Módulo de consultas distribuídas do Infinispan

Israel Danilo Lacerra

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Francisco Carlos da Rocha Reverbel

São Paulo, janeiro de 2013

Módulo de consultas distribuídas do Infinispan

Esta versão da dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 26/11/2012. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Resumo

Lacerra, I. D. **Módulo de consultas distribuídas do Infinispan**. 2012. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

Com a grande quantidade de informações existentes nas aplicações computacionais hoje em dia, cada vez mais tornam-se necessários mecanismos que facilitem e aumentem o desempenho da recuperação dessas informações. Nesse contexto vem surgindo os bancos de dados chamados de NOSQL, que são bancos de dados tipicamente não relacionais que, em prol da disponibilidade e do desempenho em ambientes com enormes quantidades de dados, abrem mão de requisitos antes vistos como fundamentais.

Neste trabalho iremos lidar com esse cenário ao implementar o módulo de consultas distribuídas do JBoss Infinispan, um sistema de cache distribuído que funciona também como um banco de dados NOSQL em memória. Além de apresentar a implementação desse módulo, iremos falar do surgimento do movimento NOSQL, de como se caracterizam esses bancos e de onde o Infinispan se insere nesse movimento.

Palavras-chave: NOSQL, Infinispan, cache, cache distribuído, consultas distribuídas, *data grid*, sistemas de grade de dados, Lucene.

Abstract

Lacerra, I. D. **Module that supports distributed queries in Infinispan**. 2012. Master thesis - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

With the big amount of data available to computer applications nowadays, there is an increasing need for mechanisms that facilitate the retrieval of such data and improve data access performance. In this context we see the emergence of so-called NOSQL databases, which are databases that are typically non-relational and that give up fulfilling some requirements previously seen as fundamental in order to achieve better availability and performance in big data environments.

In this work we deal with the scenario above and implement a module that supports distributed queries in JBoss Infinispan, a distributed cache system that works also as an in-memory NOSQL database. Besides presenting the implementation of that module, we discuss the emergence of the NOSQL movement, the characterization of NOSQL databases, and where Infinispan fits in this context.

Keywords: NOSQL, distributed queries, cache, distributed cache, Infinispan, data grid, Lucene.

Sumário

Lista de Figuras vii

1 Introdução 1
- 1.1 Objetivos do trabalho . . . 2
- 1.2 Estrutura do texto . . . 2

2 NOSQL e caches distribuídos 5
- 2.1 Sistemas de Cache . . . 5
- 2.2 NOSQL . . . 6
  - 2.2.1 Funcionamento . . . 7
  - 2.2.2 Estrutura de dados . . . 10
  - 2.2.3 *MapReduce* . . . 11
- 2.3 Sistemas de grade de dados . . . 13

3 JBoss Infinispan 15
- 3.1 JSRs relevantes para o Infinispan . . . 15
  - 3.1.1 JSR 107 . . . 15
  - 3.1.2 JSR 347 . . . 16
- 3.2 Uso da API . . . 17
- 3.3 Políticas de despejo . . . 18
- 3.4 Armazenamento em disco . . . 20
- 3.5 Opções de organização do cache . . . 21
  - 3.5.1 Modo Local . . . 21
  - 3.5.2 Modo de replicação . . . 21
  - 3.5.3 Modo de invalidação . . . 21
  - 3.5.4 Modo distribuído . . . 22
- 3.6 *Hash* consistente . . . 23
- 3.7 Comunicação . . . 25

4 Buscas 27
- 4.1 Índice secundário . . . 27
- 4.2 *MapReduce* . . . 30
- 4.3 Buscas no Infinispan . . . 31
  - 4.3.1 Apache Lucene . . . 31
  - 4.3.2 Uso do Hibernate Search . . . 35
  - 4.3.3 Vantagens da abordagem do Infinispan . . . 36

5 Arquitetura Interna do Infinispan 37
- 5.1 `CacheManager` e `Cache` . . . 37
- 5.2 `ReplicableCommand` . . . 38
- 5.3 Visitantes e `InterceptorChain` . . . 40

5.3.1 DistributionInterceptor . . . 41
5.3.2 QueryInterceptor . . . 42
5.3.3 CallInterceptor . . . 42
5.4 RpcManager . . . 43
5.5 DataContainer . . . 43

6 Consultas locais no Infinispan **45**
6.1 Indexação . . . 45
6.2 Consultas . . . 46
6.3 CacheQuery . . . 47
6.4 Iteradores . . . 48

7 Consultas distribuídas no Infinispan **51**
7.1 Indexação . . . 53
7.2 Consulta . . . 53
7.3 ClusteredQueryCommand . . . 53
7.4 ClusteredQueryCommandInvoker . . . 54
7.5 ClusteredCacheQueryImpl . . . 54
7.6 ClusteredCacheQueryImpl.getResultSize() . . . 55
7.7 Iteradores distribuídos . . . 55
7.8 QueryBox . . . 58
7.9 ClusteredCacheQueryImpl.list() . . . 59
7.10 Limitações . . . 59
7.10.1 Impactos de mudanças no aglomerado . . . 59
7.10.2 Paginação . . . 61

8 Considerações Finais **63**
8.1 Contribuições deste trabalho . . . 63
8.2 Trabalhos relacionados . . . 63
8.3 Trabalhos futuros . . . 64
8.3.1 Política de despejo . . . 64
8.3.2 Indexação global . . . 65
8.3.3 Integração com a API JPA . . . 65

Referências Bibliográficas **67**

Lista de Figuras

3.1 `java.util.concurrent.ConcurrentMap` - API 16
3.2 Exemplo de uso da API 17
3.3 Exemplo de invalidação [10] 22
3.4 *Hash* consistente em um aglomerado com 3 nós 23
3.5 *Hash* consistente em um aglomerado com 3 nós e 3 nós virtuais por nó. . . . 24

4.1 Exemplo: Adicionando um documento ao índice 32
4.2 Buscando um documento no índice 33
4.3 Exemplo de classe que será indexada pelo Hibernate Search 35

5.1 Fluxo de uma operação realizada no Infinispan 38
5.2 `ReplicableCommand` API 39
5.3 `VisitableCommand` API 39
5.4 Comandos criados pelo `Cache` a cada operação. 40
5.5 `Visitor` API [9] 41
5.6 `CommandInterceptor` API 41

6.1 Consulta local aos índices secundários 45
6.2 Exemplo de consulta no Infinispan 47
6.3 Métodos de `CacheQuery` 47
6.4 Métodos de `QueryIterator` 48

7.1 Fluxo de informações em uma consulta distribuída 52
7.2 Exemplo do uso da `PriorityQueue` dentro de um `DistributedIterator` 57
7.3 Fluxo de um comando gerado por um `DistributedLazyIterator` em um aglomerado com 4 nós 58

Capítulo 1

Introdução

Atualmente a internet conta com aplicações que demandam enormes quantidades de dados, alta velocidade de acesso e grande disponibilidade de dados. Nesse cenário começaram a surgir soluções para aumentar a eficiência no acesso aos bancos de dados e a disponibilidade dos dados. Soluções chamadas de NOSQL e diversas ideias de cache tentam resolver essas demandas.

Os bancos ditos NOSQL não possuem uma definição muito clara. Eles se caracterizam principalmente por serem bancos de dados distribuídos e não relacionais.

O cache é um componente que armazena dados uma vez acessados para que futuros acessos possam ser feitos de maneira mais rápida, evitando que se repita algum processamento ou se busque a informação em um local mais "distante" computacionalmente como, por exemplo, um banco de dados em um disco.

O JBoss Infinispan [14] é um projeto escrito nas linguagens Java e Scala que, além de prover um serviço de cache local, tem como foco principal oferecer um sistema de cache distribuído que pode ser usado por aplicações que rodam em aglomerados (*clusters*) que tratam e armazenam enormes quantidades de dados. Ele possui ainda algumas características que o fazem figurar também como um banco de dados NOSQL.

No contexto de um aglomerado, um sistema de cache ou banco de dados distribuído oferece vantagens significativas se comparado a um simples sistema de armazenamento local. Considere, por exemplo, um aglomerado com 10 máquinas contendo 3GB de memória cada uma delas. Se nesse aglomerado tivéssemos em cada nó um cache local, cada instância da aplicação teria disponível apenas as informações armazenadas localmente, e contaria com, no máximo, 3GB de espaço para armazená-las. Agora, considere que nesse mesmo aglomerado esteja ativo um cache distribuído. Teríamos 30GB de informações em cache. Poderíamos, conforme a necessidade, adicionar mais máquinas e ter a capacidade do cache ou banco de dados aumentando. Teríamos, portanto, um sistema de dados mais escalável.

Outra vantagem de sistemas de armazenamento distribuído é a possibilidade de termos maior disponibilidade dos dados. Com várias instâncias de armazenamento formando um aglomerado, poderíamos replicar os dados em mais de uma instância e, então, manteríamos a disponibilidade de todos os dados mesmo que alguma máquina se tornasse indisponível.

1.1 Objetivos do trabalho

O Infinispan oferece a possibilidade de fazermos buscas nos dados armazenados usando propriedades (campos) dos objetos lá salvos. No entanto, para que essa funcionalidade esteja ativa, é necessária a criação de um índice para a coleção de todos os objetos que estão armazenados no Infinispan.

Um índice é, a grosso modo, uma estrutura criada a fim de facilitar consultas (*queries*) a uma coleção de dados. Nele guardamos informações adicionais sobre a coleção de objetos de modo a permitir a execução mais eficiente das consultas. Seu funcionamento será explicado mais adiante neste trabalho.

Quando começamos este trabalho o Infinispan não oferecia consultas distribuídas, mesmo que estivesse rodando em um aglomerado. As consultas eram feitas sempre no índice local do nó, que nem sempre continha informações de todos os dados armazenados no aglomerado.

O objetivo deste trabalho é fazer com que as consultas trabalhem de maneira realmente distribuída. O índice deve ser distribuído no aglomerado de maneira que cada nó mantenha um subíndice apenas para a coleção dos objetos residentes em seu cache local. Ao se fazer uma busca no aglomerado, a ideia é que a busca seja disparada em todos os nós e depois as respostas sejam agrupadas em um estilo *map-reduce*.

1.2 Estrutura do texto

O Capítulo 2 explica o surgimento do cenário que deu origem ao Infinispan. Explicaremos como surgiram os bancos de dados NOSQL e como se caracterizam e são usados alguns dos principais bancos de dados desse estilo. Além disso falaremos também do uso de caches e caches distribuídos e de como tudo isso deu origem aos sistemas de grade de dados.

O Capítulo 3 apresenta alguns detalhes do funcionamento do Infinispan, de sua configuração e de algoritmos usados por ele.

Em seguida, no capítulo 4, explicaremos como os bancos de dados NOSQL oferecem a possibilidade de buscas e como o Infinispan o faz, por meio do Apache Lucene e do Hibernate Search.

No capítulo 5 mostraremos alguns detalhes da arquitetura interna do Infinispan. Depois, no capítulo 6, com o intuito de preparamos o leitor para o capítulo 7, apresentaremos como funcionam as buscas locais no Infinispan, que já existiam antes deste projeto. No capítulo 7 mostraremos a implementação do trabalho, discutindo o funcionamento do novo módulo de consultas distribuídas, dando aspectos da arquitetura da solução, quais os desafios descobertos e superados. Finalmente, no capítulo 8 falaremos dos resultados obtidos, comparando com outras soluções além de apresentarmos propostas para trabalhos futuros.

Capítulo 2

NOSQL e caches distribuídos

Para entendermos melhor o cenário no qual se insere o Infinispan, falaremos do surgimento dos bancos de dados NOSQL e, além disso, falaremos a respeito do uso de caches simples e caches distribuídos. Na verdade, sistemas de cache e bancos de dados NOSQL são tecnologias que se complementam e se misturam. Ambas possuem como um de seus principais objetivos aumentar a eficiência e a disponibilidade de dados.

2.1 Sistemas de Cache

Aplicações que efetuam uma grande quantidade de escritas e leituras de dados geralmente acabam por ter seu desempenho limitado por um gargalo em seu banco de dados. Isso acontece em parte devido a estrutura tipicamente não distribuída do banco de dados e em parte pelo acesso ao disco feito por ele, que é bem mais lento que acesso a dados em memória [39].

Uma das possibilidades para tentar amenizar esses problemas são os sistemas de cache. Um cache é uma estrutura utilizada para armazenar dados que inicialmente estão armazenados em alguma estrutura mais distante computacionalmente, com a intenção de aumentar a eficiência de futuros acessos a esses dados. No caso em que funcionam como uma camada entre a aplicação e o banco de dados, os caches são tipicamente estruturas que armazenam pares (chave, valor) na memória. Cada valor fica associado à sua respectiva chave com auxílio de uma função de hash. Dessa forma, em uma segunda leitura de um mesmo dado, a aplicação pode recuperar o dado do cache e evitar o acesso ao banco de dados. Dois destaques dentre os sistemas de cache são o JBoss Cache [13] e o EhCache [4], ambos projetos de código aberto.

Com o passar do tempo e com as aplicações passando a demandar mais dados e maior volume de acesso aos dados, os sistemas de cache passaram a prover opções de armazenamento distribuído em um aglomerado, com o intuito de distribuir a carga de processamento e espaço utilizados. Nesse cenário, um grande destaque é o MemCached [17], um sistema de cache distribuído que foi originalmente desenvolvido em 2003 para o *LiveJournal*, um

serviço que mescla rede social e blogs. Hoje ele é usado por grandes aplicações da Internet, como Facebook, Youtube, Twitter e Wikipedia, entre outras.

A ideia por trás de um sistema de cache como o Memcached é criar um aglomerado de nós com instâncias do cache. Os dados ficam distribuídos entre os nós e, usando uma função de *hash* aplicada à chave que indexa o valor, o aglomerado consegue descobrir onde armazenar ou recuperar um dado.

O Memcached é escrito em C, mas hoje há bibliotecas para clientes em praticamente todas as linguagens de uso comum. Os clientes fazem acesso ao cache usando o *MemCached Protocol*, um protocolo de comunicação baseado em TCP.

2.2 NOSQL

Com o uso crescente de grandes aplicações na Internet, tais como comunidades sociais e gigantes de compras, a demanda por um sistema de dados sempre disponível, que consiga lidar com enormes quantidades de dados em tempo real, tem determinado o surgimento de novas soluções para o armazenamento de dados de maneira mais eficiente nesse cenário.

Muitas dessas soluções tem sido voltadas aos bancos de dados. A quantidade de dados e de acessos aos dados cresceu de tal forma que os tradicionais bancos de dados relacionais passaram a ser um ponto crítico. A partir daí começaram a surgir os bancos de dados classificados como NOSQL. Diferentemente do que esse nome pode sugerir, esses bancos de dados não se caracterizam por não oferecerem consultas via SQL (ainda que tal fato seja usual entre esses bancos). Por isso, alguns autores entendem essa sigla como "Não apenas SQL" (*Not only SQL*). Esses bancos de dados apostam em paradigmas diferentes, principalmente por não serem bancos de dados relacionais. Além disso, esses bancos de dados são geralmente distribuídos, com suporte a replicação e horizontalmente escaláveis [31]. Em prol da eficiência, da escalabilidade e da disponibilidade, esses bancos de dados abrem mão de algumas características oferecidas pelos tradicionais bancos relacionais que antes eram tidas como necessárias.

Na verdade, o uso de bancos de dados não relacionais não é uma ideia nova, pois tais sistemas existem pelo menos desde os anos 60 [38] e, inclusive, precedem os bancos de dados relacionais. Já o termo NOSQL surgiu apenas nos últimos anos, devido ao destaque que esse tipo de banco de dados não relacional alcançou.

O movimento NOSQL foi iniciado por alguns gigantes da Web 2.0. Um dos pioneiros foi o Google, que em 2004 começou a desenvolver o Google BigTable [27]. A Amazon também fez seu próprio banco de dados dito NOSQL, o Dynamo [30]. Trata-se de um banco de dados

não relacional e distribuído que armazena pares chave e valor. Juntos, Dynamo e BigTable influenciaram o surgimento de uma série de bancos de dados inspirados em seus conceitos. Entre esses sistemas está o próprio Infinispan, que foi fortemente influenciado pelo Dynamo.

2.2.1 Funcionamento

O funcionamento dos bancos de dados NOSQL pode variar bastante. Mas, basicamente, todos esses sistemas buscam satisfazer as seguintes propriedades em um cenário com grandes quantidades de dados: durabilidade, disponibilidade, eficiência e escalabilidade.

A estrutura não relacional e a distribuição dos dados facilita a busca desses objetivos. Tipicamente os bancos de dados NOSQL oferecem algum tipo de replicação dos dados, fazendo com que certo dado esteja armazenado em mais de um nó. A replicação proporciona um balanceamento da carga de leitura, já que a aplicação é direcionada para diferentes nós em buscas por um mesmo dado. No caso de sistemas que armazenam os dados somente em memória, a replicação diminui a probabilidade de perda de informações e portanto provê, dentro de certas condições, a durabilidade dos dados. A maneira com que esses sistemas distribuem os dados também facilita a adição de mais nós quando necessário.

Porém, ao usar bancos de dados NOSQL, a aplicação abre mão de alguns requisitos, antes vistos como fundamentais.

Teorema CAP

O teorema CAP, enunciado por Eric Brewer em 2000 [25] e provado por Seth Gilbert e Nancy Lynch em 2002 [32], é um importante teorema a respeito de sistemas de dados distribuídos. Ele diz que um sistema de dados distribuídos só pode satisfazer dois destes três requisitos ao mesmo tempo: consistência (requisito C), disponibilidade (requisito A, de *availability*) e tolerância a partição (requisito P, de*partition-tolerance*).

Consistência. A consistência citada por esse teorema é as vezes erroneamente confundida com a consistência tradicionalmente oferecida por bancos de dados relacionais, que garante que os dados armazenados serão sempre consistentes no sentido de que nunca quebrarão nenhuma restrição de relacionamento entre as tabelas do banco de dados. Mas, no teorema CAP, o termo consistência é usado com um significado diferente. A consistência aqui é a garantia que, após uma escrita em um determinado nó do aglomerado, qualquer leitura desse mesmo item, em qualquer nó do aglomerado, deverá devolver o novo valor. Ou seja, todo o aglomerado enxerga uma única versão de todos os dados.

Disponibilidade. O sistema deve estar disponível para qualquer leitura ou atualização de dados.

Tolerância a partição. O sistema permanece em funcionamento mesmo em um estado de particionamento do aglomerado.

É importante entender exatamente o que o teorema afirma. Em um dado momento, um sistema é capaz de prover no máximo dois dentre os três requisitos CAP. Mas um particionamento na rede é algo que não ocorre o tempo todo. Portanto, na maior parte do funcionamento de um sistema distribuído, de fato só precisamos de consistência e disponibilidade [26]. Para entender melhor, imagine um aglomerado onde houve uma falha de comunicação na rede e uma consequente divisão entre duas partições por um certo intervalo de tempo. Nesse caso, o sistema, se continuar funcionando, pode se comportar basicamente de duas maneiras:

- Permitindo que a escrita ocorra em ambas as partições, ou seja, continuar disponível mas sacrificar a consistência dos dados.
- Evitando que pelo menos uma das partições realize operações de escrita. Nesse caso, o sistema continuaria funcionando apesar da partição e continuaria consistente, mas não estaria disponível para escritas.

Não há como garantir disponibilidade e consistência sem que todos os nós estejam se comunicando. Portanto, o teorema CAP diz respeito a como um sistema de dados distribuídos lida com um particionamento no aglomerado.

Mas a existência ou não de um particionamento é um conceito que depende da latência, isto é, depende do tempo de comunicação entre duas partições. Mesmo em um aglomerado com comunicação funcionando normalmente, há certa latência entre os nós. Há uma certa condição de particionamento, que pode ser maior ou menor dependendo das características da rede. Portanto, se deixarmos que ocorra escrita em dois nós concomitantemente, haverá um certo período de tempo no qual estaremos em um estado de partição e provendo disponibilidade à aplicação. Isso pode nos levar a um estado de inconsistência temporária dos dados. Por outro lado, se um nó ficar impossibilitado de executar alterações devido a uma trava no banco de dados adquirida por um outro nó, nesse exato momento, o sistema está em um estado de particionamento, provendo consistência mas sacrificando temporariamente a disponibilidade.

Como os bancos de dados NOSQL são estruturas de armazenamento distribuído, o teorema CAP explica parte dos compromissos que esses bancos de dados estabelecem para atingir seu objetivo. Como vimos na discussão acima, teríamos uma indisponibilidade ou uma inconsistência por um curto período de tempo. É uma escolha entre disponibilidade ou consistência. Um sistema de dados distribuído não pode optar por sofrer ou não particionamentos. Além disso, uma opção por consistência em detrimento da disponibilidade

tem também efeitos negativos sobre o desempenho, seja pelo custo inerente aos protocolos (como *two-phase commit*, por exemplo) que garantem consistência, seja devido à períodos de indisponibilidade temporária visíveis às aplicações (no caso de espera por travas, por exemplo). Ainda que não seja a estratégia adotada por todos os bancos de dados NOSQL, grande parte deles opta por disponibilidade e desempenho e, então, possuem períodos de inconsistência. São os chamados bancos de dados consistentes em momento indeterminado (*eventually consistent*).

Tipicamente os bancos de dados deixam essas escolhas serem feitas de maneira diferente em cada situação. Há bancos de dados com estratégias mais elaboradas, como o Cassandra, que é um banco de dados feito e usado pelo Facebook, mas de código aberto sendo inclusive mantido atualmente pela Apache. O Cassandra foi desenvolvido pelo Facebook inicialmente com o intuito de armazenar os índices para uma das buscas da rede social que envolvia muitos dados. E como muitos desses bancos de dados NOSQL [38], o Cassandra foi idealizado com o intuito de rodar de maneira distribuída em máquinas medianas e de baixo custo [37].

O Cassandra permite definir a estratégia para cada operação e então podemos conseguir diferentes configurações de CAP dependendo da natureza do dado armazenado. Para cada escrita, por exemplo, a aplicação pode escolher entre:

- escrita em apenas um nó (como a replicação é feita de maneira assíncrona, nesse caso teríamos uma inconsistência temporária);
- escrita em pelo menos dois nós;
- escrita em pelo menos todos os nós de um *data center*;
- escrita que deve ser replicada pelo menos em alguns nós de cada *data center*;
- escrita totalmente consistente.

Além disso, o Cassandra também oferece opções semelhantes para as operações de leitura.

Devido ao grande número de bancos de dados NOSQL que relaxam o requisito de consistência, um termo muito usado para caracterizar as propriedades desses bancos de dados é o BASE. Ele foi definido a partir das ideias do teorema CAP pelo próprio Eric Brewer, em oposição ao ACID, que se refere às propriedades dos bancos de dados tradicionais. O ACID consiste em:

- atomicidade (*atomicity*);
- consistência (*consistency*);

- isolamento (*isolation*);
- durabilidade (*durability*).

Por outro lado, o termo BASE, que faz uma analogia à oposição entre ácido e base, se refere nos seguintes pontos:

- basicamente disponível (*basically available*);
- estado leve (*soft-state*);
- consistente em momento indeterminado (*eventually consistent*).

2.2.2 Estrutura de dados

Apesar de haver consideráveis diferenças na maneira que os bancos de dados NOSQL guardam os dados, podemos distinguir três tipos básicos de estrutura [38]: armazenamento de pares (chave, valor), bancos de dados orientados a colunas e bancos de dados orientados a documentos.

Armazenamento de pares (chave, valor)

Bancos de dados desse tipo são sistemas que armazenam valores indexados por uma chave. Tendo como base a chave usada para armazenar o valor, esses bancos de dados usam estratégias com *hash* para localizar e recuperar o objeto. O funcionamento é semelhante a um sistema de cache distribuído, porém o armazenamento é feito em disco.

O principal destaque entre os bancos de dados NOSQL desse tipo é o Amazon Dynamo que, além de ter influenciado grande parte das funcionalidades do próprio Infinispan, influenciou, entre outros, o Riak, um banco de dados de código aberto mantido pela Basho [21]. Devido ao seu papel de influência no Infinispan e em grande parte dos bancos de dados desse tipo, alguns detalhes do Dynamo serão explicados no decorrer deste trabalho.

Bancos de dados orientados a colunas

Esses bancos de dados oferecem uma estrutura que consiste em um mapa multidimensional indexado por uma chave. A cada chave está associado um conjunto de pares nome e valor. Cada um desses nomes é considerado como o nome de uma coluna. O conjunto de pares associado a uma chave é considerado como uma linha. A aplicação cliente consegue recuperar um dado específico usando o par (chave, nome), ou recuperar todas os pares (nome, valor) disponíveis para uma chave, ou ainda recuperar todos os valores de uma coluna, ou seja, recuperar uma série de pares (chave, valor) associados a uma coluna. Tipicamente esses bancos armazenam as colunas de maneira sequencial, tornando mais eficiente a leitura de

todos os dados de uma coluna. Não há a noção de um esquema que definiria um conjunto de colunas comum a todas as linhas do banco de dados ou de uma certa parte do banco de dados. Portanto, cada linha pode ter um conjunto qualquer de colunas.

Alguns desses bancos de dados oferecem a possibilidade de se criar famílias de colunas, permitindo à aplicação criar grupos de colunas para facilitar algumas operações de busca e inserção. Em um cadastro de clientes, por exemplo, poderíamos ter uma família de colunas chamada "informações pessoais", que agruparia todas as colunas com informações desse tipo.

Dois destaques desse tipo de banco de dados são o Google BigTable e o Cassandra. Para cada valor armazenado, esses dois bancos de dados armazenam também o horário da gravação. Esse horário é usado principalmente para resolução de problemas de consistência, usando o chamado MVCC (*Multiversion concurrency control*) [28]. O MVCC é um modo otimista de lidar com concorrência na escrita de dados em um banco de dados. Quando um processo tem a intenção de fazer uma atualização em uma entrada do banco de dados, essa atualização fica condicionada ao fato dessa versão da entrada ser a mais nova possível.

Bancos de dados orientados a documentos

Como o nome diz, tais bancos armazenam documentos. Tipicamente são armazenados documentos no formato JSON[1], e cada documento pode ser recuperado a partir de um id único que o identifica. Esses bancos de dados geralmente não exigem nenhuma estrutura em comum desses documentos a não ser o que o próprio formato JSON especifica. Isso significa que a aplicação pode criar e armazenar documentos com quaisquer quantidades e tipos de campos. Alguns desses bancos de dados também permitem que a aplicação crie coleções de documentos. Seria algo semelhante a uma tabela em um modelo relacional. Embora não exista nenhuma exigência para o padrão dos documentos em uma coleção, geralmente as coleções são resultados de separações lógicas feitas pelo cliente e é de se esperar que todos os documentos de uma coleção tenham ao menos estruturas semelhantes entre si, ou seja, tenham campos em comum.

Exemplos importantes desse tipo de banco de dados são o Apache CouchDB [1], projeto da Apache escrito em Erlang e o MongoDB, projeto de código aberto escrito em C++.

2.2.3 *MapReduce*

Uma característica típica de bancos de dados NOSQL é oferecer algum suporte a operações em um modelo *MapReduce*. Esse modelo de programação se popularizou a partir de

[1] **JSON** (*JavaScript Object Notation*) é um padrão de arquivos de textos derivado da linguagem JavaScript. Consiste em uma lista de pares (chave, valor). Foi desenvolvido para facilitar a troca de dados entre processos.

um artigo escrito por engenheiros do Google e publicado em 2008 [29].

O nome *MapReduce* foi inspirado nas funções *map* e *reduce* presentes em LISP e em muitas linguagens de programação funcional. Em LISP, a função *map* recebe como parâmetros uma função de mapeamento f e uma lista l. O resultado do *map* será uma lista com os resultados da aplicação de f em cada valor da lista l. A função *reduce* recebe uma função de redução r (que recebe dois parâmetros) e uma lista t e devolve um valor. A lista t será iterada e "reduzida" pela função r. A função r será aplicada aos dois primeiros valores da lista t e o resultado combinado com o próximo valor da lista usando a função r novamente e, assim por diante. Um exemplo de uso do *reduce* seria passarmos a ele uma função que soma dois elementos e uma lista de números. Nesse caso, o *reduce* iria devolver a soma de todos os valores da lista.

Dentro do Google havia uma série de tarefas programadas frequentemente que envolviam uma grande quantidade de dados (muitas vezes armazenados no BigTable) e computações distribuídas. Percebeu-se que muitas dessas tarefas poderiam ser padronizadas usando uma aplicação de *map* seguida de uma aplicação de *reduce*. A partir daí, o Google desenvolveu uma biblioteca para que, usando esse modelo de programação, essas atividades se tornassem menos trabalhosas para os programadores. Os desenvolvedores passaram a se preocupar apenas com a escrita das funções de mapeamento e redução, além de algumas configurações com relação aos dados de entrada e saída. A biblioteca cuida de distribuir a computação das funções de mapeamento e redução entre os nós do aglomerado, cuidando de coisas como coleta dos dados, tolerância a falhas, paralelização, distribuição de carga, etc.

Para entender melhor o uso do modelo *MapReduce*, imagine que queremos contar o número de ocorrências de uma palavra p em uma série de documentos armazenados em um banco de dados NOSQL. Nossa função de mapeamento receberia um documento e retornaria o número de ocorrências da palavra nesse documento. A função seria aplicada a todos os documentos armazenados no banco de dados, e cada nó retornaria uma lista com as respostas de cada aplicação da função. No próximo passo, nossa função de redução receberia as listas com as respostas de cada nó e somaria os resultados, obtendo assim o número total de ocorrências da palavra p nos documentos armazenados.

O *MapReduce* é uma poderosa ferramenta para bancos de dados distribuídos. O modelo pode sofrer algumas variações dependendo do ambiente em que é usado e do objetivo que se deseja alcançar, mas mantém sempre a ideia central que é distribuir computações, conseguindo valores intermediários em uma primeira etapa, e depois consolidar esses resultados intermediários em um único resultado.

2.3 Sistemas de grade de dados

Sistemas de grade de dados (*data grids*) surgiram a partir de uma evolução em caches distribuídos. Geralmente os chamados sistemas de grade de dados são caches distribuídos que possuem:

- suporte a replicação;
- controle transacional;
- suporte a buscas (*queries*) por outros campos além da chave;
- suporte a operações *map/reduce*;
- possibilidade de persistência em disco;
- elasticidade (adição e remoção de nós em tempo de execução).

A partir de agora a linha entre um sistema de grade de dados e um banco de dados NOSQL fica tênue. Com uma estrutura que armazena pares (chave, valor) e que oferece todos os recursos citados acima, podemos classificar sistemas de grade de dados como bancos de dados NOSQL em memória.

E é aqui que se encaixa o JBoss Infinispan, tema deste trabalho. Outro grande destaque entre os sistemas de grade de dados é o Oracle Coherence [20], um produto proprietário.

Capítulo 3

JBoss Infinispan

O Infinispan é, de certa forma, uma evolução do JBoss Cache. Há muito código e funcionalidade aproveitados do JBoss Cache, além das mentes por trás do projeto, que são essencialmente as mesmas. O que antes era um sistema de cache local evoluiu para um sistema de grade de dados. O Infinispan provê replicação, controle de transação (JTA - *Java Transaction API* [12]) e uma API JPA (*Java Persistence API* [22]) para persistência no próprio Infinispan. Essas características o tornam, de fato, um sistema de grade de dados e, portanto, um banco de dados NOSQL em memória. No entanto, mesmo que não seja seu foco principal, podemos usar o Infinispan como um cache local tal qual a utilização do JBoss Cache.

O Infinispan é implementado parte em Java e parte em Scala, mas oferece acesso a outras linguagens que não rodem em cima da JVM por meio de protocolos de comunicação.

3.1 JSRs relevantes para o Infinispan

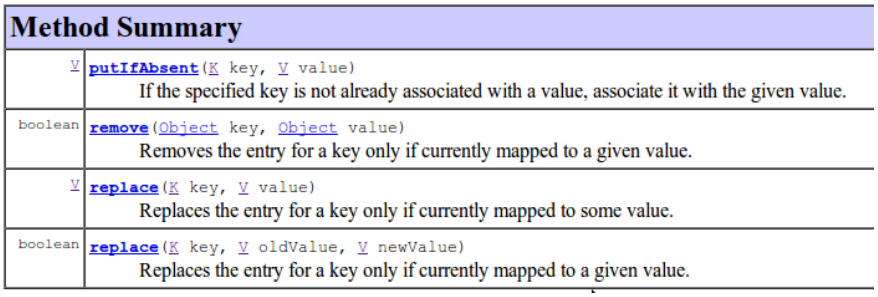
JSRs (*Java Specification Request*) são especificações de interfaces e tecnologias propostas pela comunidade Java para se tornarem padrões na plataforma Java. Nesta seção falaremos de duas JSRs voltadas para a padronização de algumas das funcionalidades oferecidas pelo Infinispan.

3.1.1 JSR 107

O Infinispan provê seu serviço armazenando pares (chave, valor) e implementa parte da especificação JSR 107 (JCACHE - Java Temporary Caching API) [15] da Sun. Essa especificação começou a ser desenhada em 2001 e, após ficar alguns anos adormecida, voltou a ter atividades e inclusive há a intenção de incluir a API na próxima versão da plataforma Java EE [1].

[1] **Java EE:** *Java Enterprise Edition*, padrão que especifica o conjunto de funcionalidades de uma plataforma Java voltada para aplicações corporativas

Mesmo sem ainda estar completa, vários sistemas (como por exemplo o EhCache e o Google App Engine [5]) implementam parcialmente a especificação JSR 107. Essa especificação define, por enquanto, uma interface `javax.cache.Cache` que estende a interface `java.util.concurrent.ConcurrentMap` e adiciona algumas funcionalidades importantes de um cache.



Method Summary

V	**putIfAbsent**(K key, V value) If the specified key is not already associated with a value, associate it with the given value.
boolean	**remove**(Object key, Object value) Removes the entry for a key only if currently mapped to a given value.
V	**replace**(K key, V value) Replaces the entry for a key only if currently mapped to some value.
boolean	**replace**(K key, V oldValue, V newValue) Replaces the entry for a key only if currently mapped to a given value.

Figura 3.1: `java.util.concurrent.ConcurrentMap` - API

Quando usamos um `ConcurrentMap` com a função de um cache, geralmente precisamos de mais funcionalidades pra tornar o cache realmente robusto, tais como políticas de despejo (para evitar estouros de memória), suporte a chamadas assíncronas, transações, operações atômicas, etc. São essas funcionalidades adicionais que tornam um cache diferente de um simples `ConcurrentMap`. O objetivo da JSR 107 é justamente padronizar uma API de acesso a essas funcionalidades [35].

3.1.2 JSR 347

A linha que distingue um sistema de cache de um sistema de grade de dados é bem tênue. Com a intenção de evitar que a JSR 107 se torne muito complexa ao se expandir para um ambiente distribuído e acabe englobando mais funcionalidades do que as necessárias para um cache, Manik Surtani, o líder de desenvolvimento do Infinispan, propôs a criação da JSR 347 (*Data Grids for the Java Platform*), que especifica o comportamento e uma interface comum para as ferramentas de grades de dados [16].

Usando a JSR 107 como ponto de partida, a ideia da JSR 347 é estender a especificação para um ambiente distribuído usando conceitos do Infinispan. O conjunto de funcionalidades ainda não está decidido, mas abordará temas como particionamento, replicação, consistência em momento indeterminado, entre outros [34]. Embora esses temas surjam também no contexto dos bancos de dados NOSQL, é importante notar que a JSR 347 não irá especificar o que são e como devem ser implementados esses bancos.

O Infinispan deverá implementar tanto a JSR 107 quanto a JSR 347.

3.2 Uso da API

O Infinispan pode ser usado como um sistema de armazenamento separado da aplicação, como tradicionalmente são usados os bancos de dados NOSQL. Nesse caso ele pode ser acessado por meio de uma API REST (*Representational State Transfer*), do protocolo *Memcached* ou do protocolo HotRod, que foi desenvolvido pela própria equipe do Infinispan.

Podemos também usar o Infinispan de maneira embarcada em nossa aplicação. Nesse caso a aplicação deve rodar em cima da JVM (*Java Virtual Machine*). O uso da API Java do Infinispan é bem intuitivo, como podemos ver na figura 3.2. Podemos usar os métodos tradicionais herdados da interface `ConcurrentMap`. Porém, é importante notar que alguns desses métodos herdados não possuem o comportamento esperado em um ambiente distribuído, pois tal comportamento poderia prejudicar o desempenho da grade de dados. Métodos como `size()`, `keySet()` e `entrySet()` não garantem valores exatos. Nenhum deles obtém informações globais da grade, apenas informações locais. E, além disso, eles também não adquirem nenhuma trava e, portanto, podem disponibilizar valores não consistentes com o estado atual da grade.

```
// instanciando o gerenciador do infinispan
EmbeddedCacheManager manager = new DefaultCacheManager();

// obtendo o cache
Cache<?, ?> cache = manager.getCache();

// adicionando valores
cache.put("chave1", "valor1");

// esse valor só ficara no cache por 5 segundos
cache.put("chave2", "valorMortal", 5, TimeUnit.SECONDS);

// esse valor ficará no cache por 60 segundos ou no máximo 10 segundos sem ser acessado
cache.put("chave3", "valorMortal3", 60, TimeUnit.SECONDS, 10, TimeUnit.SECONDS);

// esse valor será adicionado ao cache, de maneira assíncrona
// (a chamada desse método não será bloqueante)
cache.putAsync("chave4", "async");

// recuperando um valor no cache: "valor1"
String valor1 = cache.get("chave1");
```

Figura 3.2: Exemplo de uso da API

Temos a possibilidade de criar entradas mortais ou imortais na grade de dados. Uma entrada imortal, adicionada por um simples `put(Object key, Object Value)`, é uma entrada que só deixará o Infinispan quando for manualmente removida ou quando a política de despejo do aglomerado o fizer. Podemos também adicionar uma entrada mortal que tem um "prazo de validade" baseado no seu tempo de vida na grade de dados ou também no tempo de inatividade, ou seja, em uma quantidade de tempo máxima que o objeto permanece armazenado sem que ninguém o acesse (via `get()`, por exemplo). Como vemos no exemplo

da figura 3.2, podemos configurar esse prazo de validade das entradas através de argumentos passados ao método `put()`. Há também a possibilidade de configurarmos previamente um valor padrão que será usado para todas as entradas armazenadas:

```
<expiration lifespan="1000" maxIdle="500"/>
```

Nesse caso, por exemplo, definimos que por padrão todas as entradas serão mortais e serão despejadas 1000 milissegundos após serem adicionadas na grade, ou após 500 milissegundos inativas. Esse comportamento só será adotado para as entradas que forem adicionadas sem que seja passado o período de vida para o método `put()`.

Nesse exemplo mostramos como definir essa configuração via XML. Mas, além de poder realizar todas as configurações em um arquivo XML, o Infinispan pode ser configurado também via Java. Ao instanciar o cache podemos alterar qualquer um dos comportamentos que são definidos no XML.

3.3 Políticas de despejo

O Infinispan usa políticas de despejo (*eviction*) para evitar que exista falta de espaço e não consigamos mais armazenar valores. Essas políticas de despejo são configuradas independentemente para cada instância do cache, ou seja, para cada nó do aglomerado. Quando um dado é eliminado da memória devido a essa política, o despejo ocorre apenas no nó em questão e possíveis réplicas armazenadas em outros nós não sofrem quaisquer alterações.

Por outro lado, informações de prazo de vida de um valor, que são configuradas ao adicionar um dado no Infinispan, são válidas globalmente, ou seja, em todos os nós do aglomerado. Quando um valor é removido devido ao fim do seu prazo de validade, ele é eliminado de todos os nós do aglomerado.

O uso de uma política de despejo não é obrigatória e por padrão ela é desabilitada. No entanto, aplicações robustas correm sérios riscos de sofrer com falta de memória se não houver despejo na memória. O Infinispan não fará qualquer checagem com relação ao espaço livre. Se habilitada, a política de despejo possui 3 configurações possíveis, que são explicadas a seguir.

`EvictionStrategy.LRU`. Com essa configuração, sempre que houver despejo, as entradas que estiverem há mais tempo sem ser acessadas serão despejadas.

`EvictionStrategy.LIRS`. A estratégia LRU funciona bem em muitas aplicações. No entanto, essa estratégia supõe que a entrada que está há mais tempo sem ser acessada será a que tem menor probabilidade de ser acessada novamente. Essa suposição não é boa em

alguns casos. Muitas vezes, entradas criadas mais recentemente não serão mais usadas e entradas que estão há mais tempo sem ser acessadas são requisitadas periodicamente. Baseado nesse fato, o algoritmo LIRS (*Low Inter-reference Recency Set*) [36], divide a memória em dois conjuntos de dados: um conjunto maior, com as dados que são acessados periodicamente e um conjunto menor, com dados novos. Com o passar do tempo, um dado pode passar do conjunto recente para o conjunto de dados que são mais acessados. Sempre que há a necessidade de se liberar espaço no Infinispan, esse espaço é liberado no conjunto de dados recentes. Essa estratégia lembra a essência da coleta geracional feita por coletores de lixo de algumas linguagens de programação.

`EvictionStrategy.UNORDERED`: Com essa estratégia as entradas serão despejadas em uma ordem arbitrária. Com as estratégias anteriores temos uma certa perda de desempenho, já que cada nó terá que manter algumas informações para poder fazer a escolha de quais dados serão despejados. Por isso a criação da estratégia UNORDERED, que foi feita em uma das versões mais recentes. Nesse caso temos despejo, evitando que a memória seja esgotada, mas não temos nenhuma perda de desempenho com relação à manutenção das ordens de acesso ou criação dos dados.

Por padrão, se a política de despejos estiver habilitada, mas nenhuma estratégia estiver configurada, o Infinispan usa a estratégia LIRS.

Um exemplo de configuração da política de despejo seria:

```
<eviction strategy="LRU" maxEntries="2000"/>
```

Nesse caso teríamos uma estratégia do tipo LRU que comportaria no máximo 2000 objetos armazenados no cache. Ou seja, quando o cache atingir o limite de 2000 entradas armazenadas, ocorrerá o despejo da entrada que está há mais tempo sem ser acessada.

Quando ocorre uma política de despejo, dependendo da estratégia escolhida, haverá uma trava no cache por um intervalo de tempo, e a aplicação não conseguirá escrever nesse intervalo. O processo que faz o despejo precisa travar a escrita para analisar qual dado será despejado sem que uma mudança o atrapalhe. Com o intuito de minimizar esse problema, há mais um detalhe envolvido na política de despejo. O Infinispan divide cada instância do cache em segmentos de mesmo tamanho, os quais são verificados individualmente para efeitos de despejo. Ou seja, o Infinispan não verifica se o cache atingiu o valor máximo de entradas, mas verifica se cada segmento do cache atingiu a fração do valor máximo obtida dividindo-se esse valor pelo número de segmentos. Assim, quando há despejo, a trava de escrita ocorre apenas no segmento em questão. Em contrapartida, ao adotar essa estratégia, o Infinispan não garante que o número máximo de entradas será exatamente o configurado, já que sempre que um segmento atinge seu limite, o despejo ocorre sem que necessariamente

o cache esteja em seu limite.

Quando um dado é despejado da memória, não necessariamente ele é totalmente expurgado do Infinispan. Isso acontece porque o Infinispan pode ser usado com uma camada de persistência por trás dele, como veremos na próxima seção. Porém, dados eliminados devido ao prazo de validade são realmente expurgados da grade de dados.

3.4 Armazenamento em disco

O Infinispan permite ao usuário a opção de trabalhar com mecanismos de armazenamento em disco por trás da grade de dados. Essa configuração é feita individualmente em cada nó do aglomerado, podendo apontar para diferentes sistemas de armazenamento. Além disso, cada nó pode ter mais de uma base de armazenamento por trás. Quando uma tentativa de leitura é feita em um nó e o dado não se encontra na memória, todas as bases de dados serão lidas até que um valor válido seja encontrado. Em caso de escrita de um dado, a escrita é efetuada em todas as bases secundárias.

O Infinispan oferece vários ajustes finos com relação ao uso de bases secundárias em disco. O principal deles diz respeito à relação da base de dados secundária com a política de despejo, se ela estiver ativada. Se usarmos alguma base secundária de dados com a configuração padrão, a base secundária terá armazenada um super conjunto dos dados armazenados na memória. Ela terá armazenado os dados que estão na memória e também os dados que em algum momento estiveram na memória mas foram despejados pelo Infinispan segundo a política de despejo configurada.

Alternativamente, podemos ativar a configuração de passivação (*passivation*). Com ela, os dados que estão em memória e os dados que estão armazenados na base secundária formarão dois conjuntos complementares. Ou seja, a base secundária em disco terá apenas os dados que foram despejados da memória. Nesse caso, sempre que um dado for despejado, ele será transferido à base secundária. Quando a aplicação buscar um dado que está apenas no disco, isto é, um dado que foi despejado da memória, o dado será reativado, sendo movido da base secundária para a memória.

Usando o modo de passivação temos um ganho no momento da escrita de um dado, já que o dado não precisará ser escrito na base secundária. No entanto, se houver uma falha no nó, a possibilidade dos dados em memória serem perdidos é maior em relação aos dados em disco e, portanto, usando a passivação o Infinispan fica menos protegido com relação a durabilidade dos dados.

É importante notar que se o Infinispan estiver funcionando sem política de despejo, o

armazenamento secundário será uma cópia simples do armazenamento em memória.

3.5 Opções de organização do cache

O Infinispan oferece basicamente 4 possibilidades de funcionamento: modo local, modo de replicação, modo de invalidação e modo distribuído. Além disso, para cada uma dessas possibilidades, podemos ainda configurar se a comunicação entre os nós será síncrona ou assíncrona.

3.5.1 Modo Local

Nessa configuração temos um sistema de cache simples, rodando em uma máquina. Nesse caso o sistema funciona de maneira parecida com o JBoss Cache. Se o Infinispan estiver rodando em modo local em cada nó de um aglomerado, teríamos caches espalhados nas máquinas, trabalhando de maneira independente, com informações locais apenas.

3.5.2 Modo de replicação

Aqui teríamos o estado replicado entre todos os nós do aglomerado. Cada nó teria seu cache com as informações replicadas, ou seja, a cada momento em que houvesse alguma alteração no estado do cache, essa alteração seria replicada pra todos os outros nós do aglomerado. Nessa configuração temos perda de desempenho ao modificar alguma entrada no cache, pois a modificação deverá ser repassada a todos os integrantes do aglomerado. Mas, por outro lado, existe um ganho ao recuperarmos uma informação no cache, pois essa será uma interação sempre local.

Essa configuração não é escalável para grandes quantidades de nós, devido à grande quantidade de trocas de informações entre os nós a cada alteração nos dados armazenados. Além disso, a capacidade de armazenamento fica limitada à memória de um único nó do aglomerado.

3.5.3 Modo de invalidação

Nesse caso cada nó tem sua instância do cache e a única ligação entre os nós é a invalidação de dados. Se um nó atualiza ou remove uma informação no cache, este nó notifica o restante do aglomerado que houve atualização no dado e, então, os outros nós invalidam essa entrada em seu armazenamento local.

Essa configuração só faz sentido se o Infinispan estiver sendo usado como uma camada de cache para amenizar leituras em um local mais "distante", como um banco de dados comum a todos os nós. Assim haveria a consistência entre os valores do cache e do banco de dados.

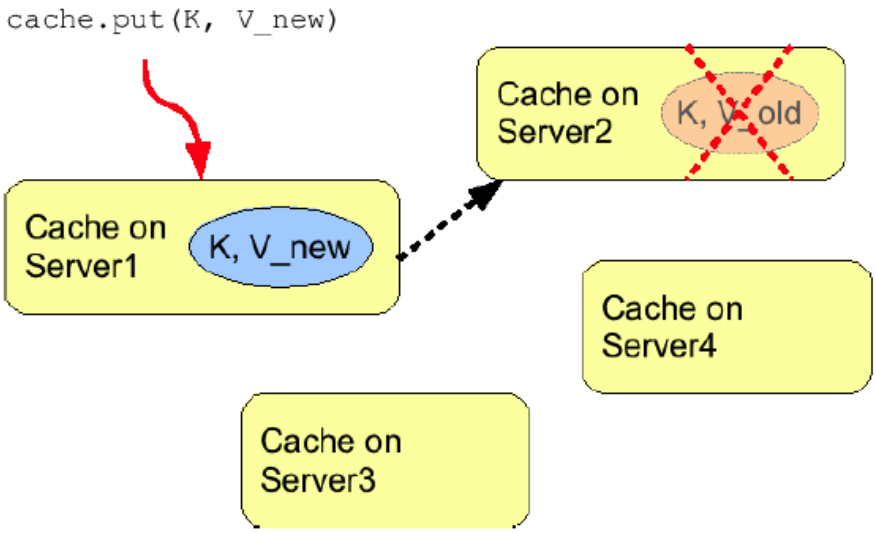


Figura 3.3: Exemplo de invalidação [10]

Em contrapartida, ainda que o nó não replique as alterações como na configuração explicada no item anterior, o nó deverá avisar todos os outros nós do aglomerado sobre a invalidação.

3.5.4 Modo distribuído

Essa é a configuração que permite maior escalabilidade e, portanto, a mais interessante para grandes quantidades de dados e nós. Aqui o Infinispan se comporta como um banco de dados NOSQL em memória, e tem vários detalhes de sua arquitetura influenciados pelo Dynamo.

Nessa configuração os dados ficam distribuídos entre os nós do aglomerado e cada nó é responsável por manter em seu cache uma certa porção de dados. Essa definição é feita usando-se o *hash* consistente, um dos principais conceitos herdados do Dynamo e que será melhor explicado na próxima seção deste trabalho.

Opcionalmente esse modo permite ainda mais um nível de cache. Esse nível é denominado cache L1 (por analogia com os caches nível 1 presentes nas CPUs atuais) e é mantido localmente em um nó para guardar valores frequentemente acessados mas que se encontram em outro nó do aglomerado. Com isso, temos um ganho ao se recuperar informações não locais repetidamente no cache, pois se um nó precisa muitas vezes de um dado cujo armazenamento é feito em outro nó segundo a política de distribuição dos dados, evitamos que a busca seja sempre feita no nó que de fato é responsável pelo armazenamento.

Por outro lado, quando há alguma alteração (atualização ou remoção) no estado de algum valor armazenado na grade de dados, o nó que mantém o valor atualizado ou removido

terá que se responsabilizar pela invalidação desse valor em todos os outros nós, para que ele possa ser removido dos respectivos caches L1. Além disso, o uso do cache L1 aumenta também o consumo de memória total do Infinispan, já que os valores passam a ser guardados em mais nós.

3.6 *Hash* consistente

O *hash* consistente é a estratégia usada pelo Dynamo e muitos bancos de dados influenciados por ele para distribuir os dados nos nós do aglomerado. Ou seja, com o auxílio de uma função de hash, o sistema é capaz de decidir em qual nó uma entrada do banco de dados deve ser lida ou salva.

Explicaremos nesta seção como o Infinispan usa essa estratégia (que é levemente diferente em relação ao uso no Dynamo). Primeiramente considera-se um anel formado pela imagem de uma função de *hash*, de forma que após o último valor da imagem, voltamos ao primeiro valor dela. Ou seja, o anel é formado unindo as duas "pontas" da imagem da função de hash.

Cada nó é mapeado no anel aplicando a função de *hash* no seu endereço. E cada dado armazenado no Infinispan é mapeado no anel aplicando a função de *hash* na chave que o indexa. E então, um nó é responsável por dados mapeados entre ele e o próximo nó no sentido anti-horário. No caso do Infinispan, é usada a função de hash MurmurHash3 [18].

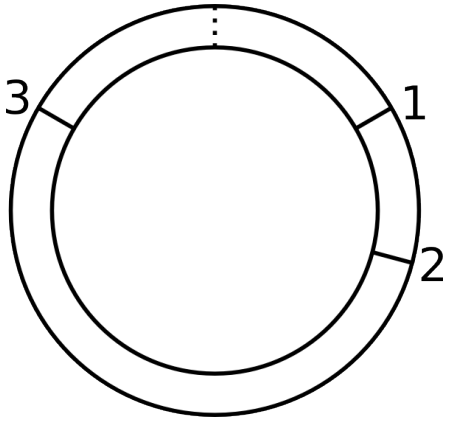


Figura 3.4: *Hash* consistente em um aglomerado com 3 nós

Na figura 3.4, a linha tracejada indica o ponto de junção do anel. Ou seja, no sentido horário, imediatamente após a linha tracejada, temos o começo da imagem da função de *hash* e imediatamente antes desta linha, temos o fim da imagem da função de *hash*. Além disso, todos os valores mapeados no espaço entre os nós um e três serão armazenados no nó um.

A replicação dos dados é feita também com base neste anel. Cada dado fica replicado em N nós, onde N é configurável. Quando calculamos a localização de um dado no anel, replicamos o dado também nos nós responsáveis pelas próximas $N-1$ partições no sentido horário. Se tivermos um aglomerado com $N = 2$, no caso do exemplo da figura 3.4, um dado mapeado no espaço entre os nós um e três teria o nó um como responsável, mas também seria replicado no nó número dois. Assim, em caso de falha no nó um, teríamos uma cópia de segurança no nó dois.

Uma das outras intenções da estratégia de *hash* consistente é fazer o menor número possível de movimentação de dados quando houver um *rehash* no aglomerado, isto é, quando houver a entrada ou saída de um nó. Com essa estratégia, apenas os nós vizinhos ao nó que causou o *rehash* seriam afetados. O restante do aglomerado não sofreria quaisquer alterações.

Mas, mesmo assim, ainda podemos otimizar o processo de distribuição. O algoritmo supõe que os valores armazenados são espalhados de maneira uniforme pelo anel. Isso acontece porque tipicamente teremos um número de dados grande o bastante para espalhar até mesmo as chaves mais populares do aglomerado. Porém, não podemos afirmar o mesmo em relação ao número de nós do aglomerado. Na figura 3.4 temos um exemplo do que isso pode acarretar. Repare que o nó três é responsável por uma área razoavelmente maior que os demais nós. Além disso, se houvesse falha neste nó, o *rehash* causaria uma grande transferência de dados em direção ao nó um.

Para minimizar esses problemas o Dynamo e o Infinispan usam o conceito de nó virtual. A ideia é que cada nó do aglomerado seja representado por V (um valor configurável) nós virtuais. Na figura 3.5, vemos o mesmo aglomerado da figura 3.4, usando nós virtuais ($V = 3$).

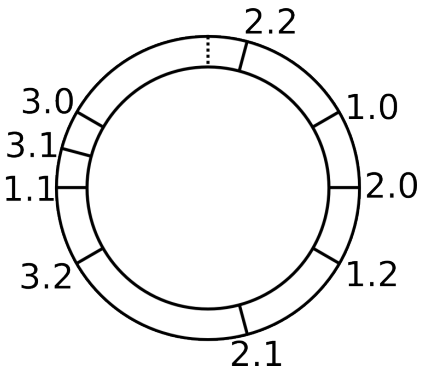


Figura 3.5: *Hash* consistente em um aglomerado com 3 nós e 3 nós virtuais por nó.

Para mapearmos os nós virtuais usamos uma concatenação do endereço real do nó com um índice i tal que i é maior ou igual a 0 e menor que V. Assim, por exemplo, o nó real 3

está divido entre 3 nós virtuais: 3.0, 3.1 e 3.2.

Podemos verificar na figura que com os nós virtuais, a distribuição dos dados ficou mais regular. Quanto maior o número de nós virtuais, melhor ficará a distribuição. Para percebermos isso, basta pensarmos que a imagem de um hash é discreta. Portanto, se usarmos nós virtuais o suficiente para que cada nó virtual seja responsável apenas por um valor da imagem, teríamos a distribuição ideal, já que cada nó real seria responsável pela mesma quantidade de valores da imagem. No Infinispan, o número padrão de nós virtuais para cada nó físico é 48.

Outra característica do uso de nós virtuais é uma melhor divisão dos dados em caso de um *rehash*. Em caso de falha em um nó, os dados armazenados em cada nó virtual são repassados ao respectivo vizinho, o que faz com que os dados não sejam repassados a apenas um nó físico. No exemplo da figura 3.5, se houver uma falha no nó 2 e ele então deixar o aglomerado, seus dados seriam divididos entre os nós virtuais 1.2, 3.2 e 1.0. Ou seja, os dados seriam distribuídos entre os nós 1 e 3.

Mas com a utilização de nós virtuais, a replicação dos dados deve tomar algumas precauções. Repare que, sem o devido cuidado, um dado do nó virtual 3.2 poderia ser replicado para o nó virtual 3.1, o que de fato não aumentaria a disponibilidade do dado. Portanto, a replicação deve evitar que os nós virtuais usados na replicação sejam de um mesmo nó. O Infinispan, além de evitar este problema, procura evitar que a replicação ocorra em máquinas que estão até em um mesmo *rack*.

O Infinispan permite configurar quantos nós virtuais por nó físico existirão, quantas replicações existirão de cada entrada, se a replicação irá ou não tentar evitar máquinas próximas (em um mesmo *rack*). Mas o cliente não pode especificar diretamente em qual nó será armazenado um valor. A escolha é feita sempre usando a chave e o *hash* consistente. Porém, há a possibilidade de se obter previamente uma chave que será mapeada em um nó desejado. E, nesse caso, a aplicação pode então adicionar o valor no Infinispan usando essa chave e, indiretamente, forçar a localização do valor. No entanto, não há qualquer garantia da localização desse valor após um *rehash*. O aglomerado sempre consegue calcular a posição de um determinado dado usando apenas sua chave, sem que informações sobre localização sejam mantidas.

3.7 Comunicação

A comunicação entre os nós do Infinispan é feita com o auxílio do JGroups [23]. O JGroups é um projeto de código aberto que fornece um *framework* para grupos de processos se comunicarem por meio de uma camada de transporte configurável (TCP, UDP ou

JMS). Ele oferece serviços de comunicação de grupo e de manutenção dos elementos de um grupo. O Infinispan é distribuído com duas configurações básicas do JGroups predefinidas, uma usando TCP e outra usando UDP. A partir dessas configurações podemos fazer vários outros ajustes finos fornecidos pelo JGroups ou até mesmo criar uma outra configuração.

O Infinispan permite configurar também algumas opções que privilegiam o desempenho em detrimento da consistência. Essas opções dizem respeito ao sincronismo na comunicação entre os nós. Podemos configurar a grade para que toda requisição de replicação, distribuição ou invalidação seja feita de maneira assíncrona. Há ainda uma configuração que vai mais adiante e permite que o passo de empacotamento (*marshalling*) feito antes de enviar as requisições para os outros nós também seja assíncrono. Essa última configuração pode trazer ainda mais problemas de consistência. Com a seriação assíncrona o Infinispan não garante que a ordem das requisições feitas seja mantida, isto é, duas chamadas de um mesmo nó podem sem transmitidas para o aglomerado em ordem invertida, dependendo do tempo que a cada chamada vai perder na etapa de seriação.

Capítulo 4

Buscas

Um dos recursos que os tradicionais bancos relacionais ofereciam e não é oferecido de maneira tão eficiente pelos bancos de dados NOSQL é a busca usando um ou mais campos dos dados armazenados. Bancos de dados NOSQL tipicamente só oferecem buscas eficientes pelas chaves às quais os valores foram associados no momento do armazenamento. Como dissemos no capítulo 2, a falta de suporte a consultas SQL não é o que caracteriza de fato esses bancos, mas o suporte a tais consultas fica bastante prejudicado com o uso desse tipo de banco.

As chaves associadas aos valores geralmente não são interpretadas pelos bancos NOSQL. Elas são usadas apenas pela função de *hash* para mapear os valores. Isso dificulta inclusive a execução de buscas mais complexas, que dependam do significado de uma chave (por exemplo, buscas por um intervalo de valores de uma chave). A indexação é feita apenas usando o valor exato da chave. Além disso, geralmente apenas por meio dessas chaves podemos incluir, atualizar ou remover dados.

No entanto, algumas aplicações necessitam de buscas mais complexas, que envolvem um ou mais campos dos valores armazenados. Por isso, o Infinispan e muitos outros bancos de dados NOSQL oferecem algumas alternativas para que esse tipo de busca possa ser feita (mesmo que não seja de uma maneira tão eficiente quanto a maneira com que geralmente os bancos de dados relacionais oferecem esse recurso).

Apesar das muitas diferenças entre os bancos de dados NOSQL, podemos identificar duas estratégias principais que vem sendo usadas por esses bancos [40]: indexação secundária e uso de *mapreduce*.

4.1 Índice secundário

Quando falamos de índices em bancos de dados ou outras estruturas que permitem buscas, estamos falando de algo semelhante a um índice de um livro. Trata-se de uma técnica que permita buscar algum conteúdo de forma mais rápida e inteligente.

A ideia é evitar fazer buscas sequenciais como, por exemplo, folhear um livro até encontrar um conteúdo em especial. Esse procedimento pode ser bem lento. O mesmo acontece com algoritmos que buscam palavras em estruturas de dados. Usualmente se faz antes alguma espécie de índice dos dados armazenados para que a busca tenha um melhor desempenho.

Alguns bancos de dados NOSQL facilitam à aplicação a criação de índices secundários para busca dos valores armazenados. Essa indexação é chamada de secundária pois a indexação principal dos valores é formada de fato apenas pela chave associada ao valor.

Bancos de dados relacionais geralmente mantém seus índices usando árvores de busca. Praticamente todos os bancos relacionais empregam árvores-B ou alguma variação dessas árvores (inclusive o MySQL [19]).

No caso de bancos de dados NOSQL que usam índices, geralmente usa-se a estratégia de índices invertidos. Essa estratégia consiste em indexar os documentos pelas palavras que neles aparecem. Ou seja, com essa estrutura, para cada palavra temos uma lista dos documentos em que ela aparece, mas não conseguimos saber facilmente quais palavras um documento contém (daí o termo "invertido").

Bancos de dados NOSQL tipicamente armazenam o índice no próprio banco, usando uma estrutura de chave e valor. Nesse caso, armazena-se os índices de forma que os termos pelos quais se pode fazer buscas se tornem chaves que indexam referências para os documentos que possuem esses termos. Ou seja, os termos são chaves associadas a valores que, por sua vez, são chaves que identificam documentos, como no exemplo abaixo:

Valores no BD	
chave	valor
Paulo	idade=15, nome=Paulo
Joãol	idade=14, nome=João
Tânia	idade=19, nome=Tânia
Joaquim	idade=13, nome=Joaquim
Joãom	idade=23, nome=João

Índice invertido (nome)	
chave	valor
Paulo	Paulo
João	(Joãol, Joãom)
Tânia	Tânia
Joaquim	Joaquim

Índice invertido (idade)	
chave	valor
15	Paulo
14	Joãol
19	Tânia
15	Joaquim
23	Joãom

A maneira com que os bancos de dados NOSQL mantém esses índices tem uma notável perda em relação aos bancos de dados relacionais. Armazenando os índices em pares de chave e valor, não conseguimos de maneira eficiente fazer buscas por intervalo usando inequações. Como os índices ficam guardados no banco, e são acessados por funções de *hash*, não temos qualquer informação de ordenação. No exemplo acima não há uma maneira eficiente de buscarmos quem tem idade inferior a 18 anos. As idades são apenas chaves de uma tabela de *hash*. Árvores são estruturas que facilitam esse tipo de busca [40], porém há um custo para manter a árvore.

A criação de índices invertidos é feita de maneira transparente em alguns bancos de dados. A aplicação apenas deve definir o que será indexado e o banco de dados NOSQL cuida do armazenamento dos índices. Mas há também bancos de dados que não possuem suporte nativo à índices secundários e, neste caso, a aplicação fica responsável por manter uma estrutura de índice invertido no banco de dados.

Os bancos de dados que fornecem suporte nativo à criação dos índices geralmente não misturam o índice na mesma estrutura onde se encontram os dados e criam uma espécie de banco de dados secundário. Tipicamente cada nó é responsável por guardar os índices dos valores que estão armazenados localmente. Nesse caso, uma busca por um termo deve ser distribuída para todos os nós do aglomerado. O Cassandra usa essa abordagem.

O Riak, além de oferecer a opção descrita acima, oferece também a opção de particionar os índices em função dos termos (estratégia também conhecida como índice global). Nesse caso, seria criado outro grupo de nós virtuais que fica responsável pelo armazenamento do índice. Como a chave de cada índice é o termo, no nosso exemplo, um *hash* consistente nos indicaria em qual nó estão indexados os usuários com nome João. Mesmo que os dois

registros de nome João estejam armazenados em nós (físicos) diferentes no aglomerado, o índice para esse termo ficaria em um único nó. Ou seja, se fizermos uma busca por todos os usuários que tenham nome João, só um nó deverá receber essa consulta.

Já o Cassandra cria uma família de colunas para armazenar cada índice, ou seja, para cada coluna do banco de dados que for indexada, será criada uma família de colunas para guardar o índice. Isso faz com que os índices secundários nativos do Cassandra não sejam indicados para indexar campos que possuem um domínio muito grande, pois cada termo possível terá uma nova coluna na família de colunas que armazena o índice. Poderíamos chegar a uma família de colunas com milhões de colunas, o que causaria problemas de desempenho. [24]

4.2 *MapReduce*

Bancos NOSQL tradicionalmente oferecem algum suporte a sequências de operações *map* e *reduce*. Basicamente há duas maneiras de entendermos um processo de *mapreduce* em um banco de dados distribuído. O processo pode ser aplicado no nível das entradas do banco de dados. Nesse caso cada entrada armazenada passaria por uma fase de *map*. Outra maneira de empregar o modelo *mapreduce* é aplicar o *map* a cada nó, ou seja, o nó é quem sofre o *map* e não as entradas nele armazenadas (pelo menos não diretamente).

Uma técnica bastante usada para fazer buscas ou análises mais complexas em um banco de dados distribuídos é aplicar um *mapreduce* nos dados armazenados. No entanto, essas buscas usualmente não são usadas para resultados imediatos. O uso se aplica mais a análises complexas nos dados, em um processo que roda em segundo plano. Isso acontece porque aplicar sequências de *map* e *reduce* em todos as entradas do banco de dados, mesmo de maneira distribuída, é um processo mais custoso se comparado a uma busca em um banco de dados com índices secundários. Porém, se restringirmos o número de entradas que passarão pelo *mapreduce*, podemos ter processos não tão custosos. Por isso, uma estratégia utilizada é fazer previamente uma busca usando os índices secundários, e restrições ou análises mais complexas seriam alcançados após a aplicação do *mapreduce* nesse conjunto já restrito de dados.

No caso dos bancos de dados com índices secundários armazenados no mesmo nó que armazena os dados indexados, cada consulta a esse índice passa por um *mapreduce* aplicado aos nós do aglomerado. A função *map* coletaria os resultados no índice de cada nó e a função *reduce* consolidaria os resultados no nó que disparou o processo. O processo de consolidação pode não ser um simples processo de concatenação dos resultados, já que a busca pode definir, por exemplo, uma ordenação. Os índices secundários não oferecem suporte a ordenação, mas esse processo pode ser feito em memória usando essa estratégia.

4.3 Buscas no Infinispan

O Infinispan oferece à aplicação a manutenção de índices secundários e também oferece uma API para a aplicação de *mapreduce* nas entradas armazenadas.

Para armazenar os índices secundários, a exemplo dos demais bancos de dados NOSQL, o Infinispan usa a estratégia de índices invertidos. Porém, a estrutura dos índices não é mantida pelo Infinispan. Para oferecer desempenho e uma API de buscas bem funcional, o Infinispan optou por usar duas ferramentas, o Apache Lucene [2] aliado ao Hibernate Search [7], ambos projetos de código aberto.

4.3.1 Apache Lucene

O Lucene é uma biblioteca Java de código aberto para indexação e busca de texto. É a ferramenta Java mais usada para esse tipo de atividade [33], e se caracteriza por oferecer uma API simples e algoritmos que acompanham o "estado da arte" da área de recuperação de informações.

Indexação

Usando a API do Lucene, podemos definir o índice basicamente criando documentos representados por objetos do tipo `Document` que agregam campos representados por objetos do tipo `Field` que, de fato, são os valores indexados. Internamente, os índices são organizados em uma estrutura de diretórios que pode ser armazenada em local configurável pelo usuário (memória, disco, etc).

Os documentos armazenados no Lucene não possuem relacionamentos entre si e nem precisam seguir o mesmo padrão. Em um mesmo índice podemos ter documentos com campos completamente diferentes.

Cada campo (`Field`) pertencente a um documento que é indexado possuí algumas configurações que podem definir a maneira com que o campo será tratado, como podemos ver no exemplo da figura 4.1. A configuração `Field.Index.ANALYZED` define que o campo será analisado antes de ser indexado. Um campo pode passar por uma série de analisadores configuráveis que, entre outras coisas, podem:

- Indexar um campo dividindo-o em palavras (*tokens*). Por exemplo, um campo formado por "Infinispan, um poderoso banco de dados", seria indexado pelas palavras Infinispan, poderoso, banco, dados. Nesse caso, o Lucene trata de retirar também palavras sem

```
IndexWriter writer = new IndexWriter(directory, analyzer, true,
                                     IndexWriter.MaxFieldLength.UNLIMITED);

// cria um documento que será adicionado ao
// índice
Document doc = new Document();

// adiciona campos ao documento
// nome
doc.add(new Field("nome", "israel",
                  Field.Store.YES,
                  Field.Index.ANALYZED));

// sobrenome
doc.add(new Field("sobrenome", "lacerra",
                  Field.Store.YES,
                  Field.Index.ANALYZED));

// adiciona documento ao índice e fecha o "escritor"
writer.addDocument(doc);
writer.close();
```

Figura 4.1: Exemplo: Adicionando um documento ao índice

significado importante, como artigos e preposições, de acordo com o idioma escolhido.

- Padronizar as palavras de forma que todo o conteúdo esteja em letra minuscula.
- Extrair a raiz(radical) de cada palavra e usá-la na indexação. Por exemplo, palavras como “sonho”, “sonhei”, “sonhador” seriam indexadas apenas usando a raiz “sonh”.

Também é possível criar analisadores personalizados e adicioná-los à cadeia de analisadores.

Repare que se indexarmos o campo passando por algum tipo de análise, os termos de fato indexados podem ser diferentes do campo original. Por isso o Lucene permite que o campo original seja salvo junto com o documento para, se necessário, termos acesso ao valor original do campo. Como podemos ver na figura, os campos indexados possuem a configuração `Field.Store.YES`, que define que o campo original será armazenado.

Há também a possibilidade de armazenarmos um campo sem indexá-lo. Nesse caso, não podemos fazer qualquer busca com relação a esse campo, mas o valor do campo fica armazenado junto com o documento. Podemos, por exemplo, indexar um livro pelo seu título e também armazenar o seu preço. Nesse caso não faremos buscas pelo preço, mas sempre que um livro for devolvido pela busca, saberemos também o seu preço.

O armazenamento dos valores ocorre em uma estrutura separada dos índices. Portanto, após obter o dado em uma busca no índice, o Lucene recupera o documento e seus campos através de um id em uma outra estrutura que mantém. É importante notar que os resultados de uma busca podem não estar armazenados de maneira sequencial nessa estrutura.

Buscas

Como podemos ver no exemplo da figura 4.2, a busca em um índice do Lucene possui uma API bem simples.

```
// cria uma consulta
IndexSearcher searcher = new IndexSearcher(directory);
Query query = new QueryParser("nome", new StandardAnalyzer()).parse("israel");

// faz a consulta
int nResults = 10;

TopScoreDocCollector collector = new TopScoreDocCollector(nResults);
searcher.search(query, collector);
assertEquals(1, collector.getTotalHits());

// obtém os documentos...
docID = collector.topDocs().scoreDocs[0].doc;
Document docRetrieved = searcher.doc(docID);

searcher.close();
```

Figura 4.2: Buscando um documento no índice

Além da busca simples descrita no exemplo, o Lucene permite buscas:

- **por intervalos (inequações):** O Lucene armazena os índices em ordem lexográfica, e pode fazer buscas em quaisquer intervalos do índice.
- **por prefixos**
- **formadas por combinações:** combinações booleanas ou negações (AND, OR e NOT) de outras buscas;
- **por frase:** O Lucene armazena a distância entre palavras nos documentos armazenados. Por isso podemos fazer buscas que procuram determinadas palavras em uma mesma frase (palavras não tão distantes em um texto)
- **por similaridade:** usando algoritmos de similaridade entre dois termos, o Lucene permite também que possamos fazer buscas que retornem documentos com termos semelhantes ao buscado.
- **com caracteres-curinga (*wildcards*):** O Lucene pode efetuar buscas definidas com * (zero ou mais caracteres) ou ? (zero ou um carácter). Exemplos: buscas pelo termo "joão*lui?" retornariam documentos com "joão pedro luiz" e "joão luis".

Os resultados de uma busca são encapsulados em um componente `TopDocs` que, por sua vez, possui um vetor de `ScoreDocs`. Cada `ScoreDoc` possui o id de um documento e a pontuação (*score*) que esse documento obteve na busca. O id é um identificador usado e gerado internamente pelo Lucene para identificar o documento armazenado. O *score* é um

valor que mede o grau de relevância que esse documento tem para a consulta.

Os cálculos para a pontuação de cada documento segundo sua relevância levam em consideração itens como:

- quantidade de vezes que o termo aparece no documento;
- quantidade de vezes que o termo aparece proporcionalmente no documento (se o campo tem pouquíssimas palavras e o termo aparece entre elas, o termo é importante para esse documento);
- quantidade de termos buscados presentes no documento (para buscas por mais de um termo).

O vetor de `ScoreDoc` por padrão vem ordenado pela relevância do documento (do documento mais relevante para o menos relevante). O usuário pode também definir critérios de ordenação usando um ou mais campos do documento.

Os critérios de ordenação são passados pelo usuário por meio do componente `Sort` que encapsula uma lista ordenada de campos que serão utilizados na ordenação dos documentos.

O Lucene oferece ainda a possibilidade de se combinar uma ou mais consultas aplicando operações booleanas (E, OU, NÃO). Além disso, temos a possibilidade de se aplicar filtros mais simples antes da aplicação de uma consulta, fazendo com que apenas documentos que passem pelo filtro estejam entre os possíveis resultados da consulta. Isso poderia ser útil, por exemplo, para limitar os resultados de uma busca em uma aplicação web apenas aos dados aos quais o usuário possa ter acesso.

Os filtros são encapsuladas sob a interface `Filter`. O usuário pode escrever seus próprios filtros e aplicá-los em uma consulta, mas o Lucene já possui implementados uma série de filtros que podem ser uteis em uma busca.

Outra funcionalidade interessante que o Lucene oferece é a paginação. Paginação é o mecanismo pelo qual sites de busca retornam os resultados separados por página. Esse recurso permite obter respostas mais rapidamente, evitando processamento prematuro de informações que tem menos relevância na busca, ou seja, de informações que estão em páginas mais distantes.

4.3.2 Uso do Hibernate Search

O Hibernate Search é uma biblioteca implementada como uma camada sobre o Lucene. Sua intenção é:

- facilitar o uso dos índices do Lucene, que lidam basicamente com texto, fazendo com que possamos armazenar e buscar objetos Java no índice, sem nos preocupar em convertê-los para texto;
- manter a sincronização entre um banco de dados e os índices do Lucene;

Esses dois itens são importantes para o Infinispan. Muito embora o segundo item tenha sido desenvolvido com foco no uso associado com a implementação JPA do Hibernate, que não engloba o Infinispan.

Com o Hibernate Search podemos indexar objetos Java no Lucene de maneira bem simples, com uso de anotações. No exemplo da figura 4.3 temos uma classe que será indexada de maneira semelhante ao exemplo da seção anterior.

```
@Indexed
@ProvidedId
public class Pessoa implements Serializable {

  private static final long serialVersionUID = 6416216469201441157L;

  @Field(index = Index.TOKENIZED, store = Store.YES)
  private String nome;

  @Field(index = Index.TOKENIZED, store = Store.YES)
  private String sobrenome;
```

Figura 4.3: Exemplo de classe que será indexada pelo Hibernate Search

As únicas informações obrigatórias em uma classe que será indexada pelo Hibernate Search são a anotação `Indexed`, que define que a classe será indexada e também uma anotação que define qual é o id da entidade no banco de dados. Essa informação é importante para as funcionalidades de sincronização com JPA que o Hibernate Search oferece. Por meio desse id o Hibernate Search saberá como identificar essa entidade no banco de dados. Há a possibilidade de informar o id anotando o campo correspondente com `@DocumentId` ou anotando a entidade com `@ProvidedId`. Usando a segunda opção, o id deverá ser informado no momento da indexação.

Além das anotações obrigatórias, o Hibernate Search oferece anotações para praticamente todas as configurações possíveis para a indexação de um documento no Lucene.

4.3.3 Vantagens da abordagem do Infinispan

A escolha do Infinispan por uma biblioteca de busca já madura se baseia em diversos argumentos. Em grande parte dos bancos NOSQL, os índices secundários não permitem buscas com certo grau de complexidade. Salvo exceções de bancos que efetuam alguns passos em memória, os índices secundários dão suporte apenas a buscas simples usando igualdades. Graças ao uso do Lucene, o Infinispan não apresenta essas restrições.

Apesar de também se basear em índices invertidos, e alguns de seus recursos, como aplicação de filtros nos campos indexados e ordenação já existirem em alguns outros bancos de dados como o Riak, o Lucene é amplamente usado em muitas ferramentas de buscas [2] e já possui estrutura e desempenho bastante otimizados e uma API bastante rica em funcionalidades.

Capítulo 5

Arquitetura Interna do Infinispan

Daremos aqui uma breve noção da arquitetura interna do Infinispan, focando no que diz respeito à implementação deste trabalho.

Na figura 5.1 podemos ver o fluxo de uma chamada ao Infinispan:

- O `Cache` cria um `ReplicableCommand` e um `InvocationContext`, que contém as informações necessárias de uma chamada à grade de dados;
- O `Cache` passa o `InvocationContext` e o `ReplicableCommand` criados para o `InterceptorChain` onde o comando é submetido a cada um dos `CommandInterceptors` que estiverem registrados no `InterceptorChain`;
- O último interceptador finalmente chama o método `ReplicableCommand.perform()` que fará a operação no `DataContainer` de fato.

A seguir daremos detalhes da arquitetura envolvida em cada um desses passos.

5.1 `CacheManager` e `Cache`

A interface básica para os clientes acessarem o Infinispan é oferecida pelos componentes `CacheManager` e `Cache`.

O `CacheManager` é responsável por criar um `Cache` e configurá-lo de acordo com um arquivo de configurações previamente definido ou ainda de acordo com configurações passadas diretamente a ele. No processo de criação do `Cache`, o `CacheManager` cria o `ComponentRegistry` que age como um *framework* de injeção de dependências, fazendo com que vários componentes relacionados ao ciclo de vida do `Cache` sejam criados sob demanda e armazenados.

A interface `Cache` estende a interface `BasicCache` e é a interface utilizada para o uso do Infinispan de maneira embarcada na JVM. Para utilização no modelo cliente-servidor através

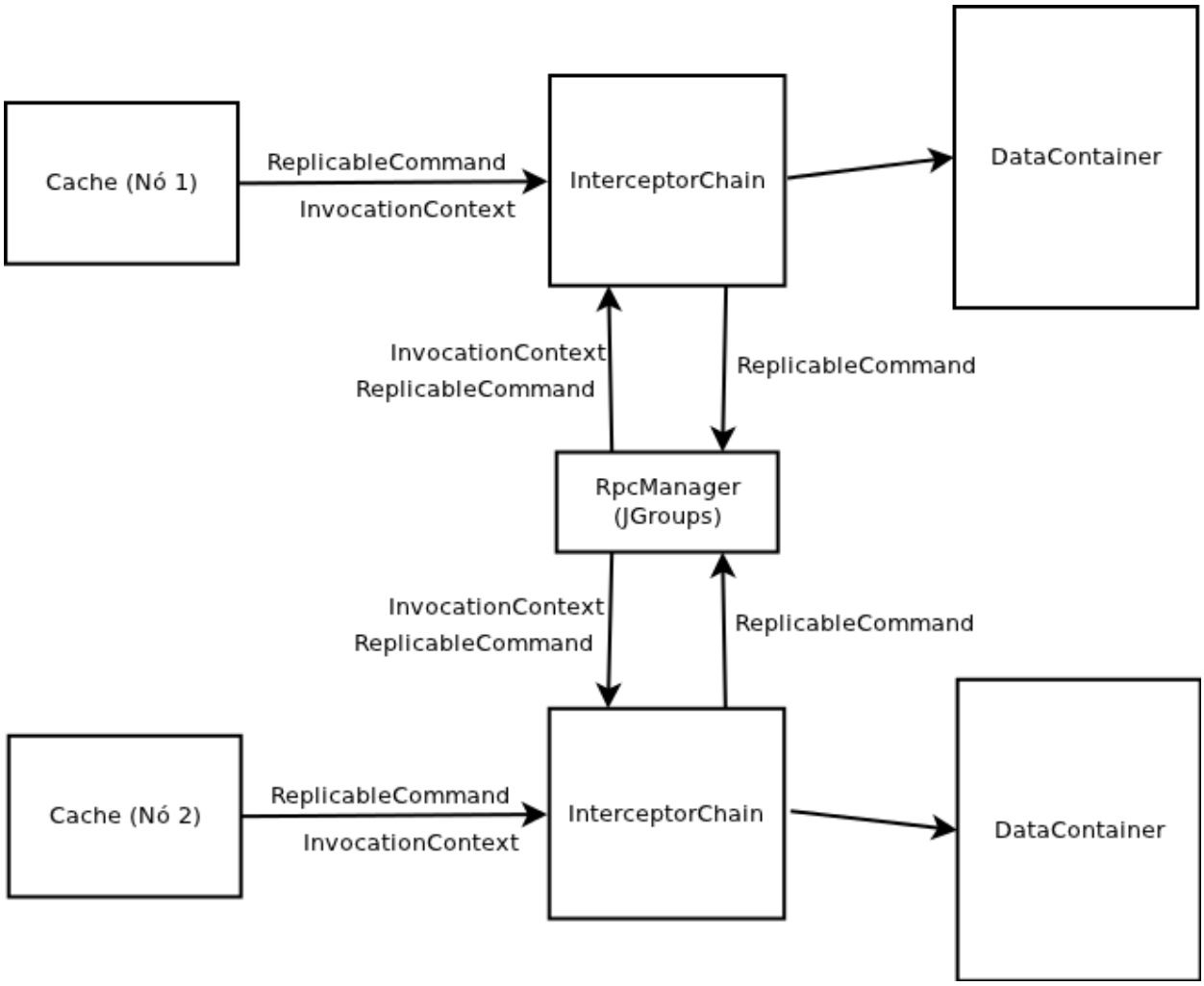


Figura 5.1: Fluxo de uma operação realizada no Infinispan

do protocolo de comunicação HotRod, o Infinispan implementa a interface `RemoteCache` que também deriva da `BasicCache` mas, devido a natureza do modelo cliente-servidor, possui algumas limitações e otimizações específicas. A interface `BasicCache` estende a `java.util.concurrent.ConcurrentMap` e possui os métodos básicos para acesso e modificação dos dados armazenados no Infinispan, como `put()`, `get()`, `remove()` e `replace()`, entre outros.

Apesar de fornecer a interface para acesso à grade de dados, o `Cache` não é o componente responsável pela estrutura de armazenamento dos valores. A responsabilidade do `Cache` é agir como uma primeira camada entre o Infinispan e o usuário.

5.2 `ReplicableCommand`

Cada operação realizada pelo usuário por meio da interface `Cache` é encapsulada em um `ReplicableCommand` (figura 5.2), que possui a lógica necessária para a execução da operação além da chave que mapeia o valor que sofrerá a operação. Além disso o `Cache`

cria um `InvocationContext`, que é um objeto que armazena algumas informações como o estado atual da transação e o endereço do nó que originou a operação.

Method Summary

byte	**getCommandId**() Used by marshallers to convert this command into an id for streaming.
Object[]	**getParameters**() Used by marshallers to stream this command across a network
Object	**perform**(InvocationContext ctx) Performs the primary function of the command.
void	**setParameters**(int commandId, Object[] parameters) Used by the CommandsFactory to create a command from raw data read off a stream.

Figura 5.2: `ReplicableCommand` API

Cada comando do Infinispan possui um id único definido estaticamente, que serve como identificador nas várias interações remotas entre os nós de um aglomerado, e facilita a seriação do comando (com ele é possível saber qual comando instanciar após receber o objeto pela rede). O método `perform()` efetivamente possui a lógica do comando. Os métodos `getParameters()` e `setParameters()` ajudam no processo de seriação do comando para transmissão pela rede.

Os comandos criados pelo `Cache` implementam também a interface `VisitableCommand`, que possui apenas dois métodos a mais, como podemos ver na figura 5.3.

Method Summary

Object	**acceptVisitor**(InvocationContext ctx, Visitor visitor) Accept a visitor, and return the result of accepting this visitor.
boolean	**shouldInvoke**(InvocationContext ctx) Used by the InboundInvocationHandler to determine whether the command should be invoked or not.

Figura 5.3: `VisitableCommand` API

Ao implementar essa última interface, os comandos passam a "aceitar a visita" de objetos denominados visitantes. Na maioria dos casos os visitantes são também interceptadores. O Infinispan mantém no componente `InterceptorChain` uma cadeia de visitantes à qual um comando, acompanhado do `InvocationContext`, é submetido após ser criado pelo `Cache`. Essa cadeia de visitantes é criada de acordo com as configurações do Infinispan e define alguns comportamentos para cada operação realizada. Na figura 5.4 podemos observar a hierarquia da interface `VisitableCommand` e ver a lista de comandos que podem

ser criados por um `Cache` a cada operação.

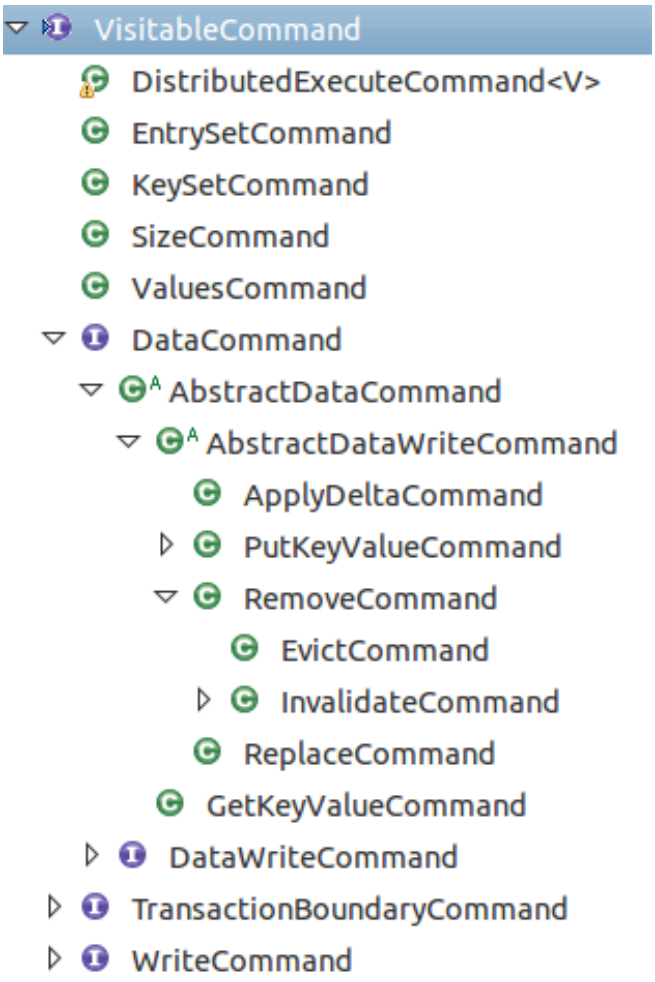


Figura 5.4: Comandos criados pelo `Cache` a cada operação.

5.3 Visitantes e `InterceptorChain`

Os visitantes, que são implementados sob a interface `Visitor` (figura 5.5), são componentes que adicionam efeitos colaterais a cada comando (`VisitableCommand`) que "visitam".

Repare que um `Visitor` deve implementar um comportamento para cada comando visitável do Infinispan. A maioria deles implementa comportamento para apenas alguns comandos e deixa métodos vazios em relação aos outros comandos.

Os `Visitors` que formam a cadeia de visitantes `InterceptorChain` devem implementar também a interface `CommandInterceptor` (figura 5.6), que dá suporte à disposição dos visitantes em uma estrutura de cadeia.

Figura 5.5: `Visitor` API [9]

A cadeia de interceptadores é mantida sempre na mesma ordem de sua criação e pode possuir interceptadores diferentes dependendo da configuração do Infinispan. Configurações como habilitar transações, escrita secundária em disco e indexação para buscas, entre outras, podem adicionar diferentes interceptadores na cadeia. A seguir vamos entender um pouco sobre alguns interceptadores que são mais relevantes para este trabalho.

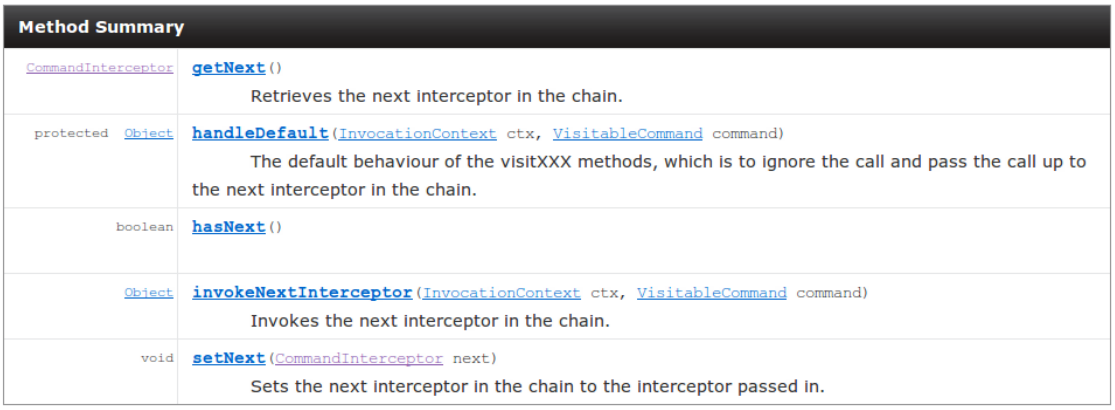


Figura 5.6: `CommandInterceptor` API

5.3.1 `DistributionInterceptor`

O `DistributionInterceptor` é um interceptador que é adicionado à cadeia quando a grade de dados está trabalhando no modo distribuído. Para outras configurações, como modo de replicação, modo de invalidação ou modo local, são adicionados outros interceptadores.

Esse interceptador é responsável por alguns comportamentos da grade de dados no modo distribuído, como manutenção do cache L1 (tanto local quanto invalidação de dados remotos) ou distribuir as chamadas entre os outros nós do aglomerado de acordo com a necessidade. Ao adicionar, atualizar ou recuperar valores, o interceptador verifica quais os nós são responsáveis pelo dado e então replica o comando para os nós correspondentes usando o componente `RpcManager`.

5.3.2 `QueryInterceptor`

Quando o usuário habilita a opção de indexação secundária, teremos na cadeia um interceptador que cuidará da manutenção dos índices Lucene, de acordo com as alterações ocorridas na grade de dados.

Ao criar a cadeia de interceptadores, o Infinispan escolhe entre `LocalQueryInterceptor` e `QueryInterceptor` de acordo com a configuração escolhida pelo usuário. Como veremos no próximo capítulo, o Infinispan oferece duas possibilidades de uso de índices secundários, a indexação local e a indexação replicada.

Para a configuração de indexação local há o interceptador `LocalQueryInterceptor`. Esse interceptador ignora comandos que não foram gerados localmente, isto é, ignora comandos que foram transmitidos via `RpcManager`. É importante notar que, dessa maneira, a indexação é local em relação ao nó que originou a alteração no dado. Porém, não necessariamente o dado é armazenado na instância local do cache, já que essa localização depende do *hash* consistente.

Por outro lado, o `QueryInterceptor` age em todos os comandos submetidos à cadeia, sejam eles comandos originados localmente ou transmitidos via `RpcManager`. Dessa maneira é mantido um índice cujas entradas estarão replicadas da seguinte maneira: a entrada correspondente a um certo dado será guardada em todos os nós que armazenam réplicas desse dado.

No próximo capítulo discutiremos o funcionamento e as consequências do uso de cada configuração.

5.3.3 `CallInterceptor`

O `CallInterceptor` é um interceptador colocado sempre no final da cadeia independentemente das configurações do Infinispan. Ele é responsável pela execução do comando, mediante chamada ao método `ReplicableCommand.perform()`. Quando efetua essa chamada, o `CallInterceptor` passa como parâmetro o `InvocationContext`.

5.4 RpcManager

O RpcManager é um componente que funciona como camada de interface para o JGroups e é responsável por distribuir todos os comandos pelo aglomerado de maneira otimizada. Ele é usado por vários interceptadores e componentes do Infinispan. A API RpcManager possui opções para que as chamadas remotas sejam tratadas de maneira síncrona ou assíncrona, de acordo com a natureza do comando distribuído e da configuração dos interceptadores. Essa escolha é feita pelo interceptador ao chamar o RpcManager. Essa interface possui métodos que recebem ou devolvem objetos que implementam alguma das seguintes interfaces:

- **ReplicableCommand**;
- **Address**: interface que define o endereço de algum outro nó específico do aglomerado;
- **Response**: interface que encapsula uma resposta de uma chamada remota feita a outros nós do aglomerado.

5.5 DataContainer

Após o comando ser criado, submetido à cadeia de interceptadores e possivelmente distribuído por outros nós do aglomerado, ele finalmente age sobre os dados armazenados no Infinispan. A estrutura responsável pelo armazenamento é o DataContainer.

O DataContainer armazena os dados em um BoundedConcurrentHashMap baseado na implementação de ConcurrentHashMap disponível na ainda não lançada nova versão do Java (JDK 8). O BoundedConcurrentHashMap é responsável por manter atualizadas as informações para a política de despejo e também por executá-la e notificar outros componentes do Infinispan sobre o despejo.

Capítulo 6

Consultas locais no Infinispan

Neste capítulo explicaremos como funcionam as consultas locais feitas no Infinispan. Na figura 6.1 temos uma noção do fluxo de informação que ocorre na execução de uma consulta.

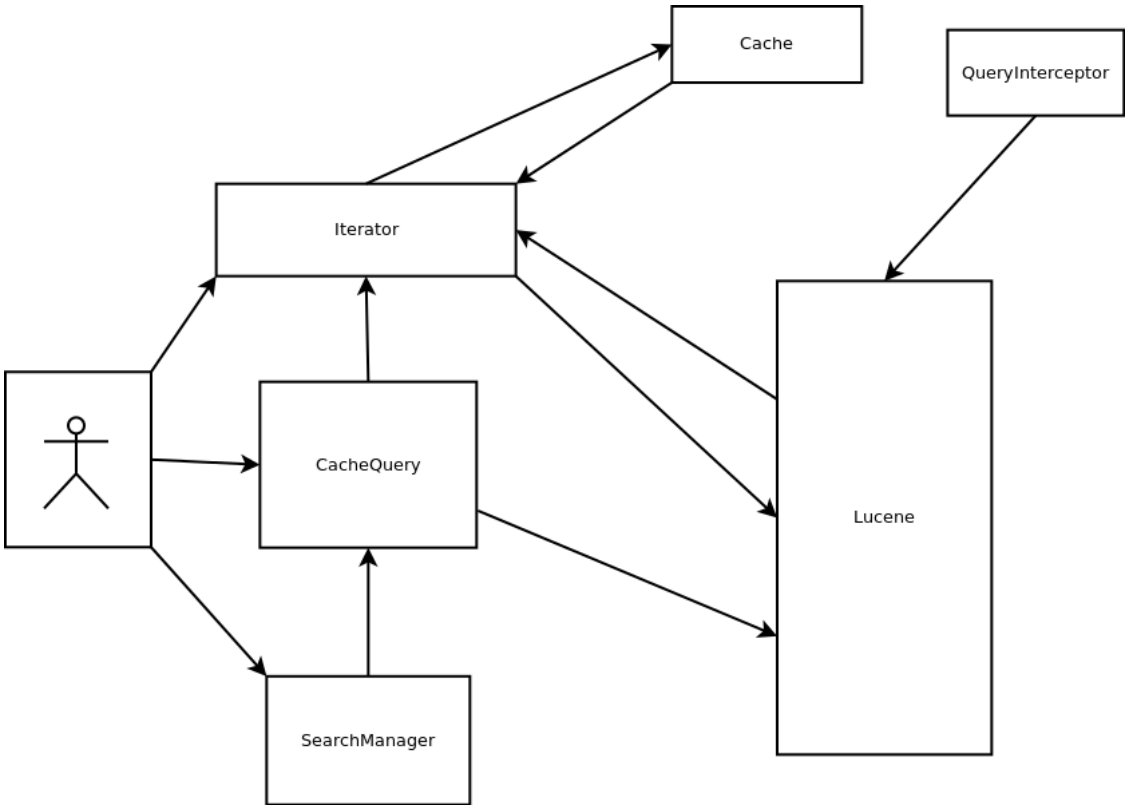


Figura 6.1: Consulta local aos índices secundários

A seguir explicaremos detalhes dessa arquitetura.

6.1 Indexação

Como vimos no capítulo 5, o `QueryInterceptor` é o interceptador responsável por manter os índices sempre atualizados no Lucene. Esse interceptador indexa objetos armazenados no Infinispan que possuam as anotações do Hibernate Search. Cada campo do objeto

é indexado de acordo com a anotação e configuração utilizada.

No entanto, algumas configurações não podem ser aplicadas. No caso do Infinispan, os valores dos campos dos objetos nunca serão armazenados no Lucene. Isso acontece devido ao modo em que a recuperação dos valores é feita, como veremos nas próximas seções. O Infinispan obtém como resposta do Lucene apenas a chave que indexa o valor na grade de dados. E então, os resultados são recuperados a partir do próprio `Cache`. Para facilitar esse processo, o id do objeto será sempre a chave à qual o valor está associado.

Temos duas possibilidades de configuração para a indexação, que correspondem a dois interceptadores diferentes que podem ser inseridos na cadeia de interceptadores.

Indexação Replicada. Com essa configuração o Infinispan adiciona na cadeia o interceptador `QueryInterceptor`, que faz com que o nó transmita para o Lucene quaisquer alterações que receba. Alterações em dados armazenados no Infinispan só são percebidas por nós responsáveis pelo valor alterado (segundo o *hash* consistente) e pelo nó que gerou a alteração. Outros nós não recebem o comando em sua cadeia de interceptadores e, consequentemente, não armazenam índices secundários correspondentes às alterações feitas pelo comando. Essa configuração é chamada de indexação global na documentação do Infinispan.

Indexação local. Nessa configuração o Infinispan adiciona na cadeia de interceptadores o `LocalQueryInterceptor` que é subclasse do `QueryInterceptor`. A diferença é que esse interceptador só indexa dados para comandos gerados localmente. Isso significa que apenas o nó que gerou o comando de atualização de um dado no Infinispan terá o dado atualizado nos índices secundários. O nó que solicitou a alteração ou inclusão do dado, não necessariamente é responsável pelo armazenamento do dado na grade de dados. Portanto, o nó pode acabar indexando o dado no Lucene, mas não sendo o responsável pelo armazenamento do próprio dado.

6.2 Consultas

O ponto de partida para um usuário iniciar uma consulta no Infinispan é o componente `SearchManager`. Esse componente é responsável por receber uma consulta na estrutura do Lucene (encapsulada na classe `org.apache.lucene.search.Query`) e devolver um `CacheQuery` que é o componente que fornece a API para executar as operações de consulta no Infinispan. A consulta armazenada no `org.apache.lucene.search.Query` pode ser criada com qualquer mecanismo do Lucene ou do Hibernate Search. Um exemplo pode ser visto na figura 6.2. Considere que a classe Pessoa é a mesma da figura 4.3.

```
// obtém o SearchManager
SearchManager searchManager = org.infinispan.query.Search.getSearchManager( cache );

// cria um QueryBuilder (Hibernate Search) para o a classe Pessoa
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass( Book.class ).get();

// cria uma consulta que busca objetos da classe Pessoa que possuam danilo
// no nome ou sobrenome
org.apache.lucene.search.Query luceneQuery = queryBuilder.phrase()
                  .onField( "nome" )
                  .andField( "sobrenome" )
                  .sentence( "danilo" )
                  .createQuery();

// cria a consulta no Infinispan
CacheQuery query = searchManager.getQuery( luceneQuery, Book.class );

// obtém a lista de resultados
List<Pessoa> objectList = query.list();

for ( Pessoa pessoa : objectList ) {
      System.out.println( pessoa.getNome() + " " + pessoa.getSobrenome());
}
```

Figura 6.2: Exemplo de consulta no Infinispan

6.3 CacheQuery

O `CacheQuery` funciona como uma interface para os componentes do Hibernate Search e do Lucene. Por meio dessa interface o usuário consegue efetuar sua busca e obter dados em relação a busca, como podemos ver na figura 6.3.

Method Summary	
void	**disableFullTextFilter**(String name) Disable a given filter by its name.
org.hibernate.search.FullTextFilter	**enableFullTextFilter**(String name) Enable a given filter by its name.
int	**getResultSize**() Gets the integer number of results.
QueryIterator	**iterator**() Returns the results of a search as a QueryIterator.
QueryIterator	**iterator**(int fetchSize) Returns the results of a search as a QueryIterator with a given integer parameter - the fetchSize.
QueryIterator	**lazyIterator**() Calls the lazyIterator(int fetchSize) method but passes in a default 1 as a parameter.
QueryIterator	**lazyIterator**(int fetchSize) Lazily loads the results from the Query as a QueryIterator with a given integer parameter - the fetchSize.
List<Object>	**list**() Returns the results of a search as a list.
void	**setFilter**(org.apache.lucene.search.Filter f) Allows lucene to filter the results.
void	**setFirstResult**(int index) Sets a result with a given index to the first result.
void	**setMaxResults**(int numResults) Sets the maximum number of results to the number passed in as a parameter.
void	**setSort**(org.apache.lucene.search.Sort s) Allows lucene to sort the results.

Figura 6.3: Métodos de `CacheQuery`

Além dos métodos de aplicação de ordenação, filtros e paginação, que apenas deixam visíveis funcionalidades existentes no Hibernate Search e no Lucene e que mudarão o com-

portamento da consulta, temos os métodos que devolvem:

- iteradores para a lista de resultados;
- a lista de resultados;
- o tamanho da lista de resultados.

Um ponto importante desses métodos é o comportamento do método `getResultSize()`. Devido a características do Lucene, usar esse método é uma alternativa menos custosa do que obter uma lista ou um iterador e então verificar o tamanho da lista.

6.4 Iteradores

O Infinispan oferece sob a interface `QueryIterator` (figura 6.4), que por sua vez é uma interface que estende `java.util.ListIterator`, um mecanismo para iterar os resultados de uma consulta ao Lucene. O `CacheQuery` oferece duas alternativas de iteradores para os resultados de uma busca: o `EagerIterator` e o `LazyIterator`.

Method Summary

void	**afterFirst**() Jumps to the one-after-the-first result
void	**beforeLast**() Jumps to the one-before-the-last result
void	**close**() This method must be called on your iterator once you have finished so that Lucene resources can be freed up.
void	**first**() Jumps to the first result
boolean	**isAfterFirst**()
boolean	**isBeforeLast**()
boolean	**isFirst**()
boolean	**isLast**()
void	**jumpToResult**(int index) Jumps to a specific index in the iterator.
void	**last**() Jumps to the last result

Figura 6.4: Métodos de `QueryIterator`

O `EagerIterator` age de maneira ávida ao fazer a consulta. Quando é instanciado, dispara a consulta no Lucene e já recupera todas as chaves que fazem parte do resultado. A partir daí o `EagerIterator` passa a iterar a lista de chaves e obter os resultados na grade de dados à medida que o usuário requisita.

O `LazyIterator` trabalha de maneira preguiçosa. Ao ser instanciado, a busca é disparada no Lucene, mas as chaves dos valores não são imediatamente recuperadas. Como dissemos no capítulo 4, os valores armazenados com os documentos no Lucene não são armazenados junto com os índices. Dessa forma, obter os ids empregados pelo Hibernate Search, ou seja, as chaves dos valores, corresponde a uma leitura não sequencial no disco. É exatamente essa leitura que o `LazyIterator` procura evitar ou adiar. As chaves só são recuperadas no Lucene à medida que o usuário requisite mais resultados.

Ao criar um iterador usando um `CacheQuery` temos a opção de passar o parâmetro `fetchSize`. Esse parâmetro define o tamanho de um *buffer* que é usado internamente pelo iterador. Tanto o `LazyIterator` quanto o `EagerIterator` utilizam esse valor. Quando o usuário solicita o primeiro valor, o iterador já recupera os primeiros `fetchSize` valores e cria um *buffer*. Os próximos valores requisitados pelo usuário são recuperados e armazenados nesse *buffer*. Quando o iterador atinge o fim do *buffer*, novamente são recuperados os próximos `fetchSize` valores e assim sucessivamente. Quando o usuário cria um iterador sem especificar o valor do `fetchSize`, o `CacheQuery` adota 1 como valor.

O método `list()` trabalha de maneira análoga ao `EagerIterator`, mas de maneira ainda mais ávida. O método dispara a consulta, obtém todas as chaves que fazer parte do resultado, recupera os valores na grade de dados usando as chaves e os devolve em uma lista.

Capítulo 7

Consultas distribuídas no Infinispan

Antes de começarmos este projeto, o módulo de consultas aos índices secundários não trabalhava de maneira realmente distribuída no Infinispan. As consultas eram sempre feitas ao índice local apenas, como explicado no capitulo anterior. Dessa maneira, alguns dados indexados em outros nós do aglomerado eram ignorados e, além disso, não tínhamos as vantagens de elasticidade desejáveis em um ambiente distribuído.

O objetivo deste trabalho foi a implementação do módulo de consultas distribuídas do Infinispan. O projeto foi baseado nos requisitos discutidos com a equipe do Infinispan para o módulo de consultas distribuídas [8]. De maneira resumida, buscamos atingir os seguintes pontos:

- Cada nó guardará os índices de objetos armazenados localmente apenas.
- As consultas feitas ao índice serão feitas de maneira distribuída.
- A consulta é criada em um nó e então transmitida aos demais nós do aglomerado.
- Cada nó recebe a consulta, a executa em seu índice local e, então, devolve os resultados ao nó gerador da consulta.
- Os resultados são consolidados no nó que criou a consulta, em um estilo *MapReduce*.
- Os resultados podem ser recuperados conforme a demanda (carga preguiçosa) ou já são todos devolvidos (carga ávida) para o nó que gerou a consulta.
- As duplicidades de resultados devem ser tratadas.
- A ordem dos resultados deve ser consistente, ou seja, buscas feitas com ordenações específicas devem devolver os resultados segundo essa ordenação. Buscas feitas sem ordenação deveriam, idealmente, devolver os resultados ordenados por relevância, como normalmente acontece em uma busca no Lucene.

Os últimos três itens necessitam mais atenção. A possibilidade de recuperarmos os resultados sob demanda requer mecanismos mais complexos. O penúltimo item se refere ao fato de que geralmente teremos replicação de valores e, portanto, um objeto deve estar em mais de um nó. Ao olharmos os resultados de uma busca, não queremos resultados duplicados. Por fim, um ponto muito importante é a ordenação dos resultados. Em uma consulta com ordenação, temos que ter um mecanismo que garanta que ao consolidarmos os resultados a ordem seja preservada.

Nas próximas seções vamos mostrar uma visão global da implementação deste trabalho e também explicar os passos que envolvem cada processo de uma consulta distribuída. Na figura 7.1 podemos ver um diagrama com o fluxo de informações entre os componentes em uma consulta distribuída.

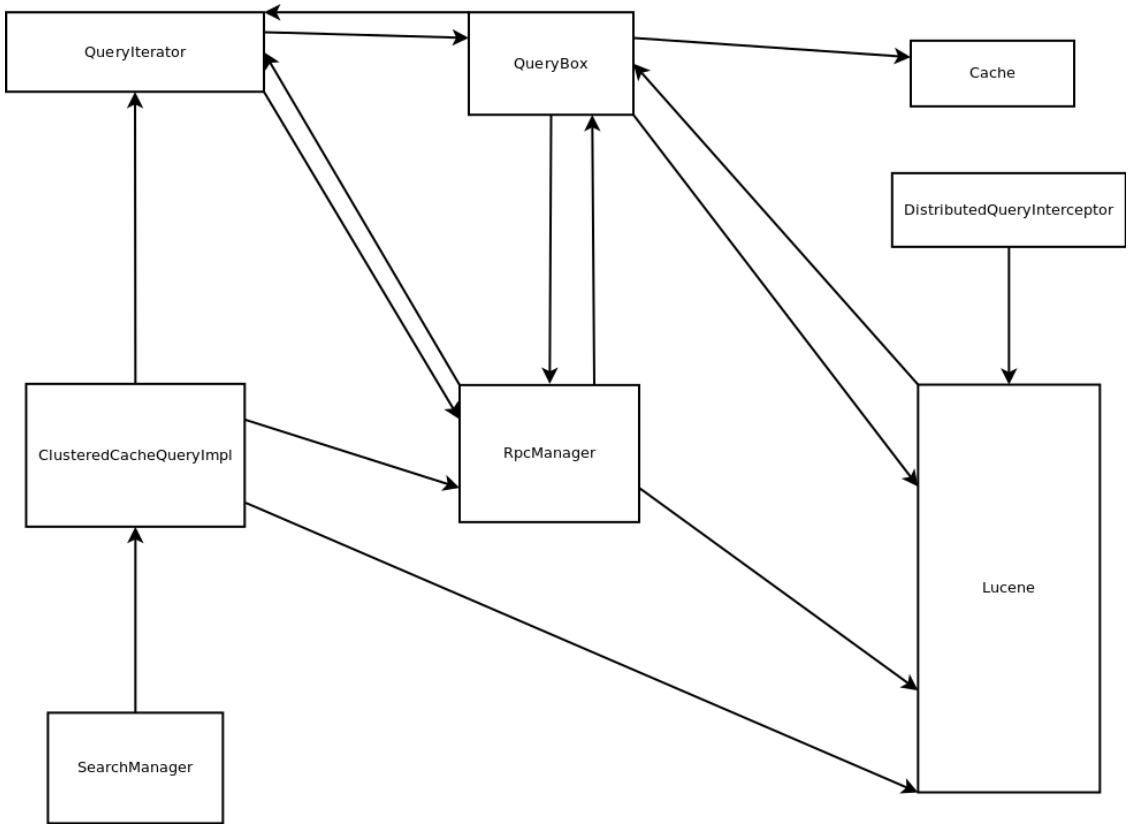


Figura 7.1: Fluxo de informações em uma consulta distribuída

Uma das intenções do trabalho foi prover o serviço de consultas distribuídas sob a mesma API usada pelo usuário para consultas locais. Portanto, a implementação oferece as mesmas funcionalidades das consultas locais.

7.1 Indexação

Como um dos objetivos do trabalho foi fazer com que cada nó indexe apenas os dados que são armazenados em sua instância de cache, as configurações de indexação oferecidas inicialmente não eram suficientes.

A indexação replicada é a mais próxima do nosso objetivo. Porém, ela acaba indexando os dados em mais nós do que o necessário para nossa implementação. Com ela, os índices são armazenados não só nos nós responsáveis pelo armazenamento do dado, mas também no nó que gerou a atualização ou adição do dado. A indexação nesse último nó não é necessária para o nosso trabalho.

Por isso, para nossa implementação, criamos uma nova opção de configuração, a indexação distribuída. Para essa configuração criamos o `DistributedQueryInterceptor`, que estende o `QueryInterceptor`. Esse interceptador verifica se o dado a ser indexado será armazenado localmente e só nesse caso o índice é armazenado.

Dessa maneira, cada nó irá indexar em sua instância do Lucene apenas índices referentes à porção de dados pela qual é responsável segundo o anel do *hash* consistente.

7.2 Consulta

Como nas consultas locais, o passo inicial para o usuário é feito com o uso do `SearchManager`. E exatamente aqui se encontra a única diferença no uso da API para utilizar uma consulta distribuída. Para uma consulta local o usuário chamava o método `SearchManager.getQuery()`. Para obter uma consulta distribuída, o usuário apenas terá que utilizar um outro método, o `SearchManager.getDistributedQuery()`. Esse método irá retornar uma implementação de `CacheQuery` que fará consultas distribuídas ao aglomerado. Todas as funcionalidades de um `CacheQuery` foram implementadas e, a partir daqui, o uso de consultas distribuídas fica transparente ao usuário.

7.3 `ClusteredQueryCommand`

O `RpcManager` é um componente fundamental no módulo de consultas distribuídas. Como vimos no capítulo 5, o `RpcManager` é o componente responsável por distribuir todas as chamadas em um aglomerado. Cada chamada deve estar encapsulada em um `ReplicableCommand`. O módulo de consultas distribuídas transmite diferentes tipos de chamadas pelo aglomerado, mas todas essas chamadas são encapsuladas em uma mesma implementação do `ReplicableCommand`, o `ClusteredQueryCommand`. Criar um `ReplicableCommand` para cada tipo de chamada do módulo criaria uma série de comandos com comportamentos semelhantes e que dificultariam a manutenção dessa estrutura.

Criamos então uma estrutura em que um `ClusteredQueryCommand` pode representar diferentes ações do módulo de consultas distribuídas. A ação de um `ClusteredQueryCommand` é definida através do enum `ClusteredQueryCommandType`, que pode assumir os seguintes valores:

- `CREATE_LAZY_ITERATOR`;
- `CREATE_EAGER_ITERATOR`;
- `DESTROY_LAZY_ITERATOR`;
- `GET_SOME_KEYS`;
- `GET_RESULT_SIZE`.

O comportamento do `ClusteredQueryCommand` para cada uma dessas ações será explicado detalhadamente nas seções seguintes.

7.4 `ClusteredQueryCommandInvoker`

Esse componente é responsável por fazer a intermediação entre o `RpcManager` e a classe `ClusteredCacheQueryImpl`, que é a classe central do módulo de consultas distribuídas, como veremos na próxima seção.

O surgimento desse componente se deu principalmente para otimizar um limitante do processo de execução do `ClusteredQueryCommand`. Dada a natureza do `RpcManager`, ao dispararmos um comando por ele, o comando é executado apenas em outros nós e não no nó local. Porém, para o nosso trabalho, seria interessante um componente que recebesse um comando e executasse localmente e em todos os outros nós.

O `ClusteredQueryCommandInvoker` recebe um `ClusteredQueryCommand`, dispara o comando localmente e, paralelamente, distribui o comando mediante o `RpcManager`. Após receber as respostas do `RpcManager` e da própria execução local, as respostas são consolidadas e então devolvidas ao componente que requisitou a chamada.

7.5 `ClusteredCacheQueryImpl`

Nossa intenção é prover as consultas distribuídas de maneira transparente, oferecendo a mesma API utilizada atualmente para as consultas locais. Para isso, criamos uma implementação para a interface `CacheQuery` que faz a consulta de maneira distribuída e provê

todas as funcionalidades.

Na figura 6.3 vimos todos os métodos da API `CacheQuery`. Os dois primeiros métodos e os últimos quatro não possuem grandes desafios para a nossa implementação, o `ClusteredCacheQueryImpl`. Eles apenam atualizam informações que deverão ser transmitidas junto com a consulta caso algum método que efetivamente execute uma ação seja chamado (aqui englobamos os métodos restantes da API). Restam o método `getResultSize()`, os iteradores e o método `list()`, cujas implementações são discutidas a seguir.

7.6 ClusteredCacheQueryImpl.getResultSize()

Para o método `getResultSize()`, basicamente criamos uma consulta local em cada nó. Cada um dos nós devolve o resultado da chamada do próprio `getResultSize()` invocado na consulta local. O nó que fez a chamada original ao `getResultSize()` soma todas as respostas.

Quando o `getResultSize()` é disparado, o `ClusteredCacheQueryImpl` cria um `ClusteredQueryCommand` do tipo `GET_RESULT_SIZE`. Esse comando é então passado ao `ClusteredQueryInvoker` que retorna uma lista com o tamanho da lista de resultados de cada nó do aglomerado. O `ClusteredCacheQueryImpl` então soma os resultados.

Se, antes da execução do `getResultSize()`, já tiver sido executado algum outro método que efetivamente obtenha a lista de resultados, as quantidades de resultados são somadas e o valor total é guardado para posteriores chamadas ao `getResultSize()`.

Se quisermos forçar que a consulta não retorne a lista completa de resultados, podemos configurar o tamanho da lista de resultados por meio do método `setResultSize()`. Há um detalhe importante quanto à isso. Não é possível saber quantos valores precisamos de cada nó para se atingir o último valor dessa iteração. Portanto, ao criar um iterador, o `ClusteredCacheQueryImpl` solicita `maxResults` valores de cada nó do aglomerado e, portanto, a coleção global de resultados acaba obtendo mais resultados que o requisitado.

7.7 Iteradores distribuídos

Os iteradores distribuídos foram a maior dificuldade para a implementação do módulo de consultas distribuídas. Um dos requisitos do projeto é que os dois iteradores (ávido e preguiçoso) funcionem plenamente no modo distribuído, inclusive mantendo a ordem consistente dos dados, o que não é algo trivial.

O iterador menos complexo é o ávido. A ideia é essencialmente disparar as consultas em

todos os nós do aglomerado e consolidá-las no nó que gerou a busca.

O `ClusteredCacheQueryImpl` dispara a consulta em todos os nós por meio de um `ClusteredQueryCommand` do tipo `CREATE_EAGER_ITERATOR`. A resposta retornada por cada nó possui o `TopDocs` retornado pela consulta local, além das chaves dos valores retornados. A transmissão do `TopDocs` na resposta é necessária pois precisamos do vetor de `ScoreDoc` para consolidar os valores de todos mantendo a ordem dos resultados. Lembre-se que o vetor de `ScoreDoc` possui as informações necessárias para a ordenação segundo a relevância.

Com essas respostas de todos os nós já armazenadas, o `ClusteredCacheQueryImpl` instancia um `DistributedIterator` com essas informações.

O `DistributedIterator` mantém uma `org.apache.lucene.util.PriorityQueue`. Esse é um componente interno do Lucene que é capaz de ordenar valores de `ScoreDoc` de acordo com a ordenação passada para o Lucene no momento da consulta (essas informações também estão atreladas ao `ScoreDoc`).

Uma das alternativas seria colocar todos os `ScoreDocs` que vieram de todos os nós já na `PriorityQueue`. Porém, com o intuito de evitar a ordenação prematura de grandes quantidades de dados, optamos por fazer um iterador não tão ávido. O `DistributedIterator` mantém na `PriorityQueue` sempre um resultado de cada nó apenas. Ou seja, a `PriorityQueue` possui sempre o próximo resultado de cada nó do aglomerado e diz, dentre esses resultados, qual é o próximo resultado a ser retornado pelo iterador. Essa estrutura pode ser melhor visualizada na figura 7.2.

A implementação do iterador preguiçoso foi feita na classe `DistributedLazyIterator`, que estende a classe `DistributedIterator` e herda grande parte de seu comportamento. Porém, a implementação do iterador preguiçoso é mais complexa. A estratégia escolhida foi implementar uma espécie de *streaming* entre o nó que gerou a consulta e os demais nós.

Como a própria recuperação da chave no Lucene é um processo mais custoso do que a simples recuperação dos `ScoreDocs`, o `DistributedLazyIterator` obtém inicialmente a lista de `ScoreDoc` sem as chaves dos valores. As chaves são requisitadas aos nós do aglomerado sob demanda, ou seja, conforme a necessidade, por meio de um `ClusteredQueryCommand` do tipo `GET_SOME_KEYS`.

Uma alternativa poderia ser a implementação de uma estratégia ainda mais preguiçosa, fazendo com que o próprio vetor de `ScoreDoc` não fosse inteiramente recuperado de início.

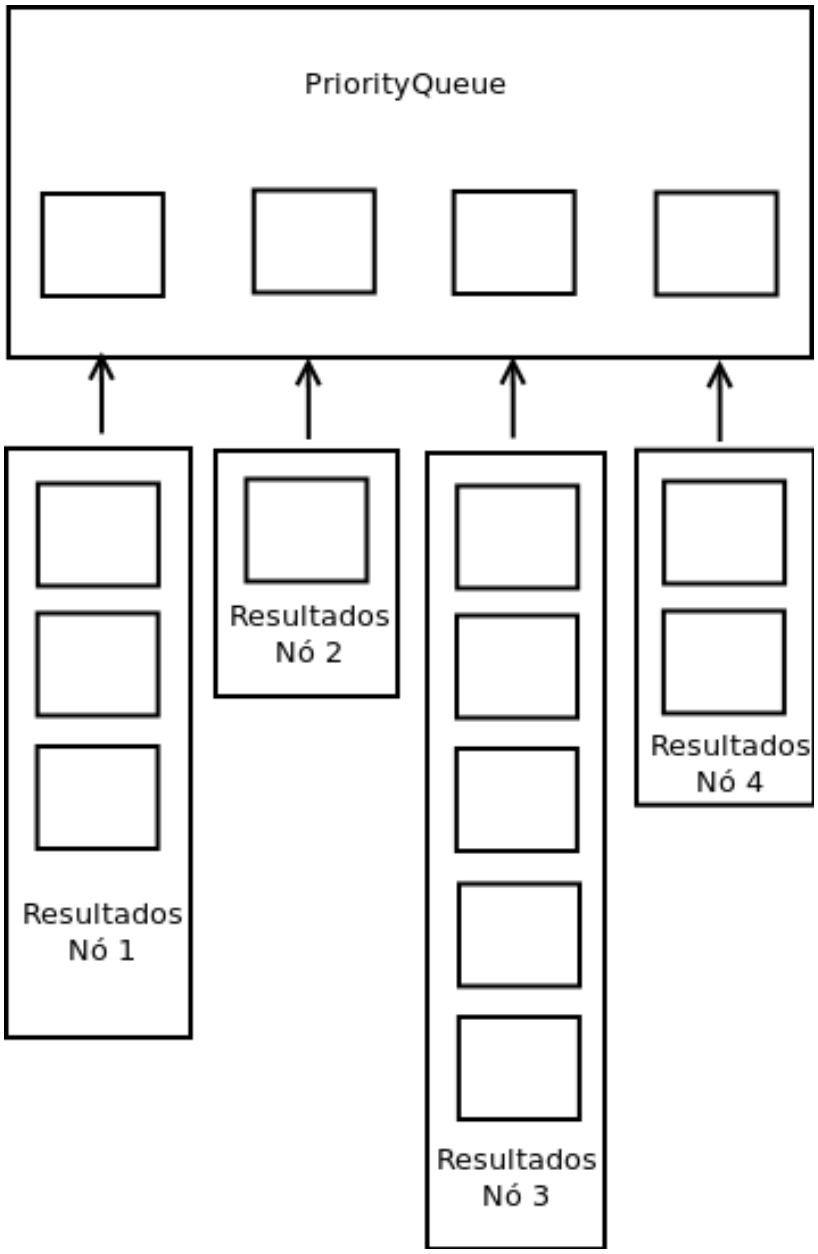


Figura 7.2: Exemplo do uso da `PriorityQueue` dentro de um `DistributedIterator`

Porém, dessa maneira não evitaríamos processamento, já que os `ScoreDocs` já foram criados. Haveria apenas uma possível economia inicial de tráfego na rede, pois evitaríamos ou adiaríamos a transmissão de `ScoreDocs` que não necessariamente seriam utilizados. Mas o *overhead* gerado pela comunicação entre os nós para a transmissão de cada `ScoreDoc` separadamente faz com que essa estratégia não seja tão atraente.

O comportamento dos iteradores quanto ao parâmetro `fetchSize` é semelhante ao comportamento que acontece com as consultas locais. Um *buffer* é mantido com `fetchSize` valores e sempre que o fim do *buffer* é atingido, ele é alimentado com novos valores. O `fetchSize` não influi na quantidade de chaves que são recuperadas com o comando `ClusteredQueryCommand` do tipo `GET_SOME_KEYS`. Em nossa implementação atual

esse comando obtém sempre uma chave apenas por requisição.

Para manter o *streaming* do iterador preguiçoso, foi necessária a criação de um componente que mantivesse ativas as consultas nos outros nós. Esse componente é o `QueryBox`, que é melhor explicado na próxima seção.

7.8 `QueryBox`

A função do componente `QueryBox` é manter o estado de todas as consultas ativas para transmitir novos resultados sempre que tais resultados forem requisitados. Cada nó do aglomerado possui uma instância de `QueryBox` ativa. Cada uma dessas instâncias guarda o estado local das consultas ativas no aglomerado.

A identificação das consultas é feita por meio de um UUID(*Universally Unique Identifier*), que é um id com 128 bits. No momento da criação do iterador preguiçoso no nó que está gerando a consulta, é criado de maneira aleatória o UUID da consulta usando a implementação nativa da Sun, a `java.util.UUID` [11].

Os demais nós do aglomerado recebem então a consulta e o UUID dela. A consulta é disparada localmente e indexada no `QueryBox` pelo UUID correspondente. Todas as requisições para obter os próximos valores da consulta, ou seja, todos os comandos `ClusteredQueryCommand` do tipo `GET_SOME_KEYS` que o nó receber, trará consigo o UUID da consulta, e o nó então obtém a consulta no `QueryBox`. A figura 7.3 ilustra o caminho que um `ClusteredQueryCommand` criado por um `DistributedLazyIterator` percorre quando é disparado.

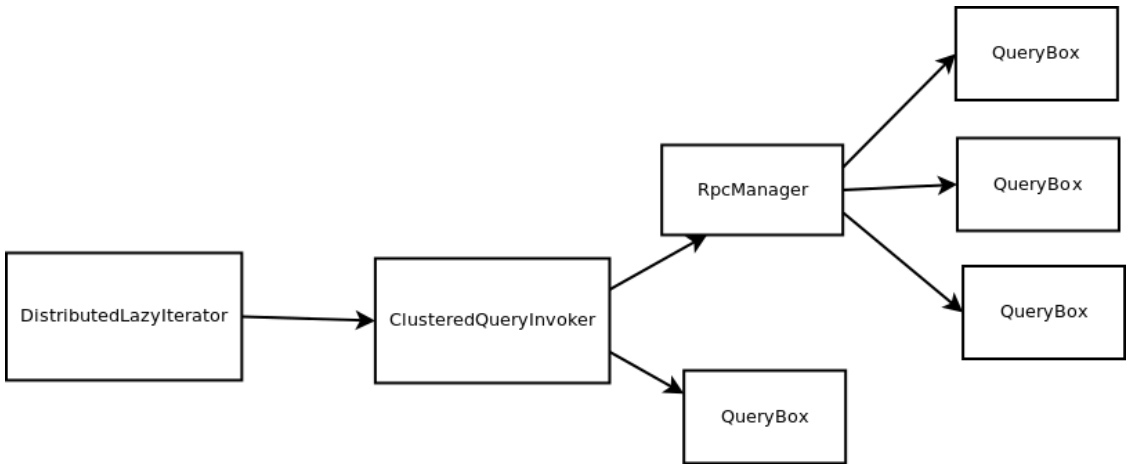


Figura 7.3: Fluxo de um comando gerado por um `DistributedLazyIterator` em um aglomerado com 4 nós

É importante notar que um nó não deve guardar uma consulta remota por tempo inde-

terminado. Caso contrário, a lista de consultas guardadas só aumentará, e isso nos levará a problemas de memória e desempenho.

No `ClusteredCacheQueryImpl` há o método `close()` que, quando chamado, avisa o restante dos nós que a consulta não será mais utilizada. O aviso é transmitido através de um `ClusteredQueryCommand` do tipo `DESTROY_LAZY_ITERATOR` e quando um nó recebe esse comando, descarta a consulta em questão.

Porém, a mera existência do método `close()` não é suficiente. O sistema deve implementar alguma política de despejo que independa do usuário disparar tal método. Em algum momento o nó precisa considerar que determinada consulta não será mais utilizada. Em nossa implementação atual usamos uma estratégia LRU. Temos uma lista que armazena as consultas ordenadas pelo último acesso. Quando o `QueryBox` atinge um limite de N consultas acessadas, onde N é um valor fixado na própria classe, a consulta que está há mais tempo sem ser acessada é expurgada.

7.9 `ClusteredCacheQueryImpl.list()`

A implementação do método `list()` é, na verdade, bem simples. Um iterador ávido é criado pelo `ClusteredCacheQueryImpl`. Por meio desse iterador, todos os valores são obtidos e armazenados em uma lista que é então devolvida ao usuário.

Uma outra possibilidade seria fazer com que cada nó já retornasse a lista de valores. Essa estratégia foi descartada pois com a lista de valores não há um mecanismo fácil para manter a ordem consistente no momento de consolidar os valores. Os dados referentes à ordenação estão encapsulados em um `org.apache.lucene.search.Sort` passado pelo usuário e algumas vezes possuem estruturas complexas. Além disso, a ordenação por relevância seria muito difícil de ser aplicada.

Usando o iterador pudemos aproveitar a estrutura já criada, que se utiliza dos `ScoreDocs` e da `PriorityQueue` para consolidar os valores.

7.10 Limitações

7.10.1 Impactos de mudanças no aglomerado

Mudanças no estado do aglomerado, como um *rehash* ou a simples saída ou entrada de valores no Infinispan, podem influenciar alguns comportamentos das consultas distribuídas.

A ocorrência de um *rehash* não causa problemas para a manutenção dos índices. As mudanças geradas por um *rehash* são feitas utilizando os comandos tradicionais de acesso

à grade de dados. Esses comandos são submetidos à cadeia de interceptadores e, então, as mudanças são passadas para o Lucene automaticamente quando o `QueryInterceptor` intercepta comandos de adição ou remoção.

Porém, um processo de *rehash* pode afetar um iterador distribuído do tipo preguiçoso que esteja ativo. Como as mudanças geradas pelo *rehash* são imediatamente transmitidas para os índices, um nó pode ter uma consulta ativa em seu `QueryBox` que aponte para valores que não estão mais em seu índice. Se isso ocorrer, o iterador irá lançar uma exceção e deixará de funcionar.

O iterador ávido sofrerá muito menos com esse tipo de situação. Logo que é criado, o iterador já obtém a lista de chaves para os resultados. Com essas chaves os valores podem ser recuperados independentemente de sua localização no aglomerado. Entretanto, o iterador pode falhar se o *rehash* ocorrer no exato momento em que as consultas são disparadas no aglomerado. Se isso ocorrer, o iterador pode falhar inesperadamente. Esse é o mesmo risco que corre o método `ClusteredCacheQueryImpl.list()`.

A entrada ou saída de dados na grade também podem afetar o andamento de um iterador distribuído que esteja ativo. Se um valor é adicionado ao Infinispan após um iterador já estar ativo, esse valor não será recuperado pelo iterador. O iterador só terá os dados que estavam armazenados no momento de sua construção.

No entanto, se um dado deixar o Infinispan após a criação de um iterador, alguns problemas podem ocorrer. Iteradores ávidos poderão ter o comportamento inconsistente em relação ao `ClusteredCacheQueryImpl` que o criou. Ao não obter sucesso em recuperar um valor usando sua chave, o iterador irá simplesmente pular para a chave seguinte. Porém, isso fará com que o iterador retorne menos valores do que o `ClusteredCacheQueryImpl` que o criou indica ser possível por meio do método `getResultSize()`.

Já com o `DistributedLazyIterator`, a saída de um valor durante sua execução causará uma exceção. Como nesse iterador as chaves são recuperados no Lucene sob demanda, ao não encontrar uma chave no Lucene, não temos como saber se houve um *rehash* ou a simples saída de um valor. Nesse caso, o mais seguro é parar o iterador. Se continuarmos a execução do iterador, podemos acabar ignorando um valor que apenas mudou de localização mas continua armazenado no Infinispan. Isso faz com que o iterador não tenha garantia de consistência com relação aos dados que estavam armazenados no momento da criação do iterador.

Tentar evitar esses problemas traria muita complexidade à manutenção e, pior que isso, poderia diminuir o desempenho do processo de *rehash* do Infinispan. A ideia é que o próprio

usuário decida o que fazer quando se deparar com as exceções.

7.10.2 Paginação

Os mecanismos envolvidos na paginação (explicada no final da seção 4.3.1) dos resultados possuem desempenho limitado em uma consulta distribuída.

Se o usuário solicitar uma consulta que dê os resultados a partir do n-ésimo valor, o iterador internamente irá comparar todos os resultados de todos os nós até atingir o n-ésimo valor global. Infelizmente não há uma maneira eficiente de se fazer essa iteração pois não é possível saber qual o dado e onde está armazenado o n-ésimo valor, já que esse resultado depende de uma ordenação global dos valores. O usuário deve evitar o emprego de paginações, especialmente se a quantidade de páginas for muito elevada.

Capítulo 8

Considerações Finais

O objetivo principal deste trabalho foi desenvolver o módulo de consultas distribuídas do Infinispan. Para atingir tal objetivo estudamos o surgimento desse sistema e como ele é inserido no movimento NOSQL. Além disso, estudamos como esses bancos de dados tipicamente implementam consultas baseadas em índices secundários em seus sistemas. A seguir sumarizamos as contribuições deste trabalho, fazemos uma breve discussão sobre trabalhos relacionados e propomos trabalhos futuros para tornar mais robusto e completo o módulo de consultas distribuídas do Infinispan.

8.1 Contribuições deste trabalho

A implementação deste trabalho pode ser encontrada em https://github.com/israeldl/infinispan/tree/ISPN-200. O Infinispan (https://github.com/infinispan/infinispan), que atualmente se encontra na versão 5.1.6, incorporou nosso módulo de consultas distribuídas em setembro de 2011 na versão 5.1.0. De lá até hoje, solucionamos alguns problemas que foram identificados com o auxílio da comunidade de usuários desse software livre. Com o uso cada vez maior da ferramenta, acreditamos que o módulo está próximo de atingir uma razoável maturidade.

Com a implementação deste trabalho, o Infinispan foi o primeiro sistema a prover nativamente suporte a indexação e consultas distribuídas a índices Lucene. Acreditamos que o uso do Lucene e do Hibernate Search são grandes vantagens do sistema. O Lucene já é uma ferramenta madura e hoje é resultado de anos de pesquisa na área de indexação e buscas de documentos. Além disso, o Hibernate Search oferece uma porção de facilidades que auxiliam o usuário na utilização do Lucene.

8.2 Trabalhos relacionados

O Apache Solr [3], projeto também de código aberto que fornece uma estrutura cliente-servidor para uso do Lucene, oferecia a possibilidade de consultas em um ambiente distribuído. No entanto, a distribuição dos índices entre os nós do aglomerado era de responsa-

bilidade do usuário. Além disso, no início deste trabalho, o Solr não provia funcionalidades de um banco de dados NOSQL. Porém, atualmente a versão 4(alpha) do Solr, deu outro foco ao projeto, que passou a figurar como um sistema de grade de dados NOSQL e também promete passar a cuidar da indexação e focar em ambientes distribuídos quando chegar em sua versão final. Tal fato ajuda a fortalecer a ideia de que o Lucene é uma boa opção para uso em ambientes NOSQL.

Outros sistemas NOSQL tem estratégias bem parecidas, como pudemos ver no capítulo 4, e poucos, como o Riak, oferecem funcionalidades ricas como um sistema aliado ao Lucene pode oferecer.

Em janeiro de 2012, Christian von der Weth e Anwitaman Datta propuseram em um artigo [40] sobre buscas em bancos NOSQL uma abordagem diferente das tradicionalmente adotadas. Porém, a abordagem foca em estratégias que otimizam a indexação de dados por mais de um termo, ou seja, otimizam a criação e manutenção de índices invertidos onde as chaves do índice são conjuntos de valores. Esses índices são uteis em buscas com operadores booleanos. Por exemplo, uma busca por pessoas que tenham João e Maria no nome. Nesse caso, o índice invertido poderia ter uma chave composta por João e Maria. Apesar da abordagem se mostrar eficiente para tais tipos de índice, ela é focada apenas em um subconjunto de buscas NOSQL. Buscas por termos simples, que englobem mais de um campo (por exemplo, buscar por João no campo nome e no campo sobrenome) ou que envolvam algum tipo de ordenação são deixadas de lado por esse artigo.

8.3 Trabalhos futuros

8.3.1 Política de despejo

A política de despejo adotada para manter as consultas que podem estar ativas no aglomerado pode ser melhorada. Atualmente a quantidade máxima de consultas que podem permanecer ativas é fixa. Seria interessante que esse valor pudesse ser configurado pelo usuário. Além disso, a estratégia LRU adotada para selecionar as consultas que serão despejadas também não pode ser alterada atualmente. Idealmente, deveríamos prover outras estratégias, como a própria LIRS que é fornecida pelo Infinispan para o despejo dos dados armazenados na grade.

Uma possibilidade também pensada é criar outra instância de Infinispan que ficaria focada apenas em armazenar os índices. Ou seja, deixaríamos de guardar os índices em uma tabela de *hash* e usaríamos o próprio Infinispan. Dessa maneira, a política de despejo ficaria transparente para nossa implementação e seria feita já por mecanismos internos do Infinispan.

8.3.2 Indexação global

Alguns bancos de dados NOSQL oferecem a possibilidade do índice ser distribuído mas não depender da localidade do dado. Nesse caso a distribuição do índice dependeria apenas dos termos que ele indexa. Em outras palavras, a ideia seria criar algo como um *hash* consistente que distribuísse o índice secundário aplicando o *hash* no termo indexado. Essa estratégia é conhecida também como indexação global.

Em nossa implementação, os índices referentes a um dado são sempre armazenados no nó que é responsável pelo próprio dado. È uma estratégia que se mostrou eficaz em muitos casos. Porém, a natureza e comportamento das aplicações que utilizam o Infinispan são muito variadas, e determinadas aplicações podem se tornar mais eficientes com uma indexação global. Opcionalmente iremos oferecer essa possibilidade de indexação ao usuário. A implementação depende principalmente da criação de um novo `QueryInterceptor` que indexe os dados baseando-se nessa nova estratégia. Além disso, sabendo em qual nó do aglomerado se encontra determinado índice, algumas otimizações na buscas são possíveis.

8.3.3 Integração com a API JPA

O Infinispan fornece interface JPA para persistência no próprio Infinispan. Essa implementação é feita pelo projeto Hibernate OGM [6], que tem como objetivo prover interfaces JPA para bancos de dados NOSQL. Atualmente o projeto só funciona sobre o Infinispan.

As consultas feitas usando a interface JPA ainda não utilizam o módulo de consultas distribuídas. O Hibernate OGM ainda se utiliza das consultas locais. Temos a intenção de adaptar em breve o Hibernate OGM para que ele se utilize do novo módulo.

Referências Bibliográficas

[1] *Apache CouchDB*, http://couchdb.apache.org/. 11

[2] *Apache Lucene*, http://lucene.apache.org/java/docs/index.html. 31, 36

[3] *Apache Solr*, http://lucene.apache.org/solr/. 63

[4] *EhCache*, http://http://ehcache.org/. 5

[5] *Google App Engine*, http://code.google.com/appengine/. 16

[6] *Hibernate OGM (Object/Grid Mapper)*, http://www.hibernate.org/subprojects/ogm.html. 65

[7] *Hibernate Search*, http://www.hibernate.org/subprojects/search.html. 31

[8] *Infinispan - Distributed Queries*, https://issues.jboss.org/browse/ISPN-200. 51

[9] *Infinispan Architectural Overview*, http://community.jboss.org/wiki/ArchitectralOverview. vii, 41

[10] *Infinispan Clustering Modes*, http://community.jboss.org/wiki/Clusteringmodes. vii, 22

[11] *Java Platform SE 6 - UUID*, http://docs.oracle.com/javase/6/docs/api/java/util/UUID.html. 58

[12] *Java Transaction API (JTA)*, http://www.oracle.com/technetwork/java/javaee/jta/index.html. 15

[13] *JBoss Cache*, http://www.jboss.org/jbosscache. 5

[14] *JBoss Infinispan - open source data grids*, http://www.jboss.org/infinispan. 1

[15] *JSR 107: JCACHE - Java Temporary Caching API*, http://jcp.org/en/jsr/detail?id=107. 15

[16] *JSR 347: Data Grids for the Java Platform*, http://www.jcp.org/en/jsr/detail?id=347. 16

[17] *Memcached - a distributed memory object caching system*, http://memcached.org/. 5

[18] *MurmurHash and SMHasher webpage*, http://code.google.com/p/smhasher/. 23

[19] *MySQL 5.0 Reference Manual*, http://dev.mysql.com/doc/refman/5.0/en/innodb-table-and-index.html. 28

[20] *Oracle Coherence*, http://www.oracle.com/technetwork/middleware/coherence/overview/index.html. 13

[21] *Riak*, http://basho.com/products/riak-overview/. 10

[22] *The Java Persistence API - A Simpler Programming Model for Entity Persistence*, http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html. 15

[23] *The JGroups Project*, http://www.jgroups.org/. 25

[24] Ed Anuff, *Cassandra indexing techniques*, Cassandra Summit SF, 2011. 30

[25] Eric A. Brewer, *Towards robust distributed systems (abstract)*, Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (New York, NY, USA), PODC '00, ACM, 2000, pp. 7–. 7

[26] ———, *CAP Twelve Years Later: How the Rules Have Changed*, Computer (2012), 23–29. 8

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, ACM Trans. Comput. Syst. **26** (2008). 6

[28] Carlos André Reis Fernandes Oliveira da Silva, *Data Modeling with NoSQL: How, When and Why*, (2011). 11

[29] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: simplified data processing on large clusters*, Commun. ACM **51** (2008), 107–113. 12

[30] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, *Dynamo: amazon's highly available key-value store*, SIGOPS Oper. Syst. Rev. **41** (2007), 205–220. 6

[31] M. De Diana e M. A. Gerosa, *NOSQL na Web 2.0: Um Estudo Comparativo de Bancos Não-Relacionais para Armazenamento de Dados na Web 2.0*, (2010). 6

[32] Seth Gilbert and Nancy Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, SIGACT News **33** (2002), no. 2, 51–59. 7

[33] E. Hatcher, O. Gospodnetic, and M. McCandless, *Lucene in Action*, second ed., Manning, 2010. 31

[34] Rick Hightower, *Java Data Grid Specification: JSR-347*, InfoQ - http://www.infoq.com/news/2011/10/java-data-grid (2011). 16

[35] ———, *JSR-107, JCache: Alive and Going to be Part of Java EE 7*, InfoQ - http://www.infoq.com/news/2011/08/jcache-jsr-lives/ (2011). 16

[36] Song Jiang and Xiaodong Zhang, *LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance*, SIGMETRICS Perform. Eval. Rev. **30** (2002), no. 1, 31–42. 19

[37] Avinash Lakshman and Prashant Malik, *Cassandra: a decentralized structured storage system*, SIGOPS Oper. Syst. Rev. **44** (2010), no. 2, 35–40. 9

[38] N. Leavitt, *Will nosql databases live up to their promise?*, Computer **43** (2010), no. 2, 12 –14. 6, 9, 10

[39] S. Robbins, *RAM is the new disk...*, InfoQ - http://www.infoq.com/news/2008/06/ram-is-disk (2008). 5

[40] C. von der Weth and A. Datta, *Multiterm Keyword Search in NoSQL Systems*, Internet Computing, IEEE **16** (2012), no. 1, 34 –42. 27, 29, 64