UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

DIEGO JUNIOR DO CARMO OLIVEIRA

Junções por Similaridade com Expressões Complexas em Ambientes Distribuídos

Goiânia
2018

sistema de bibliotecas ufg

PRPG
PRÓ-REITORIA DE PÓS-GRADUAÇÃO

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[X] Dissertação** **[ ] Tese**

2. Identificação da Tese ou Dissertação: Junções por Similaridade com Expressões Complexas em Ambientes Distribuídos

Nome completo do autor: Diego Junior do Carmo Oliveira

Título do trabalho: Junções por Similaridade com Expressões Complexas em Ambientes Distribuídos

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Diego C Oliveira

Assinatura do(a) autor(a)[2]

Ciente e de acordo:

Leonardo A Rocha

Assinatura do(a) orientador(a)[2]

---

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

Casos de embargo:
- Solicitação de registro de patente
- Submissão de artigo em revista científica
- Publicação como capítulo de livro
- Publicação da dissertação/tese em livro

[2] A assinatura deve ser escaneada.

DIEGO JUNIOR DO CARMO OLIVEIRA

Junções por Similaridade com Expressões Complexas em Ambientes Distribuídos

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Leonardo Andrade Ribeiro

Goiânia
2018

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Junior do Carmo Oliveira, Diego
Junções por Similaridade com Expressões Complexas em Ambientes Distribuídos [manuscrito] / Diego Junior do Carmo Oliveira. - 2018.
LXI, 61 f.

Orientador: Prof. Dr. Leonardo Andrade Ribeiro.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2018.
Bibliografia.
Inclui algoritmos, lista de figuras, lista de tabelas.

1. junção por similaridade. 2. sistemas distribuídos. 3. apache spark. 4. big data. I. Andrade Ribeiro, Leonardo, orient. II. Título.

CDU 004

MINISTÉRIO DA EDUCAÇÃO

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UFG

ATA Nº 06/2018

ATA DA SESSÃO DE JULGAMENTO DA DISSERTAÇÃO DE MESTRADO DE DIEGO JUNIOR DO CARMO OLIVEIRA

Aos trinta e um dias do mês de agosto de dois mil e dezoito, às catorze horas, na sala 150 do Instituto de Informática da Universidade Federal de Goiás, Campus Samambaia, reuniu-se a banca examinadora designada na forma regimental pela Coordenação do Curso para julgar a dissertação de mestrado intitulada **"Junções por Similaridade com Expressões Complexas em Ambientes Distribuídos"**, apresentada pelo aluno Diego Junior do Carmo Oliveira como parte dos requisitos necessários à obtenção do grau de Mestre em Ciência da Computação, área de concentração Ciência da Computação. A banca examinadora foi presidida pelo orientador do trabalho de dissertação, Professor Doutor Leonardo Andrade Ribeiro (INF/UFG), tendo como membros os Professores Doutores Wellington Santos Martins (INF/UFG) e Ahmed Ali Abdalla Esmin (DCC/UFLA). O professor Ahmed Ali Abdalla Esmin participou à distância por webconferência. Aberta a sessão, o candidato expôs seu trabalho. Em seguida, o aluno foi arguido pelos membros da banca e:

(X) tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **aprovação** do candidato, sem restrições.

( ) não tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **reprovação** do candidato.

Os trabalhos foram encerrados às 16:20 horas. Nos termos do Regulamento Geral dos Cursos de Pós-Graduação desta Universidade, lavrou-se a presente ata que, lida e julgada conforme, segue assinada pelos membros da banca examinadora.

Prof. Dr. Leonardo Andrade Ribeiro Leonardo A Ribeiro

Prof. Dr. Wellington Santos Martins

Prof. Dr. Ahmed Ali Abdalla Esmin A. A.

Diego Junior do Carmo Oliveira

Graduado em Engenharia da Computação pela Pontifícia Universidade Católica de Goiás com ênfase em Administração e Gerência de Redes. Possui experiência nas áreas de gerenciamento de redes empresariais, análise e desenvolvimento de sistemas e educação. Durante a graduação foi monitor bolsista das disciplinas de sistemas digitais e programação de computadores.
Lattes: http://lattes.cnpq.br/1430917350375617
LinkedIn: https://www.linkedin.com/in/diego-junior-a9594033

À minha família. Mãe, seu amor incondicional foi o que possibilitou que eu chegasse até aqui. Pai, obrigado por ser meu exemplo de honra, caráter e dignidade.

Agradecimentos

Agradeço primeira à minha minha família, a minha mãe Suzana, meu pai Junior e a minha irmã Isabela. Obrigado por serem meus alicerces e sempre estarem torcendo pelo meu sucesso.

Ao meu orientador, Prof. Dr. Leonardo Andrade Ribeiro, sou grato pela oportunidade, paciência e pelos conhecimentos transmitidos.

Aos meus colegas de mestrado, que se tornaram grandes amigos durante essa jornada, Dyego Almeida, Priscila Lima, Hugo Ribeiro, Felipe Borges, Lucas Pacífico, Alan Cortes e Josué Queiroz, agradeço pela força e pela dedicação sempre que precisei.

À todos os professores e funcionários da UFG, em especial aos do Instituto de Informática.

À CAPES pelo apoio financeiro.

Resumo

Oliveira, D.J.C. **Junções por Similaridade com Expressões Complexas em Ambientes Distribuídos**. Goiânia, 2018. 60p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Um problema recorrente que degrada a qualidade das informações em banco de dados é a presença de duplicatas, isto é, múltiplas representações de uma mesma entidade do mundo real. Apesar de ser computacionalmente oneroso, para realizar a identificação de duplicatas é fundamental o emprego operações de similaridade. Além disso, os dados atuais são tipicamente compostos por diferentes atributos, cada um destes contendo um tipo distinto de informação. A aplicação de expressões de similaridade complexas é importante neste contexto uma vez que permitem considerar a importância de cada atributo na avaliação da similaridade. No entanto, em virtude da grande quantidade de dados presentes em aplicações *Big Data*, fez-se necessário realizar o processamento destas operações em ambientes de programação paralelo ou distribuído. Visando solucionar estes problemas de grande relevância para as organizações, este trabalho propõe uma nova estratégia de processamento para identificação de duplicatas em dados textuais utilizando junções por similaridade com expressões complexas em um ambiente distribuído.

Palavras–chave

junção por similaridade, sistemas distribuídos, apache spark, big data

Abstract

Oliveira, D.J.C. **Set Similarity Joins with Complex Expressions on Distributed Platforms**. Goiânia, 2018. 60p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

A recurrent problem that degrades the quality of the information in databases is the presence of duplicates, i.e., multiple representations of the same real-world entity. Despite being computationally expensive, the use of similarity operations is fundamental to identify duplicates. Furthermore, real-world data is typically composed of different attributes and each attribute represents a distinct type of information. The application of complex similarity expressions is important in this context because they allow considering the importance of each attribute in the similarity evaluation. However, due to a large amount of data present in Big Data applications, it has become crucial to perform these operations in parallel and distributed processing environments. In order to solve such problems of great relevance to organizations, this work proposes a novel strategy to identify duplicates in textual data by using similarity joins with complex expressions in a distributed environment.

Keywords

similarity joins, distributed platforms, apache spark, big data

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

CAPÍTULO 1

Introdução

A medida que nos tornamos mais dependentes de dispositivos eletrônicos no nosso dia a dia e que aumenta o número de pessoas conectadas a redes de computadores, também cresce significativamente a quantidade de dados gerados e utilizados por sistemas computacionais. A transformação de grandes volumes de dados em informações que possam ser úteis para organizações é o que motivou o surgimento de conceitos como Big Data [13]. Nesta área, quantidades relevantes de dados provenientes de diferentes fontes passam por uma série de etapas de processamento para serem usados, por exemplo, em sistemas de suporte a decisão ou *marketing* de relacionamento.

Para que projetos na área de Big Data obtenham sucesso é fundamental que os dados a serem analisados sejam fidedignos em relação às reais entidades por eles representadas. Um problema conhecido que degrada a qualidade de um conjunto de dados é a existência de múltiplas representações de uma mesma entidade do mundo real, as quais damos o nome de duplicatas. Estas podem levar a diversos inconvenientes como geração incorreta de relatórios gerenciais, estatísticas infladas, reenvio de correspondência, cobrança em duplicidade, entre outros.

O exemplo ilustrado na Figura 1.1 representa o banco de dados de uma unidade acadêmica. Em um primeiro cadastro as informações sobre o nome e endereço do discente foram inseridas corretamente. Posteriormente, em um outro sistema, o mesmo discente foi cadastrado com um erro de grafia em seu nome e com uma abreviação em seu endereço. Em um caso hipotético, se essas bases de dados fossem integradas e sobre elas realizadas algum tipo de consulta a presença desta informação duplicada poderia levar a um erro em algum relatório gerencial.

Duplicatas podem ser causadas por erros de digitação, abreviações ou diferentes conversões de escrita. Este fator dificulta a identificação uma vez que as mesmas não são totalmente idênticas entre si e, dessa forma, simples comparações byte-a-byte não seriam suficientes. Em casos como este o conceito de similaridade poderia ser empregado baseando-se na premissa de que duplicatas são mais similares entre si do que não duplicatas. Operações de junção por similaridade são fundamentais neste contexto pois retornam todos os pares de registros de uma coleção cuja semelhança esteja acima de

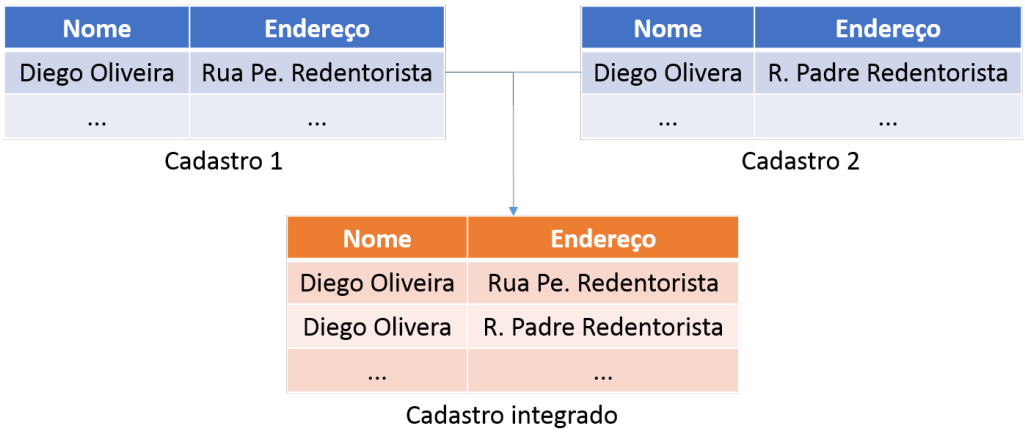


Figura 1.1: Exemplo de duplicatas em uma base de dados integrada.

um limiar específico. Para poder quantificar a similaridade entre registros uma função de similaridade é utilizada no predicado da junção. Junções por similaridade podem ser empregada em diversos contextos diferentes, dentre as quais podemos destacar: integração de dados [9]; identificação de plágio [31]; limpeza de dados [10] e linkagem de registros [30].

Um tipo popular de função de similaridade é a baseada em conjuntos que, após a conversão dos dados de entrada em conjuntos, a similaridade entre dois objetos é calculada a partir da intersecção de seus conjuntos. Este trabalho é focado no emprego de junções por similaridade sobre dados textuais. Optou-se por utilizar funções de similaridade baseadas em conjuntos em virtude de sua boa escalabilidade, versatilidade e eficiência [18].

1.1 Justificativa

Os algoritmos do estado da arte, em sua maioria, consideram cada entidade possuindo apenas um atributo para o cálculo de similaridade. Em outros casos é realizada a concatenação de múltiplos atributos para a uma posterior análise. Acontece que objetos do mundo real são tipicamente compostos por múltiplos atributos, cada um desses possuindo importância diferente para o cálculo da similaridade. Diversos fatores podem ser utilizados para avaliar a importância de um atributo, mas em geral um atributo que identifica um registro de forma mais seletiva é considerado mais importante, por exemplo, em uma coleção de registros compostos por dois atributos que representam, respectivamente, o nome e a cidade de nascimento de uma pessoa, o atributo nome seria mais importante visto que a quantidade de pessoas que nascem na mesma cidade é bem maior que a quantidade de pessoas homônimas.

Pessoa ID	RID	Nome	Logradouro	Cidade
1	1	João Ávila	Av. Pedro Paulo de Souza	Goiânia
	2	J. Ávila	Av. Pedro Paulo de Souza	Goiânia
2	3	Pedro Paulo	Av. Esperança	Goiânia
3	4	Pedro Paulo	Av. João Naves Ávila	Uberlândia

Tabela 1.1: Registros com informações sobre pessoas.

Na situação mostrada como exemplo na Tabela 1.1, os quatro registros deveriam representar pessoas distintas, no entanto os registros `RID` 1 e 2 referenciam a mesma pessoa. Caso uma junção por similaridade fosse realizada sobre o atributo `Nome` apenas o par formado pelos registros 3 e 4 seria retornado mesmo tratando-se de pessoas diferentes. Em contrapartida, o par que se deseja encontrar (1 e 2) não será retornado devido a convenção de escrita utilizada (abreviação). Em outra situação, caso os atributos `Nome` e `Logradouro` fossem concatenados para evitar o problema anterior, seria o par formado por 1 e 4 a ser retornado erroneamente, desta vez pela semelhança do nome próprio (1) com o nome do logradouro (4). Um meio de solucionar o problema mostrado seria realizar a análise da similaridade em cada atributo isoladamente e posteriormente, através de uma expressão complexa de similaridade, retornar o resultado definitivo.

Contudo, deve ser ressaltado que junções por similaridade são uma operações computacionalmente onerosas e portanto adicionar todas essas avaliações extras pode tornar impeditivo o processamento dos dados, especialmente se tratando de contextos onde grandes quantidades de dados devem ser analisados, com em *Big Data*. Visando solucionar este obstaculo de performance, novos trabalhos surgiram com a finalidade de convergir as técnicas de junção por similaridade com sistemas computacionais distribuídos.

Com objetivo de resolver os problemas que foram discutidos até aqui este trabalho propõe um algoritmo de junção por similaridade distribuído sobre múltiplos atributos focado no processamento de grandes volumes de dados, como os presentes em Big Data. O estado da arte não apresenta até o momento nenhum outro trabalho que explore ao mesmo tempo múltiplos atributos e distribuição da carga de processamento. Além disso a abordagem utilizada generaliza algoritmos existentes para possibilitar o emprego de expressões de similaridade baseadas em fórmulas booleanas.

O Apache Spark [33] foi adotado como arcabouço distribuído padrão na implementação dos algoritmos de processamento e de distribuição de dados. Embora o tradicional modelo MapReduce [6] do Hadoop ainda seja o padrão adotado nos trabalhos da área [28, 23, 7, 22], o Spark vem se mostrando um alternativa viável visto que apresenta um desempenho superior em diversos cenários de processamento [25].

1.2 Objetivos

1.2.1 Geral

O objetivo geral deste trabalho é criar um algoritmo de processamento distribuído que seja escalável e eficiente no processamento de operações de similaridade sobre grandes volumes de dados textuais estruturados.

1.2.2 Específicos

Os seguintes objetivos específicos foram definidos para se alcançar o que foi proposto no objetivo geral.

- Construir um ambiente de testes para pesquisa experimental em arcabouços distribuídos;
- Investigar o arcabouço de processamento distribuído Apache Spark com a finalidade tirar aproveitar os seus recursos;
- Avaliar novas técnicas de particionamento para realizar a distribuição da carga de processamento;
- Elaborar um modelo de custo para guiar a estratégia de particionamento;
- Executar experimentos que simulem diversos tipos de cenários diferentes e validar a solução proposta com a posterior análise dos mesmos.

1.3 Organização do Trabalho

O restante deste trabalho está organizado da seguinte forma: no Capítulo 2 serão apresentadas e formalizadas todas as técnicas de identificação de duplicatas que foram pesquisadas, assim como os arcabouços de processamento distribuído utilizados; no Capítulo 3 serão discutidos os trabalhos do estado da arte relacionados; o algoritmo proposto será apresentado no Capítulo 4 juntamente com a estratégia de seleção de atributo de assinatura; os experimentos realizados, bem como a discussão acerca dos resultados obtidos serão mostrados no Capitulo 5; finalmente, o trabalho será concluído no Capítulo 6.

CAPÍTULO 2

Referencial Teórico

Neste capítulo serão abordadas as técnicas do estado da arte em de junção por similaridade, processamento distribuído, bem como os arcabouços de processamento distribuído utilizados neste trabalho.

2.1 Operações de Similaridade

Esta seção será dedicada a apresentar conceitos em operações de similaridade nas quais este trabalho se baseia.

2.1.1 Funções de Similaridade

O conceito de similaridade pode ser explorado para identificar registros duplicados em uma base de dados tendo como base a premissa de que duplicatas são mais semelhantes entre si do que aos demais registros. Este trabalho é focado exclusivamente na identificação de atributos textuais e para este propósito utilizamos um tipo função de similaridade capaz de retornar um valor que quantifique o quanto dois registros são semelhantes.

Existem dois tipos principais de funções de similaridade no estado da arte atual: as baseadas em distância de edição e as baseadas em conjuntos [7]. As de distância de edição levam em consideração a quantidade mínima de operações necessárias para transformar uma sequência de caracteres em outra como parâmetro para mensurar a similaridade [15, 31, 11]. Operações tipicamente consideradas para cálculo da distância de edição são a inserção, a exclusão e a substituição de caracteres. As baseadas em conjuntos convertem os dados a serem processados para conjuntos e então realizada o cálculo da similaridade considerando interseção dos mesmos [3, 2, 31, 18, 29]. A escolha correta da função de similaridade a ser utilizada depende do contexto do problema. De acordo com trabalhos do estado da arte, ambos os tipos de função de similaridade apresentam acurácia compatível [5]. Neste trabalho optou-se pela utilização funções por similaridade

baseadas em conjuntos pois além de apresentarem boa versatilidade, ainda são menos custosas computacionalmente [26].

2.1.2 *Tokenização*

Um conhecido e amplamente utilizado meio de converter registros textuais em conjuntos é através do processo de *tokenização*, isto é, realizar a transformação de uma *string* em unidades indivisíveis de informação denominadas *tokens*. Uma maneira de realizar este processo seria considerar cada palavra de uma *string* como um *token*. Por exemplo, "*Pedro da Silva*" seria convertido em {"*Pedro*", "*da*", "*Silva*"}. Entretanto essa não é uma técnica interessante no contexto de identificação de duplicatas pois impediria que palavras que contivessem, por exemplo, apenas um erro de digitação fossem contabilizadas em uma intersecção.

Neste trabalho adotou-se *q-grams* [27] para realizar a *tokenização* pois essa técnica permite identificar partes semelhantes de uma *string* mesmo em casos em que apenas um caractere seja diferente. Os *q-grams* consistem em uma sequencia de caracteres obtida através do deslizamento de uma janela de tamanho q sobre um texto. Adicionalmente, para possibilitar uma melhor a identificação dos tokens do começo e do final da *string* são adicionados $q-1$ caracteres especiais no prefixo e no sufixo da mesma. Outro fator considerado é a presença de tokens duplicados. Por exemplo, a palavra "*banana*" seria convertida no 3-gram {"*ban*", "*ana*", "*nan*", "*ana*"}. Para resolver esse problema cada token é vinculado a um número sequencial representando a ordem em que foi gerado e neste caso teremos consequentemente o conjunto {"*ban*1", "*ana*1", "*nan*1", "*ana*2"}. O Exemplo 2.1 demostra o passo a passo da transformação de uma *string* em um conjunto de *tokens* utilizando 3-grams.

Exemplo 2.1 (Tokenização utilizando 3-grams) *Considere a string:*

$$r = \text{"}ALEX\ SALES\text{"}$$

Primeiramente são adicionados q-1 caracteres especiais no prefixo e no sufixo:

$$r = \text{"}\#\#ALEX\ SALES\#\#\text{"}$$

Em seguida os tokens são gerados (note a presença do número anexado para realizar a distinção entre tokens iguais):

$$3grams(r) = \{\text{"}\#\#A1\text{"},\text{"}\#AL1\text{"},\text{"}ALE1\text{"},\text{"}LEX1\text{"},\text{"}EX\ 1\text{"},\text{"}X\ S1\text{"},\text{"}\ SA1\text{"},\ \text{"}SAL1\text{"},\text{"}ALE2\text{"},\text{"}LES1\text{"},\text{"}ES\#1\text{"},\text{"}S\#\#1\text{"}\}$$

Função	Definição
Jaccard	$\frac{w(r \cap s)}{w(r \cup s)}$
Dice	$\frac{2 * w(r \cap s)}{w(r) + w(s)}$
Cosseno	$\frac{w(r \cap s)}{\sqrt{w(r) * w(s)}}$

Tabela 2.1: Funções de similaridade com conjuntos.

2.1.3 Funções de Similaridade Baseadas em Conjuntos

Uma função de similaridade baseada em conjuntos primeiramente transforma cada registro em um conjunto de tokens para em seguida obter uma determinada noção de similaridade através de sua intersecção. Sejam dois conjuntos r e s, podemos definir $sim(r, s)$ como uma função de similaridade baseada em conjuntos que retorna valores variando entre $[0, 1]$, onde 0 significa totalmente diferente e 1 totalmente igual. Um popular exemplo de função deste tipo é *Jaccard*, onde a similaridade é calculada através da razão da intersecção pela união de dois conjuntos. Outras funções populares são *Dice* e *Cosseno*. As definições dessas funções são mostradas na Figura 2.1.

Opcionalmente, cada token pode ser vinculado a um peso específico que represente sua contribuição para o cálculo de similaridade. Baseando-se na premissa de que características raras são mais importantes para a avaliação da similaridade [21], o IDF (*Inverse Document Frequency*) calcula o peso de cada *token* de forma inversamente proporcional a sua frequência na base de dados. A Definição 2.2 mostra o cálculo de IDF para um token t.

Definição 2.2 (Cálculo do IDF)

$$idf(t) = \ln\left(1 + \frac{N}{df(t)}\right),$$

onde N é o total de conjuntos de uma coleção de dados e $df(t)$ representa a quantidade de conjuntos em que token t aparece na coleção de dados.

O peso de um *token* t é referenciado como $w(t)$. O peso total de um conjunto r é dado pelo somatório dos pesos de todos os seus *tokens* conforme demostrado na Definição 2.3. Em um contexto onde não seja interessante o uso de pesos bastaria atribuir arbitrariamente o peso 1 para cada um dos tokens dos conjuntos da coleção. Neste caso observa-se que $|r| = w(r)$ para conjuntos sem peso, sendo $|r|$ a quantidade total de *tokens* presente em r.

Definição 2.3 (Cálculo do peso de um conjunto)

$$w(r) = \sum_{t \in r} w(t)$$

Finalmente, no Exemplo 2.4 é demostrado o passo a passo do cálculo de similaridade utilizando a função *Jaccard*.

Exemplo 2.4 (Função de similaridade baseada em conjuntos) *Sejam as strings:*

$$r = \text{"}DIEGO\text{"}$$
$$s = \text{"}DYEGO\text{"}$$

Primeiramente a transformação em conjuntos é realizada utilizando um processo de tokenização com 1-grams (por simplicidade, caracteres especiais não foram incluídos no sufixo e prefixo das strings):

$$r = \{D,\ I,\ E,\ G,\ O\}$$
$$s = \{D,\ Y,\ E,\ G,\ O\}$$

Ao realizar o cálculo dos pesos de cada token temos os valores:

t	D	I	Y	E	G	O
$w(t)$	0,7	1,1	1,1	0,7	0,7	0,7

Realizando a intersecção e a união temos:

$$(r \cap s) = \{D,\ E,\ G,\ O\}$$
$$(r \cup s) = \{D,\ I,\ Y,\ E,\ G,\ O\}$$

Finalmente, ao aplicar a função Jaccard obtêm-se:

$$Jaccard(r, s) = 2,8/5 \approx 0,6$$

Opcionalmente, utilizando peso 1 para todos os tokens, isto é, $|r| = w(r)$:

$$Jaccard(r, s) = 4/6 \approx 0,7$$

2.1.4 Junção por Similaridade

No contexto deste trabalho, junção por similaridade é uma técnica que emprega uma função de similaridade sobre uma coleção de dados textuais com a finalidade de identificar pares que possuam a similaridade igual ou superior a um determinado limiar

de similaridade, o qual será referenciado pelo termo *threshold* no restante do texto. A Definição 2.5 a seguir descreve formalmente o seu funcionamento.

Definição 2.5 (Junção por similaridade) *Seja R uma coleção de registros textuais, $R.a$ o domínio do atributo a em R e $r.a$ um conjunto gerado pelo valor de um atributo a de um registro $r \in R$. Seja $sim : R.a \times R.a \rightarrow [0,1]$ uma função de similaridade e α um determinado threshold. A junção por similaridade em $R.a$ retornará todos os pares de registros*

$$\langle (r,s), \alpha' \rangle \mid (r,s) \in R \times R \wedge sim(r.a, s.a) = \alpha' \geq \alpha$$

No restante deste trabalho o termo "junção por similaridade"será usado para referenciar uma junção por similaridade que utiliza uma função de similaridade baseada em conjuntos do tipo Jaccard.

2.2 Estratégia de Filtragem-e-Verificação

Após transformação dos registros em conjuntos a junção por similaridade está pronta para ser executada na coleção de dados. A forma ingênua de realizar essa operação seria comparar cada conjunto com todos os outros da base de dados. No entanto essa abordagem teria uma complexidade $O(n^2)$, o que poderia deixar inviável uma análise em contextos onde se trabalha com grandes quantidade de dados como Big Data.

Um método que vem sendo amplamente adotado por trabalhos da área e que será utilizado neste trabalho é a *filtragem-e-verificação* [28, 18, 14, 7, 22]. Nesta abordagem, o processamento é dividido em duas etapas: na primeira é realizada uma filtragem dos candidatos de modo que pares que tenham uma discrepância muito grande entre si não tenham que ser verificados; já na segunda etapa, o cálculo da similaridade entre os conjuntos é realizado. Existem diversas técnicas de filtragem no estado-da-arte da arte atual, como será demostrado apresentado a seguir.

2.3 Limites de Similaridade

Aqui serão apresentados os limites de similaridade utilizados na fase de filtragem. Por simplicidade, apenas conjuntos sem peso serão considerados nos exemplos desta seção.

Função	Limite de intersecção
Jaccard	$\frac{\alpha}{1+\alpha} * (\|r\| + \|s\|)$
Dice	$\frac{\alpha * (\|r\| + \|s\|)}{2}$
Cosseno	$\alpha * \sqrt{(\|r\| * \|s\|)}$

Tabela 2.2: Intersecção mínima para funções de similaridade com conjuntos.

2.3.1 Limite de Intersecção

O limite de interseção determina que duas entidades só podem ser semelhantes se entre elas houver uma intersecção mínima, denotada por *interMin*. Dessa forma o problema de junção por similaridade pode ser transformado para um problema de intersecção entre conjuntos. A Definição 2.6 descreve o seu funcionamento.

Definição 2.6 (Limite de intersecção) *Sendo r e s conjuntos de tokens, sim uma função de similaridade baseada em conjuntos e α um determinado threshold, então:*

$$sim(r,s) \geq \alpha \leftrightarrow |s \cap r| \geq interMin(r,s)$$

A Tabela 2.2 mostra a função de intersecção mínima para cada função de similaridade baseada em conjuntos. O Exemplo 2.7 demostra o funcionamento desta técnica de filtragem para a função *Jaccard*.

Exemplo 2.7 (Limite de intersecção) *Sejam os conjuntos r, s e seus respectivos de tokens:*

$$r = \{D,\ I,\ E,\ G,\ O\}$$
$$s = \{D,\ Y,\ E,\ G,\ O\}$$

Para um threshold $\alpha = 0,8$ e considerando conjuntos sem pesos, temos:

$$interMin(r,s) = \frac{0,8}{1+0,8}(5+5) = 4,44$$

2.3.2 Limite de Tamanho

Outra forma de filtrar candidatos é baseia-se no princípio de que para serem semelhantes dois conjuntos também necessitam ter tamanhos (ou pesos) semelhantes.

Função	Tam. Mínimo	Tam. Máximo
Jaccard	$\alpha * \lvert r \rvert$	$\frac{\lvert r \rvert}{\alpha}$
Dice	$\frac{\alpha * \lvert r \rvert}{2 - \alpha}$	$\frac{(2 - \alpha) * \lvert r \rvert}{\alpha}$
Cosseno	$\alpha^2 * \lvert r \rvert$	$\frac{\lvert r \rvert}{\alpha^2}$

Tabela 2.3: Tamanho mínimo e máximo para de similaridade com conjuntos.

Dessa forma, a quantidade (ou peso) mínima e máxima de tokens que um conjunto s deve possuir para ser semelhante a um segundo r deve estar dentro no intervalo [*tamMin*, *tamMax*], formalizado na Definição 2.8. A Tabela 2.3 mostra os tamanhos máximos e mínimos para cada função de similaridade baseada em conjuntos.

Definição 2.8 (Limite de tamanho) *Sendo sim uma função de similaridade baseada em conjuntos, r e s conjuntos de tokens e α um determinado threshold, então:*

$$sim(r, s) \geq \alpha \leftrightarrow tamMin(r) \leq |s| \leq tamMax(r)$$

O Exemplo 2.9 demostra do com limite de tamanho.

Exemplo 2.9 (Limite de tamanho) *Seja o conjunto r e seu respectivos tokens $r = \{D, I, E, G, O\}$, para um threshold $\alpha = 0,8$ e considerando conjuntos sem pesos, temos:*

$$tamMin(r) = 0,8 * 5 = 4$$
$$tamMax(r) = 5 \,/\, 0,8 = 7$$

2.4 Esquema de Assinaturas

Assim como em outras contribuições do estado da arte de junções por similaridade baseadas em conjuntos [1, 7, 12, 22], neste trabalho optou-se por usar um esquema baseado em assinaturas. Neste tipo de abordagem primeiramente são geradas assinaturas para cada conjunto de entrada e, em seguida, conjuntos que possuam assinaturas em comum são agrupados em pares que obedecem a seguinte propriedade: sendo sim uma função de similaridade baseada em conjuntos e um *threshold* α, para que $sim(r, s) \geq \alpha$, então r e s devem compartilhar pelo menos uma assinatura em comum. A diferença entre os métodos existentes está centrado basicamente na estratégia utilizada para geração das assinaturas.

2.4.1 Filtragem por Prefixo

Como mostrado em [3], com a utilização da filtragem por prefixo é possível diminuir o número de pares candidatos observando-se apenas uma parte do conjunto de entrada original. Para realizar a exploração deste conceito primeiramente deve-se aplicar uma ordenação global O sobre todos os conjuntos da coleção a ser analisada. A Definição 2.10 apresenta os detalhes para sua implementação enquanto o Exemplo 2.11 mostra a sua aplicação.

Definição 2.10 (Filtragem por Prefixo) *Considerando que os tokens de todos os conjuntos de uma coleção de dados estão sobre um ordenamento O e que $\gamma = \lceil interMin(r,s) \rceil$. Então, para que $|r \cap s| \geq \gamma$, os primeiros $|r| - \gamma + 1$ elementos de r e os primeiros $|s| - \gamma + 1$ elementos de s devem compartilhar pelo menos um token em comum.*

Exemplo 2.11 (Aplicando a filtragem por prefixo) *Considerando os registros:*

$$r = \text{"}Diego\ Oliveira\text{"}$$
$$s = \text{"}Dyego\ Oliveira\text{"}$$

Aplicando a tokenização utilizando $3 - grams$ e ordenando alfabeticamente os elementos, teremos os conjuntos:

$$r = \{\text{"}DIE\text{"}, \text{"}EGO\text{"}, \text{"}EIR\text{"}, \text{"}ERA\text{"}, \text{"}GO\text{"}, \text{"}IEG\text{"}, \text{"}IVE\text{"}, \text{"}LIV\text{"}, \text{"}\ OL\text{"}, \text{"}OLI\text{"}, \text{"}OO\text{"}, \text{"}VEI\text{"}\}$$
$$s = \{\text{"}DYE\text{"}, \text{"}EGO\text{"}, \text{"}EIR\text{"}, \text{"}ERA\text{"}, \text{"}GO\text{"}, \text{"}IVE\text{"}, \text{"}LIV\text{"}, \text{"}\ OL\text{"}, \text{"}OLI\text{"}, \text{"}OO\text{"}, \text{"}VEI\text{"}, \text{"}YEG\text{"}\}$$

Se considerarmos $\alpha = 0.7$, então teremos que:

$$|pref(r)| = 12 - \lceil 9,88 \rceil + 1 = 3$$
$$|pref(s)| = 12 - \lceil 9,88 \rceil + 1 = 3$$

Consequentemente teremos:

$$pref(r) = \{\text{"}DIE\text{"}, \text{"}EGO\text{"}, \text{"}EIR\text{"}\}$$
$$pref(s) = \{\text{"}DYE\text{"}, \text{"}EGO\text{"}, \text{"}EIR\text{"}\}$$

Logo, podemos concluir que r e s são candidatos pois compartilham 2 tokens em comum em seus prefixos.

Da forma como foi apresentado, o tamanho do prefixo vai depender de cada par analisado. Como o tamanho do prefixo varia de forma inversa com o $interMin(r,s)$, quanto menor for o tamanho de s maior será o prefixo de r. Sendo assim, uma maneira de determinar um prefixo para r que possa ser utilizado para identificar qualquer conjuntos s que satisfaça $|r \cap s| \geq \gamma$ é considerando $pref(r) = |r| - \lceil tamMin(r) \rceil + 1$ [3].

Além disso, uma forma de maximizar a eficiência da filtragem por prefixo é escolhendo uma ordenação O baseada na frequência global dos elementos da coleção, pois com os elementos mais raros no prefixo é menor a chance de existir uma intersecção com outro conjunto candidato.

2.4.2 MassJoin

Proposto por [7], esta estratégia definiu uma nova abordagem para funções de similaridade baseadas em conjuntos e também um novo esquema de geração de assinaturas. Enquanto os outros trabalhos utilizam a intersecção dos conjuntos como referência pra cálculo da similaridade, a inovação do MassJoin é utilizar a diferença entre os mesmos.

Também conhecida como distância *Hamming*, esta diferença entre dois conjuntos r e s pode ser definida como: $H(r,s) = |r-s| + |s-r|$. A Figura 2.1 representa graficamente a qual parte dos conjuntos as diferenças e a intersecção se referem.

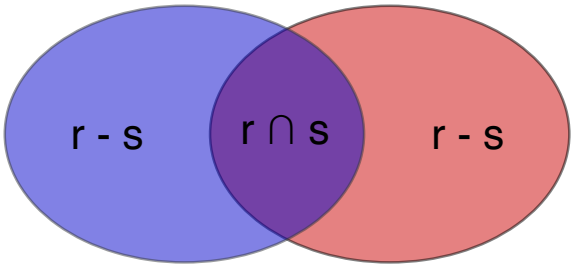


Figura 2.1: Diferenças e interseção entre dois conjuntos.

Nesta abordagem, para que dois conjuntos sejam similares, a distância *Hamming* deve ser menor ou igual a um limite superior U. A definição 2.12 mostra o cálculo do para a função *Jaccard*.

Definição 2.12 (Limite superior para função *Jaccard*) *Sejam r e s dois conjuntos obtidos atraves de dois registros textuais de uma coleção dados e α um determinado threshold. O limites superiores para $|r-s|$ e $|s-r|$ são respectivamente:*

$$|r-s| \leq \left\lfloor \frac{|r| - \alpha|s|}{1+\alpha} \right\rfloor$$

$$|s-r| \leq \left\lfloor \frac{|s| - \alpha|r|}{1+\alpha} \right\rfloor$$

Consequentemente, o limite superior da distância Hamming U para a função Jaccard e um threshold α será:

$$|r-s| + |s-r| \leq U = \left\lfloor \frac{|r| - \alpha|s|}{1+\alpha} \right\rfloor + \left\lfloor \frac{|s| - \alpha|r|}{1+\alpha} \right\rfloor$$

No Exemplo 2.13 é mostrado um caso de utilização do MassJoin.

Exemplo 2.13 (Similaridade com MassJoin) *Sejam os registros* $r = "DIEGO"$, $s = "DYEGO"$ *e seus respectivos conjuntos formados através de tokenização com 1-grams:*

$$r = \{D,\ I,\ E,\ G,\ O\}$$
$$s = \{D,\ Y,\ E,\ G,\ O\}$$

Realizando o cálculo da distância Hamming teremos:

$$|r - s| = 1$$
$$|s - r| = 1$$
$$H(r,s) = 2$$

Realizando o cálculo do limite superior para um threshold $\alpha = 0,6$*:*

$$U = \left\lfloor \frac{5-0,6*5}{1+0,6} \right\rfloor + \left\lfloor \frac{5-0,6*5}{1+0,6} \right\rfloor$$
$$U = \lfloor 1,25 \rfloor + \lfloor 1,25 \rfloor = 2$$

Neste caso podemos concluir que r *e* s *são similares pois* $H(r,s) \leq U$

O processo de geração de assinaturas no MassJoin é baseado em 'seguimentos' de *tokens* adjacentes. O primeiro passo dessa estratégia seria utilizar o limite superior U para dividir um conjunto já devidamente ordenado r em $U+1$ seguimentos. Dente os possíveis caminhos para realizar esta divisão, na Definição 2.14 é mostrado um esquema onde os seguimentos gerados são sempre de tamanhos aproximadamente iguais.

Definição 2.14 (Particionamento com MassJoin) *Seja* r *um conjunto ordenado de tokens, um certo limite superior* U *e considerando* $l = |r|$*, temos que:*

$$k = l - \left\lfloor \frac{l}{U+1} \right\rfloor * (U+1)$$

Neste caso, o tamanho dos dos i *primeiros* $U+1-k$ *seguimentos será de:*

$$l_i = \left\lfloor \frac{l}{U+1} \right\rfloor$$

Enquanto o tamanho do último seguimento será:

$$l_i = \left\lceil \frac{l}{U+1} \right\rceil$$

As informações sobre cada seguimento gerado são usadas para formar uma assinatura de acordo com a seguinte notação $\langle r[p_i, l_i], i, l \rangle$, onde p_i e l_i são, respectivamente, a posição inicial e o tamanho de um seguimento de r, i corresponde a posição sequencial

do seguimento e l seria o tamanho total de r. A Figura 2.2 mostra as assinaturas geradas para um conjunto r.

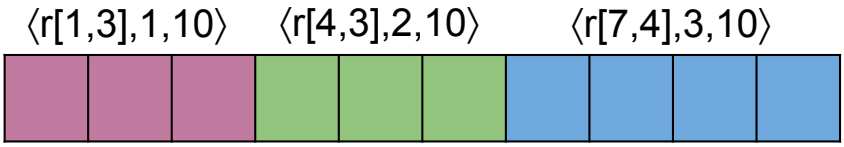


Figura 2.2: Exemplos de assinaturas no particionamento com MassJoin.

De forma análoga as assinaturas para r, as assinaturas para um conjunto s seguem a notação $\langle r[x_i, l_i], i, l \rangle$. O diferencial neste caso é o fato de que os seus seguimentos podem iniciar em qualquer posição (x_i). Para s e r serem similares os mesmos devem compartilhar ao menos um seguimento. O problema das assinaturas para s se iniciarem em uma posição arbitrária é a grande quantidade de combinações que seriam possíveis, o que geraria uma grande quantidade de seguimentos. A solução neste caso é construir as assinaturas com base no posicionamento das assinaturas em r, ou seja, serão geradas assinaturas para cada posição no intervalo $x_i \in [p_i - U_{r-s}, p_i + U_{s-r}]$. A Figura 2.3 demostra este funcionamento.

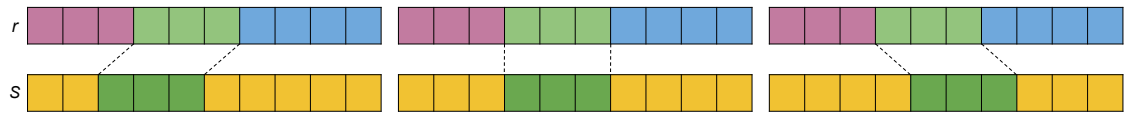


Figura 2.3: Assinaturas do conjunto s derivadas a partir do conjunto r.

2.5 Junção por Similaridade com Expressões Complexas

Até o presente momento consideramos que dois registros r e s de uma coleção R são similares apenas realizando a junção por similaridade no domínio de um atributo específico $R.a$. No entanto, conforme visto previamente na Seção 1.1, registros do mundo real normalmente são compostos por diferentes atributos e cada um destes poderia ter uma importância diferente no contexto de identificação de registros de duplicados.

O termo "expressões complexas"utilizado neste trabalho refere-se a uma expressão de similaridade que possui mais de um predicado a ser analisado. A Figura 2.4 demostra, de forma preliminar, o seu funcionamento. Primeiramente serão gerados conjuntos S_n^i obtidos de cada atributo a_i de um determinado registro r_n. Em seguida, para se obter os resultados, a similaridade destes conjuntos é avalida considerando uma expressão complexa de similaridade $\phi : R \times R \rightarrow \{true, false\}$ em forma normal disjuntiva contendo

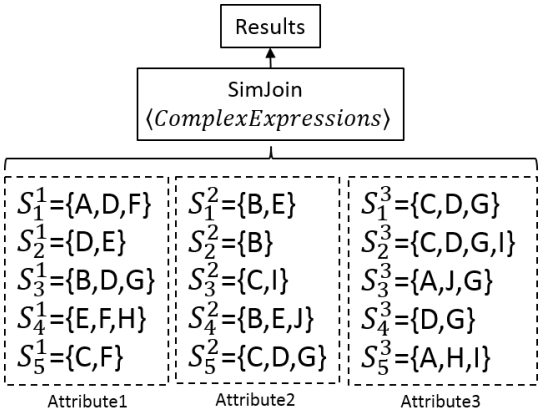


Figura 2.4: Junção por similaridade sobre múltiplos atributos.

um conjunto de cláusulas $C = \{c_1, ..., c_m\}$, cujos literais são definidos por predicados de similaridade e negações não são permitidas conforme mostrado na Definição 2.15.

Definição 2.15

$$\phi = \bigvee_{c \in C} \bigwedge_{p \in c} p.$$

O conceito de junção por similaridade sobre múltiplos atributos utilizada neste trabalho é sintetizada na Definição 2.16.

Definição 2.16 (Junção por similaridade sobre múltiplos atributos) *Seja R uma coleção de registros e ϕ uma expressão de similaridade. A junção por similaridade sobre múltiplos atributos retorna todos os pares de registros $(r, s) \in R \times R$, tal que $\phi(r, s) = true$.*

2.6 Arcabouços de Processamento Distribuído

Nesta seção serão abordadas as tecnologias para processamento distribuído de dados utilizadas neste trabalho. Apesar de não fazer parte da implementação proposta, os conceitos presentes no MapReduce são importantes para compreensão do arcabouço Spark e por esse motivo serão detalhados primeiro. Além disso serão discutidos os pontos chave que diferem os dois arcabouços.

2.6.1 MapReduce

Proposto por [6], o MapReduce é um modelo de programação para criação de aplicações capazes de processar grandes quantidades de dados de forma paralela e distribuída. Este modelo possui três fases principais de processamento: o `map`, `shuffle`

e o `reduce`. Na fase de `map` os registros do conjunto de dados são divididos e enviados para diferentes unidades de processamento denominadas de *workers*, então cada registro é mapeado em um par de chave/valor e em seguida os que possuem a mesma chave são transformados em pares de chave/lista de valores. Na fase de `shuffle` os dados gerados por cada *mapper* que possuem a mesma chave são enviados para um mesmo *reducer*. Se este *reducer* estiver em um *worker* diferente o dado será enviado pela rede. Finamente, na fase de `reduce`, cada *worker* realiza uma determinada operação sobre a chave/lista de valores resultante da etapa anterior para gerar o resultado final. A Figura 2.5 ilustra este funcionamento.

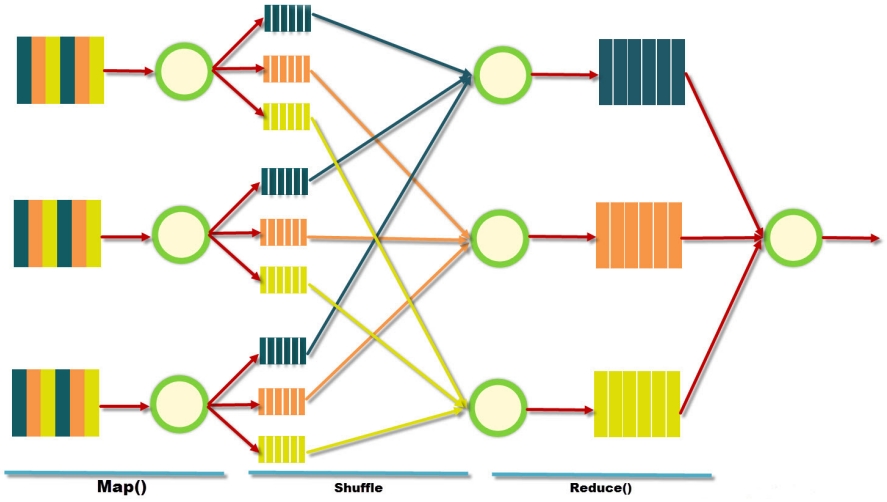


Figura 2.5: Funcionamento do MapReduce.

O Apache Hadoop[1] é a principal e mais popular implementação do MapReduce. Seu arcabouço de software permite escrever aplicações de propósito geral para serem processadas em grandes clusters e com tolerância a falhas. Suas aplicações em sistemas Big Data são bem amplas, dentre elas podendo-se destacar análise de dados, rastreamento, indexação, sistemas de reputação e mineração de dados. O arcabouço do Hadoop está dividido em três principais núcleos: O sistema de arquivos distribuído *Hadoop Distributed File System* (HDFS), o *cluster manager Yarn* e o próprio *MapReduce core*. É importante ressaltar que também é possível utilizar o *MapReduce core* em conjunto com outros sistemas de arquivos distribuídos como o *Amazon S3*[2] ou *Azure storage*[3]. Além disso outros *clusters managers* como o *Mesos*[4] também são suportados.

[1] https://hadoop.apache.org/
[2] https://aws.amazon.com/s3
[3] https://azure.microsoft.com/pt-br/services/storage/
[4] http://mesos.apache.org/

Sistema de arquivos distribuídos padrão do Hadoop, o HDFS é capaz de armazenar grandes quantidades de dados, com tolerância a falhas e oferecendo um alto desempenho. Possuindo interfaces de acesso através de API's de programação, aplicativos *shell* ou até mesmo *web browsers*. O HDFS permite que os dados sejam lidos e persistidos da mesma forma que em um sistema de arquivos contínuo.

Disponível a partir da segunda versão do Hadoop, o *Yarn cluster manager* tem como objetivo o gerenciamento do recursos computacionais do cluster. Dessa forma ele realiza o monitoramento dos mesmos e garante que serão dinamicamente alocados para uma determinada tarefa quando necessário. Atualmente o Yarn pode gerenciar as cargas de trabalho tanto do *MapReduce core* como também de outros arcabouços de processamento distribuídos como o Spark.

2.6.2 Apache Spark

Nascido em 2009 no AMPLab da universidade de Berkeley e atualmente um popular projeto da fundação Apache[5], o Spark é um arcabouço para processamento de conjuntos de dados de larga escala em arquiteturas distribuídas. Diferente do Hadoop, que necessita de ferramentas externas como Hive[6] para certos tipos de dados, o arcabouço Spark é integrado e fornece ferramentas para processamento em *batch*, *streaming*, *machine learning*, SQL e grafos. Além disso sua API suporta os mais variados sistemas de armazenamento de dados, como o HDFS (do próprio Hadoop), Cassandra[7], HBase[8], etc. O gerenciamento dos recursos computacionais do cluster pode ser feito tanto em modo *standalone* como também utilizando os *clusters managers Yarn* ou *Mesos*. Escrito em Scala e executando em uma máquina virtual Java, no Spark é possível desenvolver aplicações utilizando tanto essas duas linguagens como também Python e R. Adicionalmente um *shell* interativo também é fornecido para acesso rápido. O arcabouço Spark é mostrado no diagrama da Figura 2.6.

Como é baseado na arquitetura distribuída *shared-nothing*, cada um dos nós de processamento de um *cluster* Spark (também chamados de *workers*) funcionam de forma independente, tendo sua sua própria memória principal e disco. Este fato permite que mesmo hardware básico (que é mais barato e facilmente encontrado no mercado) seja utilizado para criação de um *cluster* e também facilita quando um upgrade, reparo ou substituição é necessário.

[5] https://spark.apache.org/
[6] https://hive.apache.org/
[7] http://cassandra.apache.org/
[8] https://hbase.apache.org/

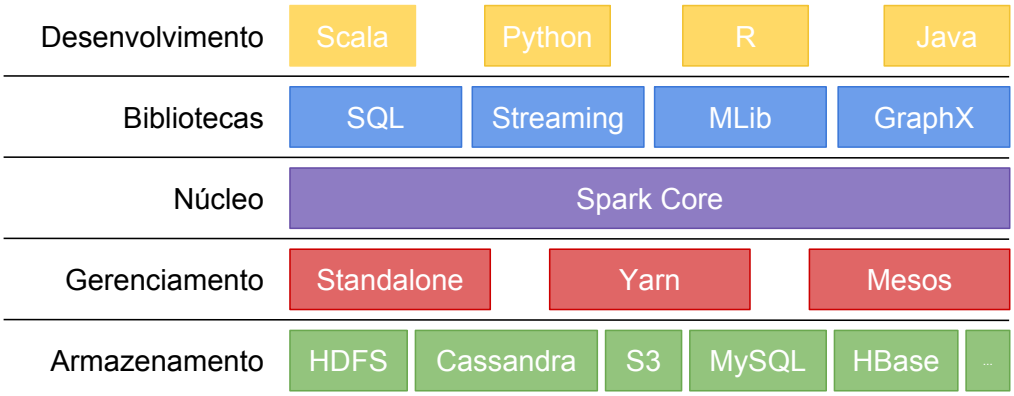


Figura 2.6: Arcabouço Apache Spark.

Resilient Distributed Datasets

O principal componente de funcionamento do Spark são os *resilient distributed datasets* (RDDs) [33], uma abstração para processamento de dados distribuídos, em memória e com tolerância a falhas. Mais especificamente, os RDD's são coleções de objetos distribuídos, particionados e imutáveis que podem ser criados a partir de dados de entrada específicos ou de outros RDD's. Existem dois tipos de operações que podem ser realizadas sobre um RDD: `transformações` e `ações`.

As `transformações` sempre criam um novo RDD a partir de outro. Além disso são operações do tipo 'preguiçosa', ou seja, ao invés de serem executadas no momento em que são chamadas, apenas a informação do que deve ser realizado por elas é armazenada. As tarefas passadas para uma transformação serão executadas apenas quando os dados da mesma forem requeridos por uma operação de `ação`. A Tabela 2.6.2 mostra alguns exemplos de transformações e ações do arcabouço Spark.

Tipo	Nome	Descrição
Transformação	map	processa cada elemento de um RDD utilizando uma determinada função.
	filter	aplica um filtro sobre os elementos de um RDD.
	groupByKey	agrupa uma coleção de pares (chave/valor) por chave.
Ação	count	retorna a quantidade de elemento em um RDD.
	first	retorna o primeiro elemento de um RDD.
	saveAsTextFile	armazena o conteudo e um RDD em um sistema de arquivos.

Tabela 2.4: Exemplos de operações no arcabouço Spark.

Spark vs. Hadoop

Por padrão os RDD's armazenam seus dados prioritariamente em memória principal e além disso é possível estabelecer um fluxo de operações de `transformação` e `ação`, isso facilita a resolução de problemas onde os mesmos dados devem ser processados várias vezes (como programação interativa, necessária em algoritmos de regressão e mineração de dados). Em contrapartida, no Hadoop seria necessário encadear uma série de trabalhos de MapReduce e entre cada um deles os dados teriam que ser gravados e lidos várias vezes na memória secundária. A Figura 2.7 mostra graficamente as diferenças entre ambos modelos de processamento. Em testes anteriores o Spark demostrou performance superior em relação ao Hadoop em diversos cenários de processamento [33, 25].

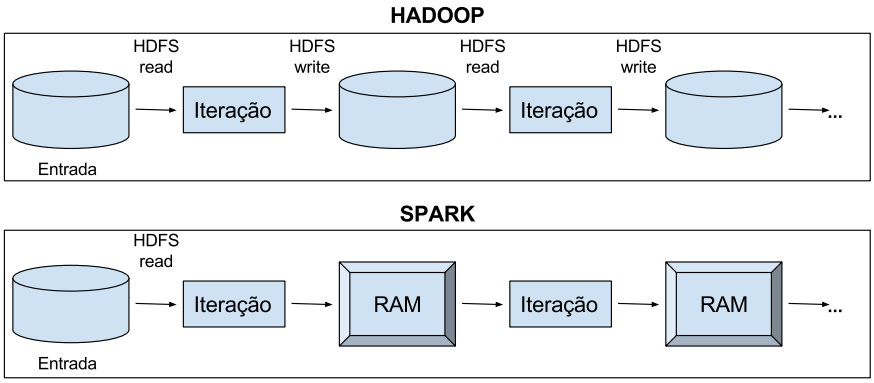


Figura 2.7: Processamento Hadoop x Spark.

Tolerância a Falhas

Muitos arcabouços para processamento distribuído utilizam *fine-grained updates* para prover tolerância a falhas, isto é, os dados e atualizações realizadas durante o processamento são replicados através dos nós do cluster, causando grande uso de banda de rede e espaço de armazenamento. Por outro lado, o Spark utiliza *coarse-grained updates*, ou seja, os cada uma das `transformações` invocadas durante a execução do programa são armazenadas em um *log* formando uma espécie de linha do tempo. Caso algum RDD seja perdido bastará utilizar esta informação para recriá-lo a partir do RDD de origem.

CAPÍTULO 3

Trabalhos Relacionados

Algoritmos de junção por similaridade baseados em conjuntos vem sendo exaustivamente investigados pela literatura [8, 24, 3, 1, 2, 32, 18]. Nos trabalhos do estado-da-arte prevalece o uso de arcabouços de filtragem-e-verificação baseados no uso de listas invertidas. Devido a etapa de verificação ser tipicamente onerosa computacionalmente, a maioria das abordagem vem focando na primeira etapa onde diversos filtros são aplicados com o objetivo de diminuir ao máximo o espaço de comparação, como filtragem por prefixo [24, 3], *length-filter* [24], filtro por posição [32] e *min-prefix* [18]. Para aproveitar melhor o ganho de desempenho obtido com os filtros de candidatos muitos trabalhos focaram exaustivamente nesta etapa [32], no entanto ao elevar muito o processamento na fase de filtragem, o tempo total de execução pode ser afetado negativamente [18].

Além dos algoritmos *stand-alone*, trabalhos anteriores também implementaram junção por similaridade em conjunto com tecnologia de banco de dados relacionais. Em [19] foi proposta uma expressão de junção por similaridade declarativa em SQL. Já em [3] foi proposto um operador físico dentro do mecanismo de consulta. Outros trabalhos exploraram o paralelismo massivo das modernas unidade de processamento gráfico (GPU's) para melhorar o desempenho [20].

Outros tipos de abordagem, mais próxima da linha de pesquisa deste trabalho, visam melhorar o desempenho dos algoritmos de junção por similaridade em plataformas de processamento distribuído [28, 7, 22]. Todos os trabalhos até aqui vem utilizando o MapReduce para implementar seus algoritmos, não foi encontrado até o presente momento nenhum trabalho que implemente alguma destas técnicas utilizando Spark. O foco principal das técnicas de junção por similaridade distribuídas, incluindo este trabalho, está na estratégia de particionamento dos dados adotada para poder dividir o trabalho através dos nós de processamento do distribuídos pelo *cluster*. A abordagem mais comum é utilizar um esquema onde registros que contenha uma assinatura em comum sejam enviados para um mesmo *worker*.

Assim como neste trabalho, em [28] é utilizado a filtragem por prefixo para geração de assinaturas, no entanto é considerado apenas um simples predicado de similaridade sobre os dados, que são representados por apenas um conjunto. O trabalho apresentado

em [7] propôs um esquema de assinatura baseado no tamanho da diferença simétrica dos conjuntos (também conhecida como distância *Hamming*). Neste trabalho foram realizados experimentos comparando diretamente esses os dois esquemas de particionamento.

Recentemente foi apresentado em [22] uma abordagem diferente baseada em particionamento vertical. Aqui, primeiramente os *tokens* dos conjuntos de entrada são divididos em 'seguimentos'. Em seguida os dados são agrupados verticalmente em 'fragmentos' de acordo com a ordem de seus seguimentos que foram gerados anteriormente. Cada fragmento é enviado para um *worker* onde é calculada a intersecção parcial dos conjuntos utilizando os seus seguimentos. Finalmente, as interseções parciais são agregadas para gerar a resposta final. A principal vantagem desta estratégia é evitar a grande replicação de conjuntos que acontece em [28, 7]. Além disso gerar seguimentos de tamanho similar melhora o balanceamento de carga através dos *workers* do *cluster*. Infelizmente, não está claro como adaptar esta estratégia para o contexto deste trabalho, que considera os dados de múltiplos atributos e expressões de similaridade complexas.

O trabalho apresentado em [12] é o único que considera junção por similaridade sobre múltiplos atributos. Aqui as assinaturas são geradas através do produto cartesiano do prefixo derivado de cada atributo de um registro. Um índice é construído através de uma árvore de prefixos com a finalidade de eliminar candidatos baseando-se em múltiplos predicados de similaridade. Heurísticas são apresentadas com objetivo de diminuir o tamanho da árvore-índice e determinar uma ordem para sua construção e avaliação de expressão de similaridade na fase de verificação. O algoritmo proposto foi projetado para executar em apenas uma máquina e aceita somente expressões de similaridade conjuntivas. Em contrapartida, neste trabalho é apresentado um algoritmo distribuído e que oferece suporte a expressões de similaridade conjuntivas e disjuntivas. Para efeito de comparação com a solução proposta, este esquema de geração de assinaturas também será avaliado nos experimentos deste trabalho.

CAPÍTULO 4

Processamento Distribuído de Junção por Similaridade Sobre Múltiplos Atributos

Neste capítulo será apresentado o algoritmo DSJoin – *Distributed Similarity Join*. Primeiramente será demostrada a estratégia de particionamento utilizada para realização de junções por similaridade em ambientes distribuídos. Em seguida o algoritmo proposto será detalhado mais a fundo e será descrito o modelo de custos usado para guiar o particionamento de dados. Por fim, outras estratégias de particionamento serão apresentadas.

4.1 Particionamento de Dados

Devido ao grande volume de dados presentes em aplicações Big Data e a necessidade de se realizar rapidamente o seu processamento, foi necessário a adaptação das técnicas de junção por similaridade para serem realizadas em outros tipos de tecnologia, como sistemas distribuídos. A solução proposta neste trabalho concentra-se em arquiteturas distribuídas do tipo *shared-nothing*, onde várias máquinas formam um cluster conectado através de uma rede de alta velocidade. Cada máquina (ou nó) é independente, ou seja, tem sua própria memória e disco privados. Normalmente, cada nó pode conter várias CPUs (arquitetura *multi-core*) e cada um deste também pode estar associado a processadores virtuais com a finalidade de aumentar o grau de paralelismo. Como já vem sendo utilizado nos capítulos anteriores deste trabalho, cada nó do cluster se refere a um *worker*.

A estratégia de particionamento é crucial em uma arquitetura do tipo *shared-nothing* pois afeta diretamente o custo de comunicação, isto é, o número de registros enviados através da rede e o custo de computação, isto é, o número de comparações executadas em cada *worker*. De maneira análoga a trabalhos anteriores [28, 7], a estratégia geral adotada neste trabalho consiste em derivar um *conjunto assinaturas* de cada registro e usá-las para guiar o particionamento de dados. Registros com pelo menos uma assinatura em comum são enviados para um mesmo *worker*, onde a expressão de similaridade será

avalidada entre eles, enquanto que registros que não compartilham qualquer assinatura não serão comparados.

A estratégia de filtragem por prefixo (Seção 2.4.1) pode ser adaptado facilmente para guiar particionamento de dados utilizando cada um dos elementos de um determinado prefixo como assinatura. Além disso, como os *tokens* de um conjunto são ordenados por ordem crescente em relação a sua frequência global, o seu prefixo possui os *tokens* mais raros em toda a coleção analisada. Desta forma menos registros são transmitidos pela rede, um menor número de pares de candidatos são gerados para serem processados e ambos os custos de comunicação e computação são reduzidos. Vale ressaltar que esta mesma estratégia já foi utilizada em [28], no entanto este trabalho não considerava registros compostos por múltiplos atributos.

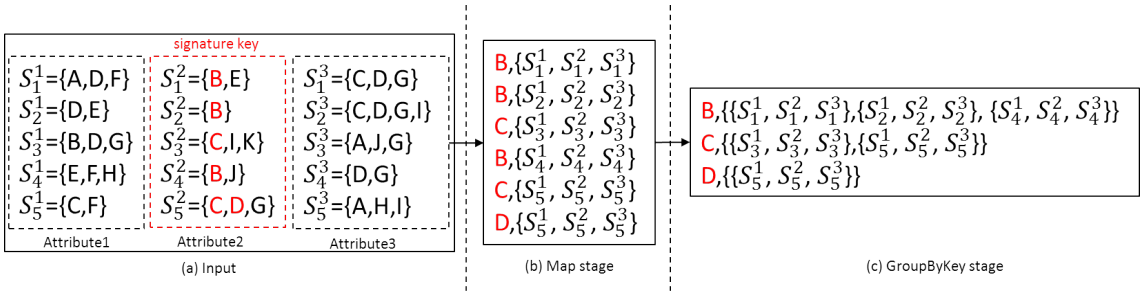


Figura 4.1: Particionamento dos dados utilizando um atributo de assinatura.

No contexto deste trabalho cada registro poderia dar origem a múltiplos prefixos, cada um destes gerados a partir de seu respectivo atributo. Neste caso para poder utilizar o particionamento com filtragem por prefixo em uma coleção de registros R associada a um determinado esquema, apenas um *atributo de assinatura* é escolhido dentre os demais para que os *tokens* do seu prefixo sejam utilizados para guiar o particionamento dos dados. Observe que o atributo de assinatura também pode ser associado a mais de um predicado dentro de uma expressão de similaridade (ver Seção 2.5).

A utilização de um atributo de assinatura é vantajosa em diversos contextos, por exemplo, considere que um dos atributos tenha sido associado a um *threshold* mais elevado no predicado da junção. Como *thresholds* mais elevados geram prefixos menores consequentemente, ao escolher este atributo mencionado como atributo de assinatura, uma menor quantidade de registros teria que ser enviada pela rede diminuindo os custos de comunicação e computação. Outra vantagem seria em relação as estratégias que utilizam atributos concatenados como base para distribuição do processamento pois os mesmos tentem a ter tamanhos maiores e portanto prefixos maiores também.

Na Figura 4.1, o `Atributo2` pertencente a um determinado esquema de coleção de registros é utilizado como atributo de assinatura (a). Em seguida, os *tokens* do prefixo do atributo de assinatura são utilizados para criar pares de `(chave/valor)` para

cada registro associado (b). Finalmente, os registros que contém uma mesma assinatura associada são agrupados em um mesmo *worker* estando prontos para a etapa verificação (c). Na Seção 4.3 será discutido a melhor maneira de se escolher o atributo de assinatura.

Esta estratégia de distribuição na forma como foi apresentada até este momento apresenta algumas limitações. Por exemplo, atributos que geram poucas assinaturas poderiam fazer com que o algoritmo de particionamento enviasse uma grande quantidade de registros para apenas um *worker*, causando um desbalanceamento de carga no *cluster*. Em contrapartida, atributos que geram uma grande quantidade de assinaturas diferentes também aumentariam a replicação de registros e consequentemente tornaria o tráfego na rede muito maior durante a fase de agrupamento, também conhecida como *flush*. Este problema não é exclusivo da área de identificação de duplicatas, a busca pelo correto balanceamento de carga de processamento é um desafio para a própria área de processamento distribuído de dados. A Figura 4.2 demostra como uma carga de processamento desbalanceada pode afetar negativamente o tempo de processamento. Note que o *worker 2* recebeu uma partição de registros muito maior que os outros e o tempo total de execução foi determinado justamente por ele. Mais adiante será demostra a solução que adotamos para mitigar este problema.

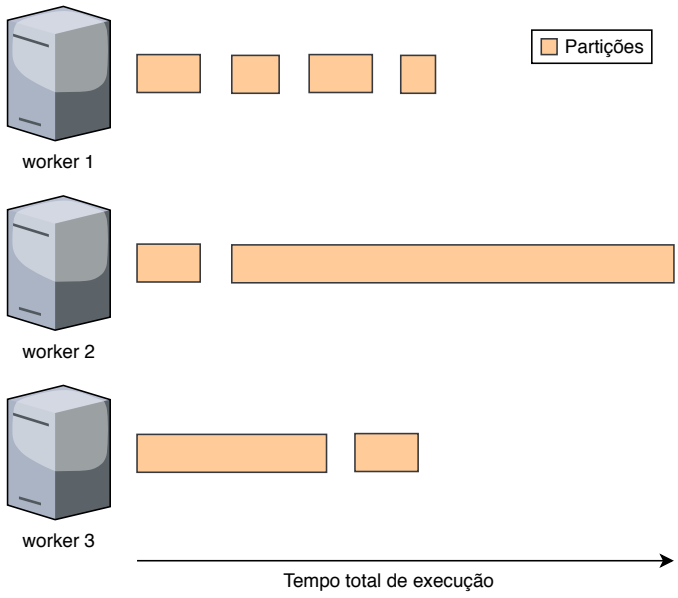


Figura 4.2: Exemplo de balanceamento ineficiente de carga.

4.2 O Algoritmo DSJoin

Como pode ser visto no Algoritmo 4.1, o DSJoin está dividido em duas fases principais de processamento: *particionamento* e *verificação*. Na fase de particionamento (linhas 1–2) cada registro de entrada contido em um RDD é passado para a função

Algoritmo 4.1: Junção por similaridade distribuída sobre múltiplos atributos.

Input: Um RDD R contendo uma coleção de registros, uma expressão de similaridade ϕ e um inteiro i que identifica o atributo de assinatura.
Output: Uma lista S' contendo todos os pares (r, s) tal que $\phi(r, s) = true$.

```
   // Fase de particionamento
1  list(key, r) ← R.flatMap(funcPart(r, i))
2  list(key, list(r)) ← groupByKey(list(key, r))
   // Fase de verificação
3  foreach (key, list(r)) ∈ List(key, list(r)) do
4      S ← flatMap(evalSim(key, list(r)))
   // Extraindo resultado final do RDD
5  S' ← collect(S)
   // Função de particionamento
6  Função funcPart(r, i)
7      foreach key ∈ pref(r.a_i) do
8          list(key, r) ← (key, r)
9      return list(key,r)
   // Função de similaridade
10 Função evalSim(key,list(r))
11     foreach candidate pair(r, s) ∈ list(r) do
12         if φ(r, s) = true then
13             S ← S ∪ ((r, s))
14     return S
```

`funcPart` através da transformação *flatMap*. A função `funcPart` (linhas 6–9) extrai os elementos do prefixo do atributo de assinatura e retorna uma lista de pares `(key, record)` (linhas 7–8) — cada par corresponde a um elemento (*token*) do prefixo do atributo de assinatura como chave e o seu respectivo registro contendo todos os seus atributos. Então, a transformação *groupByKey* é chamada para agrupar todos os registros que possuem uma mesma chave em uma lista.

Na fase de verificação, registros candidatos são comparados sobre os termos de uma expressão de similaridade (linhas 3–4). Para realizar esta tarefa, a função `evalSim` (linhas 10–14) é chamada tendo como parâmetro de entrada uma lista de registros e sua chave comum geradas na etapa anterior. Neste ponto a junção por similaridade é executada de forma distribuída e independente em cada um dos *workers* do *cluster*. Assim como em algoritmos que executam em memória principal, diversas técnicas de filtragem (ver Seção 2.2) podem ser empregadas para reduzir o número de candidatos nesta fase. Além disso é possível utilizar diferentes algoritmos de similaridade como `ppjoin` [32] e `mpjoin` [18] para primeiramente avaliar o predicado de similaridade do atributo de assinatura e em seguida avaliar o restante da expressão de similaridade em cada par de candidato que

restar. Finalmente é chamada a ação *collect* para enviar os resultados obtidos para o *driver* da aplicação *Spark* (linha 5).

Da forma como foi apresentado, o algoritmo pode produzir pares de resultados duplicados no resultado final. Por exemplo, dois registros similares que contém mais de um token em comum no prefixo do atributo de assinatura serão enviados para diferentes *workers* e, consequentemente, aparecerão mais de uma vez no resultado final. Este problema pode ser resolvido utilizando a ordenação global dos tokens do conjunto(utilizado pela filtragem por prefixo). Durante a fase de verificação, só serão comparados os pares de candidatos cujo primeiro *token* que possuem em comum for igual a chave (*token*) da partição. Por outro lado, se o *token* em comum aparecer em uma posição maior que a chave então aqueles candidatos também foram enviados para outro *worker* e a verificação pode ser interrompida sem prejuízo ao resultado final.

4.3 Seleção do Atributo de Assinatura

Como demostrado previamente, o algoritmo proposto neste trabalho faz uso do atributo de assinatura para guiar o particionamento dos dados para realização do processamento distribuído, assim como também foi exemplificado como o desempenho da aplicação pode ser negativamente afetado ao se utilizar um particionamento inadequado. Neste contexto a escolha correta do atributo de assinatura se torna crucial para a abordagem que foi adotada, impactando diretamente nos custos de comunicação e computação.

A princípio, diferentes heurísticas podem ser usadas com o propósito de buscar o melhor atributo. Por exemplo, um atributo com *threshold* mais alto dentro do predicado de similaridade poderia ser escolhido pois possivelmente é o mais seletivo ou um atributo que pertença a um domínio cuja *strings* são mais curtas. Além disso, como mostrado em [12], estas heurísticas também podem ser usadas para determinar a ordem de análise do predicado de similaridade. Outra alternativa, mais sofisticada, seria utilizar modelos de predição de desempenho baseados em modelos estatísticos como mostrado em [26]. No entanto, obter as informações necessárias para executar este tipo de abordagem pode ser difícil na prática pois as mesmas dependem fortemente de complexos e abrangentes dados de treino, o que dificultaria ainda mais a execução da aplicação.

Para selecionar o melhor atributo de assinatura este trabalho definiu um novo modelo de custo baseado nas estatísticas de frequência dos tokens do prefixo de um atributo. O desempenho da solução é medida tendo implicitamente como base o impacto do *threshold* utilizado e a distribuição subjacente de dados. A tabela 4.1 demostra resumidamente cada uma destas estatísticas.

Seja T uma sequência de tokens. A frequência dos momentos de T é definida como $F_k = \sum_{t \in T} df(t)^k$, para cada $k \geq 0$, onde $df(t)$ é o número de ocorrências do token

Nome	Descrição
P	Número de partições geradas
Y	Operações de similaridade realizadas
X	Registros enviados pela rede

Tabela 4.1: Estatísticas utilizadas no modelo de custo.

t em T. Assumindo que T contém os tokens de um prefixo derivados de um determinado atributo de assinatura, então nós teremos que $F_0 = P$, $F_1 = X$ e $F_2 \approx Y$ (o número de operações de similaridade para n registros é dado por $\binom{n}{2}$). O custo para realização da junção por similaridade distribuída para um determinado atributo é demostrado na Definição 4.1. Dito isso, neste trabalho o atributo que possuir o menor custo será definido como atributo de assinatura conforme pode ser observado na Definição 4.2.

Definição 4.1 (Custo de uma junção por similaridade distribuída) *Sejam c_{com} e c_{cmp} a unidade de custo para enviar um registro pela rede e a unidade para realizar um cálculo de similaridade, respectivamente. O custo para realizar a junção por similaridade distribuída para para um determinado atributo de assinatura $R.a$ será*

$$C(R.a) = X * c_{com} + Y * c_{cmp}.$$

Definição 4.2 (Política de seleção do atributo de assinatura) *Seja um registro R composto por múltiplos atributos textuais, seu atributo de assinatura será*

$$a = \arg\min_{R.i \in R} C(R.i)$$

Conforme demostrado, o uso dos tokens do prefixo de determinado atributo para particionamento dos dados tem um aspecto fundamental em um ambiente de processamento distribuído. Nas avaliações realizadas o número de partições P alcançou a cada de centenas de milhares em diversos cenários diferentes e portanto muito maior do que o número de *workers* disponíveis em um *cluster* convencional. Como resultado diversas partições são enviadas para cada um mesmo *worker* e por terem tamanho reduzido devido a ordenação inversa de frequência o problema do desbalanceamento de caga também é mitigado. Possuir partições pequenas torna o seu custo computacional bastante baixo em relação ao custo global e por este motivo uma estratégia complexa de balanceamento de carga como a apresentada em [4] não seria interessante pois no contexto deste trabalho se assume que a carga de trabalho será distribuída de maneira uniforme entre os *workers* no arcabouço Spark.

4.4 Outras Estratégias

Aqui serão apresentas outras estratégias para realizar o processamento distribuído da junção por similaridade.

4.4.1 MassJoin

Como demostrado no Capítulo 2, o MassJoin [7] utiliza a distancia *Hamming* como base do processamento da similaridade em detrimento da interseção dos conjuntos, como fazem os algoritmos tradicionais. Além disso, o mesmo ainda propôs um método de geração de assinaturas para particionamento de dados composto por 'seguimentos' de *tokens* adjacentes. Um dos problemas do modelo proposto é a quantidade de assinaturas que são geradas principalmente quando se utiliza thresholds mais baixos (quanto menor mais assinaturas são geradas). Para amenizar esse problema os autores propuseram um modelo semelhante que ignora as informações sobre o posicionamento das assinaturas chamado *MassJoin Merge*, o que gera assinaturas menos seletivas. No entanto, esta solução gera partições arbitrariamente maiores e consequentemente o mesmo problema de balanceamento demostrado na Figura 4.2. Vale realtar que no MassJoin não existe o conceito de ordenação global e por esse motivo não é possível adaptar a mesma estratégia utilizada no atributo de assinaturas para eliminar informações repetidas no conjunto final de resultados.

4.4.2 Assinaturas Compostas

De forma análoga a [12], o particionamento dos dados em situações onde as expressões de similaridade possuem apenas uma cláusula conjuntiva pode ser realizado utilizando um esquema de *assinaturas compostas* pela combinação vários atributos ao invés de apenas um. Considerando um coleção de dados vinculada a um domínio composto por múltiplos atributos, uma assinatura composta é gerada através da combinação dos *tokens* do prefixo de dois ou mais atributos pertencentes a um registro. No exemplo da Figura 4.3 é utilizado o produto cartesiano na geração das assinaturas compostas de um registro de entrada r 4.3(a). Note que em 4.3(b) cada assinatura foi associada a uma cópia do registro r, neste caso representado pelos seus respectivos atributos.

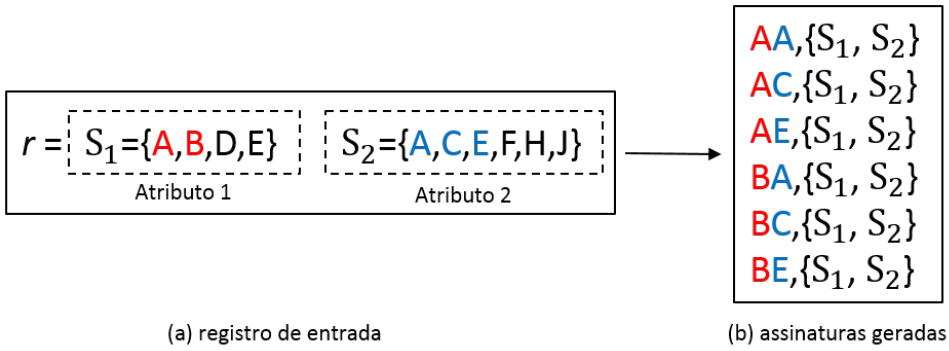


Figura 4.3: Particionamento com assinatura composta.

A principal vantagem da abordagem utilizando uma assinatura composta está na diminuição do custo computacional envolvido, uma vez que dois registros serão comparados apenas se compartilharem ao menos um *token* em todos os seus atributos. Em contrapartida, o total de partições e registros transmitidos pela rede tende aumentar radicalmente em comparação a outros métodos visto que número de assinaturas geradas cresce de forma exponencial de acordo com a quantidade de atributos de cada registro.

CAPÍTULO 5

Experimentos e Resultados

Neste capítulo serão descritos todos os experimentos que foram realizados neste trabalho juntamente com a análise do resultados obtidos. Os testes foram divididos em três partes, cada uma visando diferentes objetivos:

- **Performance e escalabilidade do algoritmo:** testes para avaliar o comportamento do algoritmo em situações com variação do número de *workers*, registros, atributos e *threshold*;
- **Estratégia de particionamento:** comparação entre as diferentes estratégias de particionamento descritas no Capítulo 4;
- **Modelo de custo:** avaliar a eficiência do modelo de custo que foi apresentado para a seleção do atributo de assinatura.

5.1 Ambiente de Testes

Nesta seção será apresentada configuração do ambiente de testes no qual os experimentos deste trabalho foram realizados.

5.1.1 Conjuntos de Dados

Para realização dos experimentos foram utilizados quatro conjuntos de dados (*datasets*) públicos com diferentes tipos de dados do mundo real: o DBLP[1] (*Digital Bibliography & Library Project*) – um repositório bibliográfico de ciência da computação, o IMDB[2] (*Internet Movie Database*) – um banco de dados sobre filmes de todo o mundo, a Wikipédia[3] – uma enciclopédia digital, multi-idioma, pública e colaborativa, e o Discogs[4] (*discographies*) – um banco de dados com informações sobre músicas.

[1] http://dblp.uni-trier.de
[2] http://www.imdb.com
[3] https://pt.wikipedia.org
[4] http://www.discogs.com

Em cada um dos *datasets* apresentados, foram selecionados aleatoriamente registros contendo múltiplos atributos textuais. Em seguida um determinado número duplicatas chamadas de "cópias sujas"foram geradas para cada um destes registros, contendo inserções, exclusões e substituições aleatórias de caracteres. A Tabela 5.1 mostra de forma resumida as informações sobre cada *dataset*: a quantidade de registros extraída, a quantidade de duplicatas geradas e o total de registros que foi avaliada como padrão nos testes. Note que foi adotado um abreviação para cada um dos *datasets* com a finalidade de se padronizar os gráficos e relatórios que foram gerados.

Datasets	**Abreviação**	**Tipo**	**Registros**	**Duplicatas**	**Total**
DBLP	DBLP	Bibliografia	250K	4	1.25M
IMDB	IMDB	Filmes	30K	10	300K
Wikipédia	WIKI	Enciclopédia	180K	5	900K
Discogs	DISC	Músicas	1M	2	2M

Tabela 5.1: Informações sobre os *datasets* analisados.

A Tabela 5.2 apresenta mais detalhes sobres as características de cada um dos atributos escolhidos dos *datasets*. Para realização dos testes optou-se por usar a quantidade máxima de atributos disponíveis como padrão.

	Atributo	Tam. Max	Tam. Méd
DBLP	title	340	79
	journal	78	22
	1st author	41	13
	2nd author	44	14
	3rd author	55	14

	Atributo	Tam. Max	Tam. Méd
IMDB	title	222	16
	actor	51	14
	distributor	108	24
	director	39	14
	producer	39	12

	Atributo	Tam. Max	Média
WIKI	title	240	51
	username	55	9
	comment	301	78

	Atributo	Tam. Max	Tam. Méd
DISC	title	676	19
	name	623	12
	1st song	927	17
	2nd song	2689	17
	3rd song	1236	17

Tabela 5.2: Detalhes sobre as strings de cada atributo.

5.1.2 Pré-processamento e Valores Padrão

Antes de iniciar o processamento, com a finalidade de homogenizar os dados, todas as *strings* foram convertidas para para caixa-alta, assim como os espaços repetidos e caracteres inválidos foram eliminados. Em seguida foram gerados os respectivos conjuntos para cada atributo que compunha cada registro utilizando tokenização com *q-grams* de tamanho três.

Na fase de verificação apenas um predicado de similaridade baseado na função *Jaccard* foi especificado para cada atributo e o valor 0.85 foi escolhido como *threshold* padrão. A quantidade de registros analisada é a mesma mostrada na Tabela 5.1, assim como sempre são utilizados 7 *workers* durante o processamento. A expressão de similaridade resultante consiste em uma única cláusula conjuntiva envolvendo todos os atributos disponíveis. Os valores $C_{com} = 2$ e $C_{cmp} = 1$ foram escolhidos para testar o modelo de custo que foi proposto.

O atributo de assinatura com o menor custo estimado foi o escolhido para guiar o particionamento de dados em todos os cenários de teste em que poderia ser aplicado. Ao menos três execuções isoladas foram realizadas para cada tipo de teste de modo que o tempo geral que será mostrado nas seções seguintes foi obtido através da média aritmética dos tempos atingidos em cada uma delas.

5.1.3 Hardware e Software

O *cluster* em que os testes foram executados possui oito nós, um destes desempenhando a função de *master* e os 7 restantes a função de *worker*. Todos os nós possuem a mesma configuração de *hardware* mostrada na Tabela 5.3.

Hardware	Configuração
Processador	Intel Xeon W3565 3.20 GHz
Quantidade de *Cores*	4
Cache	8 MB
Memória Principal	8 GB
Memória Secundária	512 GB

Tabela 5.3: Hardware utilizado nos experimentos.

O HDFS foi utilizado como sistema de arquivos distribuído e enquanto a gerência dos recursos computacionais ficou a cargo do *Yarn cluster manager* (configurado juntamente com o recurso Dynamic Resource Allocation[5]). As versões das linguagens de programação, arcabouços de processamento distribuído e de outros softwares utilizados são mostrados na Tabela 5.4.

[5]https://spark.apache.org/docs/latest/job-scheduling.html

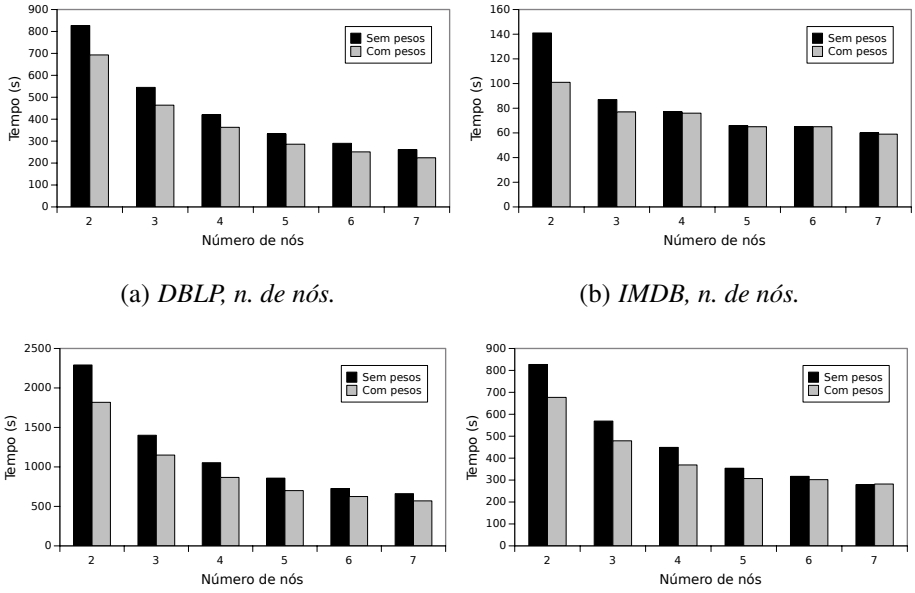
Software	Versão
Oracle Java	8 *update* 162
Scala	2.11
Apache Spark	2.1.1
Apache Hadoop	2.7.3
Ubuntu Server	16.04 LTS

Tabela 5.4: Software utilizado nos experimentos.

5.2 Avaliação de Desempenho e Escalabilidade

Com objetivo de testar o desempenho e a escalabilidade, tendo sempre como base as configurações que foram detalhadas previamente, o algoritmo DSJoin foi submetido a diversos testes que simulam diferentes cenários reais de processamento.

O primeiro deles, mostrado na Figura 5.1, verifica o desempenho variando-se a quantidade de nós de processamento, um teste muito importante visto que nem sempre é possível contar com grandes *clusters* em situações do cotidiano. Em todos os *datasets* analisados percebe-se uma consistente diminuição no tempo de execução a medida que mais nós são alocados para realizar o processamento. No entanto, a partir de certo ponto, o ganho de desempenho já não é mais tão acentuado. Esta atuação é esperada já que o custo de comunicação tende a ser maior a medida que mais nós são adicionados. Note que o comportamento é mesmo para conjuntos com e sem pesos.

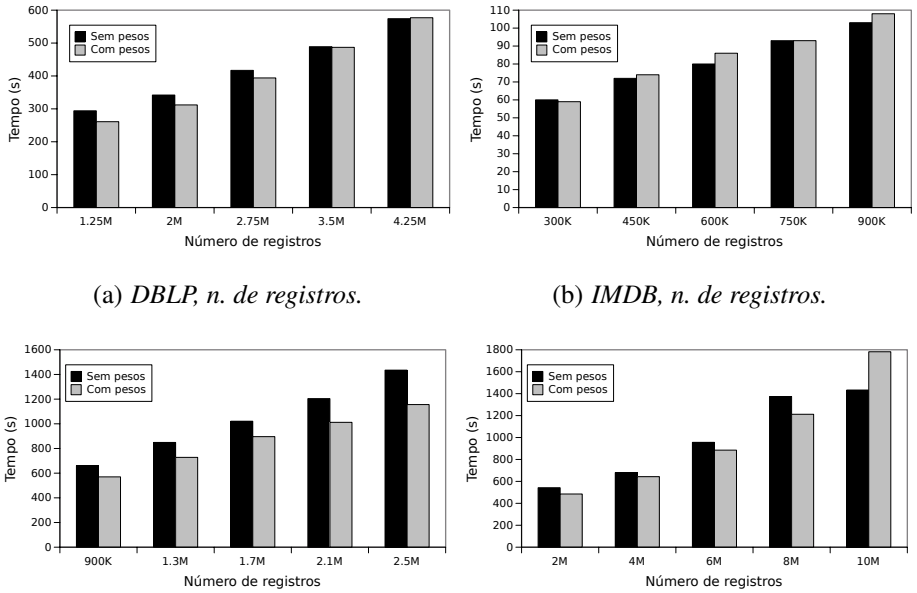


(a) *DBLP, n. de nós.* (b) *IMDB, n. de nós.*

(c) *WIKI, n. de nós.* (d) *DISC, n. de nós.*

Figura 5.1: Resultados obtidos ao variar a quantidade de nós de processamento.

No teste seguinte, exibido na Figura 5.2, foi analisado o comportamento apresentado ao se aumentar a quantidade de registros a ser processada. Neste caso o DSJoin apresentou boa escalabilidade pois em todos os *datasets* (considerando conjuntos com e sem pesos) nota-se que o tempo total de execução cresceu de forma praticamente linear em relação a quantidade de registros avaliada.



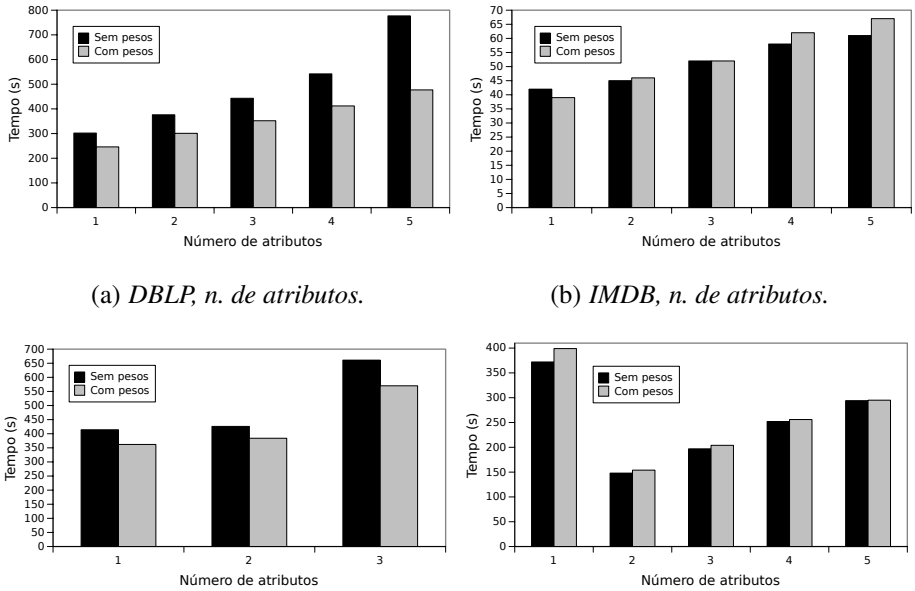
(a) *DBLP, n. de registros.* (b) *IMDB, n. de registros.*

(c) *WIKI, n. de registros.* (d) *DISC, n. de registros.*

Figura 5.2: Resultados obtidos ao aumentar a quantidade de registros a ser processada.

Expressões de similaridade contendo uma quantidade maior de atributos tendem a ter o tempo de execução maior (seja utilizando conjuntos com pesos e sem pesos), conforme mostrado nas Figuras 5.3(a), 5.3(b) e 5.3(c). Este resultado também é esperado pois mais tempo será necessário para avaliar todos os predicados de similaridade na fase de verificação.

Contudo, vale ressaltar aqui um resultado não esperado obtido na Figura 5.3(d): ao se utilizar apenas um atributo o tempo foi consideravelmente maior que todos os outros para o mesmo *dataset*. Neste caso a anormalidade se deve ao fato de o conjunto de resultados para este atributo ser muito maior que os demais (320MB contra uma média de 26MB ao se considerar mais de um atributo), o que causou aumento excessivo do tempo de execução.



(a) *DBLP, n. de atributos.* (b) *IMDB, n. de atributos.*

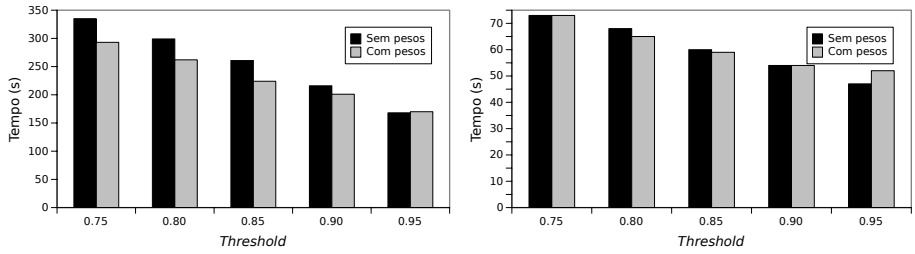
(c) *WIKI, n. de atributos.* (d) *DISC, n. de atributos.*

Figura 5.3: Resultados com diferentes quantidades de atributos.

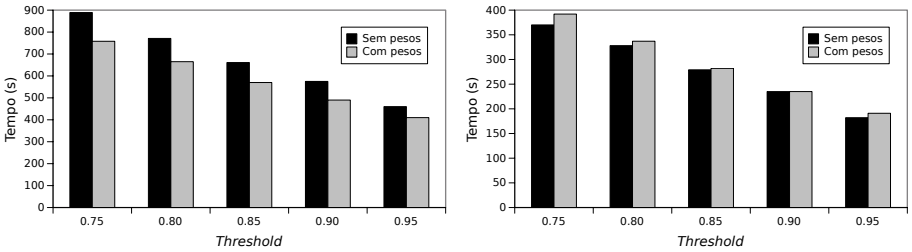
Finalmente, na Figura 5.4, é demostrado o comportamento do DSJoin variando-se o *threshold*. Novamente o resultado obtido para todos os *datasets* e considerando conjuntos com peso e sem peso foi o esperado, devendo ressaltar-se ainda a boa escalabilidade apresentada. O comportamento obtido deve-se ao fato de que *thresholds* maiores geram prefixos menores, o que diminui o custo de comunicação e computação.

De forma geral é possível notar que o desempenho (*speed-up*) para o *dataset* WIKI foi o pior em todos os cenários analisados. Neste caso, o comportamento apresentado se deve a grande quantidade de registros duplicados que a base já possuía inicialmente (antes da geração das cópias sujas), o que acarreta em uma explosão no número de resultados encontrados e consequentemente aumenta o tempo de execução.

Também vale resultar aqui o desempenho superior que foi observado na maioria dos testes realizados ao se utilizar conjuntos com pesos, em especial nos *datasets* maiores (tanto em quantidade de registros quanto em tamanho médio das *strings*). Este é um comportamento esperado e se deve ao fato de conjuntos sem pesos gerarem prefixos maiores pois os mesmos dependem diretamente do tamanho dos conjuntos analisados. Já nos conjuntos com pesos o tamanho do prefixo é dado pelo somatórios dos pesos dos elementos do prefixo. Como a estratégia particionamento baseada em atributo de assinatura depende diretamente do tamanho do prefixo, os conjuntos com pesos tendem a gerar uma menor caga de comunicação e de processamento.



(a) *DBLP, threshold.* (b) *IMDB, threshold.*



(c) *WIKI, threshold.* (d) *DISC, threshold.*

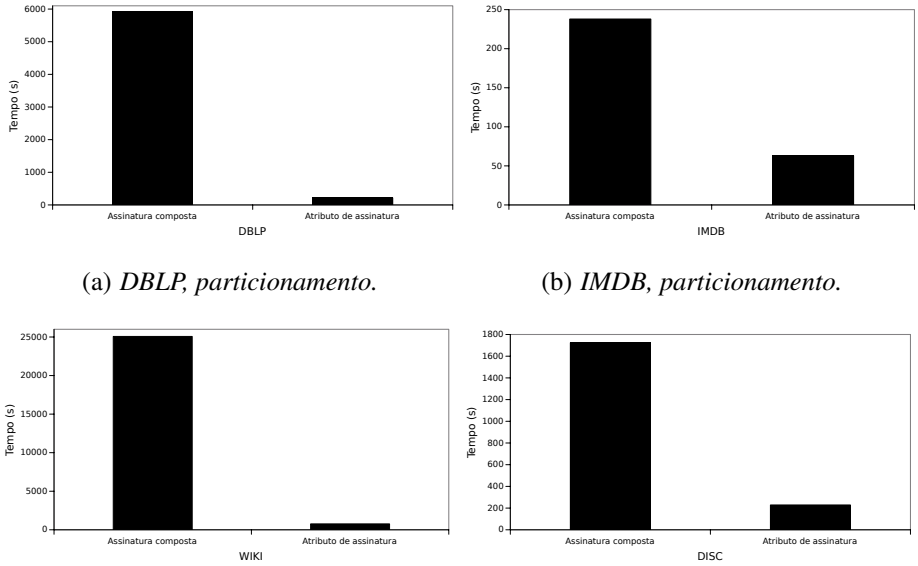
Figura 5.4: Resultados variando-se o *threshold*.

5.3 Estratégia de Particionamento

Após a avaliação do desempenho e da escalabilidade do DSJoin proseguiu-se com os testes para avaliar o desempenho da estratégia de particionamento utilizando atributo de assinatura. Para realizar a comparação de foram escolhidos as estratégias com assinatura composta [12] e a do MassJoin [7].

5.3.1 Particionamento com Assinatura Composta

Para realização deste experimento apenas os três primeiros atributos de cada *dataset* foram utilizados. Como a assinatura composta é formada pelo produto cartesiano dos tokens do prefixo de cada conjunto, um teste utilizando apenas um atributo seria equivalente ao particionamento com atributo de assinatura. Em contrapartida, a medida que mais conjuntos forem sendo utilizados a quantidade de assinaturas crescerá de forma exponencial. Adicionalmente, com objetivo de diminuir a carga de comunicação, foi realizada uma etapa extra após que geração de todas as assinaturas que consiste em eliminar partições contendo apenas um registro.



(a) *DBLP, particionamento.* (b) *IMDB, particionamento.*

(c) *WIKI, particionamento.* (d) *DISC, particionamento.*

Figura 5.5: Comparando a estratégia de particionamento com assinatura composta.

Conforme fica evidenciado nos resultados mostrados na Figura 5.5, o particionamento utilizando o atributo de assinatura tem desempenho muito superior em todos os *datasets* analisados. A utilização da assinatura composta levou a uma explosão de registros transmitidos pela rede, o que não foi compensado pela sua vantagem em permitir uma quantidade menor de avaliações de similaridade na fase de verificação.

5.3.2 Particionamento com MassJoin

O *dataset* DBLP foi escolhido para realizar a comparação com a estratégia de particionamento proposta pelo MassJoin. Adicionalmente, foi avaliado também o algoritmo derivado MassJoin Merged. Optou-se por realizar um comparativo variando-se o valor do *threshold* pois neste tipo de abordagem a quantidade de assinaturas gerada depende fortemente do mesmo. Na Figura 5.6 é possível observar o resultado obtido.

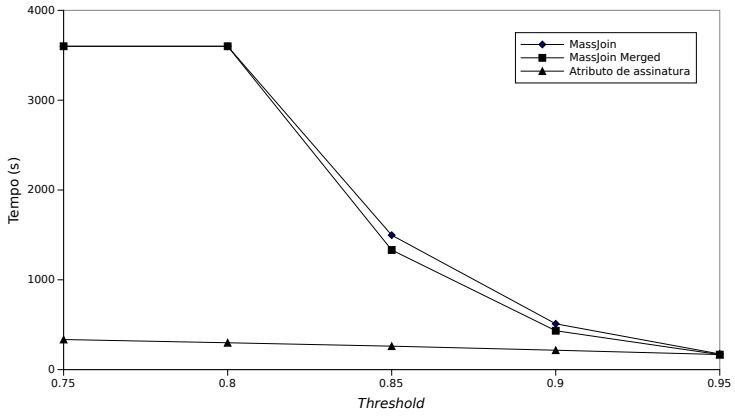


Figura 5.6: Comparando a estratégia de particionamento com MassJoin.

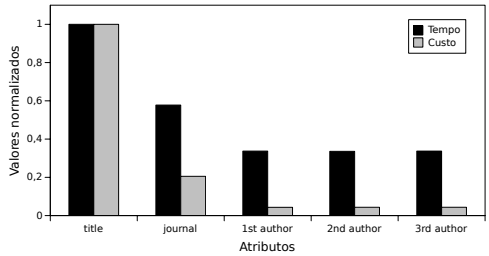
Primeiramente, como o tempo médio para processamento utilizando o atributo de assinatura é menor que cinco minutos, foi fixado um tempo máximo de execução de uma hora sem que isso comprometesse a análise dos dados. Ao se utilizar os valores $0,75$ e $0,8$ para o *threshold*, o tempo máximo foi atingido para ambos os concorrentes e desta forma a sua execução foi interrompida antes da conclusão. Para *threshold* maiores, verificou-se uma acentuada melhora no desempenho (com uma pequena vantagem do MassJoin Merged), ficando muito próximo do atributo de assinatura para o *threshold* 0.95.

A conclusão neste caso, inclusive também observada em [22], é a de que o MassJoin é incapaz de realizar operações de similaridade em grandes *datasets*, especialmente para valores baixos de thresholds. Isso se deve ao fato de ambos os seus métodos gerarem uma grande quantidade de assinaturas, comprometendo totalmente o custo de comunicação envolvido.

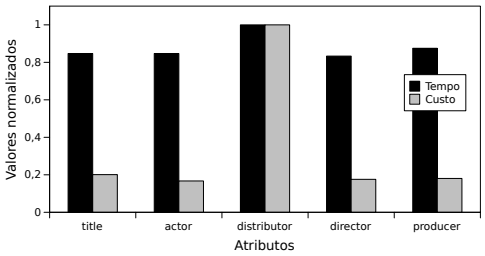
5.4 Modelo de Custo

Finalmente, este último experimento avaliou a eficiência do modelo de custo proposto para seleção do atributo de assinatura usando o filtro por prefixo. Na Figura 5.7 é possível observar, em valores normalizados, o tempo de execução juntamente com o custo estimado para todos os atributos de todos os *datasets* analisados. Em todos os casos, o modelo de custo permitiu a identificação do melhor atributo de assinatura para realização do particionamento, uma vez que o custo mostrado segue a mesma tendencia do tempo de execução.

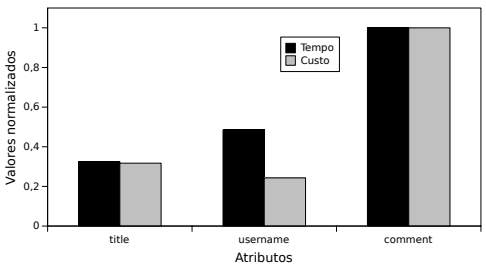
Com este experimento também ficou comprovado que nem sempre atributos compostos por *string* menores levam a melhores tempos de execução ao serem escolhidos como atributo de assinatura. Por exemplo: no *datasets* DISC, o atributo `name` possui a menor media de tamanho de *string* (Tabela 5.2) e mesmo assim obteve o pior tempo de execução (Figura 5.7(d)).



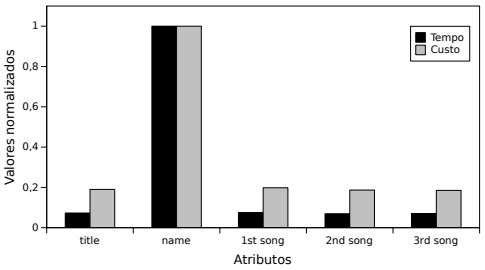
(a) *DBLP, tempo e custo.*



(b) *IMDB, tempo e custo.*



(c) *WIKI, tempo e custo.*



(d) *DISC, tempo e custo.*

Figura 5.7: Avaliação do modelo de custo.

CAPÍTULO 6

Conclusão

Operações de junção por similaridade são indispensáveis em banco de dados modernos e permitir que as mesmas executem sobre os grandes quantidade de dados é fundamental para o sucesso de projetos na área de *Big Data*. Além disso, como os dados atuais são tipicamente compostos por diversos tipos diferentes de informação é relevante considerar a importância de cada uma delas no cálculo de similaridade.

Neste trabalho foi proposto um algoritmo distribuído para processamento de junções por similaridade sobre múltiplos conjuntos, um tipo que particularmente vem recebendo pouca atenção dos trabalhos do estado da arte atual. Para lidar com o particionamento dos dados foi introduzido o conceito de atributo de assinatura: uma estratégia que apesar de simples se mostrou muito eficiente conforme evidenciaram os experimentos. Além disso, para seleção do atributo de assinatura foi definido um modelo de custo associado ao menor tempo de execução geral, que também se mostrou eficiente. Toda a parte de processamento distribuído foi implementada utilizando o arcabouço Spark e utilizando *datasets* com dados do mundo real foram realizados extensivos testes que comprovaram tanto a eficiência quanto a escalabilidade do algoritmo proposto.

Apesar do escopo deste trabalho estar baseado apenas na análise de dados textuais, se for considerado apenas o aspecto de desempenho, os algorítimos e técnicas aqui apresentadas podem ser aplicados em qualquer tipo de dado que possa ser representado na forma de conjuntos.

Em trabalhos futuros pretende-se investigar novos esquemas de particionamento que sejam associados a expressões complexas de similaridade visando sempre os melhores caminhos para realizar o processamento distribuído dos dados focando especialmente na diminuição do custo de comunicação. Também pretende-se realizar um comparativo de desempenho entre os arcabouços Spark e MapReduce para processamento de junções por similaridade baseadas em múltiplos conjuntos.

Em outra linha poderia-se avaliar novos métodos para mitigar o problema de desbalanceamento da carga no cluster durante o processamento distribuído, em particular, encontrar novas formas para diminuir a replicação de registros. Outro foco seria estudar a integração de modelos de processamento distribuído em conjunto com o paralelismo

massivo proporcionado por unidade de processamento gráfico (GPU's).

Finalmente, ressalta-se que os métodos e algoritmos aqui desenvolvidos serviram de fundamento para duas publicações. A primeira, intitulada "Uma Abordagem para Processamento Distribuído de Junção por Similaridade sobre Múltiplos Atributos"[16], publicada no XXXII Simpósio Brasileiro de Banco de Dados (SBBD) e a segunda, "*Set Similarity Joins with Complex Expressions on Distributed Platforms*"[17], publicada na "*Advances in Databases and Information Systems - 22nd European Conference*".

Referências Bibliográficas

[1] ARASU, A.; GANTI, V.; KAUSHIK, R. **Efficient Exact Set-Similarity Joins**. *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[2] BAYARDO, R. J.; MA, Y.; SRIKANT, R. **Scaling up all pairs similarity search**. *Proceedings of the 16th international conference on World Wide Web - WWW '07*, p. 131, 2007.

[3] CHAUDHURI, S.; GANTI, V.; KAUSHIK, R. **A primitive operator for similarity joins in data cleaning**. In: *Proceedings - International Conference on Data Engineering*, volume 2006, p. 5, 2006.

[4] CHU, X.; ILYAS, I. I. F.; KOUTRIS, P. **Distributed Data Deduplication**. *Proceedings of the Very Large Data Bases Endowment*, 9(11):864–875, 2016.

[5] COHEN, W. W.; FIENBERG, S. E.; RAVIKUMAR, P. D.; FIENBERG, S. E. **A Comparison of String Distance Metrics for Name-Matching Tasks**. *Proceedings of IJCAI-03 Workshop on Information Integration on the Web*, 2003.

[6] DEAN, J.; GHEMAWAT, S. **MapReduce: Simplified Data Processing on Large Clusters**. *Proc. of the OSDI - Symp. on Operating Systems Design and Implementation*, p. 137–149, 2004.

[7] DENG, D.; LI, G.; HAO, S.; WANG, J.; FENG, J. **MassJoin: A mapreduce-based method for scalable string similarity joins**. In: *Proceedings - International Conference on Data Engineering*, p. 340–351, 2014.

[8] GRAVANO, L.; IPEIROTIS, P. G.; JAGADISH, H. V.; KOUDAS, N.; MUTHUKRISHNAN, S.; SRIVASTAVA, D. **Approximate String Joins in a Database (Almost) for Free**. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases*, p. 491–500, 2001.

[9] HERNÁNDEZ, M. A.; STOLFO, S. J. **The merge/purge problem for large databases**. *ACM SIGMOD Record*, 1995.

[10] HERNÁNDEZ, M. A.; STOLFO, S. J. **Real-world data is dirty: Data cleansing and the merge/purge problem**. *Data Mining and Knowledge Discovery*, 1998.

[11] KIM, J. **An effective candidate generation method for improving performance of edit similarity query processing**. *Information Systems*, 47:116–128, 2015.

[12] LI, G.; HE, J.; DENG, D.; LI, J. **Efficient Similarity Join and Search on Multi-Attribute Data**. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, p. 1137–1151, 2015.

[13] LOHR, S. **The Age of Big Data**. *The New York Times*, p. 1–5, 2012.

[14] METWALLY, A.; FALOUTSOS, C. **V-SMART-join: A Scalable Mapreduce Framework for All-pair Similarity Joins of Multisets and Vectors**. In: *VLDB*, volume 5, p. 704–715, 2012.

[15] NAVARRO, G. **A guided tour to approximate string matching**. *ACM Computing Surveys*, 2001.

[16] OLIVEIRA, D. J. C.; BORGES, F. F.; RIBEIRO, L. A. **Uma Abordagem para Processamento Distribuído de Junção por Similaridade sobre Múltiplos Atributos**. *XXXII Simpósio Brasileiro de Banco de Dados (SBBD)*, 2017.

[17] OLIVEIRA, D. J. C.; BORGES, F. F.; RIBEIRO, L. A.; CUZZOCREA, A. **Set Similarity Joins with Complex Expressions on Distributed Platforms**. In: *Advances in Databases and Information Systems - 22nd European Conference, {ADBIS} 2018, Budapest, Hungary, September 2-5, 2018, Proceedings*, p. 216–230, 2018.

[18] RIBEIRO, L. A.; HRDER, T. **Generalizing prefix filtering to improve set similarity joins**. In: *Information Systems*, volume 36, p. 62–78, 2011.

[19] RIBEIRO, L. A.; SCHNEIDER, N. C.; INÁCIO, A. D. S.; WAGNER, H. M.; WANGENHEIM, A. V. **Bridging Database Applications and Declarative Similarity Matching**. *Journal of Information and Data Management*, 7(3):217–232, 2016.

[20] RIBEIRO-JÚNIOR, S.; QUIRINO, R. D.; RIBEIRO, L. A.; MARTINS, W. S. **Fast Parallel Set Similarity Joins on Many-core Architectures**. *Journal of Information and Data Management*, 8(3):255–270, 2017.

[21] ROBERTSON, S.; JONES, K. **Relevance weighting of search terms**. *Journal of American Society of Information Science*, 1976.

[22] RONG, C.; LIN, C.; SILVA, Y. N.; WANG, J.; LU, W.; DU, X. **Fast & scalable distributed set similarity joins for big data analytics**. *Proceedings - International Conference on Data Engineering*, p. 1059–1070, 2017.

[23] RONG, C.; LU, W.; WANG, X.; DU, X.; CHEN, Y.; TUNG, A. K. H. **Efficient and scalable processing of string similarity join**. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2217–2230, 2013.

[24] SARAWAGI, S.; SARAWAGI, S.; KIRPAL, A.; KIRPAL, A. **Efficient set joins on similarity predicates**. *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004.

[25] SHI, J.; QIU, Y.; MINHAS, U.; JIAO, L.; WANG, C.; REINWALD, B.; ÖZCAN, F. **Clash of the titans: Mapreduce vs. spark for large scale data analytics**. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[26] SIDNEY, C. F.; MENDES, D. S.; RIBEIRO, L. A.; HÄRDER, T. **Performance prediction for set similarity joins**. In: *SAC '15 Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, p. 967–972, 2015.

[27] UKKONEN, E. **Approximate string-matching with q-grams and maximal matches**. *Theoretical Computer Science*, 1992.

[28] VERNICA, R.; CAREY, M. J.; LI, C. **Efficient Parallel Set-similarity Joins Using MapReduce**. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, p. 495–506, 2010.

[29] 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 SIGMOD Conference*, p. 85–96, 2012.

[30] WINKLER, W. E. **The State of Record Linkage and Current Research Problems**. *Statistical Research Division US Census Bureau*, 1999.

[31] XIAO, C.; WANG, W.; LIN, X. **Ed-Join : An Efficient Algorithm for Similarity Joins With Edit Distance Constraints**. *Proceedings of the VLDB Endowment*, 1(1):933–944, 2008.

[32] XIAO, C.; WANG, W.; LIN, X.; YU, J. X.; WANG, G. **Efficient similarity joins for near-duplicate detection**. *ACM Transactions on Database Systems*, 2011.

[33] ZAHARIA, M.; CHOWDHURY, M.; DAS, T.; DAVE, A. **Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing**. *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, p. 2–2, 2012.