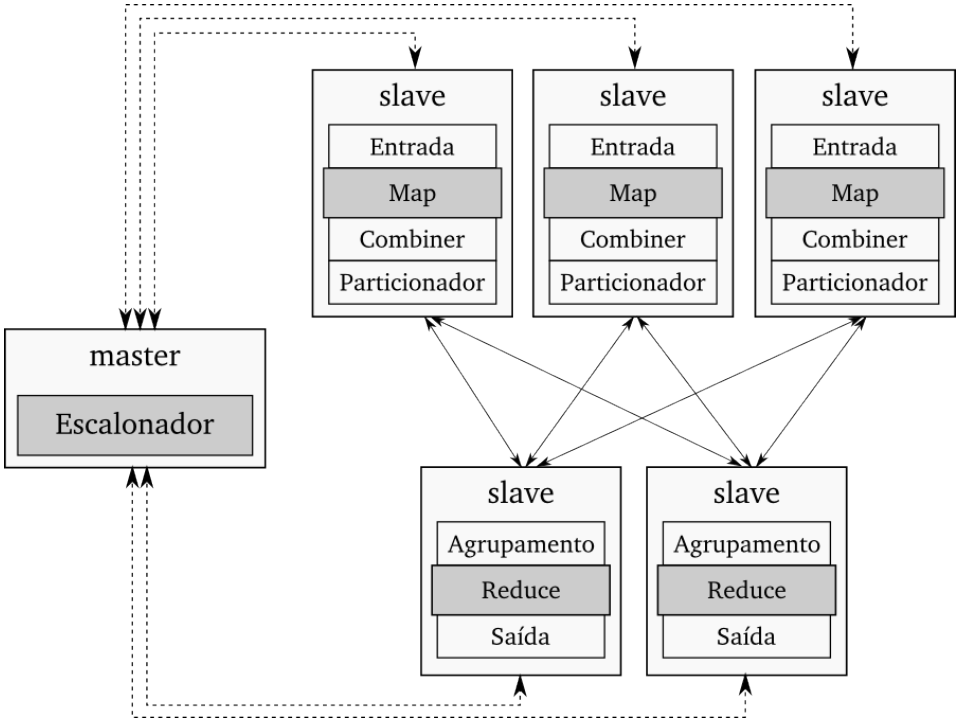

Figura 2.4: Arquitetura do *MapReduce* (LI *et al.*, 2013)

sempenho, o DFS mantém réplicas das informações em diferentes máquinas para casos de falhas no *cluster* e para melhorar o escalonamento de tarefas para as máquinas que estejam mais ociosas.

Baseado em alguma função de particionamento do DFS, os dados são divididos em tamanhos iguais e distribuídos pelas máquinas do *cluster*. Cada porção de dados é utilizada como entrada de um *mapper*. Portanto, o número de processos *Map* é igual à quantidade de blocos de entrada.

Para o funcionamento do *MapReduce*, existem dois tipos de nós: *masters* (ou mestres) e *slaves* (também chamados de *workers* ou escravos). Os *masters* são responsáveis por controlar as execuções das tarefas nos *slaves* por meio do escalonador.

Uma implementação do *MapReduce* geralmente inclui os seguintes módulos, como mostrado na Figura 2.4:

- Escalonador: é responsável por designar e controlar as tarefas de cada nó do *cluster* de acordo com estatísticas, tal como localidade dos dados e estado da rede, e também de recuperar o *cluster* de possíveis falhas. O projeto desse módulo afeta significantemente no desempenho do *MapReduce*.

- Entrada e saída: o módulo de entrada é capaz de processar diferentes formatos de entrada e dividi-la em pares chave/valor. O módulo de saída, da mesma forma, especifica os formatos de saída de cada *mapper* e *reducer* a ser escrita no DFS.

- *Map*: são tarefas que transformam um conjunto de registros de entrada em registros intermediários. Os registros de saída não precisam ser do mesmo formato da entrada, e um par de entrada pode ser mapeado para nenhum ou para vários pares de saída. A quantidade de *Maps* é definida com base no tamanho total da entrada, isto é, no número total de blocos após a divisão da entrada.

- *Reduce*: tem a função de processar um conjunto de valores intermediários que compartilham uma chave e reduzí-lo a um conjunto menor de registros. O número de processos *Reduce* pode ser definido pelo usuário, e também desabilitado, caso o processo de *Reduce* e de ordenação sejam desnecessários.

- *Combiner*: sua função é realizar uma fase semelhante a um *Reduce*, porém antes dos dados serem transmitidos pela rede. Dessa forma, ao invés de toda a saída do *Map* ser enviada, algum agrupamento é feito sobre ela e o volume de dados passado para o *Reduce* é menor. Esta fase é opcional e nem

sempre aplicável, mas quando utilizada pode otimizar consideravelmente o desempenho.

- Particionador: controla o particionamento das chaves das saídas intermediárias dos *Maps*. Geralmente a chave é utilizada para definir a partição, tipicamente por uma função *hash* sobre a chave ou valor. O número total de partições é igual ao número de tarefas Reduce para o *job*. Dessa forma, a função de particionamento deve retornar um valor inteiro no intervalo $[0, \textit{número de Reducers - 1}]$.

- Agrupamento: este módulo especifica como os dados recebidos de diferentes *mappers* deverão ser agrupados antes de serem passados para os *reducers*.

Uma das principais características do *MapReduce* é a tolerância a falhas. O processo *master* verifica periodicamente se os processos *slaves* estão ativos. Caso não houver alguma resposta, o *master* assume que o *slave* parou de funcionar e todas suas operações precisam ser descartadas e realocadas para outro nó. Tarefas *Map* precisam ser executadas novamente, pois as informações salvas foram perdidas. No caso de tarefas *Reduce* isso não é preciso, pois elas armazenam suas saídas no DFS.

A Figura 2.5 ilustra o funcionamento do *MapReduce* na execução do programa contador de frequências de palavras em um texto. A primeira etapa é recuperar os dados de entrada e fazer a divisão de acordo com o tamanho do bloco. Cada *mapper* recebe um bloco de texto e associa o valor 1 para cada palavra presente no mesmo. Após a execução de cada processo *Map*, as saídas são ordenadas e agrupadas pelo processo *shuffle* do *framework* para que cada *reducer* receba um conjunto de registros que contém a mesma chave (a palavra, neste caso). Por fim, os *reducers* operam sobre suas listas de valores e geram uma saída – no exemplo, cada *reducer* receberá a lista de ocorrências de uma palavra. O *reducer* realiza a

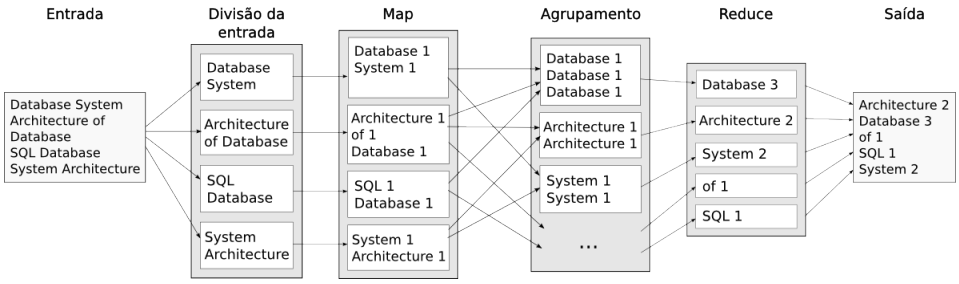

Figura 2.5: Funcionamento de um processo *MapReduce*

soma do número de ocorrências de determinada palavra no texto e salva esse valor do sistema de arquivos distribuído.

2.7 Apache Hadoop

Apache Hadoop é um *framework* de código aberto escrito em Java que suporta computação segura, escalável e distribuída. Foi projetado para escalar de uma até milhares de máquinas, cada uma oferecendo computação local e armazenamento. É capaz de oferecer um serviço de alta disponibilidade e ser executado em um *cluster* propenso a falhas (FOUNDATION, 2012).

Além da grande utilização do Hadoop por pesquisadores em universidades, atualmente ele também está presente nos projetos de grandes empresas, como Ebay, Facebook, Google, IBM, Linkedin, Twitter e Yahoo! (FOUNDATION, 2013c).

O projeto Hadoop inclui (1) *Hadoop Common*, que são os utilitários comuns que suportam os outros módulos do Hadoop; (2) *Hadoop Distributed File System* (HDFS), um sistema distribuído de arquivos que oferece acesso com boa vazão aos dados da aplicação; (3) *Hadoop YARN*, um *framework* para escalonamento de tarefas e gerenciamento de recursos de um *cluster* e (4) *Hadoop MapReduce*, que é

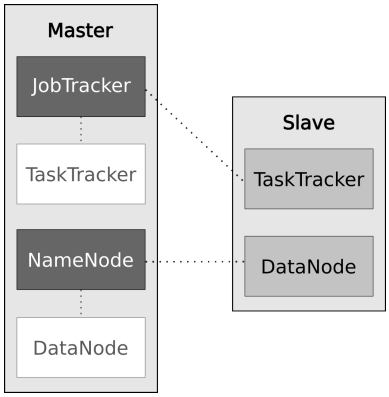

Figura 2.6: Processos do Apache Hadoop executados em cada *master* e *slave*

o sistema baseado em YARN para processamento paralelo de grandes quantidades de informação.

Para o funcionamento do Hadoop são necessários basicamente quatro processos: *NameNode* e *JobTracker* nos nós *masters*, e *DataNode* e *TaskTracker* nos nós *slaves*, conforme a Figura 2.6. O *JobTracker* inicializa os processos (ou *jobs*) e coordena suas execuções. Nos *slaves*, cada *TaskTracker* executa uma porção do *job* que foi dividido. O *NameNode* é responsável por armazenar os metadados dos blocos, por meio de um índice invertido que relaciona os blocos e os respectivos *slaves* que os contém em seu *DataNode*. Os *DataNodes* são a essência do sistema de arquivos, pois são responsáveis pelo armazenamento e recuperação dos blocos quando requisitados pelos clientes ou *NameNodes*, e também de reportar periodicamente aos *masters* a lista dos blocos que está armazenando (WHITE, 2012, p. 46).

Integrante do projeto Hadoop, o HDFS (*Hadoop Distributed File System*) é um sistema de arquivos distribuído projetado para armazenar grandes quantidades de dados (*terabytes* ou até *petabytes*), além de prover boa taxa de vazão para essas informações. Os arquivos são armazenados de forma redundante para garantir du-

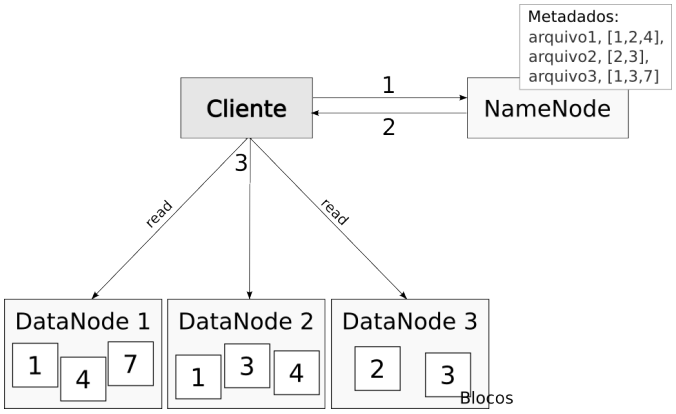

Figura 2.7: Funcionamento básico do HDFS

rabilidade em casos de falhas e disponibilidade para execuções em paralelo (INC., 2011).

A Figura 2.7 ilustra o funcionamento do HDFS. Quando há uma requisição de cliente para leitura ou escrita no HDFS, o *NameNode* verifica no seu índice quais *DataNodes* contém determinados blocos e responde ao cliente em quais nós ele poderá recuperar ou armazenar a informação (WHITE, 2012).

Outro componente do Hadoop que foi utilizado neste trabalho é o sistema de *cache* distribuído, chamado de *Distributed Cache*. Ele permite a distribuição e armazenamento local de arquivos que serão utilizados somente para leitura por outras partes da aplicação. O Hadoop copiará os arquivos necessários para o nó *slave* antes que qualquer tarefa seja executada nesse nó. A eficiência está no fato de essa cópia ser feita apenas uma vez por *job* (FOUNDATION, 2013b).

2.8 *Fuzzy-joins* e *MapReduce*

O trabalho de (VERNICA; CAREY; LI, 2010) é um dos pioneiros na área de junções por similaridade utilizando o modelo de programação *MapReduce*. Conluiu-

se nele que a utilização de *MapReduce* em junções por similaridade é uma solução eficiente para conjuntos volumosos de dados, ainda mais quando baseadas no *prefix-filtering* e *ppjoin+*, e que também possui boa taxa de *speedup* e *scaleup*. O autor utiliza uma abordagem de três estágios:

Estágio 1: Ordenação dos *tokens*

Nesta etapa são computadas as frequências dos *tokens*, isto é, o número de vezes que cada *token* aparece no documento. Em seguida, os *tokens* são ordenados de modo crescente pelas frequências, visto que a ordenação por frequência é a mais utilizada para possibilitar o *prefix filtering*. A entrada é o arquivo original com os registros para a junção de similaridade e a saída é o conjunto de *tokens* ordenados por suas frequências. Essa ordenação é utilizada para a filtragem pelo prefixo, que utiliza os *tokens* mais raros no prefixo e, com isso, parte da tabela de frequências poderá ser descartada.

Estágio 2: *Junção por similaridade*

É a etapa onde é feita a junção por similaridade propriamente dita. Registros com identificadores (RID, ou *Record ID*) semelhantes são agrupados e mapeados juntamente com o grau de similaridade entre os dois registros. A entrada é o conjunto ordenado de *tokens*, obtido da primeira etapa, e o resultado é o conjunto de RIDs que satisfazem o predicado da junção por similaridade.

Estágio 3: Recuperação dos registros

Na etapa final o objetivo é recuperar todas as informações (demais atributos) dos registros similares, pois até o momento tem-se apenas os identificadores e o grau de similaridade entre eles.

Neste trabalho esse estágio não é considerado, pois o objetivo do projeto é unicamente reconhecer os registros similares. A busca pelas demais informações

dos registros pode ser facilmente implementada através de uma outra junção utilizando seus identificadores.

3 DESENVOLVIMENTO DO TRABALHO

Para atingir o objetivo geral, primeiramente a estrutura do algoritmo proposto foi planejada, definindo o particionamento dos conjuntos, a quantidade de fases necessárias, os tipos e estruturas de dados e as definições das entradas e saídas de cada estágio. Durante o desenvolvimento do trabalho, três estratégias para a divisão dos estágios do algoritmo foram implementadas e avaliadas.

Neste trabalho foi utilizada a versão do *mpjoin* para conjuntos com peso (*w-mpjoin*), diferentemente da implementação de (VERNICA; CAREY; LI, 2010) que trabalha com o *ppjoin* sobre conjuntos sem peso.

3.1 Implementação do Algoritmo

Utilizou-se como base o algoritmo desenvolvido em (RIBEIRO; HÄRDER, 2011), na linguagem de programação Java e altamente configurável para trabalhar sobre conjuntos com e sem peso, algoritmos *ppjoin*, *ppjoin+* e *mpjoin*, e funções por similaridade *Jaccard*, *Dice* e *Cosine*. Como o objetivo deste projeto é distribuir o processamento da junção por similaridade, utilizou-se o código existente ao invés de reimplementar os algoritmos e etapas bases do problema.

Durante o desenvolvimento do trabalho, foi necessário implementar três estratégias de distribuição do algoritmo. A primeira estratégia utiliza dois *mappers* para agrupar um conjunto de tuplas de *tokens* e uma tabela de frequências e direcionar esses dados para um *reducer*. Essa abordagem mostrou-se ineficiente devido ao processamento gasto na criação da lista, principalmente quando o volume de dados é grande. Uma nova estratégia foi desenvolvida para evitar o uso de dois *mappers* no Estágio 3. Na segunda estratégia, as tuplas de *tokens* não são obtidas do Estágio 1, mas sim geradas novamente. Com isso, evita-se um *mapper* e, con-

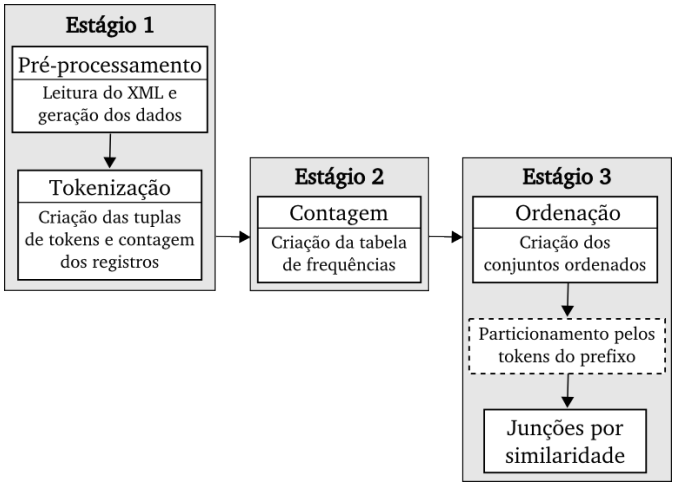

Figura 3.1: Visão geral dos estágios do algoritmo

sequentemente, uma fase, pois os conjuntos ordenados não precisarão ser criados no *reducer*. Porém, essa abordagem possui como limitação o tamanho da entrada, que geralmente é menor do que o tamanho de um bloco do HDFS, fazendo com que um único processo *mapper* crie todos os conjuntos ordenados. Como essa fase é onerosa, essa estratégia não obteve bom desempenho. Por fim, a última estratégia utiliza o *cache* distribuído do Hadoop para manter a tabela de frequências salva após o Estágio 2. Cada processo *mapper* carrega a tabela durante a inicialização do mesmo e a salva em memória, permitindo que todas execuções da função *map* utilizem o mesmo objeto salvo. Utilizando as tuplas de *tokens* como entrada do *mapper*, a entrada será dividida em diversos blocos e, então, vários mappers processarão esse estágio. Esta última estratégia obteve o melhor desempenho dentre as três, como está descrito nas próximas seções.

3.2 Estratégia 1 (base)

Como mostra a Figura 3.1, a implementação base do algoritmo é composta por 3 estágios:

- Estágio 1: Tokenização;
- Estágio 2: Criação da tabela de frequências;
- Estágio 3: Criação dos conjuntos ordenados e execução da junção por similaridade.

É importante ressaltar que na abordagem de três etapas do (VERNICA; CAREY; LI, 2010), o estágio inicial *tokeniza* a entrada e já cria a tabela de frequências. Na implementação deste trabalho, isso não é possível devido à utilização de múltiplos conjuntos, que requer a geração das tuplas de *tokens* de acordo com os atributos, e também a contagem da quantidade total de registros para o cálculo do peso de cada conjunto – a partir daqui o peso de um conjunto será referenciado como norma, que é obtida pela soma dos pesos dos *tokens* baseada na abordagem do IDF. Portanto, pode-se dizer que os Estágios 2 e 3 deste trabalho correspondem, respectivamente, às etapas 1 e 2 do trabalho do (VERNICA; CAREY; LI, 2010). Como citado anteriormente, o terceiro estágio do artigo em questão não é implementado neste trabalho.

Os dois primeiros estágios precisam ser executados uma única vez para cada conjunto de dados de entrada. No caso de alterar o *threshold*, por exemplo, pode-se executar apenas o último estágio (junção por similaridade).

3.2.1 Estágio 1: Geração das Tuplas de *Tokens*

O estágio inicial é responsável por ler o conjunto de dados extraídos do arquivo XML e utilizar o tokenizador para gerar as tuplas de *tokens* que compõem cada conjunto. Essas tuplas são escritas no HDFS pelo *mapper*. O *Reduce* deste estágio contabiliza o número total de registros e armazena esse valor no *cache* – o Estágio 3 utiliza a quantidade de registros para calcular o peso do conjunto na geração

dos conjuntos ordenados. A Figura 3.2 ilustra o primeiro estágio do algoritmo implementado.

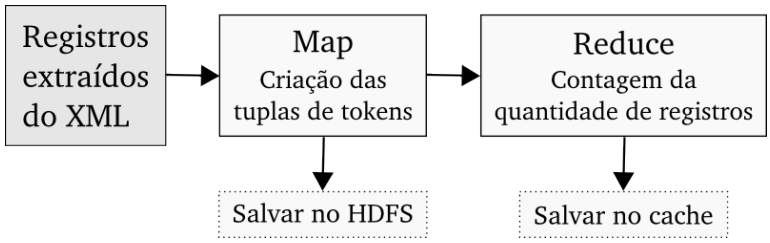

Figura 3.2: Primeiro estágio da estratégia base

Por exemplo, seja um registro extraído `$$Pearl Harbor##Michael Bay$$` e *q-grams* de tamanho 3 – existem delimitadores de tamanho 2 para garantir que toda a *string* seja tokenizada. As tuplas de *tokens* geradas são da seguinte forma: `[[1$$P, 1$PE, 1PEA, 1EAR, 1ARL, ...], [2##M, 2#MI, 2MIC, 2ICH, 2CHA, ..., 2Y$$]]`. O número que antecede cada *token* representa o atributo que contém esse *token*, para que o cálculo de intersecção na abordagem sobre múltiplos conjuntos considere apenas *tokens* que pertencem ao mesmo atributo.

3.2.2 Estágio 2: Construção da Tabela de Frequências

O Estágio 2 é composto por 2 fases. A primeira delas tem a função de contar a frequência de cada *token* no conjunto de tuplas de *tokens*, obtido do Estágio 1. Cada *mapper* percorre os *tokens* de cada tupla e atribui o valor 1 para esse *token*. A etapa de ordenação e agrupamento se encarrega de entregar a cada *reducer* um conjunto de ocorrências de cada *token*, permitindo então que o *reducer* conte a frequência desse *token* no conjunto de dados. A saída, portanto, é uma tabela de frequências parcial que contém um *token* como chave e a frequência desse *token* no conjunto de dados como valor (Figura 3.3).

A Fase 2 cria a tabela de frequências, isto é, uma tabela *hash* formada por relações entre *tokens* e frequências. Utiliza as saídas da primeira fase para gerar

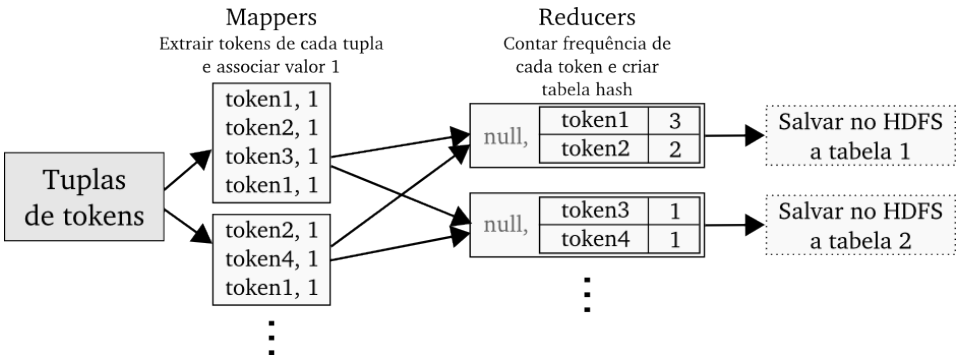

Figura 3.3: Primeira fase do estágio de construção da tabela de frequências (estratégia base)

um objeto que possui a tabela de frequências e o número de objetos contidos na mesma. O *Reduce* dessa fase faz o agrupamento de todos os mapas de frequências gerados em cada execução da fase anterior – para isso utiliza-se uma chave em comum, como *null* neste caso, para que todas as tabelas sejam agrupadas pela fase *shuffle* do MapReduce. A saída é um objeto que reúne todas as tabelas de frequências intermediárias em uma única tabela (Figura 3.4).

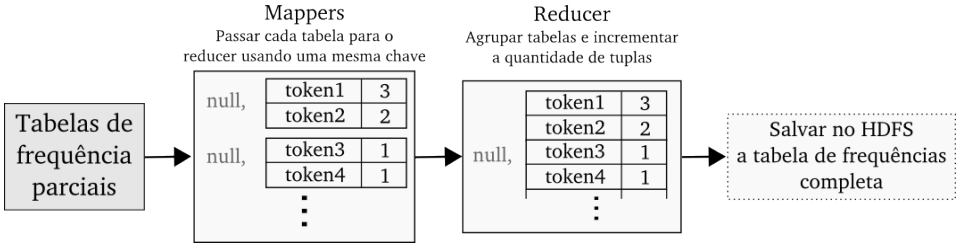

Figura 3.4: Segunda fase do estágio de construção da tabela de frequências (estratégia base)

3.2.3 Estágio 3: Junções por Similaridade

Esse estágio é responsável pela junção por similaridade propriamente dita. Na Estratégia 1, esse estágio contém 3 fases. A primeira fase tem a função de gerar os conjuntos ordenados utilizando as saídas dos dois estágios anteriores: conjunto de *tokens* originais e a tabela de frequências (saídas dos Estágios 1 e 2, respecti-

vamente). Como observa-se na Figura 3.5, é preciso dois *mappers* para agrupar uma tabela de frequências com um conjunto de tuplas de *tokens* – no Hadoop é possível utilizar mais de um *mapper* através da classe *MultipleInputs*. A chave de cada objeto de entrada do *mapper* é um *hash*, uma codificação da tupla de *token* emitida na saída do Estágio 1. Cada *mapper* desempenha um papel de particionador, emitindo o identificador de um *reducer* destino como chave e um objeto serializado como valor – a serialização é necessária porque os *reducers* precisam receber valores do mesmo tipo.

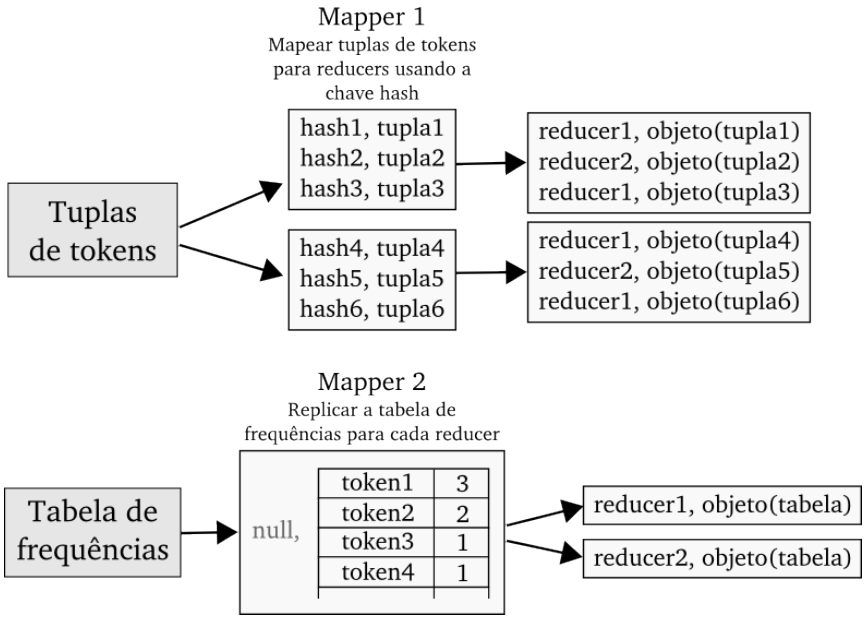

Figura 3.5: Dois *mappers* constituem a primeira fase do Estágio 3 (Estratégia base)

Portanto, cada processo *Reduce* receberá uma lista de objetos serializados, que contém uma instância da tabela de frequências e um bloco de tuplas de *tokens* (Figura 3.6). Utilizando o número de conjuntos, salvo em *cache* pelo estágio inicial, o algoritmo gera os conjuntos ordenados para que o algoritmo de junção por similaridade possa explorar a filtragem pelo prefixo, visto que os conjuntos são ordenados utilizando suas normas.

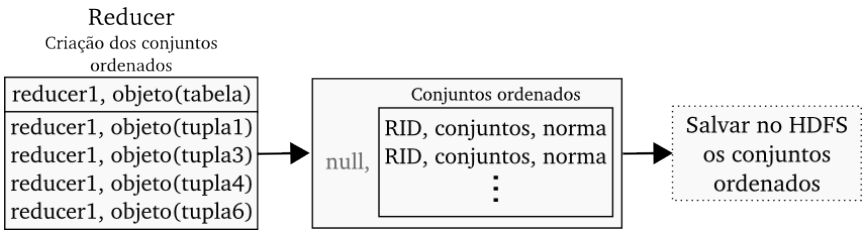

Figura 3.6: Os conjuntos ordenados são gerados no Reduce da primeira fase do estágio de junção por similaridade (estratégia base)

Conforme ilustra a Figura 3.7, a Fase 2 realiza as junções por similaridade sobre os conjuntos ordenados gerados na fase anterior. Os *mappers* tem a função de calcular os pesos de cada conjunto baseando-se na tabela de frequências e na quantidade total de registros obtida do Estágio 1. Em seguida, cada *mapper* cria conjuntos ordenados e calcula os tamanhos dos seus prefixos. É nesta fase que ocorre o particionamento dos conjuntos. Cada prefixo é percorrido e o conjunto ordenado é replicado uma vez para cada *token* do prefixo. Dessa forma, cada *reducer* receberá conjuntos ordenados que possuem pelo menos um *token* em comum nos seus prefixos.

É na etapa de *reducer* que aplica-se o algoritmo *mpjoin* sobre conjuntos com peso. Cada *reducer* processa conjuntos ordenados de registros que compartilham um *token* nos seus prefixos, utilizando o conceito de *Prefix Filtering*. Como os conjuntos estão ordenados por uma mesma ordem global, a probabilidade desses registros serem similares é considerável. A saída dessa fase é um conjunto de resultados das junções por similaridade.

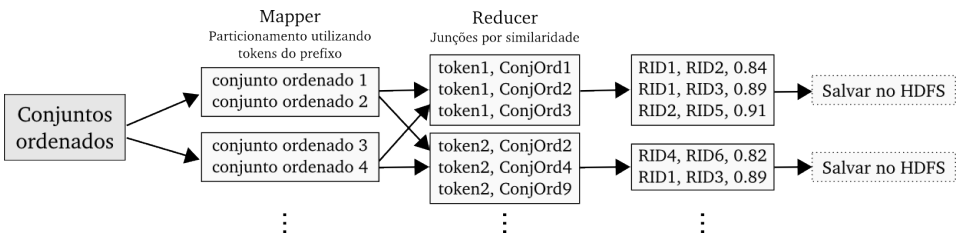

Figura 3.7: Segunda fase do estágio de junções por similaridade (estratégia base)

Por fim, a Fase 3 faz o agrupamento dos resultados gerados e remove os resultados duplicados – que existirão quando o mesmo conjunto ordenado é enviado para *reducers* distintos, devido ao particionamento que replica os mesmos conjuntos ordenados quando dois registros possuem mais de um *token* em comum nos seus prefixos. Portanto, tem como entrada o conjunto de resultados da junção (Fase 2) e resulta em um único conjunto de resultados com valores semelhantes aos obtidos na execução do algoritmo sequencial – por isso é feito por apenas um *reducer*. Esta fase está ilustrada pela Figura 3.8.

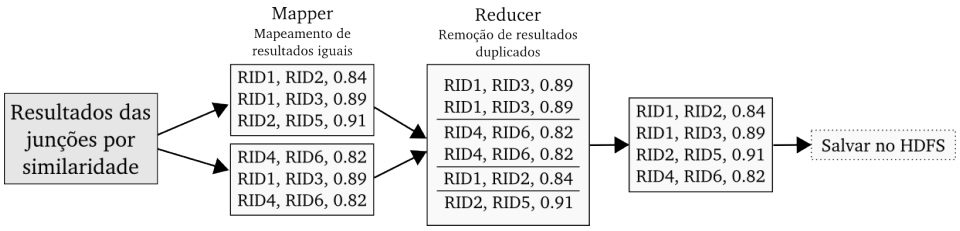

Figura 3.8: A última fase remove os resultados duplicados das junções por similaridade

A utilização de dois *mappers* para agrupar a tabela de frequências e as tuplas de *tokens* não se mostrou eficiente devido ao custo de replicação da tabela para cada *reducer* e iterar sobre as tuplas para utilizar uma única instância dessa tabela. Quando executado sobre grande volume de dados, essa abordagem não obteve o desempenho esperado – conforme descrito no próximo capítulo. Por isso, modificações foram feitas nas fases sobrecarregadas para permitir maior exploração do *MapReduce*.

A Figura 3.9 ilustra em alto nível o funcionamento da estratégia de 6 fases descrita nesta seção.

A seguir serão apresentadas outras duas estratégias para distribuição do algoritmo.

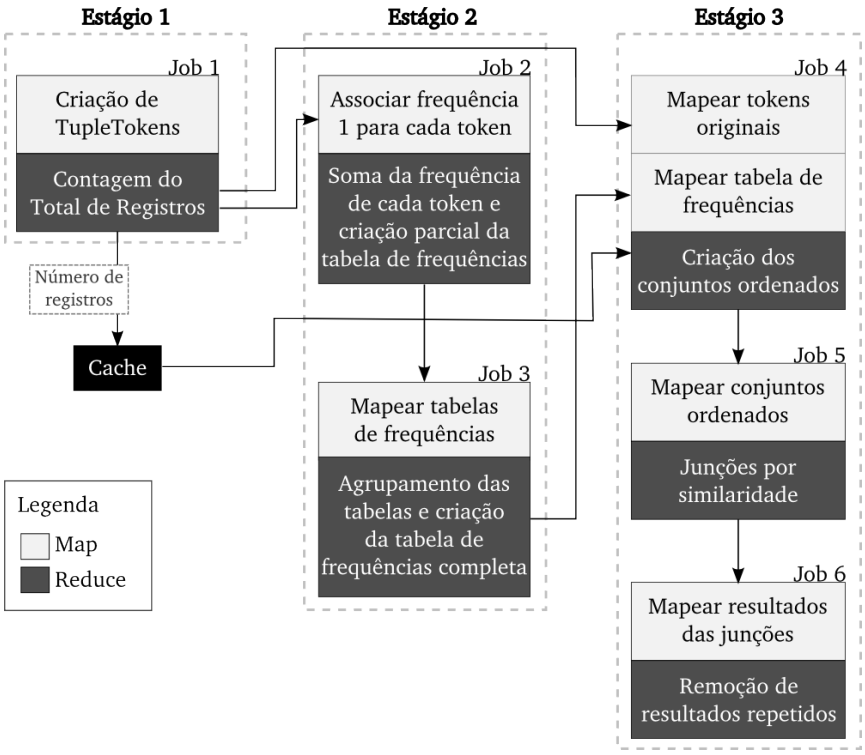

Figura 3.9: Etapas da estratégia 1 (base)

3.3 Estratégia 2: Geração das Tuplas de *Tokens* Novamente no Estágio 3

Visto que a estratégia anterior é limitada pela quantidade de memória da máquina – devido ao agrupamento das tuplas de *tokens* em uma lista –, o objetivo é utilizar os *mappers* para gerar os conjuntos ordenados, e não os *reducers*.

Para isso, uma alteração possível consiste em gerar novamente os conjuntos de *tokens* na primeira fase do Estágio 3, e não carregar do *HDFS* o resultado do Estágio 1. Portanto, nessa alteração evita-se o *mapper* que recebe as tuplas de *tokens* (primeiro *mapper* do *Job* 4 na Figura 3.9 ou o *Mapper 1* da Figura 3.5).

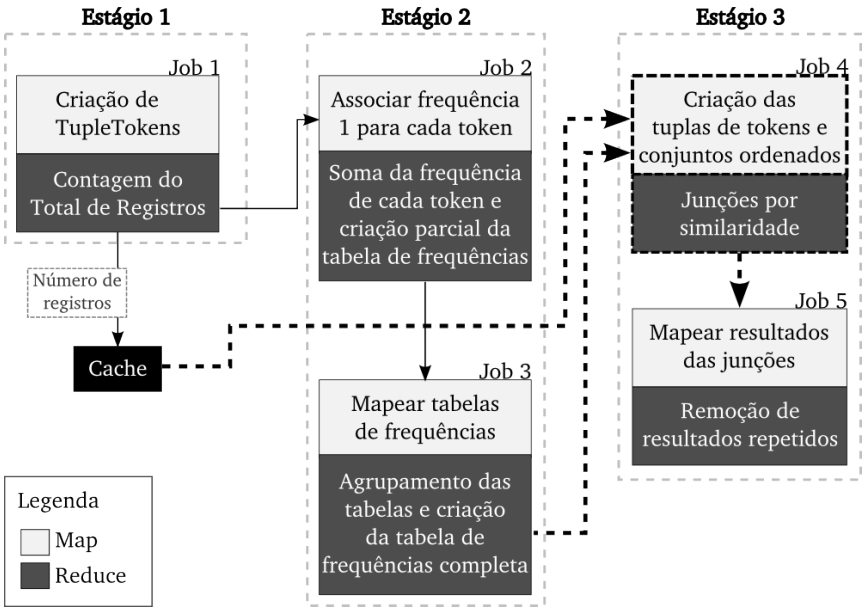

Figura 3.10: Etapas da Estratégia 2, que gera os conjuntos de *tokens* também no terceiro estágio

Com essa melhoria, uma fase do Estágio 3 é eliminada, pois o início desse estágio não receberá duas entradas (conjunto de *tokens* e tabela de frequências) e então os conjuntos ordenados poderão ser criados no *Map*. Portanto, as junções por similaridade são feitas no *Reduce* e os *jobs* 4 e 5 da estratégia base são unidos (Figura 3.10).

Pode-se afirmar que o tamanho da tabela de frequências é mínimo se comparado ao conjunto de tuplas de *tokens*. Apesar de ter como vantagem a diminuição do tráfego de dados na rede, apenas um *Map* será responsável por gerar os conjuntos ordenados, pois a tabela de frequências geralmente é menor do que o tamanho de um bloco do HDFS.

3.4 Estratégia 3: Armazenamento em *Cache* da Tabela de Frequências

Como na estratégia anterior apenas um *mapper* é responsável por gerar os conjuntos ordenados, foi preciso elaborar uma maneira mais eficiente de dividir a execução do algoritmo pelo *MapReduce*.

Essa alteração consiste em salvar em *cache* a tabela de frequências gerada no fim do Estágio 2. Utiliza-se o *Distributed Cache* do Hadoop (FOUNDATION, 2013b) nesta etapa. Como o tamanho da tabela de frequências é na ordem de *megabytes*, é possível mantê-la em memória. Para isso, o *cache* é lido uma única vez durante a inicialização do *mapper* – pelo método *setup* da classe *hadoop.mapreduce.Mapper* – e a tabela de frequências é mantida em memória para ser utilizada por cada execução do método *map*.

A principal diferença para a estratégia anterior é que a entrada do Estágio 3 será as tuplas de *tokens* (Estágio 1) e não a tabela de frequências (Estágio 2). Como o volume de dados gerado pelo primeiro estágio é muito maior do que a tabela de frequências, vários *mappers* serão utilizados para gerar os conjuntos ordenados, o que garante maior escalabilidade nesta fase. A Figura 3.11 destaca as alterações feitas a partir do algoritmo base.

3.5 Particionamento dos Conjuntos

Para que cada máquina do *cluster* efetue uma operação, é preciso particionar os dados de entrada. Devido à sobrecarga gerada (disco, rede e processamento), não seria eficiente enviar todos os conjuntos para todos os nós, mesmo se cada nó operasse apenas sobre uma parte desses conjuntos.

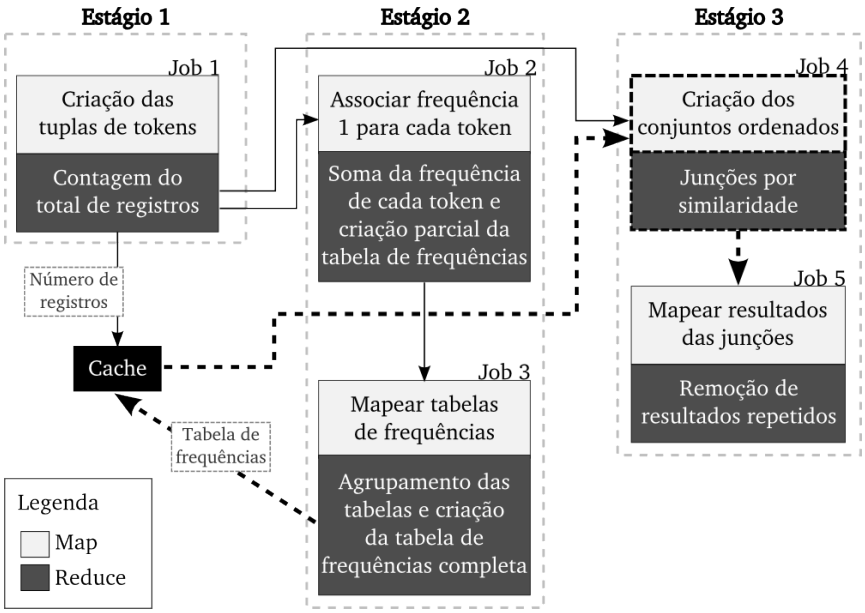

Figura 3.11: Etapas do algoritmo da Estratégia 3, que armazena em *cache* a tabela de frequências

O particionamento por *tokens* segue a abordagem de (VERNICA; CAREY; LI, 2010). O conjunto é replicado uma vez por *token* no prefixo e cada réplica é direcionada para algum processo *Reduce*. Assim, o *Reduce* receberá candidatos que compartilham um mesmo *token* no prefixo, tendo então alguma probabilidade de serem similares. Neste caso, um mesmo conjunto é replicado n vezes, onde n é o número de *tokens* que constituem o prefixo.

Como ilustrado pela Figura 3.12, cada *token* dos prefixos passa por uma função de particionamento. Esta função obtém o valor *hash* do *token* e calcula o módulo desse valor pela quantidade de *reducers* configurada no Hadoop, de forma que o particionador retornará um número entre zero e o total de *reducers* existentes. Então, dois *tokens* iguais terão o mesmo código *hash* e serão enviados ao mesmo *reducer*.

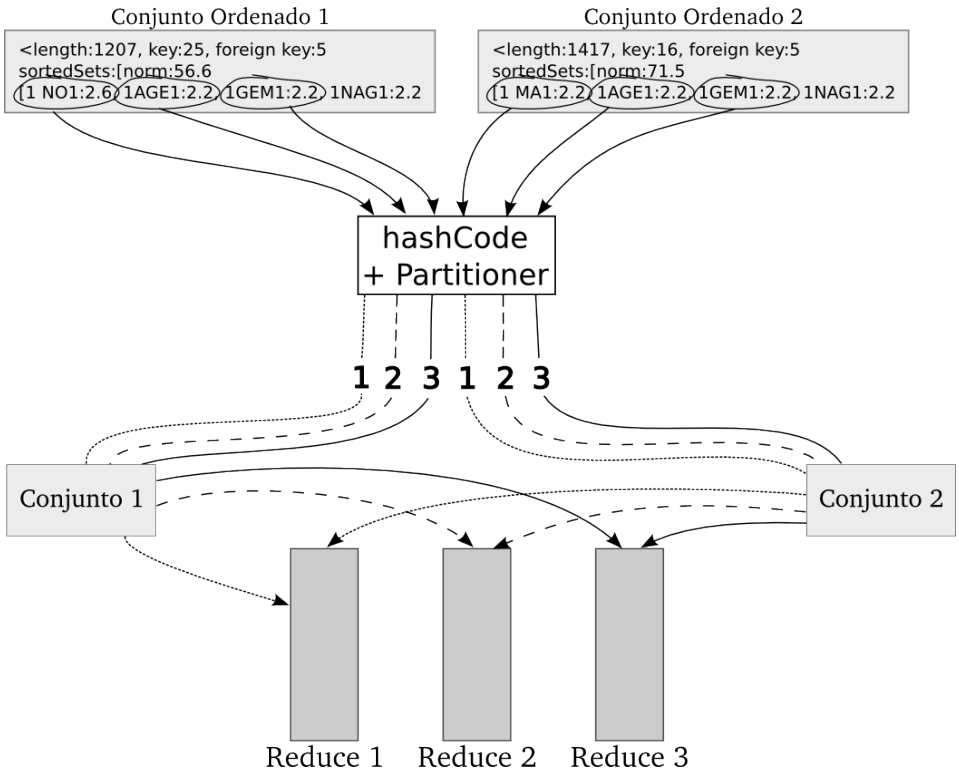

Figura 3.12: Particionamento dos conjuntos baseado nos *tokens* do prefixo

4 EXPERIMENTOS

Os testes foram realizados no ambiente de execução do grupo de pesquisas em Banco de Dados no Departamento de Ciência da Computação da UFLA. Problemas de desempenho surgiram e foram avaliados para que adaptações no algoritmo base fossem implementadas. Após refeitos os experimentos, os resultados permitiram a elaboração de gráficos comparativos para discussões e conclusões. Finalmente, o código do algoritmo passou por refatorações e melhorias para facilitar a execução e auxiliar futuros trabalhos da área.

Neste capítulo serão descritos os resultados e conclusões a respeito de cada estratégia implementada.

4.1 Ambiente de Testes

4.1.1 Hardware

Para a realização dos testes de desempenho dos algoritmos, foi utilizado um *cluster* composto por uma máquina *master* e oito *slaves*. Três dos *slaves* são servidores HP Proliant DL120 G7 E3-1240 3,1 GHz (4 núcleos físicos e 4 núcleos simulados), com 8MB cache e 8GB de memória RAM. Os outros cinco *slaves* são máquinas com processador Intel Core i3 3450M (4 núcleos físicos), 6MB cache 2,8 GHz, 4GB de memória. O computador *master* é um AMD Phenom II X4 B95, com 6GB de RAM. Devido ao processador inferior aos demais, ele foi utilizado para gerenciamento do *cluster*. Neste trabalho é considerado que cada nó corresponde a uma máquina do *cluster*.

4.1.2 *Dataset*

O algoritmo e os testes foram executados sobre a biblioteca DBLP (*DataBase systems and Logic Programming*), que provê informações bibliográficas da maioria dos jornais e periódicos da área de Ciência da Computação. O banco de dados é obtido em um arquivo XML (*eXtensible Markup Language*) que contém todos os registros de informações bibliográficas. Esse arquivo inicialmente é interpretado e adaptado para uma forma mais adequada ao algoritmo desenvolvido, removendo informações irrelevantes para os testes. A base de dados utilizada contém mais de 2 milhões de registros em um arquivo XML de tamanho 1,2GB. Para obtenção da base de dados para testes, um número determinado de registros são extraídos do XML e duplicatas são geradas por meio de modificações textuais aleatórias, tal como deleção, inserção e substituição de caracteres, de acordo com os parâmetros configurados na entrada do algoritmo.

Todos os testes foram feitos com *threshold* 0,80 para o título e 0,70 para o autor de cada conferência (registros do tipo *inproceedings* no DBLP). O primeiro valor é um dos mais utilizados na literatura (VERNICA; CAREY; LI, 2010). O *threshold* 0,70, por outro lado, foi utilizado para demonstrar a abordagem sobre múltiplos conjuntos, na qual cada atributo é associado a um *threshold*.

4.1.3 Hadoop e Configurações

A versão do Apache Hadoop utilizada neste trabalho é a 1.1.2, disponível no dia 15 de fevereiro de 2013. O número de *reducers* foi definido através da fórmula $1.75 * N * M$, onde N é o número de nós do *cluster* e M é a quantidade máxima de tarefas executadas simultaneamente em uma máquina. O valor 1.75 faz com que as máquinas mais rápidas (os servidores, neste caso) finalizem suas execuções

e iniciem novas tarefas antes das demais[1]. Na Tabela 4.1 estão as quantidades de *reducers* utilizadas neste trabalho para cada variação no número de máquinas do *cluster*. O valor de *M* foi considerado como a média de tarefas paralelas em cada máquina, visto que os servidores com 8 núcleos de processamento foram configurados para executar 7 tarefas simultaneamente, enquanto os servidores com 4 núcleos de processamento executaram 3 tarefas em paralelo.

Tabela 4.1: Relação entre tamanho do *cluster*, núcleos de processamento, quantidade de *reducers* e volume de dados (utilizado no cálculo do *scale-out*)

Nós	Threads	*Reducers*	Registros
1	8	5	2.000.000
2	16	11	4.000.000
4	24	21	8.000.000
6	32	31	12.000.000
8	44	42	16.000.000

O Apêndice A contém um tutorial de instalação e configuração inicial do Apache Hadoop em um *cluster* de computadores.

4.2 Algoritmo Sequencial

O algoritmo sequencial de (RIBEIRO; HÄRDER, 2011) foi executado para servir de parâmetro nas comparações com o algoritmo produzido neste trabalho para calcular o *speedup*.

Na execução sobre maiores volumes de dados, o algoritmo sequencial precisa utilizar sua versão secundária, denominada de *Out of Core* pelos autores (RIBEIRO; HÄRDER, 2011, p. 16). Essa implementação é baseada no algoritmo *Nested-loop join*. Ele percorre os dados de entrada múltiplas vezes e cada bloco

[1] http://wiki.apache.org/hadoop/HowManyMapsAndReduces

Tabela 4.2: Tempos de execução do algoritmo sequencial em um dos servidores (8 núcleos de processamento)

Registros	Pré-processamento (minutos)	Junções por similaridade (minutos)
500,000	1.10	1.28
1,000,000	2.28	4.58
2,000,000	5.03	18.26
4,000,000	12.96	71.66
8,000,000	28.71	316.78
12,000,000	44.78	755.58
16,000,000	60.21	1,375.43

lido é indexado e verificado da mesma forma que a versão do algoritmo que opera em memória. O algoritmo é otimizado nesta parte com a exploração dos tamanhos dos conjuntos, podendo parar a etapa de verificação de um bloco quando um conjunto de tamanho maior do que um limite é indexado. Utiliza também o algoritmo de ordenação externa chamado TPMMS (*Two-Phase Multiway Merge Sort*), que é uma variante do *Merge-Sort* utilizado para dados residentes no disco e principalmente quando o conjunto de dados é maior do que a quantidade de memória disponível. O algoritmo sequencial não executa operações em paralelo.

A Tabela 4.2 contém os tempos gastos pelo algoritmo sequencial nas etapas de pré-processamento (criação das tuplas de *tokens* e de conjuntos ordenados) e de junções por similaridade, executado em um dos servidores com melhor configuração.

4.3 *Speedup* e *Scale-out*

Um termo comum na área de algoritmos paralelos é o *speedup*, que se refere a quanto um algoritmo paralelo é mais rápido do que sua versão sequencial. Seu

valor é dado pela equação 4.1, onde $T(n)$ é o tempo necessário para completar a execução utilizando n núcleos de processamento.

$$speedup(n) = \frac{T(1)}{T(n)} \tag{4.1}$$

Na área de computação paralela e distribuída existem duas formas de escalonamento: vertical (*scale up*) e horizontal (*scale out*). A primeira refere-se ao aumento de recursos ao ambiente já existente, tal como memória e capacidade de armazenamento – poucas máquinas, *hardware* robusto. Por outro lado, o escalonamento horizontal significa acrescentar componentes ao *cluster* – diversas máquinas menos robustas. Geralmente o *scale-out* é mais custoso, pois impacta na complexidade da solução e pode requerer mudanças no projeto.

4.4 Experimentos da Estratégia 1

O algoritmo de 6 fases (Seção 3.2) mostrou-se limitado pela quantidade de memória da máquina. Nessa abordagem, a geração dos conjuntos ordenados é feita no *Reduce*, e nele os conjuntos de *tokens* e a tabela de frequências precisam ser agrupadas para que a geração seja feita corretamente. É por esse motivo que existem dois *mappers*, um para mapear os *tokens* e outro para as tabelas de frequências. Como não seria eficiente repassar a tabela de frequências uma vez para cada *token*, então ela é passada uma vez para cada *Reduce* e os *tokens* são agrupados para serem iterados em uma única passada sobre os dados. Devido a esse agrupamento em uma lista, o limite de espaço da pilha do Java (*Java heap space*) – configurado para 2GB neste trabalho – é atingido e o processo não finaliza com sucesso. Nos experimentos isso ocorreu após execução sobre o conjunto de 2 milhões de registros.

4.5 Experimentos da Estratégia 2

A modificação no experimento base, que está descrita na Seção 3.3, também não se mostrou eficiente. O problema dessa abordagem é que ela possui como entrada a tabela de frequências resultante do Estágio 2, que geralmente é menor do que o tamanho de um bloco configurado no Hadoop (128MB neste trabalho). Portanto, como já foi dito que o número de *mappers* varia de acordo com a quantidade de blocos da entrada, nesse caso existirá apenas um bloco e, consequentemente, uma única fase de *Map* será encarregada de fazer toda a geração dos conjuntos ordenados.

É notável que ter apenas um *Map* gerando os objetos mais custosos do algoritmo não é eficiente. Por exemplo, na execução com 16 milhões de registros, a saída do *Map* que gera os conjuntos ordenados é de aproximadamente 60GB de dados.

4.6 Experimentos da Estratégia 3

O modelo descrito na Seção 3.4 obteve o melhor desempenho, principalmente devido ao tamanho da tabela de frequências ser na ordem de *megabytes* e então passível de ser mantida em memória. Essa tabela, resultante do segundo estágio, é lida do HDFS uma única vez durante a inicialização de cada *mapper* e mantida em memória para ser usada por todas chamadas do mesmo. Como essa é a fase mais dispendiosa do algoritmo, o ganho de desempenho foi claramente percebido.

No gráfico da Figura 4.1 pode-se observar que o Estágio 3 (junções por similaridade) é o mais dispendioso a partir da execução com 12 milhões de registros utilizando dois servidores (cada um com 8 núcleos de processamento). Na comparação com o algoritmo sequencial, considerou-se os tempos gastos nesse estágio

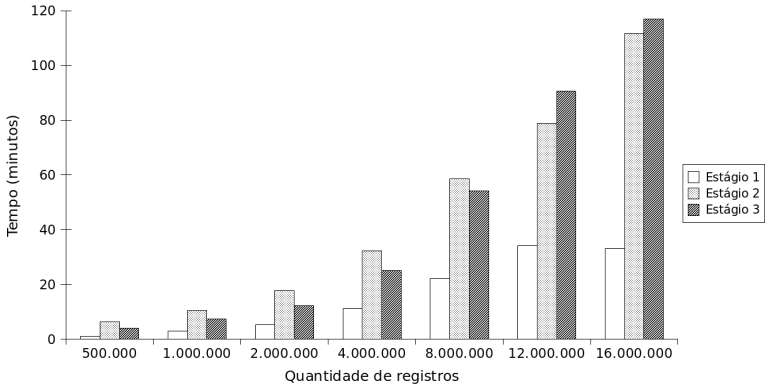

Figura 4.1: Tempo em minutos gasto por cada estágio na execução com 2 nós

do algoritmo paralelo e os tempos de execução das junções por similaridade propriamente ditas do algoritmo sequencial. Isso porque o algoritmo sequencial gera, em uma única rotina, a tabela de frequências e os conjuntos ordenados antes de executar a junção.

Os testes mostraram que o ganho de desempenho realmente existe quando a entrada possui mais de 4 milhões de registros. Abaixo dessa quantidade o algoritmo sequencial é capaz de executar utilizando a memória principal e não é necessária a versão *Out of Core*. Portanto, nesses casos o *speedup* é baixo, devido ao *overhead* do Hadoop para inicializar os *jobs* e o sistema de arquivos, além da ordenação, particionamento e transmissão dos dados pela rede.

A Figura 4.2 mostra o *speedup* do algoritmo na execução sobre 12 milhões de registros. Interessante notar que a execução com um único nó do *cluster* (um dos servidores com 8 núcleos de processamento) obteve melhor desempenho do que a versão sequencial executada localmente no mesmo servidor. Isso aconteceu devido à utilização do modelo de programação MapReduce, por meio do particionamento do volume de dados e do processamento, e também da execução em paralelo de diferentes *mappers* e *reducers* – como mostra a Tabela 4.1, para uma única máquina *slave* no *cluster* foram utilizados 5 *reducers*.

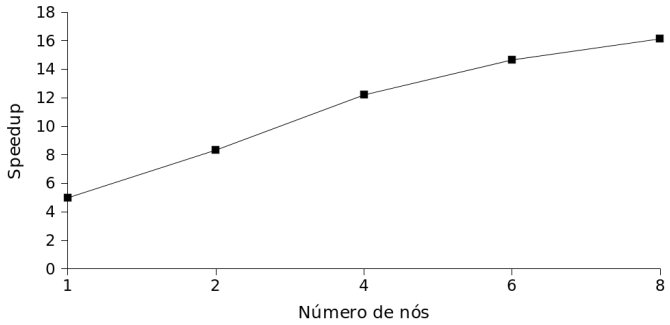

Figura 4.2: Gráfico do *Speedup*

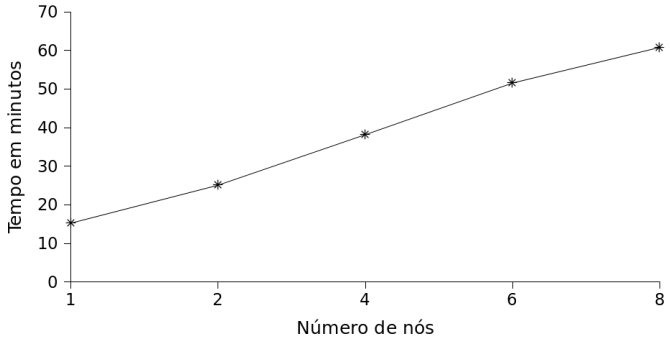

Figura 4.3: Gráfico do *Scale-out*

O gráfico da Figura 4.3 demonstra o desempenho do algoritmo quando foram acrescentados mais nós ao *cluster* à medida que aumenta-se, no mesmo fator (Tabela 4.1), a quantidade de registros de entrada.

4.7 Conclusão dos Experimentos

Como dito anteriormente, o Hadoop é capaz de escalar de um único nó até milhares deles. Neste trabalho, apesar do *cluster* ser composto por poucas máquinas, algumas conclusões foram obtidas a respeito do escalonamento horizontal do algoritmo.

Foi possível concluir que o algoritmo desenvolvido escalona bem horizontalmente (*scale out*) e que o *speedup* melhorou à medida que aumentou-se o tamanho da entrada. O grande aumento do *speedup* acontece porque o algoritmo sequencial utiliza demasiadamente de operações em disco quando o volume de dados é grande.

O estágio que exige mais processamento é o terceiro (criação dos conjuntos ordenados e junções por similaridade). Isso acontece devido ao fato de que a saída do Estágio 1 (geração das tuplas de *tokens*) tem tamanho considerável (*gigabytes*), implicando em um grande número de *Maps*. Além disso, maior volume de dados pode implicar em muitos *tokens* comuns destinados a um mesmo *reducer*, que então necessita de maior processamento durante a junção por similaridade. Porém, como observado nos gráficos, o tempo gasto nesse estágio diminui à medida que aumenta-se o número de máquinas no *cluster*, pois cada máquina terá menos *Maps* e *Reduces* para processar.

4.8 Observações e Dificuldades Encontradas

A primeira observação é que a infraestrutura de testes limitou o desempenho do algoritmo. O gráfico da Figura 4.4 mostra como a pouca quantidade de nós do *cluster* influenciou na execução do algoritmo. Isso é observado quando se tem muito mais processos *Map* do que número de nós, causando tempo de espera entre execuções que estão direcionadas para um mesmo nó. Em outros trabalhos da área de programação distribuída, os *clusters* são compostos por dezenas e até centenas de máquinas. Por isso, esse é um fator considerável para obtenção de conclusões mais precisas acerca do desempenho do algoritmo desenvolvido.

Além da quantidade de máquinas, outra observação é que a diferença de *hardware* também afetou o desempenho do processo. Conforme já descrito neste

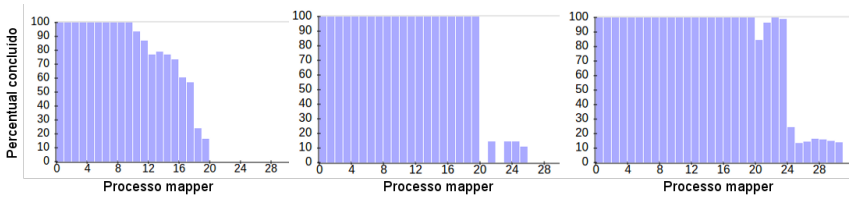

Figura 4.4: Gráfico de execução dos processos Map do Experimento 2

trabalho, o ambiente de testes foi composto por computadores pessoais (do laboratório de pesquisas) juntamente com servidores *rack*. A Figura 4.5 mostra como o processo *Reduce* 3 demandou mais tempo do que os demais devido à menor capacidade de processamento da máquina.

Figura 4.5: Gráfico de execução dos processos Reduce do Experimento 2

Ainda quanto à estrutura do *cluster*, notou-se que – ao menos com os computadores disponíveis para os experimentos – não é eficiente utilizar uma mesma máquina como *master* e *slave* ao mesmo tempo. A alteração para manter uma máquina exclusiva para ser o servidor *master* diminuiu significativamente o tempo gasto na execução do algoritmo, ainda que essa máquina fosse um computador de mesa e não um servidor.

Por fim, é interessante notar que o Hadoop também faz paralelismo a nível de *threads*, executando simultaneamente em um nó mais de um *Map* ou *Reduce* quando configurado para tal. A Figura 4.6 é uma captura de tela que mostra o estado de cada núcleo de processamento durante a execução do algoritmo em um

dos servidores configurado para processar 7 processos *Map* em paralelo[2]. Pode-se notar que a *thread* 7 está com menor carga de trabalho, devido à configuração feita para que um núcleo de processamento não seja utilizado para tarefas MapReduce, permitindo que ele execute outras tarefas sem sobrecarregar a máquina.

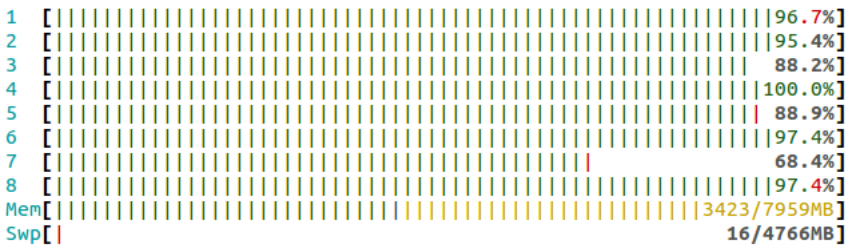

Figura 4.6: Execução paralela a nível de *threads* em um nó do cluster

[2]O parâmetro de configuração utilizado no Hadoop é *mapred.tasktracker.map.tasks.maximum*.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho mostra que o modelo de programação *MapReduce* é eficiente para processamento de grandes volumes de dados, especificamente para junções por similaridade sobre múltiplos conjuntos com peso.

Das três estratégias apresentadas, uma delas mostrou-se altamente escalável, mas que ainda pode ter melhorias no desempenho. Por exemplo, utilizar o módulo *combiner* do *MapReduce* pode diminuir pelo menos um *job* do algoritmo – além de reduzir o tráfego de dados na rede e a ser processado por cada *reducer*.

Um ponto de partida para dar continuidade neste trabalho é explorar o conceito de múltiplos conjuntos. Por exemplo, pode-se planejar um modelo de particionamento dos conjuntos baseado nos seus atributos. Outra sugestão de particionamento é pelos tamanhos dos conjuntos para utilizar o conceito de que registros similares possuem tamanhos similares (conforme descrito na Seção 2.3), e comparar com o particionamento pelos *tokens* do prefixo implementado neste trabalho. Outro trabalho é implementar a versão do algoritmo sem pesos e comparar o desempenho com a implementação com pesos. Por fim, técnicas e trabalhos da área de Inteligência Artificial podem ser integrados para, por exemplo, decidir em tempo de execução qual é o melhor conjunto para serem feitas as otimizações.

Quanto ao código implementado, uma das sugestões é efetuar testes alterando os tipos dos dados transmitidos pela rede. Neste trabalho foram utilizados tipos estendidos das classes existentes no algoritmo do estado da arte, como foi descrito nos capítulos anteriores, e também a serialização de objetos na forma binária. Pode-se adaptar o código para trabalhar com dados somente do tipo binário e, em seguida, somente com dados do tipo texto, por meio de serialização em JSON (*JavaScript Object Notation*), por exemplo. Se o volume de dados a ser armazenado no sistema de arquivos e transmitido pela rede diminuir, pode-se ter melhoria de

desempenho no algoritmo – desde que o tempo gasto no processo de deserialização seja menor do que esse ganho.

Outra melhoria sugerida para os próximos trabalhos no Hadoop é testar o paralelismo a nível de *threads* utilizando a implementação alternativa de *Mapper*, que é provido pela classe *MultithreadedMapper* (WHITE, 2012, p. 236).

REFERÊNCIAS BIBLIOGRÁFICAS

BAYARDO, R. J.; MA, Y.; SRIKANT, R. Scaling up all pairs similarity search. In: *Proceedings of the 16th international conference on World Wide Web*. New York, NY, USA: ACM, 2007. (WWW '07), p. 131–140. ISBN 978-1-59593-654-7. Disponível em: <http://doi.acm.org/10.1145/1242572.1242591>.

CHAUDHURI, S.; GANTI, V.; KAUSHIK, R. A primitive operator for similarity joins in data cleaning. In: *Proceedings of the 22nd International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2006. (ICDE '06), p. 5–. ISBN 0-7695-2570-9. Disponível em: <http://dx.doi.org/10.1109/ICDE.2006.9>.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA: USENIX Association, 2004. (OSDI'04), p. 10–10. Disponível em: <http://dl.acm.org/citation.cfm?id=1251254% -.1251264>.

FOUNDATION, A. S. *Cluster Setup*. 2013. Disponível em: <http://hadoop-.apache.org/docs/r1.1.2/cluster_setup.html>.

FOUNDATION, A. S. *DistributedCache*. 2013. Disponível em: <http://hadoop-.apache.org/docs/stable/api/org/apache/hadoop/filecache/DistributedCache-.html>.

FOUNDATION, A. S. *Powered By Hadoop*. 2013. Disponível em: <http://wiki-.apache.org/hadoop/PoweredBy>.

FOUNDATION, A. S. *Single Node Setup*. 2013. Disponível em: <http://hadoop-.apache.org/docs/stable/single_node_setup.html>.

FOUNDATION, T. A. S. *Apache Hadoop*. 2012. Disponível em: <http://hadoop-.apache.org/>.

GOOGLE. *MapReduce Python Overview*. 2012. Disponível em: <https:/-/developers.google.com/appengine/docs/python/dataprocessing>.

INC., Y. *Hadoop Tutorial from Yahoo!* 2011. Disponível em: <http://developer-.yahoo.com/hadoop/tutorial/index.html>.

LAM, H. T.; DUNG, D. V.; PEREGO, R.; SILVESTRI, F. An incremental prefix filtering approach for the all pairs similarity search problem. In: *Proceedings of the 2010 12th International Asia-Pacific Web Conference*. Washington, DC, USA: IEEE Computer Society, 2010. (APWEB '10), p. 188–194. ISBN 978-0-7695-4012-2. Disponível em: <http://dx.doi.org/10.1109/APWeb.2010.30>.

LI, F.; OOI, B. C.; OZSU, M. T.; WU, S. Distributed data management using mapreduce (a ser publicado em 2013). In: *ACM Computing Surveys*. ACM, 2013. Disponível em: <http://www.comp.nus.edu.sg/~ooibc/mrsurvey.pdf>.

RIBEIRO, L. A. *A framework for XML similarity joins*. 1-220 p. Tese (Doutorado), University of Kaiserslautern, 2010.

RIBEIRO, L. A.; HÄRDER, T. Generalizing prefix filtering to improve set similarity joins. *Inf. Syst.*, Elsevier Science Ltd., Oxford, UK, UK, v. 36, n. 1, p. 62–78, mar. 2011. ISSN 0306-4379. Disponível em: <http://dx.doi.org/10.1016-/j.is.2010% -.07.003>.

RIJSBERGEN, C. J. V. *Information Retrieval*. 2nd. ed. Newton, MA, USA: Butterworth-Heinemann, 1979. ISBN 0408709294.

ROBERTSON, S. E.; JONES, K. S. Relevance weighting of search terms. *Journal of the American Society for Information Science*, Wiley Subscription Services, Inc., A Wiley Company, v. 27, n. 3, p. 129–146, 1976. ISSN 1097-4571. Disponível em: <http://dx.doi.org/10.1002/asi.4630270302>.

SARAWAGI, S.; KIRPAL, A. Efficient set joins on similarity predicates. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004. (SIGMOD '04), p. 743–754. ISBN 1-58113-859-8. Disponível em: <http://doi.acm.org/10.1145/1007568.1007652>.

SPERTUS, E.; SAHAMI, M.; BUYUKKOKTEN, O. Evaluating similarity measures: a large-scale study in the orkut social network. 2005.

UKKONEN, E. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci.*, Elsevier Science Publishers Ltd., Essex, UK, v. 92, n. 1, p. 191–211, jan. 1992. ISSN 0304-3975. Disponível em: <http://dx.doi.org/10.1016/0304-3975(92)90143-4>.

VERNICA, R.; CAREY, M. J.; LI, C. Efficient parallel set-similarity joins using mapreduce. ACM, New York, NY, USA, p. 495–506, 2010. Disponível em: <http://doi.acm.org/10.1145/1807167.1807222>.

WANG, C.; WANG, J.; LIN, X.; WANG, W.; WANG, H.; LI, H.; TIAN, W.; XU, J.; LI, R. Mapdupreducer: detecting near duplicates over massive datasets. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. New York, NY, USA: ACM, 2010. (SIGMOD '10), p. 1119–1122. ISBN 978-1-4503-0032-2. Disponível em: <http://doi.acm.org/10-.1145/1807167.1807296>.

WANG, J.; LI, G.; FENG, J. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2012. (SIGMOD '12), p. 85–96. ISBN 978-1-4503-1247-9. Disponível em: <http://doi.acm.org/10.1145/2213836.2213847>.

WHITE, T. *Hadoop: the definitive guide*. 3rd. ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2012. ISBN 1449311520, 978-1-4493-1152-0.

XIAO, C.; WANG, W.; LIN, X.; YU, J. X. Efficient similarity joins for near duplicate detection. In: *Proceedings of the 17th international conference on World Wide Web*. New York, NY, USA: ACM, 2008. (WWW '08), p. 131–140. ISBN 978-1-60558-085-2. Disponível em: <http://doi.acm.org/10.1145/1367497-.1367516>.

A INSTALAÇÃO E UTILIZAÇÃO DO HADOOP

Neste apêndice está um passo-a-passo da instalação e configuração do Apache Hadoop em um *cluster*, com base em (WHITE, 2012; FOUNDATION, 2013d; FOUNDATION, 2013a).

A.1 Pré-requisitos

O Hadoop possui alguns requisitos para ser executado: (1) GNU/Linux para ambiente de produção, pois o suporte a Win32 é apenas para desenvolvimento; (2) Java 1.6 ou superior e (3) cliente e servidor SSH (*Secure Shell*).

É recomendado criar um grupo e um usuário dedicado ao uso do Hadoop, principalmente por questões de segurança - visto que será utilizado SSH para interação entre os nós.

```
# addgroup hadoop
# adduser --ingroup hadoop hadoop
```

A.2 Download e Instalação

O próximo passo é fazer o *download* do Hadoop. Todas as versões disponíveis estão listadas no site oficial[1]. Para acompanhar este trabalho, obtenha a versão 1.1.2. A sequência de comandos para *download* e descompactação está abaixo.

```
$ wget http://www.us.apache.org/dist/hadoop/common/hadoop-1.1.2/
    hadoop-1.1.2.tar.gz
```

[1] http://hadoop.apache.org/releases.html

```
$ tar zxvf hadoop-1.1.2.tar.gz
# cp -r hadoop-1.1.2 /usr/local/
# chown -R hadoop:hadoop /usr/local/hadoop-1.1.2
# ln -s /usr/local/hadoop-1.1.2 /usr/local/hadoop
```

Acrescentar na variável $*PATH* do sistema o diretório dos executáveis do Hadoop, através do arquivo *.bashrc* no diretório *home* do usuário *hadoop*.

```
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin
```

Os *scripts* de controle (*start-all.sh* e *stop-all.sh*, por exemplo) do Hadoop utilizam SSH para acessar os demais nós do *cluster*. Para isso é preciso habilitar o *login* sem senha para os usuários *hadoop* nas demais máquinas.

```
$ ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

A chave privada é especificada como argumento da opção $-f$, e a chave pública estará localizada no mesmo diretório porém com a extensão *pub*.

Em seguida é preciso compartilhar a chave pública para as demais máquinas. Uma alternativa é utilizar NFS (*Network File System*). Outra maneira é copiar a chave para as máquinas (conhecendo seus *hostnames* ou IPs) utilizando o próprio SSH.

```
$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@slave1
```

A partir de agora deverá ser possível acessar cada máquina via SSH sem digitar a senha. Para testar, utiliza-se os comandos *ssh master* e *ssh slave1*.

A.3 Configuração do Cluster

Os arquivos de configuração são mantidos na pasta *conf* no diretório de instalação do Hadoop. Seguindo este passo-a-passo, o diretório é */usr/local/hadoop/conf* e os principais arquivos são *core-size.xml*, *hdfs-site.xml*, *mapred-site.xml*, *hadoop-env.sh* e *slaves*. Abaixo seguem exemplos básicos (apenas as propriedades essenciais) dos arquivos de configuração.

```
1  <?xml version="1.0"?>
2  <configuration>
3    <property>
4      <name>fs.default.name</name>
5      <value>hdfs://namenode/</value>
6      <final>true</final>
7    </property>
8  </configuration>
```

Código A.1: Exemplo de arquivo core-site.xml

```
1  <?xml version="1.0"?>
2  <configuration>
3    <property>
4      <name>dfs.name.dir</name>
5      <value>/disk1/hdfs/name,/remote/hdfs/name</value>
6      <final>true</final>
7    </property>
8    <property>
9      <name>dfs.data.dir</name>
10     <value>/disk1/hdfs/data,/disk2/hdfs/data</value>
11     <final>true</final>
12   </property>
13   <property>
14     <name>fs.checkpoint.dir</name>
```

```
15      <value>/disk1/hdfs/namesecondary,/disk2/hdfs/namesecondary</
            value>
16      <final>true</final>
17    </property>
18  </configuration>
```

Código A.2: Exemplo de arquivo hdfs-site.xml

```
1  <?xml version="1.0"?>
2  <configuration>
3    <property>
4      <name>mapred.job.tracker</name>
5      <value>jobtracker:8021</value>
6      <final>true</final>
7    </property>
8    <property>
9      <name>mapred.local.dir</name>
10     <value>/disk1/mapred/local,/disk2/mapred/local</value>
11     <final>true</final>
12   </property>
13   <property>
14     <name>mapred.system.dir</name>
15     <value>/tmp/hadoop/mapred/system</value>
16     <final>true</final>
17   </property>
18 </configuration>
```

Código A.3: Exemplo de arquivo mapred-site.xml

A.4 Inicialização e Desligamento do Cluster

Após a instalação é preciso formatar o sistema de arquivos através do seguinte comando (utilizando o usuário *hadoop*):

```
$ hadoop dfs namenode -format
```

Para inicializar o *cluster* Hadoop é preciso rodar os processos de HDFS e *MapReduce* através dos *scripts* de controle.

Primeiramente, inicia-se o *HDFS*. O *script* inicia um *NameNode* na máquina *master* e um *DataNodes* em cada máquina *slave* por uma conexão *SSH* nos *hostnames* ou IPs listados no arquivo *conf/slaves*. Por fim, inicia um *NameNode* secundário em cada máquina listada no arquivo *conf/masters*.

```
$ start-dfs.sh
```

Em seguida, iniciar o *MapReduce* na máquina *master* (a que rodará o *JobTracker*). Da mesma forma, esse comando inicializa um processo *TaskTracker* em cada *slave*. Diferente do *script* anterior, neste não se utiliza o arquivo *conf/masters*.

```
$ start-mapred.sh
```

O desligamento dos processos do Hadoop é importante para não corromper arquivos, *jobs* e *logs*. A sequência de comandos é baseada na inicialização, sendo primeiro o *shutdown* no HDFS e depois no *MapReduce*.

```
$ stop-dfs.sh
$ stop-mapred.sh
```

Ambos os comandos farão o trabalho de desligar os devidos serviços nos nós *slaves* configurados.

A.5 Comandos

Tendo o Hadoop devidamente instalado, é possível interagir com o HDFS através de comandos. Algumas ações básicas de manipulação de arquivos são:

```
$ hadoop dfs -ls /
$ hadoop dfs -mkdir /user
$ hadoop dfs -copyFromLocal input.txt /user/input.txt
$ hadoop dfs -copyToLocal /user/output.txt output.txt
$ hadoop dfs -rm /user/output.txt
```

A.6 Interface Web

O servidor *web* do HDFS é capaz de monitorar o estado do HDFS e também de navegar pelo sistema de arquivos. Através de um navegador *web*, basta acessar o endereço e porta configurados no arquivo *conf/hadoop-site.xml* do Hadoop. A figura A.1 é uma captura de tela da interface *web* padrão.

A.7 Execução de um Job no Hadoop

Antes de executar um *job* é preciso inicializar os serviços do pacote Hadoop, que foram mostrados no capítulo sobre a instalação do Hadoop. Em seguida, possuindo um arquivo *jar*, basta executá-lo no Hadoop:

```
$ hadoop jar arquivo.jar NomeClassePrincipal [par metros]
```

O pacote instalado também contém um arquivo de exemplos, com programas como *terasort* e *wordcount*. Para executar o *wordcount* (contador de frequência de

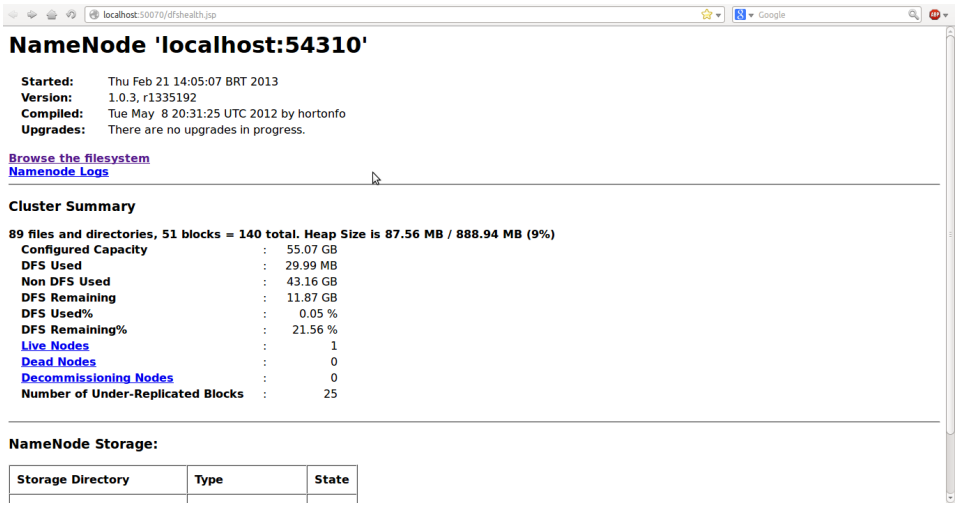

Figura A.1: Interface web do Hadoop Distributed File System

palavras em um texto), primeiramente cria-se um arquivo com o texto que será a entrada do programa e, em seguida executa-se o programa passando o arquivo de entrada e o diretório de saída, que será gravado no HDFS. Por fim, o resultado da execução pode ser visualizado pela interface *web* ou pelo último comando a seguir.

```
$ hadoop dfs -copyFromLocal texto.txt input.txt
$ hadoop jar hadoop-examples-1.1.2.jar wordcount input.txt output
$ hadoop dfs -ls output/
$ hadoop dfs -cat output/part-r-00000
```

A.8 Exemplo de Código: Contador de Frequências

O programa de contador de palavras é um dos mais utilizados para explicar o funcionamento do *MapReduce* e do Hadoop. O código a seguir, retirado da Wiki do Apache Hadoop[2], é simples e ajudará no entendimento do *MapReduce* no Hadoop. As linhas de importação foram omitidas.

[2]http://wiki.apache.org/hadoop/WordCount

O estágio de Map é feito pela classe no código A.4. Sua função é percorrer o arquivo de entrada e atribuir o valor 1 para cada palavra lida. Os tipos utilizados pelo Hadoop serão detalhados adiante, mas é importante ressaltar a utilização do tipo *IntWritable* para salvar o processamento no HDFS.

```
1  public static class Map extends Mapper<LongWritable, Text, Text,
       IntWritable> {
2      private final static IntWritable one = new IntWritable(1);
3      private Text word = new Text();
4
5      public void map(LongWritable key, Text value, Context context)
           throws IOException, InterruptedException {
6          String line = value.toString();
7          StringTokenizer tokenizer = new StringTokenizer(line);
8          while (tokenizer.hasMoreTokens()) {
9              word.set(tokenizer.nextToken());
10             context.write(word, one);
11         }
12     }
13 }
```

Código A.4: Método Map do programa contador de frequência

O processo de Reduce é responsável por agrupar todas palavras iguais e somar os valores associados a cada uma.

```
1  public static class Reduce extends Reducer<Text, IntWritable, Text
       , IntWritable> {
2      public void reduce(Text key, Iterable<IntWritable> values,
           Context context)
3        throws IOException, InterruptedException {
4          int sum = 0;
5          for (IntWritable val : values) {
6              sum += val.get();
```

```
7           }
8           context.write(key, new IntWritable(sum));
9       }
10  }
```

Código A.5: Método Reduce do programa contador de frequência

Nota-se que ambas as classes extendem da biblioteca do Hadoop com as assinaturas dos métodos na forma:

$$< TipoChaveEntrada, TipoValorEntrada, TipoChaveSada, TipoValorSada >.$$

No Map, a chave de entrada é do tipo *LongWritable* devido ao processamento inicial do Hadoop sobre o texto, que associa cada palavra (nesse caso) a um valor, geralmente um *Hash* da *string*. O valor é do tipo *Text* que é basicamente uma *string* que pode ser manipulada no HDFS.

Por outro lado, no Reduce, a chave de entrada é do tipo *Text* – o mesmo da chave de saída do Map. Como o objetivo é agrupar as palavras semelhantes, a chave é a palavra e o valor é a frequência associada (e não o contrário).

O programa principal configura o *Job*, especifica os tipos e formato dos dados, a quantidade de processos de Reduce, os locais de entrada e saída e outros valores que forem necessários.

```
1   public class WordCount {
2     public static void main(String[] args) throws Exception {
3       Configuration conf = new Configuration();
4       Job job = new Job(conf, "wordcount");
5
6       job.setOutputKeyClass(Text.class);
7       job.setOutputValueClass(IntWritable.class);
8
9       job.setMapperClass(Map.class);
```

```
10      job.setCombinerClass(Reduce.class);
11      job.setReducerClass(Reduce.class);
12
13      job.setInputFormatClass(TextInputFormat.class);
14      job.setOutputFormatClass(TextOutputFormat.class);
15
16      FileInputFormat.addInputPath(job, new Path(args[0]));
17      FileOutputFormat.setOutputPath(job, new Path(args[1]));
18
19      job.waitForCompletion(true);
20    }
21  }
```

Código A.6: Classe principal do programa contador de frequência

No exemplo acima, pode-se utilizar o próprio método de Reduce como *Combine* – A configuração está na linha 10 da classe principal. Dessa forma, após cada Map será feita uma contagem intermediária das frequências das palavras, enviando ao Reduce uma palavra já com o seu número de aparições naquele nó.

Por exemplo, se não utilizada a função de combinação, um Map terá como saída:

```
<Olá, 1>
<Mundo, 1>
<Tiau, 1>
<Mundo, 1>
```

Mas se utilizada o *Combine*, a saída desse mesmo Map será:

```
<Olá, 1>
<Mundo, 2>
```

```
<Tiau, 1>
```

Quando possível, a utilização do *Combine* é recomendada. Porém, como nem todos algoritmos permitem essa abordagem, é preciso ter cautela no uso desse recurso.

Visto que a quantidade de Maps varia de acordo com o tamanho da entrada, a utilização da função *Combine* é bastante eficiente. Suponha que existam 1000 processos Map, cada um com algumas ocorrências da palavra “Mundo”. Se não utilizada a função *Combine*, o Reduce receberá todas as ocorrências em uma única chamada. Supondo que a média de ocorrências dessa palavra em cada Map é 5, o Reducer terá 5000 palavras “Mundo” para iterar. Quando se utiliza o combinador, o Reduce receberá 1000 registros, cada um com a sumarização intermediária, diminuindo consideravelmente a quantidade de dados para processar e trafegar na rede.