Junções por Similaridade usando Processamento Distribuído e Paralelismo Massivo

Larissa Ramos Marques Silva, Leonardo Andrade Ribeiro

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

larissaramos@discente.ufg.br, laribeiro@inf.ufg.br

Abstract. *Similarity join returns all pairs of similar objects in a dataset. As this operation is computationally expensive, the runtime can be excessive on large volumes of data. This paper presents an efficient and scalable similarity join algorithm that exploits the massive parallelism of GPUs in a heterogeneous distributed environment. In this context, a coprocessing model is proposed to distribute the workload between CPU and GPU. Experimental results show that our proposal is effective and outperforms previous work.*

Resumo. *Junção por similaridade retorna todos os pares de objetos similares em um conjunto de dados. Como essa operação é custosa computacionalmente, o tempo de execução pode ser excessivo em grandes volumes de dados. Este artigo apresenta um algoritmo de junção por similaridade eficiente e escalável que explora o parelismo massivo de GPUs em um ambiente distribuído heterogêneo. Neste contexto, um modelo de coprocessamento é proposto para distribuir a carga de trabalho entre CPU e GPU. Resultados experimentais demonstram que a proposta é efetiva e supera trabalhos anteriores.*

1. Introdução

Junções por similaridade são usadas para encontrar todos os pares de objetos similares em um conjunto de dados. Dois objetos são considerados similares se o valor retornado por uma função de similaridade aplicada sobre eles for maior ou igual a um limiar predefinido. Junção por similaridade é uma operação fundamental em integração e limpeza de dados [Doan et al. 2012]. Por exemplo, uma junção por similaridade pode ser usada para encontrar representações não idênticas de uma mesmo objeto de informação.

No caso de objetos textuais, uma estratégia popular consiste em mapear as *strings* para conjuntos e, então, aplicar a junção por similaridade sobre esses conjuntos; a similaridade entre pares de *strings* é determinada pela interseção dos conjuntos correspondentes [Chaudhuri et al. 2006]. O foco deste artigo é este tipo de junção por similaridade em que os dados de entrada são representados como conjuntos.

Junção por similaridade é uma operação custosa computacionalmente. Uma abordagem ingênua para execução dessa operação aplicaria a função de similaridade sobre todos os pares de conjuntos, resultando em um tempo de execução proibitivo para bancos de dados volumosos. Ao longo dos anos, uma grande variedade de técnicas foram propostas para otimizar a execução de junções por similaridade, em particular através do emprego de filtros para redução do espaço de comparação [Chaudhuri et al. 2006, Ribeiro and Härder 2011]. Entretanto, o tempo de execução ainda cresce de maneira quadrática com o tamanho dos dados de entrada [Ribeiro and Härder 2011].

A exploração do paralelismo massivo disponível em GPUs é uma opção atrativa para acelerar a execução de junções por similaridade. De fato, trabalhos anteriores reportaram ganhos significativos de desempenho com o uso de GPU em comparação com soluções baseadas em CPU [Ribeiro-Júnior et al. 2017]. Porém, GPUs possuem memória *onboard* com capacidade limitada, inviabilizando o armazenamento nessa memória de grandes volumes de dados.

O emprego de processamento distribuído é uma abordagem natural para obter escalabilidade [Oliveira et al. 2017]. Distribuição da carga de trabalho entre os recursos computacionais e eficiência no processamento local são aspectos fundamentais neste contexto. Infelizmente, arcabouços para processamento distribuído como o *Apache Spark* [Zaharia et al. 2016] consideram somente a localidade dos dados para distribuição da carga de trabalho [Xu et al. 2018]. Detalhes sobre a capacidade computacional dos componentes em ambiente distribuído heterogêneo são desconsiderados nas decisões de alocação de tarefas, resultando em ociosidade e desperdício de recursos.

Este artigo apresenta uma proposta para junções por similaridade baseada em processamento distribuído e paralelismo massivo. Em particular, será considerado um ambiente de hardware baseado em uma arquitetura *shared-nothing* heterogênea, composta por nós de processamento equipados com CPU e GPU. Até o momento, não se tem conhecimentos de trabalhos anteriores que investigaram junções por similaridade em um ambiente distribuído heterogêneo. O principal desafio desta abordagem é lidar com a assimetria de poder computacional entre os processadores. Neste contexto, este trabalho propõe um modelo de coprocessamento para guiar a distribuição da carga de trabalho entre CPU e GPU. Resultados experimentais demonstram que o modelo proposto permite identificar precisamente a melhor alocação de carga entre os recursos computacionais disponíveis e propicia ganhos de desempenho em comparação com trabalhos anteriores.

2. Fundamentação Teórica e Trabalhos Relacionados

Strings podem ser mapeadas para conjuntos de diversas maneiras. Um método comum é coletar todas as sequências de caracteres de tamanho *q* de uma *string*; 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 r e s, a similaridade *Jaccard* entre os mesmos é dada por $Jaccard\,(r,s) = \frac{|r \cap s|}{|r \cup s|}$.

Definição 1 (Junção por Similaridade) *Seja $\mathcal{C}$ uma coleção de conjuntos e τ um limiar de similaridade. Uma junção por similaridade retorna todos os pares de conjuntos $(r,s) \in \mathcal{C} \times \mathcal{C}$ tal que $Jaccard\,(r,s) =\geq \tau$.*

Técnicas de filtragem são essenciais para execução eficiente de junções por similaridade. Uma das técnicas mais efetivas é a chamada *filtragem por prefixo* [Chaudhuri et al. 2006]. Primeiro, os elementos de todos conjuntos são ordenados de acordo com uma ordem global. Dado um conjunto r e um limiar τ, seja $pref\,(r) \subseteq r$ o subconjunto de r composto por seus primeiros $|r| - \lceil |r| \times \tau \rceil + 1$ elementos. Portanto, tem-se $Jaccard\,(r,s) \geq \tau \Rightarrow pref\,(r) \cap pref\,(s) \neq \emptyset$. Outra técnica popular é chamada filtragem por tamanho: dado um conjunto r e um limiar τ, para qualquer conjunto s tem-se $Jaccard\,(r,s) \geq \tau \Rightarrow |r| \times \tau \leq |s| \leq \frac{|r|}{\tau}$.

O trabalho em [Ribeiro-Júnior et al. 2017] apresenta `gSSJoin`, um algoritmo paralelo projetado para execução em GPUs. O `gSSJoin` constrói um índice invertido na GPU e usa esse índice para encontrar conjuntos similares sem utilizar técnicas de filtragem; desta maneira, o desempenho do algoritmo não é afetado quando o valor de τ diminui. Uma extensão chamada `sf-gSSJoin` divide os conjuntos de entrada em blocos para lidar com coleções que não cabem na memória da GPU. Durante o processamento, o `sf-gSSJoin` usa o filtro de tamanho para descartar blocos inteiros de conjuntos.

O trabalho em [Oliveira et al. 2017] apresenta `DSJoin`, um algoritmo distribuído para junções por similaridade. `DSJoin` envia conjuntos para nós de processamento baseando-se nos elementos do prefixo. Esta estratégia aplica a filtragem por prefixo indiretamente, pois dois conjuntos serão enviados para um mesmo nó de processamento e comparados se eles possuírem elementos do prefixo em comum. Uma avaliação comparativa entre dez algoritmos distribuídos para junção por similaridade é apresentada em [Fier et al. 2018]— o algoritmo com o melhor desempenho nessa comparação adota um esquema de particionamento baseado em filtragem por prefixo, como o `DSJoin`. Todos algoritmos assumem um cluster homogêneo e não consideram a disponibilidade de GPUs.

3. Junções por Similaridade em Clusters Heterôgeneos

Esta seção apresenta uma abordagem para processamento distribuído de junções por similaridade em um cluster heterogêneo equipado com CPUs e GPUs. Primeiro, será apresentado um modelo para divisão da carga de trabalho entre os processadores. Em seguida será apresentado o DSJoin^{gpu}, um algoritmo que aplica este modelo para aliar processamento distribuído e paralelismo massivo na execução de junções por similaridade.

3.1. Modelo de Coprocessamento CPU-GPU

Um desafio inerente de uma estratégia de coprocessamento em hardware heterogêneo é identificar a melhor distribuição da carga de trabalho. Por um lado, a distribuição deve evitar a ociosidade de recursos mantendo todos os processadores ocupados. Por outro lado, deve-se evitar a sobrecarga de um processador, pois o tempo total de execução será determinado pelo último processador que concluir sua carga de trabalho.

A partir da observação de que operações em banco de dados são tipicamente limitadas pela largura de banda da memória, o trabalho em [Shanbhag et al. 2020] apresenta modelos de predição do tempo execução em CPU e GPU para seleção, projeção e junção. Esses modelos baseiam-se na estimativa de que a razão entre os tempos de execução da CPU e GPU acompanhará a razão da largura de banda de memória desses processadores. Inspirado por este trabalho, será apresentado a seguir um modelo para distribuição da carga de trabalho entre CPU e GPU.

Definição 2 (Modelo de Coprocessamento CPU-GPU) *Seja CPU e GPU as frações da carga de trabalho alocadas para a CPU e GPU, respectivamente; $CPU + GPU = 1$. Seja B_{cpu} e B_{gpu} as larguras de banda de memória da CPU e GPU, respectivamente. A distribuição da carga de trabalho é definida pela seguinte proporção: $\frac{CPU}{GPU} = \frac{B_{cpu}}{B_{gpu}}$.*

Exemplo 1 *Como um exemplo concreto, considere um processador como Intel® Xeon® E5-2650 v3, com largura de banda da memória de 68 GB/s, e uma GPU Nvidia Tesla K40, com largura de banda da memória de 288 GB/s. A partir de $CPU + GPU = 1$ e*

Algoritmo 1: $DSJoin^{gpu}\ (\mathcal{C}, \tau, pg)$

Dados: Um RDD $\mathcal{C}$ contendo um coleção de conjuntos, um limiar de similaridade τ, fração da carga de trabalho destinada à CPU fc
Resultado: Um RDD S contendo todos os pares (r, s) tal que $Jaccard\,(r, s) \geq \tau$.

1 $list(key, r) \leftarrow R.flatMap(funcPart\,(r))$
2 $list(key, list(r)) \leftarrow groupByKey(list(key, r))$
3 **para cada** $(key, list(r)) \in List(key, list(r))$ **faça**
4 **se** $random() < fc$ **então**
5 $S' \leftarrow S' \cup flatMap(simJoin(key, list(r), \tau))$
6 **senão**
7 $S' \leftarrow S' \cup flatMap(sf\text{-}gSSJoin(key, list(r), \tau))$
8
9 $S \leftarrow collect(S')$
10 **Função** *funcPart(r)*
11 **para cada** $key \in pref\ (r.a_i)$ **faça**
12 $list(key, r) \leftarrow (key, r)$
13 **retorna** list(key,r)

substituindo GPU conforme a proporção da Definição 2, tem-se: $CPU + \frac{288}{68} \times CPU = 1 \equiv CPU \times \left(1 + \frac{288}{68}\right) = 1 \equiv CPU \approx 0.19$. Ou seja, a distribuição da carga de trabalho deve ser 19% para CPU e 81% para GPU.

3.2. O Algoritmo DSJoin^{gpu}

Com a definição das frações da carga de trabalho destinadas a cada tipo de processador, é possível projetar um algoritmo que integre distribuição de processamento e paralelismo massivo. Intuitivamente, a ideia consiste em incorporar o algoritmo paralelo `sf-gSSJoin` no algoritmo distribuído `DSJoin`. O Algoritmo 1 descreve o DSJoin^{gpu}, um algoritmo que implementa a abordagem proposta usando *Spark*. O DSJoin^{gpu} possui duas etapas: particionamento e verificação. Na etapa de particionamento, os elementos do prefixo são usados como chaves de particionamento para dividir a coleção de entrada em blocos que serão enviados para os nós de processamento (Linhas 1-2). Na etapa de verificação, a similaridade entre todos pares de conjuntos associados a uma mesma chave é calculada e os pares com similaridade maior que τ são incluídos no resultado (linhas 3-8). Esta etapa é realizada por um algoritmo de junção por similaridade para CPU ou pelo `sf-gSSJoin` na GPU, conforme a distribuição obtida pelo modelo de coprocessamento (representada pelo parâmetro fc).

4. Experimentos

Os experimentos usaram as bases de dados disponíveis publicamente DBLP e IMDB, que contêm informações sobre artigos científicos e filmes, respectivamente. Nas duas bases, as *strings* do atributo `título` foram extraídas e, para cada *string*, foram criadas cópias sujas contendo [1,5] modificações aleatórias (inclusão, exclusão e modificação de caracteres). Finalmente, as strings foram mapeadas para conjuntos de *3-qgrams*.

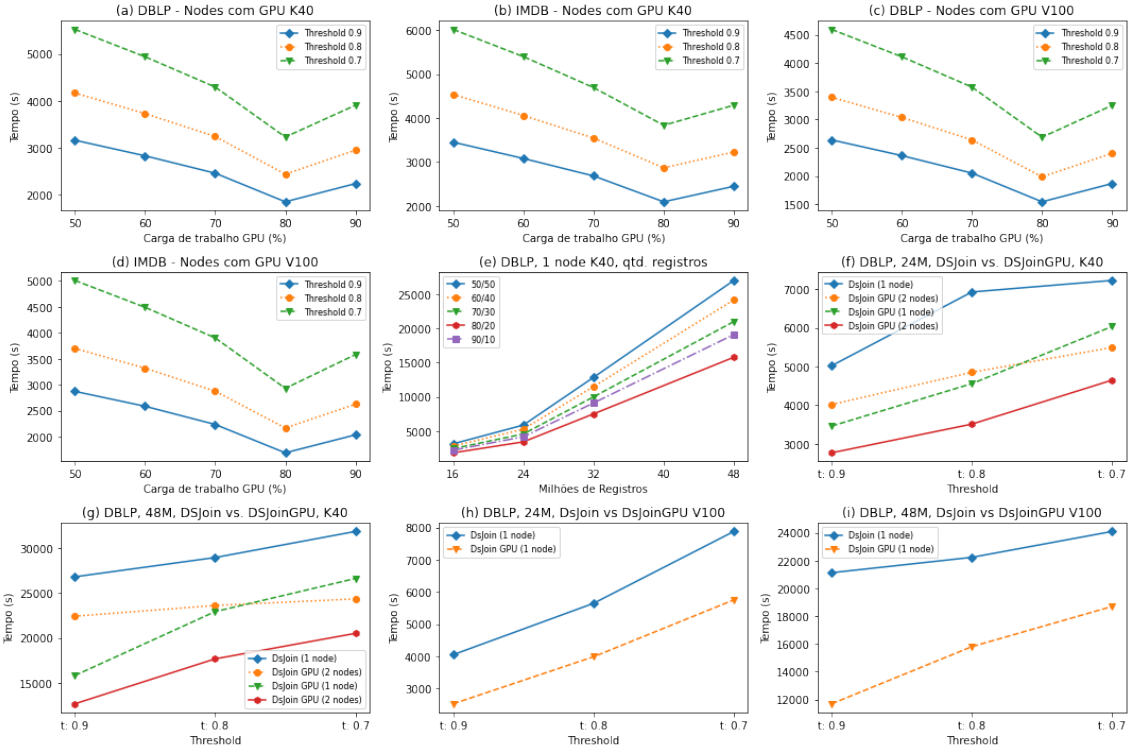

Figura 1. Resultados dos experimentos.

O `DSJoin` foi implementado em *Scala* 2.11 e *Spark* 3.3. Como algoritmo para junção por similaridade em CPU foi usado `MPJoin` [Ribeiro and Härder 2011], implementado em *Oracle Java 11*. O algoritmo para GPU `sf-gSSJoin` foi implementado em *CUDA* 11.0. A integração entre o `DSJoin` e o `sf-gSSJoin` foi realizada através da biblioteca de ligação *JCuda* 10.1.0.

Os testes foram realizados em dois *clusters*, chamados *K40* e *V100*. O *cluster K40* é composto por 2 nós de processamento, cada nó equipado com 20 núcleos em 2 processadores Intel® Xeon® E5-2650 v3, 128 GB de memória com largura de banda de 68 GB/s, e 1 GPU NVIDIA Tesla K40, 12 GB de memória com largura de banda de 288 GB/s. O *cluster V100* é composto por um nó de processamento equipado com 64 núcleos em 2 processadores AMD EPYC™ 7452, 256 GB de memória com a largura de banda de 204.8 GB/s, e 1 GPU NVIDIA Tesla V100, 12GB de memória com largura de banda de 897 GB/s. A divisão da carga de trabalho de acordo com o modelo de coprocessamento proposto é a mesma nos dois *clusters*: 19% para CPU e 81% para a GPU.

Figura 1 apresenta os resultados. O primeiro conjunto de testes buscou validar o modelo de coprocessamento. A partir de uma divisão homogênea inicial entre CPU e GPU, a carga de trabalho da CPU foi diminuída em decrementos de 10%. Os valores de τ usados foram 0.9, 0.8 e 0.7. Figuras 1(a)-(b) e 1(c)-(d) mostram os resultados obtidos nos clusters K40 e V100, respectivamente. Em todos os casos, o melhor resultado é obtido com a divisão 20%-80% entre CPU-GPU, que é a mais próxima aos valores obtidos pelo modelo proposto. O ganho de desempenho com essa divisão é de até 40% em comparação com a divisão homogênea. Esta tendência se mantém quando o tamanho da coleção de entrada aumenta, conforme mostrado na Figura 1(e) para valor de *threshold* igual a 0.9.

O segundo conjunto de experimentos comparou os algoritmos DSJoin e DSJoin^{gpu}; a divisão da carga de trabalho entre CPU-GPU do DSJoin^{gpu} segue o modelo proposto, isto é 19%-81%. As Figuras 1(f)-(g) e 1(h)-(i) apresentam os resultados nos *clusters K40* e *V100*, respectivamente, na base DBLP com 24M e 48M conjuntos. Em todos os casos, DSJoin^{gpu} é mais rápido que o DSJoin, atingindo ganhos de desempenho de até 63%. No *cluster K40*, DSJoin^{gpu} supera o DSJoin para *threshold* igual a 0.9 mesmo usando apenas um nó de processamento.

5. Conclusões

Este artigo apresentou o DSJoin^{gpu}, um algoritmo de junção por similaridade que explora processamento distribuído e paralelismo massivo. Um modelo de coprocessamento foi proposto para distribuição da carga de trabalho entre CPU e GPU. Os resultados experimentais demonstram que a proposta obteve ganhos de desempenho em todos os cenários analisados. O modelo de coprocessamento proporcionou ganho de desempenho de até 40% em comparação com a divisão homogênea da carga de trabalho e o DSJoin^{gpu} é 63% mais rápido do que um algoritmo existente que usa apenas processamento distribuído.

Agradecimento Esta pesquisa foi apoiada pelo LaMCAD/UFG.

Referências

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

Doan, A., Halevy, A. Y., and Ives, Z. G. (2012). *Principles of Data Integration*. Morgan Kaufmann.

Fier, F., Augsten, N., Bouros, P., Leser, U., and Freytag, J. (2018). Set Similarity Joins on MapReduce: An Experimental Survey. *Proceedings of the VLDB Endowment*, 11(10):1110–1122.

Oliveira, D., Borges, F. F., and Ribeiro, L. A. (2017). Uma Abordagem para Processamento Distribuído de Junção por Similaridade sobre Múltiplos Atributos. In *Proceedings of the Brazilian Symposium on Databases*, pages 300–305.

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

Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. (2017). Fast Parallel Set Similarity Joins on Many-core Architectures. *Journal of Information and Data Management*, 8(3):255–270.

Shanbhag, A., Madden, S., and Yu, X. (2020). A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the SIGMOD Conference*, pages 1617–1632.

Xu, L., Butt, A. R., Lim, S., and Kannan, R. (2018). A Heterogeneity-Aware Task Scheduler for Spark. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 245–256.

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.