Busca por Similaridade no CassandraDB*

Antonio Mourão[1], Rafael Pasquini[1], Rodolfo Villaça[2], Lasaro Camargos[1]

[1]Universidade Federal de Uberlândia (UFU) – Uberlândia, MG
[2] Universidade Federal do Espirito Santo (UFES) – Vitoria, ES

{antoniomourao@comp.,rafael.pasquini@,lasaro@}ufu.br
rodolfo.villaca@ufes.br

Abstract. *The need for scalability in the Big Data era has lead to flourishing of distributed NOSQL systems. However, none of the widely used solutions available support similarity searches, another important feature in such scenarios. Here we propose to overcome this limitation by extending Cassandra with similarity search capabilities. Our approach is based on using locality sensitive hashing to spread data to nodes, to target nodes for queries, and for efficiently recovering similar data from stable storage at the nodes. We overview how to implement the approach, our test plan, and initial results.*

Resumo. *A necessidade de escalabilidade em cenários de* Big Data *levou à prosperação dos NOSQL distribuídos. Contudo, as soluções mais amplamente usadas não suportam uma outra característica desejável, busca por similaridade. Neste artigo propomos contornar tal limitação estendendo o Cassandra com suporte a buscas por dados similares. Nossa abordagem usa* hashing *sensível à localidade para distribuir dados entre nós, direcionar buscas, e recuperar dados em armazenamento estável de forma eficiente. Descrevemos aqui como implementar esta abordagem, nosso plano de testes e resultados iniciais.*

1. Introdução

Cassandra [Lakshman and Malik 2010] é um banco de dados NOSQL (*Not Only SQL*) amplamente utilizado em cenários *BigData* devido à sua escalabilidade horizontal. Como em outros NOSQL [Decandia et al. 2007, Chang et al. 2008], essa característica vem do *consistent hashing*, particionamento aleatório dos dados entre os nós do sistema que distribui a carga uniformemente entre os mesmos e facilita sua entrada e saída do sistema [Stoica et al. 2001]. Apesar das vantagens, essa distribuição dificulta a busca por dados similares, outra característica desejável em *BigData*.

Dois objetos de um mesmo domínio são similares se seus atributos possuem um grau de semelhança elevado; caso contrário são dissimilares. A busca por similaridade consiste em recuperar objetos que sejam similares aos parâmetros da consulta. Alguns exemplos incluem a identificação de perfis de clientes com hábitos de consumo parecidos e de imagens com o mesmo tema. Dentre alguns esforços para se melhorar a busca por similaridade nestes sistemas, destacamos a HammingDHT [Villaca et al. 2013].

Na HammingDHT, a chave de cada objeto é construída por uma função de espalhamento sensível a localidade (LSH), que gera chaves similares para dados similares e

*Agradecimentos à CAPES, CNPq, FAPEMIG e FAPES pelo apoio à realização deste trabalho.

os coloca, com alta probabilidade, em um mesmo nó ou nós vizinhos. Assim, dado uma consulta, a mesma gerará uma chave que será roteada, com alta probabilidade, para os nós que contêm dados similares aos da consulta, permitindo que os dados similares sejam retornados com poucos saltos na rede. Como outros esforços, a HammingDHT foi apenas um protótipo e não é usável por aplicações reais. Propomos aqui integrar as técnicas usadas na HammingDHT ao Cassandra, obtendo um sistema usável e com suporte à busca por similaridade.

O restante deste artigo é organizado da seguinte forma: Seção 2 detalha a fundamentação teórica; Seção 3 apresenta trabalhos relacionados; Seção 4 apresenta nossa proposta de modificação do Cassandra; Seção 5 descreve os primeiros passos na avaliação do trabalho; e Seção 6 conclui o artigo com uma perspectiva dos próximos passos.

2. Fundamentação Teórica

2.1. Similaridade e Espalhamento

Neste trabalho assumimos que objetos são representáveis como vetores k-dimensionais e que a similaridade entre dois vetores é dada por uma função de distância entre os mesmos. Dado uma consulta $q \in R^k$, onde R^k é o universo de objetos k-dimensionais de interesse, e um limiar de similaridade t, procura-se por **todos** os objetos $p \in R^k : dist(p,q) \leq t$, ou por **um** dos objeto $p \in R^k$ com **menor** distância para q, isto é, $p = \underset{p \in R^k}{\operatorname{argmin}} \|dist(p,q)\|$.

Random Hyperplane Hashing Funções de Espalhamento mapeiam seus dados de entrada em *buckets*; Funções de espalhamento Sensíveis à Localidade (*Locality Sensitive Hashing*, LSH) mapeiam dados similares, com alta probabilidade, nos mesmos *buckets*. Assim, as LSH servem como método de redução de espaços com alta dimensionalidade para espaços de menor dimensão, que mantém pontos próximos no espaço original também próximos no espaço reduzido. Uma família de funções LSH é um conjunto $\mathcal{F}$ de funções *hash* tal que para dois objetos x e y, $\mathbf{Pr}_{h \in \mathcal{F}}[h(x) = h(y)] = sim(x,y)$, onde $sim(x,y) \in [0,1]$ é uma função de similaridade sobre universo de objetos.

Espalhamento por Hiperplano Aleatório (*Random Hyperplane Hashing*, RHH), é uma família LSH que mapeia um vetor $\vec{v} \in R^k$ para 1 se $\vec{u} \cdot \vec{v} \geq 0$ e para 0 se $\vec{u} \cdot \vec{v} < 0$, onde $\vec{u} \in R^k$ tem coordenadas aleatórias normalmente distribuídas com centro em 0. Segundo [Charikar 2002], dados os vetores $\vec{u}, \vec{v}, \vec{w} \in R^k$, quanto menor o ângulo $\theta(\vec{v}, \vec{w})$ entre $\vec{v}$ e $\vec{w}$, mais $\mathbf{Pr}[h_{\vec{u}}(\vec{v}) = h_{\vec{u}}(\vec{w})]$ se aproxima de $\cos\theta(\vec{v}, \vec{w})$. Em outras palavras, com alta probabilidade, quanto mais similares forem dois objetos, mais o cosseno entre os vetores representando seus objetos se aproximará de 1. Tal relação entre a representação de similaridade da função RHH e o cosseno é chamada de similaridade do cosseno, ou $sim_{\cos}$[Villaça 2013].

Similaridade de Hamming A distância de Hamming entre os vetores $\vec{v}, \vec{w} \in R^k$ é o número de dimensões nas quais eles diferem. A distância normalizada $DH_{norm}(\vec{v_1}, \vec{v_2})$ resulta da divisão da distância de Hamming por k e varia no intervalo $[0,1]$. Como em [Villaca et al. 2013], definimos similaridade de Hamming entre dois vetores $\vec{v}$ e $\vec{w}$ como $sim_h(\vec{v_1}, \vec{v_2}) = 1 - DH_{norm}(\vec{v_1}, \vec{v_2})$.

Se para cada $\vec{u}$ aleatório a função $h_{\vec{u}}(\vec{v})$ gera 1 *bit*, dado um conjunto $M = \{\vec{u_1}, \ldots, \vec{u_m}\}$ de vetores, $u_i \in R^{\mathcal{N}}$, um identificador de m *bits*, onde $\bullet$ é um operador

de concatenação, é definido como $h_M(\vec{v}) = \bullet_{i=1...m} h_{\vec{u}_i}(\vec{v})$. Assim, dados dois vetores de objetos $\vec{v}$ e $\vec{w}$, quanto maior for a similaridade do cosseno entre esses vetores, mais *bits* em comum terão os identificadores $h_M(\vec{v})$ e $h_M(\vec{w})$ gerados, e maior será a similaridade de Hamming entre eles.

Em suma, a similaridade entre objetos, mantida pela redução de dimensionalidade via RHH, é capturada pela similaridade de Hamming, e pode ser medida facilmente, contando-se os bits diferentes entre vetores de bits.

2.2. Armazenamento e Recupeção de Dados Locais

Log Structured Merge Tree *Log Structured Merge Trees* (LSM) [O'neil et al. 1996] são usadas em NOSQL para armazenar dados em discos rígidos [Chang et al. 2008, Lakshman and Malik 2010]. Em LSM, dados são escritos ordenada e sequencialmente em *String Sorted Table* (*SSTable*), quando atingem algum limite em memória volátil (tamanho, idade, etc.).

SSTable são imutáveis e, por isso, atualizações levam à existência de diferentes versões do mesmo dado em diferentes *SSTables*. Para minimizar este problema, *SSTables* são frequentemente unidas para remover duplicatas e à cada *SSTable* é associado um filtro de Bloom com os dados da *SSTable*, consultado antes da mesma ser lida para memória.

Filtro de Bloom Filtros de Bloom [Bloom 1970] armazenam informações sobre pertinência de objetos a conjuntos, de forma compacta e aproximada. Dado um universo $U = \{c_1, \ldots c_n\}$, o filtro é definido em termos de um vetor V de m bits e um conjunto de funções de espalhamento $H = \{h_1, \ldots, h_k\}$, tal que $\forall h \in H, \forall c \in U, 0 < h(c) \leq m$. Dado um conjunto $C \subseteq U$ e respectivo filtro, então $\forall 0 < i \leq m$, $V[i] = 1 \Leftrightarrow \exists h \in H, \exists e \in C, h(e) = i$. Devido à colisões nas funções de espalhamento, é possível que $\forall h \in H, V[h(e)] = 1$ mesmo que $e \notin C$, e por isso consultas ao filtro podem retornar falsos positivos.

Filtro de Bloom Sensível à Localidade Essa variação do Filtro de Bloom usa funções de espalhamento sensíveis à localidade para determinar os índices de V a terem bits ajustados para 1. Uma vez que essas funções geram valores próximos para dados similares, os mesmos serão mapeados para os mesmos índices ou para índices vizinhos. Para testar a pertinência, em vez de se verificar se os bits correspondentes a um certo dado são todos iguais a 1, verifica-se se algum dos bits na vizinhança do índice gerado, inclusive, é igual a 1; quanto maior a vizinhança considerada, menor a similaridade exigida dos dados para se obter uma resposta positiva. Além de falsos positivos, estes filtros também sofrem de falsos negativos; [Hua et al. 2012] apresenta técnicas reduzir sua ocorrência.

3. Trabalhos Relacionados

3.1. *k-dimensional trees*

Uma árvore k-dimensional distribuída (*distributed kd-tree*, DKDT) [Gao 2007] é uma árvore construída sobre uma DHT, que serve de índice para buscas por similaridade. Percorrer a DKDT requer saltos entre nós, o que aumenta a latência das buscas. Em nosso sistema, consultas são direcionadas, em um salto, aos nós que provavelmente contém os dados buscados.

3.2. HammingDHT

A HammingDHT [Villaca et al. 2013] é uma tabela de espalhamento distribuído que permite a busca de conteúdos por similaridade. Seus nós recebem identificadores aleatórios, que são posicionados em um anel virtual, ordenados de acordo com a sequência do código de Gray (identificadores sucessivos apresentam distância de Hamming igual a 1). Adicionalmente, cada nó mantém uma tabela que inclui apontadores para outros no anel, também com distância de Hamming 1. Cada nó é responsável por armazenar todos os conteúdos cujos identificadores, gerados via RHH, ficam entre ele e o seu antecessor. Como identificadores de dados similares tem, com alta probabilidade, pequena distância de Hamming, dados similares tendem a ser colocados em um mesmo nó ou naqueles com quem está em contato direto. Essa característica reduz a média do número de saltos na rede na busca por conteúdos similares, resultando em menos saltos (menos esforço) para se obter grande cobertura (*recall*) das buscas.

A HammingDHT foi avaliada por meio de simulação, mas um protótipo funcional, distribuído, nunca foi implementado, e sua validação foi restrita a uma única base de dados. Aqui, propomos testar as técnicas utilizadas na HammingDHT de forma mais abrangente, integrando-as ao Cassandra, um SGBD NOSQL distribuído, baseado também no conceito de DHT e largamente utilizado na academia e indústria.

3.3. Cassandra

Cassandra [Lakshman and Malik 2010] é um NOSQL distribuído que armazena pares chave/valor. Nós são organizados em um anel e responsáveis por faixas contíguas de *tokens*, resultados do *hashing* da chaves; todos os nós conhecem todos os outros e roteiam dados (consultas ou inserções) diretamente para o nó responsável pelo *token* correspondente. O mapeamento chave/*token*/nó é implementado por um *particionador*, presente em cada um dos nós.

Cassandra utiliza LSM para armazenar dados de forma eficiente. Diferentemente de outros SGDB, colunas no Cassandra podem ser adicionadas dinamicamente e ter seus dados escritos independentemente. Por isso a recuperação de uma linha de dados requer a leitura de todas as *SSTables* que tenham a chave associada à linha, não somente a mais recentemente modificada, tornando imprescindível o uso de filtros de Bloom.

4. Proposta e Implementação

A proposta deste trabalho é, essencialmente, i) modificar o mecanismo de particionamento do Cassandra para distribuir os dados similares para o mesmo nó ou vizinhos e, ii) implementar busca por similaridade nas *SSTables*.

Similarity Partitioner Para que o particionador roteie consultas e inserções corretamente, propomos modificá-lo para usar RHH para gerar *tokens* para os dados. Além disso, ordenamos os *tokens* de acordo com o código de Gray binário refletido, de forma que *tokens* sucessivos tenham distância de Hamming 1, ou seja, correspondam a dados similares com alta probabilidade. Uma vez que *tokens* para dados semelhantes gerados com RHH tendem a ter múltiplos bits em comum (baixa distância de Hamming), há uma alta probabilidade de estarem próximos no código de Gray binário refletido, e portanto serem mapeados para o mesmo nó ou para nós vizinhos.

Nosso *Similarity Partitioner*[1], encapsula o processo de geração dos *tokens*: dado um vetor que descreve um dado, o particionador gera o *token* via RHH usando um conjunto de m vetores pré-definidos no arquivo de configuração do Cassandra, compartilhado por todos os nós; calculado o *token*, o particionador calcula sua posição na sequência de Gray e determina qual dentre os nós do sistema é responsável pela faixa de posições em que o mesmo se encontra, que é responsável pelo *token*.

No caso de uma operação de escrita, descritor, *token* e dados são enviados para o nó, onde são armazenados em uma mem-table. Quanto esta é convertida em uma SSTable, os *tokens* são armazenados em filtros de Bloom sensíveis à localidade.

Filtro de Bloom Sensível à Localidade Em uma operação de leitura, devem ser fornecidos o identificador de um objeto do qual se queira similares, e um limiar de similaridade, entre 0 e 1. Estes valores são encaminhados ao nó responsável pelo *token* correspondente.

No nó, o *token* é comparado com os tokens dos dados armazenados em suas mem-tables e *SSTables*. Dados com similaridade de Hamming maior ou igual ao limiar de similaridade são retornados na consulta. Somente são consultadas *SSTables* cujos filtros de Bloom sensíveis à localidade retornem positivo à consulta do *token*; o mesmo limiar de similaridade é considerado pelo filtro, de forma que quanto maior é o limiar, menor é a vizinhança dos bits consideradas na consulta.

5. Avaliação Experimental

Uma vez que o protótipo não está completo, não podemos apresentar um estudo das alterações de desempenho nem das taxas de recuperação e precisão da busca por similaridade no Cassandra. Apresentamos contudo um comparativo resumido do desempenho das funções de espalhamento RHH e Murmur Hash[2], usada por padrão no Cassandra. Para os testes geramos 15000 vetores de 50 dimensões aleatórias uniformemente distribuídas entre 0 e 99 (dados), vetores com valores reais uniformemente distribuídos entre 0 e 1 (vetores aleatórios do RHH). Murmur Hash gerou chaves de 64 bits e RHH gerou chaves de 16, 32, 64 e 128 bits. A Tabela 1 mostra o tempo médio de 10 execuções de cada função para gerar o *token* de 10000 chaves (o tempo das primeiras 5000 foram ignorados).

Tabela 1. Média de tempo de 10 execuções de geração de 10000 Murmur *hashes* de 128 bits e RHH *hashes* de 16, 32, 64 e 128 bits.

	Murmur	RHH-16	RHH-32	RHH-64	RHH-128
media (ms)	17,7	275,9	534	969,2	1769,2
desvio padrão	5,441507	13,17156	8,306624	15,702229	7,054077

Observe que Murmur *hash* é aproximadamente 15x mais rápido que RHH para 16 vetores aleatórios e que à medida que o número de vetores dobra, o tempo de execução do RHH aumenta em aproximadamente 1,8 vezes, levando a uma diferença de 100x para *hashes* de tamanho 128 bits. Apesar da grande diferença no tempo de execução das duas funções, é importante destacar que o custo da busca por similaridade é intrinsecamente

[1] `https://github.com/pluxos/cassandra-sim/tree/master`
[2] `https://github.com/aappleby/smhasher`

maior que da busca exata e que o custo do *hash* provavelmente terá pouco impacto no custo total da recuperação e inserção de dados. Mesmo assim, em futuras versões procuraremos melhorar o desempenho de nossa implementação.

6. Conclusão

Neste artigo propusemos estender o Cassandra com a capacidade de busca por similaridade, discutimos os conceitos necessários para realizar essa extensão, e descrevemos o estágio atual de nossa proposta. Descrevemos também um pequeno experimento comparando o tempo de execução do Murmur Hash, usado por padrão no Cassandra, e do RHH, que usamos em nossa implementação. Em futuras comunicações descreveremos a validação e avaliação experimental do nosso sistema.

Referências

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26.

Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA. ACM.

Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Hastorun, D., Decandia, G., and Vogels, W. (2007). Dynamo: Amazon's highly available Key-Value store. *SOSP*.

Gao, J. (2007). Efficient support for similarity searches in dht-based peer-to-peer systems. In *IEEE International Conference on Communications (ICC'07*.

Hua, Y., Xiao, B., Veeravalli, B., and Feng, D. (2012). Locality-sensitive bloom filter for approximate membership query. *IEEE Transactions on Computers*, 61(6):817–830.

Lakshman, A. and Malik, P. (2010). Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40.

O'neil, P. E., Cheng, E., Gawlick, D., and O'neil, E. J. (1996). The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33:351–385.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA. ACM.

Villaça, R. S. (2013). *Hamming DHT e HCube: Arquiteturas distribuídas para busca por similaridade*. PhD thesis, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

Villaca, R. S., de Paula, L. B., Pasquini, R., and Magalhaes, M. F. (2013). Hamming dht: Taming the similarity search. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, pages 7–12.