Particionamento em GPU para o Problema da Junção Exata por Similaridade

Gabriel Viana Dantas[1], **Wellington S. Martins**[1], **Leonardo A. Ribeiro**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
CEP 74001-970 – Goiânia – GO – Brazil

g.viana.dantas@gmail.com, {wellington,laribeiro}@inf.ufg.br

Abstract. *The exact set similarity join finds all similar pairs between set collections, enabling a variety of applications. Its quadratic complexity demands efficient solutions capable of making use of the GPU's computing power. The positional filter, used in existing algorithms, requires a large data structure for parallel execution. This work explores a new partitioning scheme for the exact and symmetric self-join, incorporating the positional filter and allowing for massive data parallelism without limiting the size of the index structures used.*

Resumo. *A junção exata por similaridade encontra todos os pares similares entre coleções de conjuntos, permitindo diversas aplicações. Sua complexidade quadrática demanda soluções eficientes e capazes de fazer uso do poder computacional da GPU. O filtro posicional, utilizado em algoritmos existentes, requer uma grande estrutura de dados para a execução paralela. Este trabalho explora um novo esquema de particionamento para a auto-junção exata e simétrica, incorporando o filtro posicional e permitindo o paralelismo de dados massivo sem limitar o tamanho das estruturas de índice utilizadas.*

1. Introdução

A junção por similaridade possui aplicações em diferentes domínios, como limpeza de dados [Chaudhuri et al. 2006], extração de informações [Chakrabarti et al. 2008], detecção de duplicatas [Xiao et al. 2011], e mineração de dados [Rajaraman and Ullman 2011].

Diferentes tipos de dados podem ser convertidos para conjuntos pelo processo de *tokenização*. Objetos são fragmentados em elementos de um universo finito $\mathcal{U}$, resultando em conjuntos $r \in 2^{\mathcal{U}}$ que compõem uma coleção $R \subseteq 2^{\mathcal{U}}$. Um exemplo de conversão para dados textuais é a fragmentação em *q-grams*, subcadeias de tamanho fixo. A transformação permite o uso de soluções para a junção por similaridade em conjuntos.

Seja o universo $\mathcal{U}$ e uma função de similaridade sim, o problema da junção exata das coleções $R, S \subseteq 2^{\mathcal{U}}$ para o limiar $\theta \in [0, 1]$ pode ser formalizado pela Definição 1. O caso especial onde $R = S$ é chamado de auto-junção e permite o uso de otimizações específicas [Augsten and Böhlen 2013]. A generalização para $R \neq S$ pode ser feita pela auto-junção de $R \cup S$ seguida da remoção dos pares originados da mesma coleção.

$$sim : 2^{\mathcal{U}} \times 2^{\mathcal{U}} \rightarrow [0, 1]$$

$$join(R, S, \theta) = \{(r, s) \mid (r, s) \in R \times S, r \neq s, sim(r, s) \geq \theta\} \quad (1)$$

Funções de similaridade são usadas para comparar um par de objetos. Seu custo computacional é significativo e evitado pela aplicação de filtros, técnicas que exploram propriedades dos dados para identificar pares incapazes de atingir o limiar.

1.1. Soluções em GPU

GPUs possuem arquiteturas específicas para processamento paralelo com alta vazão de instruções e memória, qualidades interessantes para resolver o problema da junção de forma eficiente. O filtro posicional adaptado para conjuntos [Xiao et al. 2011] é utilizado por diversos algoritmos [Mann et al. 2016]. Sua complexidade de espaço se torna quadrática em um contexto paralelo, sendo necessário um esquema de particionamento.

O algoritmo em GPU *fgssjoin* [Quirino et al. 2017] utiliza um esquema de particionamento em blocos. Cada um é indexado e consultado de forma retroativa pelos blocos passados, até que todos os pares similares sejam encontrados. O espaço das estruturas geradas cresce de forma quadrática em relação ao tamanho de bloco escolhido, que deve ser suficientemente pequeno. Motivado a estudar outros particionamentos para o filtro posicional em GPU, este trabalho explora a indexação de blocos maiores.

1.2. Complexidade do Filtro Posicional

Funções de similaridade típicas como Jaccard são simétricas e diretamente proporcionais ao tamanho da interseção de um par, permitindo o uso da restrição $minoverlap$.

$$sim(r, s) \geq \theta \iff |r \cap s| \geq minoverlap(r, s) \tag{2}$$

O filtro posicional encontra tetos para a interseção entre registros da coleção, eliminando pares que não atendem à restrição. Seu uso requer uma estrutura de dados para contar o número de elementos em comum encontrados entre um registro sendo filtrado e aqueles presentes no índice.

Algoritmos de filtragem sequencial, como *ppjoin* [Xiao et al. 2011], precisam contar a interseção para pares de apenas um único registro em um dado momento. A filtragem paralela dos pares de múltiplos registros multiplica espaço necessário para contar as interseções encontradas.

2. Particionamento Triangular

Este trabalho propõe o particionamento triangular, utilizado para resolver o problema da auto-junção com funções de similaridade simétricas.

Devido à simetria da similaridade, os pares de um registro r_i são reduzidos a $\{\{r_i, r_j\} \,|\, i > j\}$. A Figura 1 mostra a disposição dos pares de índices para $1 \leq i \leq 9$, formando uma matriz triangular que representa cada combinação possível. Essa estrutura utiliza o espaço de forma mais eficiente que uma matriz quadrada completa, pois elimina os pares simétricos redundantes.

A matriz de pares pode ser utilizada para armazenar a contagem de interseções feita pelo filtro posicional. Sua forma permite o paralelismo de dados, cada processador tendo acesso exclusivo ao espaço de uma linha i da matriz para filtrar os pares do tipo $\{i, j\}$. As cargas de trabalho são desbalanceadas na ponta do triângulo, porém a diferença

	0	1	2	3	4	5	6	7	8
1	{1,0}								
2	{2,0}	{2,1}							
3	{3,0}	{3,1}	{3,2}						
4	{4,0}	{4,1}	{4,2}	{4,3}					
5	{5,0}	{5,1}	{5,2}	{5,3}	{5,4}				
6	{6,0}	{6,1}	{6,2}	{6,3}	{6,4}	{6,5}			
7	{7,0}	{7,1}	{7,2}	{7,3}	{7,4}	{7,5}	{7,6}		
8	{8,0}	{8,1}	{8,2}	{8,3}	{8,4}	{8,5}	{8,6}	{8,7}	
9	{9,0}	{9,1}	{9,2}	{9,3}	{9,4}	{9,5}	{9,6}	{9,7}	{9,8}

Figura 1. Matriz de pares

máxima para p processadores é igual a $p - 1$. Bases de dados razoavelmente grandes são ordens de magnitude maiores que p, assim como o tamanho das linhas na estrutura. A implementação deste esquema, apresentada na próxima seção, permitiu observar que a divisão é equilibrada na maior parte do tempo.

Primeiro é necessário estabelecer um mapeamento bidirecional entre a estrutura triangular e o espaço linear de memória. A Função 3 linearia de forma contígua os índices da matriz. A Função 4 é a solução máxima para i na inequação $f(i, 0) \leq M$ e possui dois papeis, calcular o maior triângulo que pode ser armazenado em um espaço de memória M e reverter um índice linearizado k para a forma $(g(k), k - f(g(k), 0))$.

$$f(i, j) = \frac{i(i-1)}{2} + j \tag{3}$$

$$g(M) = \left\lfloor \frac{\sqrt{8M+1}+1}{2} \right\rfloor \tag{4}$$

A memória disponível M pode ser muito menor que a estrutura de dados. O esquema de particionamento consiste em processar faixas de índices $\{i, i+1, ..., i+n\}$ que possam ser armazenadas em memória. Cada faixa constitui uma seção contígua de linhas da matriz e permite encontrar todos os seus pares similares. Após o processamento de uma região, ela não será revisitada e a memória M pode ser reutilizada para outra iteração até que todas as partições sejam visitadas.

3. Avaliação Experimental

Para demonstrar e analisar o particionamento triangular, ele foi implementado no algoritmo em GPU $tgjoin$. A implementação foi desenvolvida em CUDA, modelo de programação proprietário da NVIDIA [Kirk and Wen-Mei 2016]. O código fonte está disponível no endereço www.github.com/GabrielVD/tgjoin.

Foram preparadas as coleções DBLP 100K e DBLP 1M, compostas respectivamente por 10^5 e 10^6 amostras da base de dados DBLP (dblp.uni-trier.de/db). Cada registro é formado pela tokenização do título da publicação em *3-grams*. A Tabela 1 contém a média aritmética de 3 testes dos tempos de junção. O limiar utilizado é indicado no topo das colunas. As implementações referências dos algoritmos *ppjoin* e *fgssjoin* foram utilizadas como base, por compartilharem dos mesmos filtros. Foi escolhido o tamanho

Tabela 1. Tempos de execução (DBLP 100k em ms, DBLP 1M em segundos)

DBLP 100k	75%	90%	99%
tgjoin	**110**	**36**	31
fgssjoin	277	123	92
ppjoin	1782	208	**20**

DBLP 1M	90%	95%	99%
tgjoin	4.60	2.93	2.71
fgssjoin	**4.53**	**2.25**	**1.58**
ppjoin	44.98	16.69	9.17

de bloco $30\,000$ para *fgssjoin*, valor próximo de saturar a memória disponível. Todos os testes foram realizados em um computador dedicado, com sistema operacional Ubuntu 22.04 LTS, driver CUDA versão 11.7, CPU Ryzen 7 5800X, GPU RTX 3060 Ti e 32 GB de memória RAM 3200 MHz.

4. Conclusões

O esquema de particionamento triangular é capaz de realizar a auto-junção para similaridades simétricas, de forma massivamente paralela e adotando o filtro posicional. O particionamento é realizado pós-indexação, sem limitar o tamanho do índice construído. Este resultado introduz uma nova solução para o problema, aproveitando da arquitetura de GPUs.

A implementação *tgjoin* conseguiu desempenho superior para cardinalidades da ordem de 10^5. Seus resultados revelam potenciais vantagens sobre as referências existentes. A escalabilidade observada se mostrou pior que em *fgssjoin*, implementação dominante na ordem de 10^6. Isto se deve à baixa granularidade da indexação única e completa feita por *tgjoin*, impedindo o salto de blocos utilizado por *fgssjoin*.

Referências

Augsten, N. and Böhlen, M. H. (2013). Similarity joins in relational database systems. *Synthesis Lectures on Data Management*, 5(5):1–124.

Chakrabarti, K., Chaudhuri, S., Ganti, V., and Xin, D. (2008). An efficient filter for approximate membership checking. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 805–818.

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 5–5. IEEE.

Kirk, D. B. and Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

Mann, W., Augsten, N., and Bouros, P. (2016). An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 9(9):636–647.

Quirino, R. D., Ribeiro-Júnior, S., Ribeiro, L. A., and Martins, W. S. (2017). fgssjoin: A gpu-based algorithm for set similarity joins. In *ICEIS (1)*, pages 152–161.

Rajaraman, A. and Ullman, J. D. (2011). *Mining of massive datasets*. Cambridge University Press.

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):1–41.