Uma Abordagem para Processamento Distribuído de Junção por Similaridade sobre Múltiplos Atributos

Diego Junior do Carmo Oliveira[1], **Felipe Ferreira Borges**[1], **Leonardo Andrade Ribeiro**[1]

[1]Instituto de Informática
Universidade Federal de Goiás (UFG) – Goiânia – GO – Brazil

{diegooliveira,felipeferreiraborges,laribeiro}@inf.ufg.br

Abstract. *Similarity join is a fundamental operation in data integration. Most existing algorithms consider single-attribute data. However, real data is typically multi-attribute. Besides requiring more complex similarity expressions, this type of data is larger and, therefore, processing cost on a single machine can be prohibitively expensive. This paper presents a distributed similarity join algorithm on multi-attribute data using Spark. Initial experimental results show that the proposed approach is efficient and scalable.*

Resumo. *Junção por similaridade é uma operação fundamental em integração dados. Algoritmos existentes assumem, em sua ampla maioria, dados representados por um único atributo. Contudo, dados reais são tipicamente compostos por múltiplos atributos. Além de demandar expressões de similaridade mais complexas, dados desse tipo são mais volumosos e, com isso, o custo de processamento em um único computador pode tornar-se proibitivo. Este artigo apresenta um algoritmo distribuído de junção por similaridade sobre múltiplos atributos usando a plataforma Spark. Resultados experimentais iniciais mostram que a abordagem proposta é eficiente e escalável.*

1. Introdução

Sistemas de integração de dados frequentemente requerem que registros sejam comparados em termos de sua similaridade. Por exemplo, quando fontes de dados independentes, mas contendo informações em comum são integradas, representações divergentes de um mesmo objeto poderão estar presentes no banco de dados consolidado. A presença desse tipo de redundância pode causar inúmeros problemas em diversos cenários de aplicação, do reenvio desnecessário de correspondências até a corrupção de modelos de mineração de dados. A comparação baseada em similaridade é imprescindível para identificação desses dados duplicados porque os mesmos não são cópias idênticas entre si.

Junção por similaridade é uma operação fundamental nesse contexto, pois a mesma retorna todos os pares de objetos similares em uma coleção. Dois objetos são considerados similares se o valor retornado por uma função de similaridade para os mesmos for maior ou igual a um limiar predefinido. Para dados representados por atributos textuais, um tipo popular de junção por similaridade primeiramente converte o texto de cada objeto em um conjunto e, posteriormente, a similaridade entre pares de objetos é determinada pela interseção dos conjuntos correspondentes [Chaudhuri et al. 2006, Ribeiro and Härder 2011].

Tabela 1. Registros com informações sobre pessoas.

Pessoa ID	RID	Nome	Via	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

A ampla maioria dos algoritmos de junção por similaridade propostos consideram objetos representados por apenas um atributo. Entretanto, dados reais são tipicamente compostos por múltiplos atributos. Algoritmos tradicionais ainda podem ser empregados nesta situação usando um atributo ou a concatenação de múltiplos atributos na representação dos objetos. Infelizmente, ambas abordagens possuem sérias limitações. Por exemplo, considere os registros sobre pessoas ilustrados na Tabela 1. Os quatro registros representam três pessoas distintas, pois registros 1 e 2 referem-se a mesma pessoa. Caso seja empregado apenas o atributo `Nome`, apenas o par formado por registros 3 e 4 poderia ser retornado, pois registros 1 e 2 possuem textos diferentes devido à abreviação no último. Caso todos atributos sejam concatenados, o par formado por registros 1 e 4 poderia ser retornado devido à similaridade entre os valores nos atributos `Nome` e `Via`.

Uma abordagem mais interessante seria realizar a avaliação de similaridade em cada atributo isoladamente. Mais ainda, diferentes funções e limiares de similaridade podem ser definidos, possibilitando a composição de expressões de similaridade mais sofisticadas. O principal desafio é incorporar tais modificações em uma junção por similaridade sem sacrificar desempenho e escalabilidade. O emprego de expressões de similaridade complexas torna a comparação entre registros mais custosa computacionalmente, além de que dados representados por múltiplos atributos são naturalmente mais volumosos.

Este artigo apresenta um algoritmo distribuído de junção por similaridade sobre múltiplos atributos. Até o momento, não se tem conhecimento de trabalhos anteriores que investigaram o processamento desse tipo de junção por similaridade em um ambiente distribuído. A abordagem generaliza algoritmos existentes para possibilitar o emprego de expressões de similaridade baseadas em fórmulas Booleanas arbitrariamente complexas. Uma estratégia simples, mas eficiente de particionamento de dados é adotada e implementada em *Apache Spark* [Zaharia et al. 2016]. Resultados iniciais mostram que a solução proposta é eficiente e escalável quando aplicada em grandes volumes de dados.

2. Conceitos Básicos e Trabalhos Relacionados

Seja R uma coleção de registros representados por uma lista de atributos textuais $\mathcal{A} = (a_1, ..., a_k)$. $R.a$ denota o domínio do atributo a em R e $r.a$ denota o valor do atributo a em um registro r. Seja $f : R.a \times R.a \rightarrow [0, 1]$ uma função de similaridade que, dados dois valores $r.a$ e $s.a$ obtidos dos registros r e s, retorna um valor entre 0 e 1, e quanto maior o valor, maior a similaridade. Seja p um predicado de similaridade da forma $f\,(r.a, s.a) \geq \alpha$, onde f é uma função de similaridade e $\alpha \in [0, 1]$ é um limiar de similaridade. Seja $\phi : R \times R \rightarrow \{true, false\}$ uma expressão de similaridade representada por uma fórmula em forma normal conjuntiva contendo um conjunto de cláusulas $C = \{c_1, ..., c_t\}$, cujos literais são definidos por predicados de similaridade e literais negados não são permitidos, isto é, $\phi = \bigwedge_{c \in C} \bigvee_{p \in c}$.

Definição 1 (Junção por similaridade sobre múltiplos atributos)**.** *Seja R uma coleção de registros e ϕ uma expressão de similaridade. Um junção por similaridade sobre múltiplos atributos retorna todos os pares de registros $(r, s) \in R \times R$ tal que $\phi(r, s) = true$.*

O foco deste artigo são funções de similaridade baseadas em conjuntos. Com isso, o cálculo de similaridade entre dois valores textuais requer que os mesmos sejam representados por conjuntos. Existem diversas maneiras de se mapear texto para conjunto. Uma abordagem comum é coletar todas as sequências de caracteres de tamanho *q* de um texto; essas sequências são chamadas de *q-grams*. Por exemplo, o texto *"similar"* pode ser mapeado para o conjunto de *3-grams* {*'sim', 'imi', 'mil', 'ila', 'lar'*}. A função de similaridade *Jaccard* é popularmente usada para determinar a similaridade entre conjuntos. Dados dois conjuntos x e y, a similaridade *Jaccard* entre os mesmos é dada por $Jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$. Outras funções de similaridade com as mesmas características que Jaccard são *Dice* e *Cosine* [Ribeiro and Härder 2011].

A representação baseada em conjuntos permite a aplicação de diversas otimizações. Uma das técnicas mais populares e efetivas é a chamada *filtragem por prefixo* [Chaudhuri et al. 2006]. Os elementos de todos conjuntos são ordenados de acordo com uma ordem global. Dado um conjunto x e um limiar α, seja $pref(x) \subseteq x$ o subconjunto de x composto por seus primeiros $|x| - \lceil |x| \times \alpha \rceil + 1$ elementos. É possível demonstrar que se $Jaccard(x, y) \geq \alpha$, então $pref(x) \cap pref(y) \neq \emptyset$. A filtragem por prefixo permite descartar conjuntos dissimilares verificando apenas uma fração dos mesmos.

O trabalho em [Vernica et al. 2010] explora prefixos para guiar o particionamento de dados em um algoritmo distribuído de junção por similaridade. Outra estratégia de particionamento baseada na diferença simétrica entre conjuntos é apresentada em [Deng et al. 2014]. Ambos trabalhos consideram registros compostos por apenas um atributo. O único trabalho conhecido até o momento para junções por similaridade sobre múltiplos atributos é apresentado em [Li et al. 2015]. O algoritmo proposto é estritamente sequencial, não-distribuído e suporta apenas expressões de similaridade conjuntivas. Em contraste, este trabalho apresenta um algoritmo distribuído que permite expressões de similaridade mais complexas. Após a conversão dos valores textuais em conjuntos, um registro r será representado por uma família de conjuntos. Por simplicidade, o restante deste artigo usará o termo registro para referir-se à família de conjuntos correspondente.

3. Junção por Similaridade Distribuída sobre Múltiplos Atributos

De maneira análoga a trabalhos anteriores [Vernica et al. 2010, Deng et al. 2014], a estratégia geral consiste em derivar um conjunto *assinaturas* de cada registro e usar essas assinaturas para guiar o particionamento de dados. Registros com uma assinatura em comum são enviados para um mesmo processador virtual (*worker*), onde a expressão de similaridade será avalidada entre os registros, ao passo que registros que não possuem uma assinatura em comum jamais serão comparados. Note que, como os processadores executam em paralelo, o tempo total de execução é determinado pelo maior tempo de comunicação e computação entre todos os *processadores*. Uma questão central no contexto deste trabalho é a geração de registros para múltiplos atributos. Para este fim, um atributo é selecionado como *chave de assinatura* e as assinaturas são produzidas a partir do conjunto correspondente. Aqui será adotado o esquema proposto em [Vernica et al. 2010], onde cada elemento do prefixo corresponde a uma assinatura;

Algoritmo 1: Junção por Similaridade Baseada em Múltiplos Conjuntos

Entrada: Uma coleção de registros R, uma expressão de similaridade ϕ, um inteiro i identificando a chave de assinatura.

Saída: Um conjunto S contendo todos os pares (r, s) tal que $\phi(r, s) = true$.

```
// Fase de particionamento
```

1 $flatMap\,(r)$ **para cada** $t \in pref\,(r.a_i)$ **faça**

2 $\quad (Key, lista\,(r)) \leftarrow groupByKey\,(Key, r)$

```
// Fase de verificação
```

3 $flatMap\,(Key, lista\,(r))$ **para cada** $candidatos \in lista_{(r)}$ **faça**

4 $\quad$ **se** $\phi\,(r, s) = true$ **então**

5 $\quad\quad S' \leftarrow S' \cup (candidatos)$

```
// Extrair resultado final
```

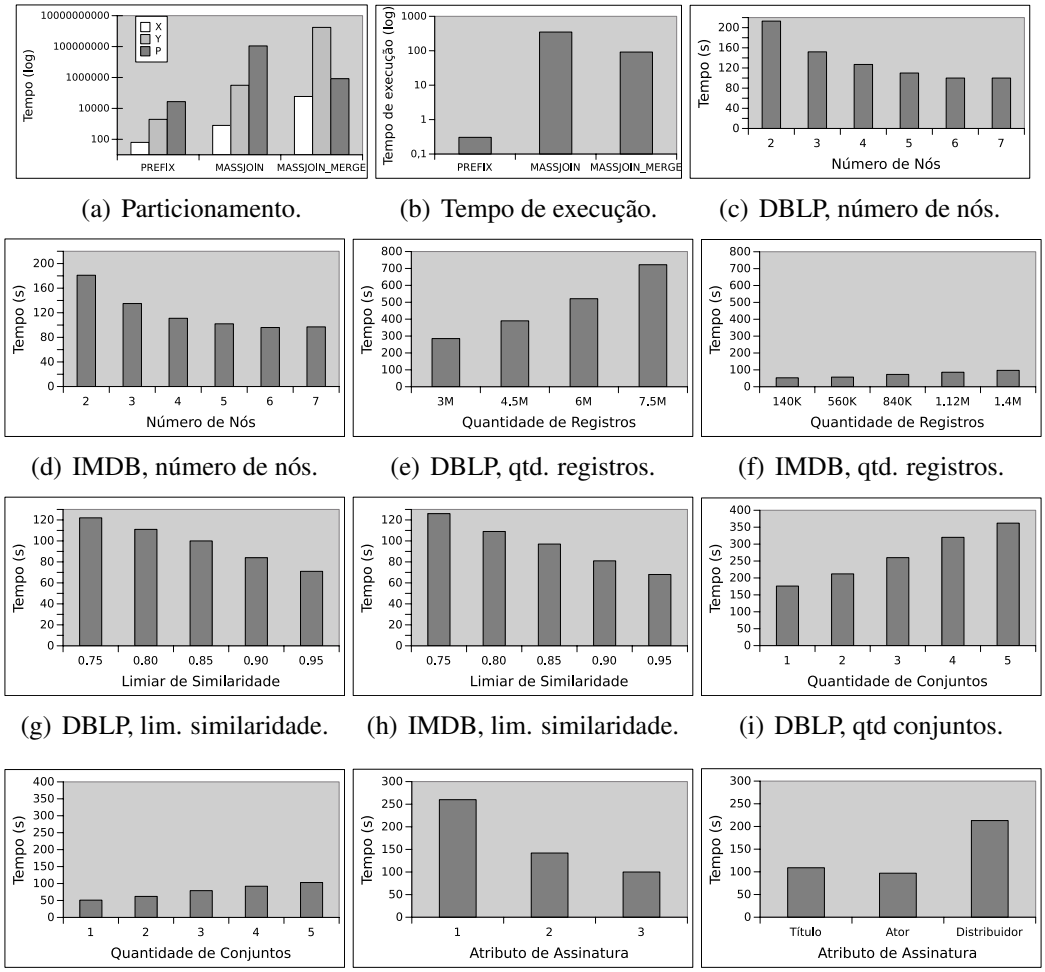
6 $S \leftarrow collect\,(S')$

uma comparação entre essa estratégia e o esquema em [Deng et al. 2014] é apresentada na Seção 4. A seleção da chave de assinatura deve minimizar o custo de máximo de comunicação e computação. Várias heurísticas podem ser usadas para este fim, por exemplo, pode-se escolher o atributo envolvido no predicado com o maior limiar de similaridade (e, portanto, possivelmente mais seletivo) ou contendo textos mais curtos. Outra opção, mais sofisticada, consiste na construção de modelos de predição de desempenho baseado em estatísticas [Sidney et al. 2015].

O Algoritmo 1 descreve os passos da abordagem proposta usando a plataforma *Spark*. O mesmo possui duas etapas: particionamento e verificação. Na etapa de particionamento, os elementos (*q-grams*) presentes no prefixo da chave de assinatura são usados para particionar os registros entre os processadores (linhas 1-2). Na segunda fase, registros associados a uma mesma assinatura são comparados em termos da expressão de similaridade (linhas 3-4). Como apresentado, o algoritmo pode produzir pares duplicados no resultado: dois registros similares podem conter mais de um *q-gram* em comum em seus prefixos e, com isso, estes registros serão enviados para diferentes processadores. Esse problema é evitado explorando a ordem global definida sobre os *q-grams* da chave de assinatura. A verificação de um par de registros avalia primeiramente o predicado associado à chave de assinatura. Caso o primeiro *q-gram* em comum encontrado for menor que a chave associada ao processador, então sabe-se que este par também foi enviado para outro processador e a avaliação pode ser seguramente interrompida.

4. Experimentos

As bases de dados usadas nos experimentos foram DBLP e IMDB, que contém artigos científicos e informações sobre filmes, respectivamente. Para o DBLP, foram usados os atributos título do artigo, nome da conferência, e nomes do primeiro autor e mais dois autores aleatórios. Para o IMDB, foram usados os atributos título, ator, distribuidor, diretor e produtor. Para cada registro extraído, foram criadas cópias sujas contendo $[1, 3]$ modificações textuais aleatórias. Os valores textuais foram mapeados para conjuntos de *3-grams*. Foram consideradas apenas expressões conjuntivas com um predicado baseado em Jaccard para cada atributo; o valor padrão do limiar de similaridade é 0.85.



(a) Particionamento. (b) Tempo de execução. (c) DBLP, número de nós.

(d) IMDB, número de nós. (e) DBLP, qtd. registros. (f) IMDB, qtd. registros.

(g) DBLP, lim. similaridade. (h) IMDB, lim. similaridade. (i) DBLP, qtd conjuntos.

(j) IMDB, qtd conjuntos. (k) DBLP, atr. assinatura. (l) IMDB, atr. assinatura.

Figura 1. Resultados dos experimentos.

O algoritmo foi implementado usando *Oracle Java 8, Scala 2.11 e Spark 2.1.1*. O *cluster* utilizado é composto por um mestre e sete escravos, todos eles equipados com processador *Intel Xeon W3565 quad-core com 3.2GHz, 8MB de cache CPU*, 8GB de memória principal, sistema operacional *Ubuntu 16.04 LTS* e recursos computacionais distribuídos gerenciados pelo *Yarn* baseado no *Hadoop 2.7.3*.

Inicialmente foram realizados testes para comparar os esquemas de assinaturas baseados no filtro de prefixo [Vernica et al. 2010] e os esquemas propostos em [Deng et al. 2014] para os algoritmos *MassJoin* e *MassJoin Merge*. Para este experimento, foram utilizados 10k registros do DBLP, contendo apenas um atributo formado pela concatenação de título e autor. Figura 1(a) mostra o X (tempo máximo de comunicação na rede), o Y (tempo máximo de processamento) e a quantidade de partições geradas P; note que os resultados estão em escala logarítmica. Pode-se observar uma grande superioridade da abordagem mais simples baseada em filtro de prefixo. O tempo total para geração de assinaturas é exibido na Figura 1(b), onde nota-se que o *MassJoin* é significativamente mais custoso computacionalmente. Deve-se ressaltar, contudo, que o *MassJoin* e *MassJoin Merge* foram implementados utilizando tipos inteiros de precisão arbitrária, que por sua vez têm desempenho significativamente inferior a tipos primitivos.

Figuras 1(c) e 1(d) reportam resultados para uma quantidade crescente de processadores. Nota-se que ao alocar mais processadores para a execução do algoritmo, em ambas as bases de dados, o tempo de execução tende a melhorar inicialmente, mas em certo ponto estabiliza e não diminui mais. Isso ocorre pois, em certo ponto, o trabalho gasto com comunicação e gerenciamento distribuído tende a influenciar, até mesmo negativamente, no resultado final. Nas Figuras 1(e), 1(f), 1(i) e 1(j) observa-se que ao se aumentar a quantidade de registros ou de conjuntos a serem processados, naturalmente o tempo de execução também aumentará. Entretanto, pode-se destacar aqui a escalabilidade da solução ao prover um aumento linear do tempo de execução. Como pode ser observado nas Figuras 1(g) e 1(h), a medida que o limiar de similaridade aumenta, o tempo de execução diminui. Isso ocorre pois prefixos menores são gerados e, por consequência, partições menores são produzidas. Figuras 1(i) e 1(j) mostram que o aumento da quantidade de atributos produz um aumento no tempo de processamento, como esperado. Finalmente, Figuras 1(k) e 1(l) mostram os resultados obtidos ao mudar o atributo de assinatura (nos experimentos anteriores fora utilizado o atributo de assinatura de melhor desempenho). Pode-se observar que o atributo de assinatura possui influência significativa no tempo de execução do algoritmo.

5. Conclusões e Trabalhos Futuros

Este artigo apresenta um algoritmo distribuído para junções por similaridade sobre múltiplos atributos utilizando a plataforma *Spark*. Em contraste com trabalhos anteriores, a abordagem proposta permite o emprego de expressões de similaridade complexas. Resultados iniciais são promissores, indicando bom desempenho em todos os cenários analisados. Trabalhos futuros serão concentrados em melhorar particionamento de dados para expressões complexas e explorar recursos de compartilhamento de dados do *Spark*.

Agradecimento Este trabalho foi parcialmente financiado pela CAPES.

Referências

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*, page 5.

Deng, D., Li, G., Hao, S., Wang, J., and Feng, J. (2014). MassJoin: A Mapreduce-based Method for Scalable String Similarity Joins. In *ICDE*, pages 340–351.

Li, G., He, J., Deng, D., and Li, J. (2015). Efficient Similarity Join and Search on Multi-Attribute Data. In *SIGMOD*, pages 1137–1151.

Ribeiro, L. A. and Härder, T. (2011). Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems*, 36(1):62–78.

Sidney, C. F., Mendes, D. S., Ribeiro, L. A., and Härder, T. (2015). Performance Prediction for Set Similarity Joins. In *SAC*, pages 967–972.

Vernica, R., Carey, M. J., and Li, C. (2010). Efficient Parallel Set-similarity Joins using MapReduce. In *SIGMOD*, pages 495–506.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache Spark: a Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65.