

Desempenho de operações distribuídas de agrupamento por similaridade em dados de alta dimensionalidade por meio da VP-tree

Ana Paula C. A. Silva¹, Humberto Razente¹

¹Faculdade de Computação
Programa de Pós-graduação em Ciência da Computação
Universidade Federal de Uberlândia (UFU) – Uberlândia, MG – Brazil

ana.acassiano@ufu.br, humberto.razente@ufu.br

Abstract. *Contemporary organizations face the challenge of analyzing significant volumes of complex data using techniques such as clustering analysis. Previous research introduced DSG (Distributed Similarity Grouping), a solution developed under the MapReduce strategy in Apache Hadoop/Spark, characterized by parallelism in identifying similar groups in massive datasets. This work introduces an evolution of this method, called DSG-VPTREE (Distributed Similarity Grouping with Vp-Tree), which incorporates the Vantage Point Tree (VP-Tree) to optimize data partitioning by pivots. The new algorithm allows optimization of both the partitioning and the analysis of overlapping partition windows. The results of the experiments show that DSG-VPTree outperforms DSG, with a significant reduction in execution time and better scalability on high-dimensional data.*

Resumo. *As organizações contemporâneas enfrentam o desafio de analisar volumes significativos de dados complexos, utilizando técnicas como análise de agrupamentos. Pesquisas antecedentes introduziram o DSG (Distributed Similarity Grouping), uma solução desenvolvida sob a estratégia MapReduce no Apache Hadoop/Spark, caracterizada pelo paralelismo na identificação de grupos similares em conjuntos de dados massivos. Este trabalho¹ introduz uma evolução desse método, denominado DSG-VPTREE (Distributed Similarity Grouping with Vp-Tree), que incorpora a Vantage Point Tree (VP-Tree) para otimizar o particionamento de dados por pivôs. O novo algoritmo permite a otimização tanto do particionamento quanto da análise das janelas de sobreposição das partições. Os resultados dos experimentos demonstram que o DSG-VPTree supera o DSG, apresentando uma redução significativa no tempo de execução e melhor escalabilidade em dados de alta dimensionalidade.*

1. Introdução

No cenário atual, caracterizado pela prevalência de *Big Data*, organizações de diversas áreas enfrentam o desafio constante de extrair conhecimento relevante de conjuntos de dados de tamanho e complexidade sem precedentes. A análise eficaz desses dados não é apenas uma vantagem competitiva, mas uma necessidade para a inovação, a tomada de

¹O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

decisões estratégicas, a manutenção da relevância no mercado. Entre as várias técnicas empregadas para analisar esses dados, a análise de agrupamentos surge como um método fundamental para identificar padrões e grupos de similaridade que facilitam a interpretação de grandes volumes de informações [Silva et al. 2019].

O uso de sistemas distribuídos para processamento de dados massivos tem ganhado popularidade em diversas aplicações, por exemplo, com o uso do paradigma Map/Reduce [Dean and Ghemawat 2004] implementado no *Apache Hadoop* e *Spark* [Zaharia et al. 2016]. O sistema é composto de um serviço de processamento e sistema de arquivos distribuído em um cluster composto por computadores interligados em rede. São beneficiados os algoritmos para os quais os dados possam ser particionados para distribuição, onde o processamento é realizado nas partições de modo distribuído, e por fim, realiza-se a agregação dos resultados. O critério de particionamento é geralmente baseado na relação de ordem e a igualdade de chaves.

Para dados em que a relação de ordem e a igualdade entre elementos tem pouca utilidade, utiliza-se uma noção de dissimilaridade, normalmente modelada como uma função de distância, que quantifica o quão dissimilar dois elementos são. Por exemplo, se o usuário tem uma imagem, recuperar uma imagem idêntica tem pouca utilidade quando comparado com a possibilidade de recuperar imagens similares.

O particionamento de conjuntos de dados baseado em dissimilaridade é uma tarefa essencial para manipulação otimizada desses dados, por exemplo, com a criação de métodos de acesso métricos dinâmicos [Traina-Jr. et al. 2002]. Em [Samet 2006] é apresentada uma extensa revisão sobre métodos de acesso que utilizam estratégias de particionamento baseadas em bolas (ou hiperesferas) e baseados em hiperplanos separadores. Esses métodos buscam a organização dos dados em agrupamentos baseados na dissimilaridade, criando geralmente hierarquias balanceadas dinamicamente. Para o particionamento de conjuntos de dados estáticos são empregados elementos de referência selecionados previamente (pivôs).

Nos últimos anos, pesquisas nesse domínio têm avançado significativamente, com destaque para o operador *Similarity Group-By* (SGB) [Tang et al. 2016], que permite a computação de agrupamentos por similaridade em consultas ad hoc. Nesse contexto destaca-se o operador *Distributed Similarity Grouping* (DSG) [Silva et al. 2019], que opera sobre plataformas de computação distribuída *Apache Hadoop* e *Spark*. Através da utilização do paradigma *MapReduce*, o DSG consegue identificar grupos similares em conjuntos de dados massivos de maneira paralela, otimizando significativamente o tempo de processamento necessário para tal tarefa.

Contudo, apesar do progresso proporcionado pelo DSG, o crescimento incessante do volume de dados e a exigência por análises cada vez mais ágeis e precisas evidenciam a necessidade de contínuo aprimoramento das soluções existentes. Nesse contexto, este trabalho propõe o DSG-VPTree (*Distributed Similarity Grouping with Vp-Tree*), uma inovação do método DSG que incorpora a estrutura de dados *Vantage Point Tree* (VP-Tree) para melhorar o particionamento de dados. Esta abordagem visa oferecer um método mais equilibrado e eficiente para operações de agrupamento por similaridade, mitigando assim as limitações de escalabilidade e desempenho das soluções anteriores.

Este artigo detalha a criação do DSG-VPTree no ambiente *Spark*, fornecendo dire-

trizes claras para sua aplicação e avalia seu desempenho em comparação com o operador DSG, que teve vantagem comprovada na comparação com outros métodos de agrupamento de dados. Os experimentos realizados no trabalho mostram que o DSG-VPTree oferece uma melhora significativa em termos de tempo de execução e escalabilidade quando comparado ao algoritmo DSG.

Para tanto, este trabalho está organizado da seguinte forma. A Seção 2 apresenta o contexto do trabalho assim como os trabalhos correlatos. A Seção 3 apresenta os detalhes do algoritmo principal, que deu origem ao trabalho. A Seção 4 apresenta os detalhes do algoritmo proposto e sua implementação em *Spark*. A Seção 5 apresenta os resultados da avaliação de desempenho e a Seção 6 apresenta as conclusões e trabalhos futuros.

2. Trabalhos Correlatos

O agrupamento de dados baseado em igualdade é uma das operações mais utilizadas em análise de dados [Gray et al. 1997]. Em se tratando de agrupamento por similaridade, métodos como o K-Means, CURE, DBSCAN tem sido amplamente estudados nas últimas décadas [Jain 2010]. [Iqbal et al. 2022] aborda o uso de agrupamentos por similaridade para processamento analítico online (Online Analytical Processing – OLAP). [Tang et al. 2016] introduziu o operador de Agrupamento por Similaridade para dados multidimensionais. O crescimento de dados em diversas áreas tem pressionado continuamente os limites das tecnologias de processamento de dados. Em resposta a isso, [Silva et al. 2019] introduziram o operador de Agrupamento por Similaridade Distribuída DSG (*Distributed Similarity Grouping*), um método projetado para otimizar a identificação de grupos de similaridade em grandes conjuntos de dados utilizando a capacidade de processamento paralelo dos sistemas *Hadoop* e *Spark*. O DSG representa um avanço significativo na análise de *Big Data*, integrando a eficácia das operações de agrupamento com a semântica de similaridade dos métodos de *clustering*, proporcionando assim tempos de execução rápidos e agrupamentos baseados em similaridade eficazes.

Este trabalho explora estruturas de dados otimizadas para buscas em espaços métricos complexos e de alta dimensionalidade. O arcabouço teórico essencial inclui a *Vantage Point Tree* (VP-Tree), introduzida por [Yanilos 1993], e a *Multi-Vantage Point Tree* (MVP-Tree), detalhada por [Bozkaya and Özsoyoglu 1999], e a M-Tree e a GH-Tree, discutidas por [Samet 2006]. Essas estruturas, conhecidas como Métodos de Acesso Métrico (MAM), são fundamentais para melhorar a eficiência de consultas por similaridade por meio do particionamento e organização dos dados, permitindo computar o resultado de consultas sem a necessidade de examinar cada elemento do conjunto. Neste contexto, a VP-Tree, com suas características de organização espacial baseadas em esferas concêntricas centradas em pontos de vantagem, demonstra uma adaptabilidade notável a diferentes distribuições de dados e uma eficácia superior em espaços de alta dimensionalidade [Samet 2006].

A M-Tree, embora robusta para ambientes dinâmicos que requerem inserções e exclusões frequentes, apresenta desafios significativos em termos de manutenção do balanceamento, especialmente quando escalada para grandes volumes de dados, o que pode comprometer o desempenho em sistemas distribuídos como *Hadoop* e *Spark* [Samet 2006]. Por outro lado, a GH-Tree, que utiliza hiperplanos para particionar o espaço, tende a se tornar ineficiente à medida que a dimensionalidade dos dados aumenta.

A complexidade dos cálculos envolvidos na manutenção de hiperplanos em altas dimensionalidades resulta em um *overhead* computacional que pode ser proibitivo [Samet 2006]. Em contraste, a VP-Tree não apenas simplifica o processo de particionamento de dados através de esferas concêntricas centradas em pontos de vantagem, mas também adapta-se eficientemente a diferentes distribuições de dados, mantendo um desempenho consistente de consulta mesmo em conjuntos de dados de alta dimensionalidade. Essas características tornam a VP-Tree particularmente atraente para a implementação do DSG-VPtree, assegurando uma integração eficaz e eficiente com as plataformas de computação distribuída utilizadas [Samet 2006].

3. Algoritmo de agrupamento distribuído por similaridade (DSG)

Este trabalho tem como objetivo aprimorar o desempenho do algoritmo *Distributed Similarity Grouping* (DSG). Nesta seção, detalhamos a abordagem algorítmica original, de acordo com [Silva et al. 2019], para estabelecer uma base sobre a qual as melhorias são introduzidas.

O algoritmo *Distributed Similarity Grouping* (DSG) inicia com a seleção de pivôs a partir dos dados de entrada, que são utilizados para particionar o conjunto de dados. Cada registro de entrada é atribuído à partição associada ao pivô mais próximo, e aqueles localizados na região de fronteira entre as partições são replicados (janela), conforme demonstrado na Figura 1, para garantir a correta detecção dos grupos de similaridade nessas regiões, uma vez que os elementos próximos ao hiperplano limite da partição precisam considerar os elementos próximos mas que estão na partição vizinha. O particionamento é realizado utilizando uma distância euclidiana generalizada para calcular a distância de um registro ao hiperplano que separa duas partições, assegurando uma distribuição eficiente dos dados.

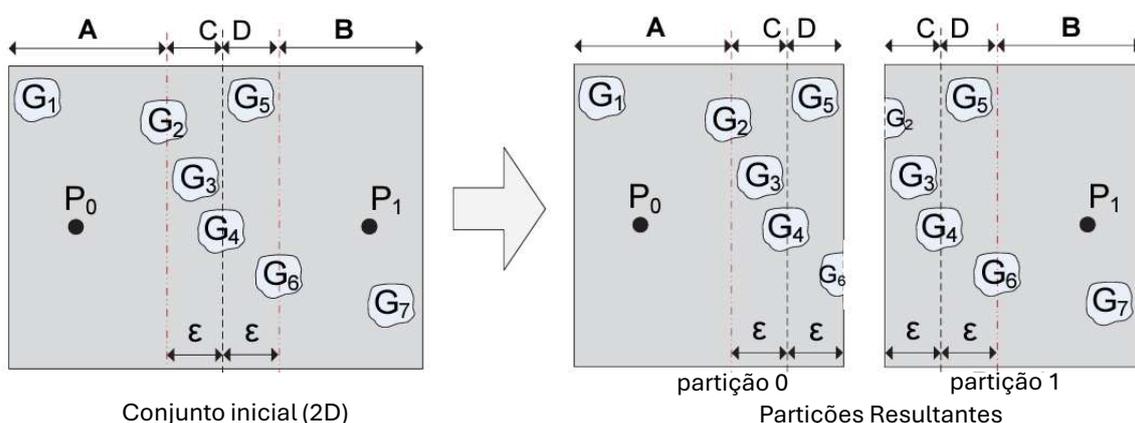


Figura 1. Particionamento pelo DSG para dois pivôs. Fonte: [Silva et al. 2019]

Na fase de formação dos grupos de similaridade, cada partição é processada individualmente. Se a partição for pequena o suficiente, um algoritmo de nó único é usado para identificar os grupos de similaridade; caso contrário, a partição é armazenada para reprocessamento em rodadas subsequentes utilizando o mesmo algoritmo DSG, até que a partição seja pequena o suficiente. Durante a verificação de pertinência aos grupos, cada elemento é comparado com todos os elementos do grupo para verificar se satisfaz a condição de similaridade, e se assim for, é adicionado ao grupo. Finalmente, cada grupo é

marcado com informações de *flag* (onde cada registro é anotado com informações sobre a sua partição base e a partição à qual foi atribuído) para garantir que os grupos sejam gerados sem duplicação em diferentes partições, permitindo uma organização eficiente e precisa dos dados distribuídos.

A Figura 1 demonstra o funcionamento do algoritmo DSG, que nesse caso tem o objetivo de particionar o conjunto de dados inicial em duas partições de modo a podermos identificar todos os grupos de similaridade (G1 a G7). No exemplo, cada grupo de similaridade deve ser gerado apenas em uma partição. O algoritmo fará o particionamento dos dados de entrada utilizando dois pivôs (P0 e P1), de modo que cada ponto pertença à partição do seu pivô mais próximo. Adicionalmente, o algoritmo duplicará os pontos nas janelas ε (C e D), de modo que a partição 0 será dividida em A+C+D, e a partição 1 será dividida em B+C+D. Para identificar os grupos de similaridade em cada partição, o algoritmo utilizará a lógica demonstrada nas tabelas 1 e 2. Dessa forma, após a formação de grupos e a validação de janelamento, os grupos de similaridade G1, G2, G3 e G4 serão gerados na partição P0, enquanto os grupos de similaridade G5, G6 e G7 serão gerados na partição P1.

Tabela 1. Identificação dos Grupos de Similaridade na Partição P0

Se o Grupo	Ação
Somente em A	Gerar grupo em P0
Em A e C	Gerar grupo em P0
Somente em C	Gerar grupo em P0
Em C e D	Gerar grupo em P0
Somente em D	Ignorar

Tabela 2. Identificação dos Grupos de Similaridade na Partição P1

Se o Grupo	Ação
Somente em C	Ignorar
Em C e D	Ignorar
Somente em D	Gerar grupo em P1
Em D e B	Gerar grupo em P1
Somente em B	Gerar grupo em P1

A utilização do *Apache Spark* e sua API de *Resilient Distributed Datasets* (RDDs) para a implementação do DSG demonstra um aumento significativo na concisão e eficiência do código, conforme ilustrado por [Silva et al. 2019]. O processo de seleção de pivôs, particionamento de dados, e a consolidação dos registros em grupos de similaridade são facilitados por funções específicas da plataforma *Spark*, como *mapPartitionsToPair* e *groupByKey* [Zaharia et al. 2016].

O algoritmo DSG, apesar de eficiente em diversos cenários, apresenta limitações em termos de velocidade de execução quando aplicado a grandes volumes de dados. A função *getClosestPivot*, responsável por determinar o pivô mais próximo para cada registro, representa um gargalo significativo, especialmente porque é executada múltiplas vezes para cada registro, primeiro para identificar o pivô mais próximo e novamente para

confirmar esta proximidade dentro de um limite definido (ε). O Algoritmo 1 apresenta a parte do algoritmo principal que é foco do presente trabalho, destacando os pontos de gargalo nas linhas 3 e 5. Na linha 3 é chamada a função *getClosestPivot* para identificar o pivô mais próximo ao registro. Já da linha 5 à linha 9, é feita a identificação de registros dentro da janela definida por ε .

Algorithm 1 DistSimGrouping

Require: *inputData*, *eps*, *numPivots*, *memT*

Ensure: similarity groups in *inputData*

```

1: pivots  $\leftarrow$  selectPivots(numPivots, inputData)           ▷ Partitioning
2: for each record r in a chunk of inputData do
3:   Pc  $\leftarrow$  getClosestPivot(r, pivots)
4:   output  $\langle P_c, r \rangle$                                        ▷ intermediate output
5:   for each pivot p in {pivots – Pc} do
6:     if (dist(r, p) – dist(r, Pc))/2  $\leq$  eps then
7:       output  $\langle p, r \rangle$                                    ▷ intermediate output
8:     end if
9:   end for
10: end for                                                    ▷ Shuffle: records with same key  $\Rightarrow$  partition
11: for each partition Pi do                                   ▷ Group Formation
12:   .....
13:   .....
14: end for

```

4. Algoritmo de agrupamento distribuído por similaridade com *Vantage Point Tree* (DSG-VP-Tree)

Para superar as limitações apresentadas do algoritmo principal, este trabalho apresenta a integração da *Vantage Point Tree* (VP-Tree) na estrutura do DSG. A VP-Tree é uma estrutura de dados que permite consultas por similaridade eficientes e pode ser particularmente útil para acelerar a identificação de pivôs próximos. Após a seleção dos pivôs, uma VP-Tree estática é construída a partir desses elementos e distribuída aos nós do *cluster* via mecanismo de *broadcast* do *Apache Spark*. Este processo garante que cada nó receba uma cópia imutável da árvore, permitindo consultas locais sem a necessidade de comunicação constante entre os nós, o que reduz a latência nas operações de busca. Como o DSG no *Spark* é executado no estilo *batch*, a VP-tree é criada e utilizada apenas para uma execução do algoritmo.

É importante destacar que a construção da VP-Tree, neste caso, considera apenas os elementos escolhidos como pivôs (e não o conjunto de dados em análise). A sua construção e distribuição apresentam um pequeno overhead inicial, que é compensado pelo ganho na sua utilização, permitindo a exclusão rápida de um número significativo de elementos sem a necessidade de recalculá-los todas as distâncias para cada elemento do conjunto de dados para o qual é preciso encontrar sua partição.

Na construção da VP-Tree, inicialmente escolhe-se um elemento como ponto de vantagem. Em seguida, calcula-se a distância de todos os demais elementos até este ponto

de vantagem (v). O próximo passo consiste em calcular a mediana dessas distâncias. Utilizando essa mediana μ , os elementos são bipartidos. A Figura 2 apresenta um esquema onde aqueles elementos cuja distância é igual ou menor que μ são posicionados no nó esquerdo, enquanto aqueles elementos com distância maior são alocados no nó direito. Essa divisão estratégica facilita consultas eficientes, pois durante uma busca, parte dos elementos pode ser excluída com base na comparação de distâncias relativas a μ , por meio da desigualdade triangular.

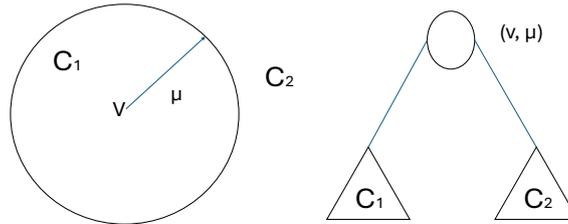


Figura 2. Distribuição de dados em uma VP-Tree. Adaptado de [Fu et al. 2000]

A construção da VP-Tree é completada por meio de um processo recursivo, onde este método de divisão é aplicado sucessivamente aos nós filhos, até que todos os elementos estejam adequadamente atribuídos.

A incorporação da VP-Tree altera tanto o algoritmo principal DSG quanto a função *getClosestPivot*. Era preciso uma estrutura estática, onde não ocorrem exclusões ou atualizações, que funcionasse eficientemente em memória principal, que fosse distribuída para todos os nós do *cluster Spark*. A VP-Tree foi escolhida por trabalhar com memória principal e possibilitar essa criação de árvore única, distribuída por *broadcast* para todos os nós do *cluster*. A criação da árvore é baseada em uma amostra dos pivôs escolhidos aleatoriamente, cujo tamanho é especificado pelo usuário.

Algorithm 2 DistSimGroupingVpTree

Require: *inputData*, *eps*, *numPivots*, *memT*

Ensure: similarity groups in *inputData*

- 1: *pivots* \leftarrow *selectPivots*(*numPivots*, *inputData*)
 - 2: *vptree* \leftarrow **createVpTree**(*pivots*, *euclidianDist*)
 - 3: **for** each record *r* in a chunk of *inputData* **do** ▷ Partitioning
 - 4: $P_c \leftarrow$ **getClosestPivotVpTree**(*r*, *pivots*, *vptree*, *eps*)
 - 5: **end for** ▷ Shuffle: records with same key \Rightarrow partition
 - 6: **for** each partition P_i **do** ▷ Group Formation
 - 7:
 - 8:
 - 9: **end for**
-

O Algoritmo 2 apresenta o algoritmo principal, onde foi incluído na linha 2 a criação da VP-Tree a partir da lista de pivôs e de uma função euclidiana. Também é possível observar, na linha 4, a chamada da função que trará o pivô mais próximo e também fará o tratamento dos pontos contidos na janela definida por ϵ . Essa função utiliza o registro sendo tratado, a VP-tree criada em memória principal e o valor de ϵ . O tratamento de

pontos contidos em janela passa a ser efetuado dentro da função `getClosestPivotVpTree`, no mesmo laço que o pivô mais próximo, já garantindo um ganho de desempenho inicial.

No Algoritmo 3 é possível verificar que inicialmente é feita a busca pelo vizinho mais próximo, na linha 1. Essa função, retorna o pivô mais próximo ao ponto de pesquisa e a distância entre os dois. Para melhorar o desempenho da verificação de pontos presentes em mais de uma partição, foi criada uma variável *Dring* que representa duas vezes o valor de ϵ mais o valor da distância entre o ponto e seu vizinho mais próximo. Com essa variável, é possível definir um limite de poda de pontos para a próxima busca por janela-mento. Assim, na linha 4, onde é feito a busca por abrangência dos vizinhos pertencentes à janela, o raio de busca é limitado pelo valor de *Dring*, ou seja, apenas os pivôs dentro da distância definida por *Dring* terão sua distância verificada na fase de verificação de pontos que pertencem à janela de outro pivô.

Algorithm 3 `getClosestPivotVpTree`

Require: $r, vptree, eps$

Ensure: closest pivots

```

1:  $P_c, D_c \leftarrow \text{getVPTreeNearestNeighbor}(r, vptree)$ 
2: output  $\langle P_c, r \rangle$  ▷ intermediate output
3:  $Dring \leftarrow (2 \times \epsilon) + D_c$ 
4:  $pivotsRing \leftarrow \text{getAllInRange}(vptree, Dring)$ 
5: for each pivot  $p$  in  $\{pivotsRing - P_c\}$  do
6:   if  $(\text{dist}(r, p) - \text{dist}(r, P_c))/2 \leq \epsilon$  then
7:     output  $\langle p, r \rangle$  ▷ intermediate output
8:   end if
9: end for

```

Espera-se que a integração da VP-Tree reduza a complexidade de tempo da função `getClosestPivot` de $\mathcal{O}(n)$ para $\mathcal{O}(\log n)$ em média, considerando a eficiência típica das árvores de *vantage point* em cenários de busca dimensional. Esta melhoria pode resultar em uma redução significativa no tempo total de processamento, especialmente em conjuntos de dados extensos onde o número de registros e pivôs é grande.

5. Avaliação de desempenho

Os experimentos foram conduzidos utilizando *Hadoop* versão 3.3.6 e *Spark* versão 3.5.1, e Linux kernel versão 5.4 virtualizado com KVM. O *cluster* consiste de um nó mestre e cinco nós de trabalho. Todos os nós estão equipados com dois processadores *Intel Xeon Quad-Core* com frequência de 2.4 GHz (8 cores). O nó mestre tem 48 GB de memória RAM e 8 TB de espaço em disco, enquanto os nós de trabalho têm 24 GB de memória RAM e 1 TB de espaço em disco.

Como a intenção foi efetuar uma comparação do algoritmo original com a nova implementação, para a avaliação foi utilizado um gerador de dados sintéticos configurado para simular diversas condições de teste: o mesmo gerador utilizado para os testes do algoritmo original. Os dados gerados representavam grupos de similaridade em espaços multidimensionais (de 64D a 256D), com a separação entre grupos definida por $2 * \epsilon$. O objetivo foi verificar a consistência dos resultados entre o DSG original e o DSG-VPTree sob condições controladas.

O volume de dados foi escalonado pelo fator N (SFN), com cada conjunto de dados SF1 contendo aproximadamente 200.000 registros multiplicados por N. Para SF1, os dados incluíam cerca de 3.000 grupos, cada um composto por 50 a 100 registros. Cada registro foi replicado de uma a três vezes para testar a robustez dos algoritmos frente a duplicações. O número de pivôs utilizados foi definido como $\text{numPivots} = 40 \times \text{SF}$, e o limiar de memória (memT) foi fixado em 50.000 unidades.

Os testes focaram em avaliar o tempo de processamento total do algoritmo e a formação de grupos de similaridade. Comparou-se o desempenho do DSG e do DSG-VPTree em termos de tempo de execução, além de verificar a integridade dos grupos formados em presença de variações de dimensionalidade e tamanho do conjunto de dados.

Aumento do fator de escala. A Figura 3 ilustra o comportamento dos dois algoritmos à medida que o fator de escala aumenta. Nesse experimento, o valor de *epsilon* foi fixado em 100 e o número de dimensões em 64D. Até 1.000.000 de registros (SF5), o tempo de execução para ambos os algoritmos permanece comparativamente próximo, mostrando um aumento gradual. Entretanto, a partir de 1.200.000 registros (SF6), observa-se que o tempo de execução para o DSG começa a crescer mais rapidamente, alcançando um tempo 40% maior que o do DSG-VPTree quando expandido para SF10. É possível notar que conforme o número de registros (SF) aumenta, o tempo de execução do DSG aumenta linearmente em função de uma constante maior que do DSG-VPTree. Esse comportamento é explicado porque além do aumento do fator de escala, existe um aumento no número de pivôs também.

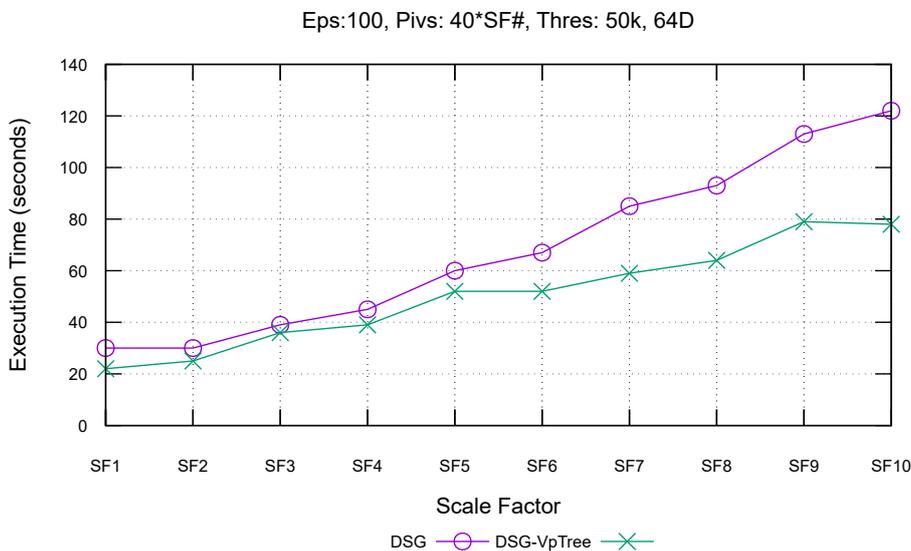


Figura 3. Tempo de execução com o aumento de tamanho do conjunto de dados

Aumento do número de dimensões. Os experimentos foram realizados fixando o fator de escala em SF10, *epsilon* em 100, enquanto o número de dimensões variou de 64D a 256D. Conforme demonstrado na Figura 4, o tempo de execução para ambos os algoritmos aumenta com o número de dimensões. No entanto, o aumento no tempo de execução do DSG-VPTree é mais moderado em comparação com o DSG tradicional. Com 256 dimensões, o tempo de execução do DSG é aproximadamente o dobro do tempo necessário para o DSG-VPTree, sugerindo que o DSG-VPTree está mais preparado para

lidar com contextos de alta dimensionalidade. É possível notar que o tempo de execução da DSG possui um crescimento mais rápido comparado ao tempo da DSG-VPTree. O gráfico mostra ainda que a DSG-VPTree cresce de maneira mais controlada e linear.

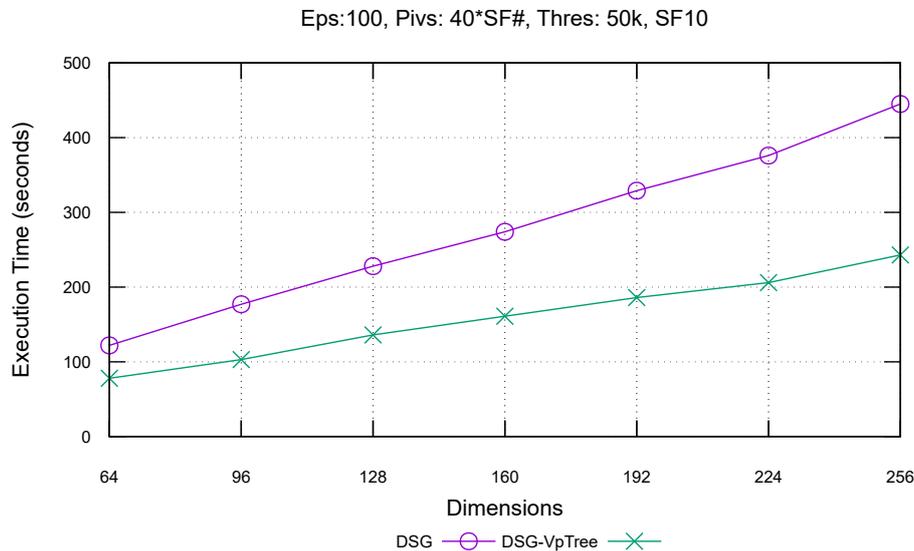
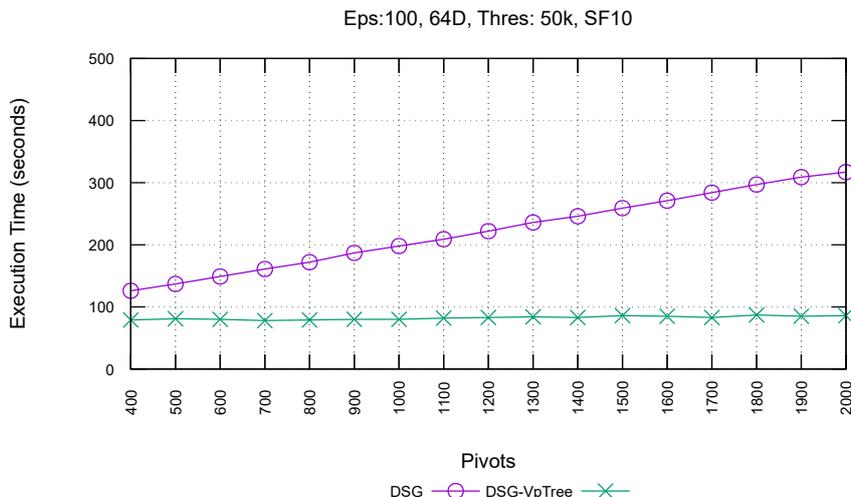
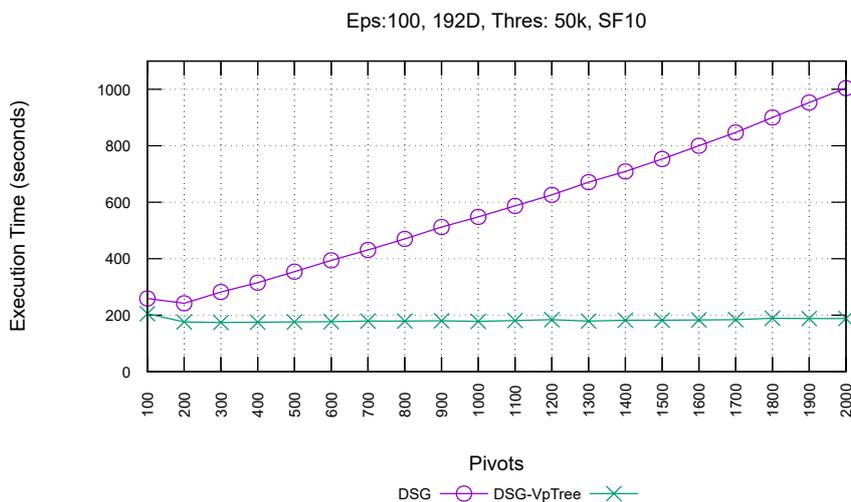


Figura 4. Tempo de execução com o aumento de número de dimensões

Aumento do número de pivôs. Foram realizados experimentos variando-se o número de pivôs em duas configurações distintas: 64 dimensões e 192 dimensões. Em ambos os casos, os valores de *epsilon* foram mantidos constantes em 100, o limiar em 50000 e o fator de escala em SF10. Conforme ilustrado na Figura 5, observa-se que, para as duas configurações dimensionais testadas, o DSG-VPTree apresenta um desempenho mais estável e uma melhor escalabilidade. Na configuração com 192 dimensões, à medida que o número de pivôs aumenta, o DSG-VPTree tem tempo de execução constante, enquanto o DSG tem um aumento de tempo que cresce linearmente com o número de pivôs. Verifica-se que o DSG tem um ponto de inflexão a partir do qual o aumento no número de partições resulta em aumento no tempo de processamento, enquanto o DSG-VPTree exibe um comportamento constante. Esse experimento evidencia os benefícios proporcionados pela utilização de uma árvore *vantage-point*. Mesmo com o aumento da quantidade de pivôs, estes são eficientemente indexados na árvore, o que melhora a eficiência das buscas. Isso ocorre principalmente devido à poda de registros não candidatos durante a busca por elementos na janela, o que reduz o número de comparações necessárias e acelera o processo de busca.



(a) 64 Dimensões.



(b) 192 Dimensões.

Figura 5. Tempo de execução com aumento de número de pivôs

6. Conclusões

Este trabalho introduziu o DSG-VPTree (*Distributed Similarity Grouping with VP-Tree*), uma evolução do algoritmo DSG que incorpora a estrutura de dados VP-Tree para melhorar o desempenho do particionamento de dados e aprimorar as operações de agrupamento por similaridade em conjuntos de dados massivos. A implementação proposta provou ser superior em termos de velocidade quando comparada ao DSG original, demonstrando um desempenho até 40% melhor em contextos de alta dimensionalidade e grandes volumes de dados nos experimentos realizados.

Como trabalho futuro pretende-se explorar o desenvolvimento de algoritmos distribuídos que se adaptem a diversos tipos de grupos de similaridade, expandindo a aplicabilidade do DSG-VPTree para outras áreas de dados. Além disso, pretende-se criar uma estratégia para seleção de pivôs que resulte em partições balanceadas, particularmente em

conjuntos com distribuições heterogêneas e variadas dimensionalidades.

Referências

- Bozkaya, T. and Özsoyoglu, Z. M. (1999). Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)*, 24(3):361–404. doi:10.1145/328939.328959.
- Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In Brewer, E. A. and Chen, P., editors, *Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150. USENIX. <http://www.usenix.org/events/osdi04/tech/dean.html>.
- Fu, A. W., Chan, P. M., Cheung, Y., and Moon, Y. S. (2000). Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB J.*, 9(2):154–173. doi:10.1007/PL00010672.
- Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53. doi:10.1023/A:1009726021843.
- Iqbal, M., Lissandrini, M., and Pedersen, T. B. (2022). A foundation for spatio-textual-temporal cube analytics. *Inf. Syst.*, 108:102009. doi:10.1016/j.is.2022.102009.
- Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern Recognit. Lett.*, 31(8):651–666. doi:10.1016/J.PATREC.2009.09.011.
- Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers.
- Silva, Y. N., Sandoval, M., Prado, D., Wallace, X., and Rong, C. (2019). Similarity grouping in big data systems. In *Intl Conf. Similarity Search and Applications (SISAP)*, volume 11807 of *LNCS*, pages 212–220. Springer. doi:10.1007/978-3-030-32047-8_19.
- Tang, M., Tahboub, R. Y., Aref, W. G., Atallah, M. J., Malluhi, Q. M., Ouzani, M., and Silva, Y. N. (2016). Similarity group-by operators for multi-dimensional relational data. *IEEE Trans. Knowl. Data Eng.*, 28(2):510–523. doi:10.1109/TKDE.2015.2480400.
- Traina-Jr., C., Traina, A. J. M., Faloutsos, C., and Seeger, B. (2002). Fast indexing and visualization of metric data sets using slim-trees. *IEEE Trans. Knowl. Data Eng.*, 14(2):244–260. doi:10.1109/69.991715.
- Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321. ACM/SIAM. <http://dl.acm.org/citation.cfm?id=313559.313789>.
- 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. *Commun. ACM*, 59(11):56–65. doi:10.1145/2934664.