Um Repositório Chave-Valor com Controle de Localidade

Patrick A. Bungama[1], **Wendel M. de Oliveira**[1], **Flávio R. C. Sousa**[2], **Carmem S. Hara**[1]

[1] Departamento de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – 81.531-990 – Curitiba, PR – Brasil

[2]Departamento de Engenharia de Teleinformática – Universidade Federal do Ceará
Caixa Postal 6007 – 60440-970 – Fortaleza, CE – Brazil

{pabungama,wendel,carmem}@inf.ufpr.br,flaviosousa@ufc.br

Abstract. *The ever increasing volume of data produced nowadays presents challenges for storing and processing this data. Traditional database solutions are not efficient to face these challenges, especially with respect to scalability. One approach to provide scalability is the adoption of a layered architecture which combines a distributed storage system with a simple access interface. This paper presents ALOCS, a distributed repository which adopts the key-value model, allowing the user application to control the allocation of data into servers. The goal is to allow the application to co-allocate data that are frequently used together. Our experimental study shows that ALOCS improves query response times by reducing the amount of remote data accesses.*

Resumo. *O aumento no volume de dados produzidos apresenta desafios no armazenamento e processamento destes dados. Entretanto, soluções tradicionais de bancos de dados não se mostraram eficientes diante de tais desafios, principalmente no requisito de escalabilidade. Uma abordagem para prover escalabilidade é a adoção de uma arquitetura estratificada, que combina um sistema de armazenamento distribuído com uma interface simples para o acesso aos dados. Este artigo apresenta o ALOCS, um repositório de armazenamento distribuído de dados que adota o modelo chave-valor e que permite a aplicação usuária gerenciar o controle de localidade dos dados, reduzindo a comunicação no processamento de consultas. Os estudos experimentais mostram a melhoria no tempo de resposta das consultas utilizando a solução proposta.*

1. Introdução

A evolução dos sistemas computacionais e a difusão do acesso à Internet levaram à produção de grandes volumes de dados. Esta avalanche de dados trouxe novos desafios, como a definição de formas eficientes para coletar, armazenar, acessar e extrair estes dados [Yin e Kaynak 2015]. Contudo, soluções tradicionais de banco de dados não têm se mostrado eficientes diante destes desafios, principalmente por não apresentarem escalabilidade aceitável [Agrawal et al. 2010]. A escalabilidade vertical, obtida por meio da adição de recursos a um único servidor, é limitada e apresenta um alto custo para a aquisição de hardware e software. Já a escalabilidade horizontal, que adiciona mais servidores ao sistema, é mais atrativa, mas apresenta desafios tecnológicos para dar suporte a um modelo de armazenamento de dados distribuído [Zhang et al. 2013, Tran 2013].

Existem diversos sistemas gerenciadores de bancos de dados (SGBDs) baseados em uma arquitetura estratificada, na qual o módulo de armazenamento possui uma interface bem definida com o módulo de processamento de consultas. Isso permite que diferentes estruturas de armazenamento possam ser utilizadas, mantendo a mesma linguagem de consulta para o usuário e aplicações. Tal arquitetura é adotada para SGBDs relacionais (MySQL[1], MariaDB[2]), XML (Zorba[3]) e mais recentemente, NoSQL (MongoDB[4]). Dessa forma, uma possível abordagem para obter escalabilidade horizontal é a integração de um módulo de armzenamento escalável para estes sistemas. Segundo Cattell [Cattell 2011], existem quatro características comuns em repositórios de dados escaláveis: (i) são baseados em uma interface simples; (ii) conferem a habilidade de escalar horizontalmente um sistema sobre muitos servidores; (iii) dispõem de índices distribuídos e; (iv) permitem um ajuste dinâmico da distribuição da carga de trabalho. Todas estas características estão associadas à natureza *shared-nothing* de redes par-a-par (P2P). Tendo em vista que as tabelas de dispersão distribuídas (*DHT - Distributed Hash Table*) foram propostas neste contexto para oferecer uma interface de uso geral para a indexação e armazenamento distribuído de dados, alguns sistemas adotaram DHTs para prover escalabilidade aos SGBDs [de S. Rodrigues et al. 2009, Arnaut et al. 2011, Ribas et al. 2011].

Uma das dificuldades relatadas pelos sistemas que adotaram uma DHT como módulo de armazenamento é o alto custo de comunicação para o processamento de consultas. Isso se deve ao fato das DHTs adotarem o modelo chave-valor e oferecerem uma interface para acesso de valores individuais a partir da chave (operações *get-put-rem*). No entanto, consultas em SGBDs envolvem um conjunto de valores relacionados. Como as DHTs em geral não oferecem um controle sobre a localidade, a recuperação deste conjunto de dados pode incorrer em uma comunicação entre servidores para a recuperação de cada elemento individualmente.

Assim, uma possível alternativa é a utilização de um sistema de arquivos distribuídos (SAD) no módulo de armazenamento do SGBD. Nos SADs, o modelo usado para armazenar dados é o arquivo. Eles permitem que aplicações armazenem e acessem arquivos remotos a partir de qualquer computador em uma rede como se fossem locais, sendo responsáveis por organizá-los, armazená-los, recuperá-los, compartilhá-los e protegê-los [Rani et al. 2014]. Arquivos de metadados são utilizados para armazenar a localização dos arquivos. O acesso aos metadados, quando centralizado, gera de 50% a 80% de todo tráfego de rede [Ousterhout et al. 1985]. Apesar do tamanho dos metadados ser geralmente pequeno em comparação com a capacidade de armazenamento do sistema, ele pode tornar-se um gargalo potencial. Evitar este tipo de gargalo é fundamental para que um sistema de armazenamento atinja alto desempenho e escalabilidade. Alguns SADs permitem que aplicações especifiquem em quais servidores armazenar arquivos e assim garantir a localidade dos dados.

A localidade de objetos armazenados em um ambiente distribuído influencia no desempenho das consultas realizadas sobre estes dados. Portanto, controlar a localidade de dados é muito importante, pois pode garantir que os dados usualmente utilizados em

[1] `http://www.mysql.com`
[2] `http://mariadb.com`
[3] `www.zorba.io`
[4] `http://www.mongodb.com`

conjunto possam ser alocados em um mesmo servidor, reduzindo a comunicação no processamento de suas consultas. Isto evita acesso a múltiplos servidores, e pode ser usado para aproximar informações de suas aplicações usuárias. Este fator é essencial para evitar a alta latência em ambientes distribuídos como em uma rede WAN [Corbett et al. 2013].

Este artigo apresenta o ALOCS, um repositório de armazenamento distribuído de dados que adota o modelo chave-valor, mas que permite a alocação de um conjunto de pares agrupados em uma única estrutura, denominada *bucket*, cuja localidade é tratada de maneira controlada e orientada pela aplicação usuária do sistema. Assim, embora a interface para a aplicação seja similar aos repositórios NoSQL (baseada nas operações *get-put-rem*), a unidade de comunicação de dados entre servidores é o *bucket*. Dessa forma, somente o acesso ao primeiro par chave-valor armazenado em um *bucket* remoto requer uma comunicação entre servidores. Como o *bucket* é mantido localmente em cache, acessos subsequentes a chaves armazenadas no mesmo *bucket* não requerem nova comunicação. Esta funcionalidade de controle de localidade, aliada a um modelo simples chave-valor, permite que o ALOCS possa ser usado como *backend* de armazenamento para SGBDs, dando suporte a um modelo de *clusterização* distribuído.

No ALOCS, o armazenamento físico de dados é feito em um SAD. Como os SADs apóiam-se em uma estrutura de metadados para associar o dado ao servidor no qual ele está armazenado, o ALOCS permite a replicação destas informações, sendo que o número de réplicas pode ser definido de acordo com o volume de leituras e escritas da aplicação. Ou seja, aplicações com muitas leituras podem possuir diversas réplicas dos metadados para evitar que ele torne-se um gargalo. No entanto, para aplicações com muitas escritas a existência de muitas réplicas torna sua atualização um gargalo e portanto uma arquitetura com poucos servidores de metadados é mais apropriado.

O artigo está organizado da seguinte maneira. A Seção 2 discute os trabalhos correlatos. A Seção 3 descreve o modelo de armazenamento definido neste artigo, assim como a arquitetura da solução proposta. A Seção 4 apresenta os resultados experimentais, que mostram o impacto do sistema de metadados sobre operações de leitura e escrita, bem como o efeito do controle de alocação no tempo de processamento das operações. As conclusões são apresentadas na Seção 5.

2. Trabalhos Correlatos

Algumas características importantes na comparação entre sistemas de armazenamento distribuído são o método utilizado para dispersar e recuperar os dados, a escalabilidade e disponibilidade. A dispersão influencia a escalabilidade do sistema, uma vez que ela é determinante no tempo de recuperação dos dados [Paiva e Rodrigues 2015]. Duas técnicas são comumente utilizadas como *backend* de sistemas NoSQL: DHTs e sistemas de arquivos distribuídos (SADs). Dentre os SADs, o HDFS (*Hadoop Distributed File System*) [Shvachko 2010] e o GFS (*Google File System*) [Ghemawat et al. 2003] não implementam meios efetivos de alocação dos arquivos. Eles fragmentam os arquivos, que são alocados em servidores com maior disponibilidade de recursos (menor carga para o HDFS ou menor uso de armazenamento para o GFS). O Ceph [Weil et al. 2006a], baseado em objetos, mapeia arquivos para um ou mais objetos e os distribui entre os participantes do *cluster* através de uma função de dispersão específica. É permitido criar políticas para a distribuição e replicação dos objetos. Esta funcionalidade garante o controle parcial sobre

a localidade dos dados, o que motivou o seu uso na implementação do ALOCS. O PVFS [Ross et al. 2000] fragmenta dados em arquivos e permite que a alocação dos arquivos seja definida através de uma interface. Contudo, ele não dá suporte à replicação.

Uma das dificuldades de utilização de SADs como *backend* para SGBDs distribuídos é que a unidade de armazenamento é o arquivo, uma granularidade maior que aquela de manipulação de SGBDs, baseada em itens de dados. Assim, bancos de dados NoSQL, como o *MemcacheDB* [Tudorica e Bucur 2011], utilizam protocolos baseados em DHT como mecanismo de dispersão de dados. Contudo, as DHTs tradicionalmente não oferecem mecanismos para o controle de alocação de dados. Soluções que mais se aproximam em atender este requisito distribuem os dados agrupados pela ordem lexicográfica das chaves, como no *Scalaris* [Schütt et al. 2008]. Esta abordagem, no entanto, obriga a aplicação a modificar ou adequar as chaves, adicionando prefixos para possibilitar seu agrupamento. Apesar de as aproximarem, não há garantias de sua alocação no mesmo servidor.

Outras soluções que procuram atender o requisito de localidade baseado em DHT incluem o AutoPlacer [Paiva et al. 2015] e Infinispan[5]. O AutoPlacer apresenta uma técnica de otimização de alocação de dados críticos para obter maior desempenho. Isto é possível coletando dados estatísticos relacionados às consultas remotas sobre estes dados. O controle de localidade não é possível na primeira escrita de um determinado item no repositório, visto que inicialmente não há nenhum dado estatístico sobre as consultas. O Infinispan é um repositório em memória. Ao contrário destas soluções, este artigo propõe um modelo de distribuição com controle de alocação desenvolvido sobre um SAD, mas que mantém a interface simples baseada no modelo chave-valor das DHTs.

3. ALOCS - Um Repositório Chave-Valor com Controle de Localidade

Esta seção apresenta o ALOCS, um repositório chave-valor com controle de localidade dos dados. Ele foi projetado para dar suporte a SGBDs nos quais a coalocação de dados envolvidos em uma consulta é fator determinante para o seu desempenho. As próximas subseções descrevem o modelo de armazenamento proposto (Seção 3.1), a arquitetura do sistema (Seção 3.2) e sua implementação (Seção 3.3).

3.1. Modelo de Armazenamento

O ALOCS adota um modelo de armazenamento baseado em quatro conceitos: (1) par chave-valor, que corresponde à unidade de acesso; (2) *bucket*, que consiste de um conjunto de pares chave-valor e corresponde à unidade comunicação entre servidores; (3) diretório, formado por um conjunto de *buckets* e é a unidade de replicação do modelo; e (4) servidor, que contém um conjunto de diretórios, e corresponde a um servidor físico do sistema de armazenamento distribuído. Os componentes deste modelo definem uma hierarquia (Figura 1), que é usada para a localização dos dados. Ou seja, um caminho do tipo `/Servidor/Diretório/Bucket/Chave`, determina o local de persistência física de um par chave-valor.

O ALOCS oferece um conjunto de operações associado a cada um destes conceitos, tais como a adição e remoção de servidores, bem como adição, remoção e cópia de

[5]`http://infinispan.org`

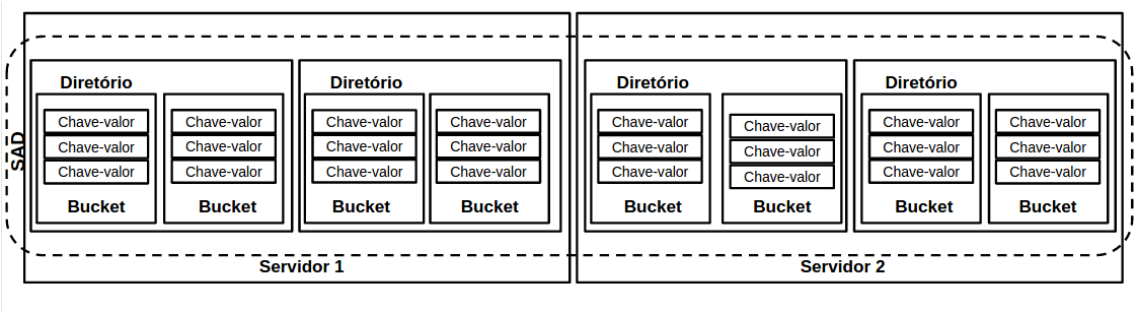

Figura 1. Modelo de armazenamento

diretórios em determinados servidores. Um *bucket* é criado por uma aplicação usuária associando-o a um intervalo de chaves através da operação `create_bucket(dir, idBucket, chaveIni, chaveFim)`. Dessa forma, para executar a operação `put (chave, valor)`, o par é criado dentro do *bucket* responsável pelo intervalo que contém a chave, que já está previamente associado a um diretório, que por sua vez, pode estar replicado em um conjunto de servidores.

A interface de aplicação oferecida pelo ALOCS para pares chave-valor é idêntica à interface dos repositórios chave-valor desenvolvidos sobre DHTs, ou seja, com as operações `get(chave)`, `put(chave, valor)` e `rem(chave)`. Contudo, a execução de uma operação `get` recupera não apenas o valor associado à chave, mas o *bucket* no qual ele está armazenado. Assim, acessos subsequentes a chaves que pertencem ao mesmo *bucket* não requerem novos acessos remotos. O objetivo é dar à aplicação o controle sobre a distribuição dos dados para que ela explore esta funcionalidade a fim de minimizar o custo de comunicação no processamento de consultas.

3.2. Arquitetura

A arquitetura do ALOCS considera um conjunto de nodos (máquinas físicas), como ilustrado na Figura 2. Além de dar suporte ao armazenamento distribuído de dados, o sistema é responsável por manter *metadados* a respeito da associação dos intervalos de chaves aos *buckets* e informações sobre a hierarquia de servidores, diretórios e *buckets*. Assim, os nodos que compõem o repositório distribuído podem ser apenas servidores de dados (como os nodos na parte inferior da ilustração) ou servidores de dados e metadados (como os nodos na parte superior). Os metadados são replicados em todos os servidores de metadados. Os servidores de dados têm como objetivo dar suporte ao armazenamento de um grande volume de dados (escalabilidade de armazenamento). Com a possibilidade dos nodos assumirem papéis distintos, é possível criar um conjunto maior de servidores de dados sem incorrer no custo de replicação dos metadados em todos eles. O custo da replicação pode ser proibitivo para aplicações com um grande volume de escritas. Assim, é possível ajustar a quantidade de servidores de metadados de acordo com o volume de escritas e leituras de cada aplicação.

Os componentes do sistema responsáveis por prover tais funcionalidades são os módulos de controle, módulos de armazenamento e módulos de metadados, que são detalhados na sequência.

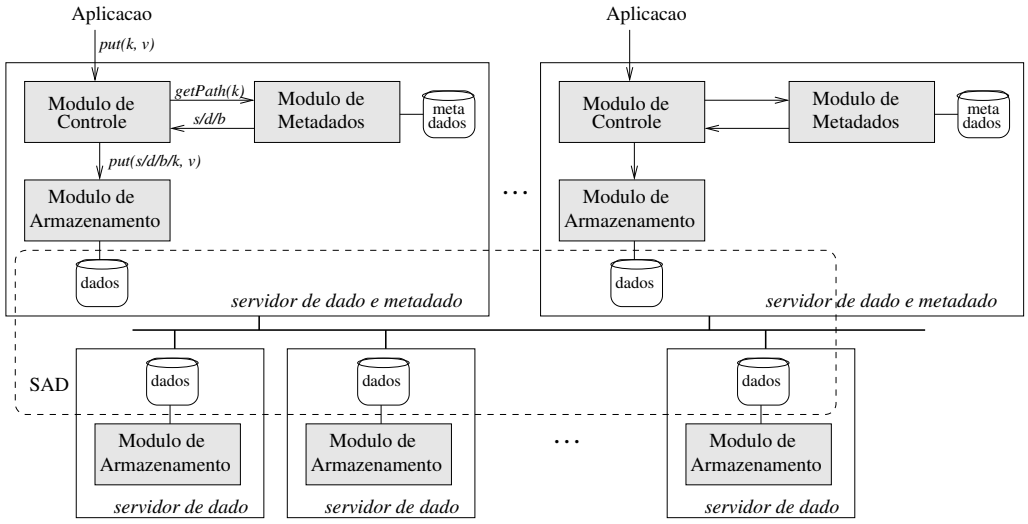

Figura 2. Arquitetura do Repositório de dados

3.2.1. Módulo de Controle

O módulo de controle é responsável por receber as requisições dos programas de aplicação e fazer a interface com os módulos de metadados e de armazenamento. Por exemplo, considere que a aplicação submeta a operação `put(k,v)`. O módulo de controle envia ao módulo de metadados a operação `getPath(k)`, que retorna o caminho servidor/diretório/bucket (`s/d/b`) no qual a chave `k` deve ser armazenada. Com posse destas informações, a operação `put(s/d/b/k, v)` é então enviada para o módulo de armazenamento.

3.2.2. Módulo de Armazenamento

A principal função do módulo de armazenamento é a interface entre o módulo de controle e o sistema de armazenamento distribuído de dados, fazendo o mapeamento entre o modelo implementado pelo ALOCS e o modelo de armazenamento físico. Em sua implementação atual, é adotado um SAD no qual cada *bucket* é armazenado em um arquivo. Assim, a recuperação de um par chave-valor envolve a leitura do arquivo que armazena o *bucket* e a extração do par chave-valor de interesse. O módulo de armazenamento é também responsável pelo gerenciamento do cache. Assim, sempre que uma chave é requisitada, cabe a ele verificar se o *bucket* correspondente já encontra-se no cache para evitar uma nova requisição ao SAD. O cache utilizado funciona apenas para operações de leitura. No caso de operações de escrita, o par chave-valor é atualizado diretamente no disco.

3.2.3. Módulo de Metadados

O módulo de metadados é responsável pelo armazenamento, distribuição e gerenciamento de informações sobre as localidades dos *buckets*. Estas informações são coletadas durante a inserção de dados no sistema, visto que o usuário especifica o local onde deseja armazenar seus dados, isto é, o *bucket*, o diretório e o servidor. No módulo de metadados, cada

bucket é armazenado com o intervalo de chaves correspondentes aos valores armazenados nele. Além disso, os metadados mantém também informações sobre o seu modelo hierárquico de armazenamento. Assim, o módulo de metadados possui 2 subcomponentes:

- **Estrutura Física**: ela é responsável pelo armazenamento do mapeamento físico de todos dispositivos de armazenamento usados no sistema, isto é, todos os servidores com seus diretórios, e estes com todos os seus *buckets*. Um exemplo desta estrutura é ilustrada na Figura 3.

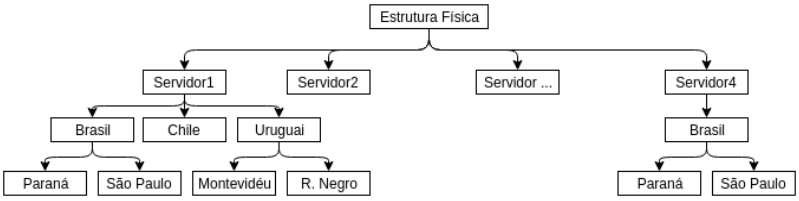

Figura 3. Estrutura Físca do Módulo de Metadados

- **Estrutura de Busca**: ela corresponde a uma estrutura de indexação para facilitar a busca de chaves. Foi adotada uma adaptação da árvore de intervalos, na qual os nodos correspondem aos intervalos de chaves [Pal e Pal 2009]. Cada nodo da árvore mantém, além do intervalo, o valor máximo dentre todos os intervalos em sua subárvore e a localização do *bucket* responsável pelo seu intervalo de chaves (*/servidor/diretório/bucket*). Um exemplo desta árvore é apresentada na Figura 4. O sistema garante que não há sobreposição entre os intervalos no momento da criação dos *buckets*. Assim, a busca na árvore é idêntica à busca em uma árvore binária, considerando apenas o valor inicial de cada intervalo armazenado em cada nodo.

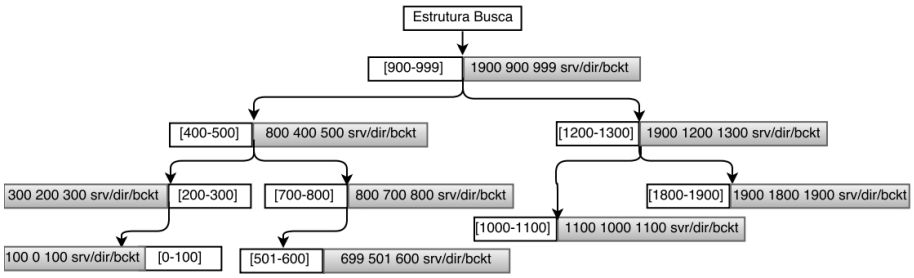

Figura 4. Estrutura de Busca do Módulo de Metadados

3.3. Implementação

O ALOCS utiliza o SAD Ceph[6] para a implementação do módulo de armazenamento e o Zookeeper[7] para o módulo de metadados. Porém, o sistema foi projetado de forma modular, com uma interface bem definida entre os três módulos que o compõem. Dessa forma, outros sistemas de armazenamento e de controle de metadados podem ser considerados no desenvolvimento de versões futuras do ALOCS, bastando para isso manter a interface definida para o módulo.

[6] `ceph.com`
[7] `http://zookeeper.apache.org`

O SAD Ceph foi escolhido por ser distribuído e possuir um algoritmo para distribuição de dados denominado CRUSH [Weil et al. 2006b], que é responsável por alocar e replicar os dados em um *cluster* de servidores. O CRUSH faz a distribuição a partir de um conjunto de regras, que indicam onde um determinado objeto e suas réplicas devem ser alocados, e no mapa do *cluster*, que permite explorar as características do ambiente físico. Esta abordagem concede ao Ceph flexibilidade e controle na alocação e replicação de dados.

O módulo de armazenamento herda do Ceph propriedades como escalabilidade, desempenho e disponibilidade de dados. O conceito usado é o *Object Storage*, onde um arquivo é mapeado para um ou mais objetos e discos convencionais são substituídos por dispositivos denominados *Object Storage Devices* (OSDs). Os objetos são agrupados em estruturas conhecidas como *Placement Group* (PG). Cada PG é associado a uma lista de OSDs, sendo um primário, e os demais réplicas. Tarefas relacionadas ao gerenciamento de dados, como alocação, replicação e recuperação, são distribuídas entre os dispositivos disponíveis, permitindo paralelização das operações com maior eficiência [Azagury et al. 2003]. Para implementar a hierarquia do ALOCS no Ceph, cada *bucket* é mapeado para um objeto do Ceph e cada diretório é mapeado para um PG, associando-o aos OSDs que correspondem aos nodos do servidor ALOCS no qual o diretório deve ser armazenado. Assim, caso o diretório seja replicado, basta associar um novo OSD ao grupo de alocação correspondente.

O objeto que corresponde ao *bucket* contém um cabeçalho onde são armazenadas informações para a localização dos pares chave-valor, denominado *slot*, e um mapa de bits para o controle dos slots livres. O *bucket* é a unidade de transmissão e seu tamanho máximo é configurável. Nos experimentos foi adotado o limite de 64 KBytes.

O módulo de metadados é implementado usando o *Apache Zookeeper*, que é um serviço de coordenação de processos com alta disponibilidade, escalabilidade e confiabilidade [Junqueira e Reed 2009]. Este serviço utiliza unidades de armazenamento denominadas de *znodes*, que são organizadas de acordo com um espaço de nomes hierárquico, semelhante às estruturas de dados de diretórios [Skeirik et al. 2013].

O módulo de metadados foi implementado na linguagem Java. A interação deste módulo com o *Zookeeper* é feita usando a versão Java do *Zookeeper* API. O módulo de controle, implementado em C, interage com o módulo de metadados através de uma interface desenvolvida utilizando o *Java Native Interface* (JNI).

4. Experimentos e Resultados

A fim de validar o modelo proposto neste trabalho, foi elaborado um ambiente de testes com 3 máquinas virtuais Debian 7.0.0, cada uma com 30GB de disco e 2GB de memória RAM. Para os experimentos, foi configurado um *cluster* com 2 servidores de armazenamento e 1 a 3 servidores de metadados. Os módulos de controle e metadados do ALOCS são sempre executados em servidores que armazenam os metadados, a fim de evitar comunicações remotas para a obtenção da localização das chaves. Os servidores de metadados formam um *cluster* (*Zookeeper*) e os de dados um *cluster* Ceph.

Em cada servidor foi criado um único diretório, contendo um conjunto de *buckets*, cada um contendo até 100 pares chave-valor. A replicação, permitida pelo Ceph, não foi utilizada na realização dos experimentos.

Tendo em vista que o modelo de coalocação de dados proposto neste artigo é baseado na manutenção de metadados que relacionam pares chave-valor ao seu armazenamento físico, nesta seção são apresentados dois experimentos que tem como objetivo determinar o custo de acesso aos metadados. Um terceiro experimento descreve o impacto da coalocação dos dados em *buckets* sobre o tempo de acesso aos dados.

Experimento 1: Custo da gravação de metadados
Para determinar o custo da gravação dos metadados, neste experimento é executada uma sequência de comandos que geram *buckets* de intervalos distintos em um mesmo diretório (operação `createBucket(dir, idBucket, chaveIni, chaveFim)`). Cada comando requer duas escritas nos metadados: uma na estrutura física, para inserir o *bucket* `idBucket` como filho de `dir` e outra escrita na estrutura de busca. Além disso, um objeto é escrito no SAD Ceph, contendo apenas o cabeçalho para manter as informações sobre os objetos que serão posteriormente inseridos no *bucket*. A sequência de comandos foi executada em três configurações: com um único servidor de metadados e comandos ordenados pelo valor da chave inicial do intervalo; com um único servidor de metadados e comandos em ordem randômica; e com três servidores de metadados e comandos em ordem randômica. Nesta última configuração, cada comando requer gravação nas três réplicas dos metadados. Os tempos de execução (em milissegundos) de sequências de 5 a 50 operações são apresentados na Figura 5. Pode ser observado que a inicialização sequencial teve custo mais alto do que as randômicas e consiste no cenário de pior caso. Isso se deve ao fato da altura da árvore da Estrutura de Busca ter crescimento linear com esta ordem de inserção, transformando-a em uma lista. Nesta configuração, para 50 operações o tempo total de execução foi de 4650 ms (93 ms por operação). No final da execução, a árvore resultante possui altura 50, o que explica o seu alto tempo de execução.

Já para a inicialização randômica, o tempo total de execução para 50 operações foi de 349 ms (7 ms por operação), com uma redução de 92.5% com relação ao tempo de inicialização sequencial. No final da sequência o altura da árvore era de 6. Isso mostra o grande impacto que o balanceamento da árvore tem sobre o tempo de geração dos *buckets*. Um ponto importante a ser observado é que o tempo total de execução para a configuração com replicação é apenas 34% maior que o tempo sem replicação. Isso mostra que em uma aplicação com muitos operações de leitura é possível replicar os metadados em diversos nodos para obter escalabilidade de processamento, sem que isto tenha um grande impacto sobre o tempo de execução das operações de escrita.

Experimento 2: Custo da leitura dos metadados
Neste segundo experimento foi avaliado o tempo de leitura aos metadados. Para isso, foram executadas sequências de operações de inserção de pares chave-valor nos *buckets* previamente criados, como relatados no Experimento 1. Todas as operações foram submetidas para um único módulo de controle do ALOCS. A Figura 6 apresenta o tempo total de execução (em milissegundos) de sequências de 10 a 100 operações de gravação `put(k, v)`. Observe que cada operação requer uma consulta aos metadados, para obter a localização do *bucket* e a gravação do par chave-valor no SAD Ceph. No gráfico, o tempo de gravação no Ceph é apresentado na barra "Sem metadados", enquanto as demais barras apresentam o tempo total de execução da sequência com os metadados gerados randomicamente e sequencialmente. Ou seja, estas barras representam o *overhead* causado pelo ALOCS para o gerenciamento de metadados sobre o armazenamento no SAD Ceph

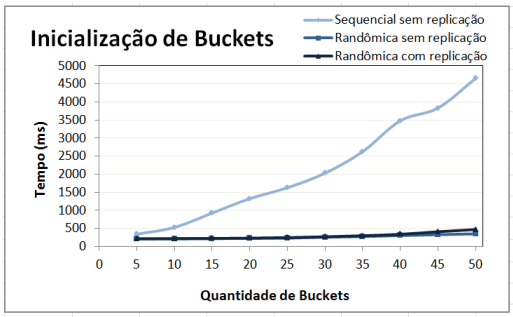

Figura 5. Inicialização - gravação de metadados

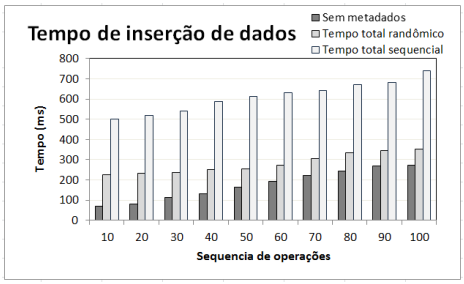

Figura 6. Impacto dos metadados sobre o SAD Ceph

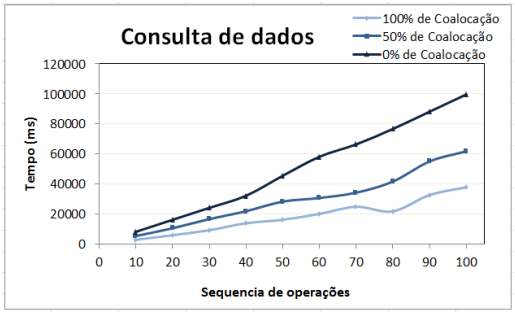

Figura 7. Efeito da localidade de dados

sem controle de localidade. Para uma sequência de 50 operações, o tempo de gravação no SAD foi de 163 ms. O tempo total quando os metadados foram gerados randomicamente foi de 253 ms, o que resulta em uma média de 6 ms por operação de gravação. Quando os metadados foram gerados sequencialmente, o tempo de acesso aos metadados é bem maior, com 13 ms por operação de gravação. Isso era esperado, tendo em vista a altura da árvore de busca resultante com os metadados inseridos desta forma. Embora o custo de acesso aos metadados pareça alto para uma sequência pequena de operações, o *overhead* sobre o tempo de execução total para sequências maiores não é tão grande, como pode ser observado nos resultados com 100 operações.

Experimento 3: Impacto da coalocação de dados

Para avaliar o impacto da coalocação no acesso a um conjunto de dados, neste experimento foram executadas sequências de operações de leitura (`get(k)`) nas quais há um porcentual distinto de coalocação das chaves requisitadas pelas operações. Foram considerados os porcentuais de 0%, 50% e 100%. Ou seja, foram geradas sequências nas quais cada par chave-valor está armazenado em *buckets* distintos (0% de coalocação), com metade dos pares coalocados com um outro par da sequência (50% de coalocação) e sequências com todos os pares alocados no mesmo *bucket* (100% de coalocação). As sequências tem tamanho variando de 10 a 100 e a Figura 7 apresenta os resultados.

Como esperado, o tempo de consultas de dados com 100% de coalocação é menor do que os demais. Isto ocorre porque como o *bucket* é a unidade de transmissão do ALOCS, apenas o primeiro acesso a um par chave-valor de um *bucket* requer uma leitura em um nó remoto. As leituras subsequentes a pares no mesmo *bucket* são executadas localmente no cache do sistema. Para 10 consultas, o tempo médio de execução de cada operação `get(k)` é de 269 ms para dados coalocados em um único *bucket* (100% de

coalocação). Este tempo tem um aumento de 52% para dados com 50% de coalocação e 297,7% para dados sem coalocação. Para sequências de 100 operações, com 100% de coalocação o tempo médio de cada operação foi de 367 ms. Este tempo sobe para 617 ms para 50% de coalocação (169% maior que o tempo com 100% de coalocação) e 996 ms para 0% de coalocação (271% maior). Estes resultados mostram o grande impacto que a coalocação tem sobre o acesso de um conjunto de dados relacionados.

5. Conclusão

O artigo propôs o ALOCS, um repositório de armazenamento distribuído chave-valor com controle de localidade de dados. O ALOCS baseia-se em um modelo de armazenamento hierárquico e nos metadados para poder gerenciar a localidade de dados. Esta proposta possibilita à aplicação a alocação de dados em um determinado servidor no repositório distribuído. Com o modelo proposto neste trabalho é possível otimizar aplicações distribuídas globalmente, aproximando os dados das aplicações usuárias. Isto evita o acesso a múltiplos servidores e diminui a incidência de transições distribuídas. Os experimentos comprovaram que o custo de controle de localidade é baixo. Isto viabiliza a solução proposta e seu modelo de armazenamento. Além disso, o uso da cache para as operações de leitura mostrou-se eficiente.

Como trabalhos futuros, planeja-se realizar experimentos em um ambiente de larga escala para avaliar o impacto da escalabilidade dos servidores de armazenamento e metadados no desempenho das inserções e consultas de dados. Além da replicação, a otimização da árvore de busca adotada pelo módulo de metadados também deverá ser tratada para mantê-la balanceada e assim reduzir seu impacto na inicialização e tempo de leitura dos *buckets*. Por fim, pretende-se adicionar e analisar uma política de atualização de cache.

Referências

Agrawal, D., El Abbadi, A., Antony, S., e Das, S. (2010). Data management challenges in cloud computing infrastructures. *Databases in Networked Information Systems*, páginas 1–10. Springer.

Arnaut, D. E., Schroeder, R., e Hara, C. S. (2011). Phoenix: A relational storage component for the cloud. *Proc. of the 4th IEEE Int. Conf. on Cloud Computing*, páginas 684–691.

Azagury, A., Dreizin, V., Factor, M., Henis, E., Naor, D., Rinetzky, N., Rodeh, O., Satran, J., Tavory, A., e Yerushalmi, L. (2003). Towards an object store. *Proc. of the 20th IEEE Int. Conf. on Mass Storage Systems and Technologies*, páginas 165–176.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27.

Corbett, J. C., Dean, J., et al. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8.

de S. Rodrigues, C. A., de Almeida, J. F., Braganholo, V., e Mattoso, M. (2009). Consulta a bases XML distribuídas em P2P. *Simpósio Brasileiro de Banco de Dados - Sessão de Demos*, páginas 21–26.

Ghemawat, S., Gobioff, H., e Leung, S.-T. (2003). The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43.

Junqueira, F. P. e Reed, B. C. (2009). The life and times of a Zookeeper. *Proc. of the 28th ACM Symposium on Principles of Distributed Computing*, página 4.

Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., e Thompson, J. G. (1985). A trace-driven analysis of the UNIX 4.2 BSD file system. *ACM SIGOPS Operating Systems Review*, 19(5):15–24.

Paiva, J. e Rodrigues, L. (2015). On Data Placement in Distributed Systems. *ACM SIGOPS Operating Systems Review*, 49.

Paiva, J., Ruivo, P., Romano, P., e Rodrigues, L. (2015). Auto Placer. *ACM Transactions on Autonomous and Adaptive Systems*, 9(4).

Pal, A. e Pal, M. (2009). Interval tree and its applications. *Advanced Modeling and Optimization*, 11(3):211–224.

Rani, L. S., Sudhakar, K., e Kumar, S. V. (2014). Distributed file systems: A survey. *International Journal of Computer Science & Information Technologies*, 5(3).

Ribas, E. A., Uba, R., Reinaldo, A. P., de Campos Jr, A., Arnaut, D., e Hara, C. (2011). Layering a dbms on a dht-based storage engine. *Journal of Information and Data Management*, 2(1):59–66.

Ross, R. B., Thakur, R., et al. (2000). PVFS: A parallel file system for linux clusters. *Proceedings of the 4th annual Linux showcase and conference*, páginas 391–430.

Schütt, T., Schintke, F., e Reinefeld, A. (2008). Scalaris: reliable transactional P2P key/value store. *Proc. of the 7th ACM SIGPLAN Workshop on ERLANG*, páginas 41–48.

Shvachko, K. V. (2010). HDFS scalability: The limits to growth. *Login*, 35(2):6–16.

Skeirik, S., Bobba, R. B., e Meseguer, J. (2013). Formal analysis of fault-tolerant group key management using ZooKeeper. *Proc. of the 13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, páginas 636–641.

Tran, V.-T. (2013). *Scalable data-management systems for Big Data*. Tese de doutorado, École normale supérieure de Cachan.

Tudorica, B. G. e Bucur, C. (2011). A comparison between several NoSQL databases with comments and notes. *Proc. of the 10th Roedunet Int. Conf.*, páginas 1–5.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., e Maltzahn, C. (2006a). Ceph: A scalable, high-performance distributed file system. *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, páginas 307–320.

Weil, S. A., Brandt, S. A., Miller, E. L., e Maltzahn, C. (2006b). Crush: Controlled, scalable, decentralized placement of replicated data. *Proc. of the ACM/IEEE Conf. on Supercomputing*, página 122.

Yin, S. e Kaynak, O. (2015). Big data for modern industry: Challenges and trends [Point of View]. *Proc. of the IEEE*, 103(2):143–146.

Zhang, H., Wen, Y., Xie, H., e Yu, N. (2013). A survey on distributed hash table (DHT): Theory, platforms, and applications. Relatório técnico, Nanyang Technological University.