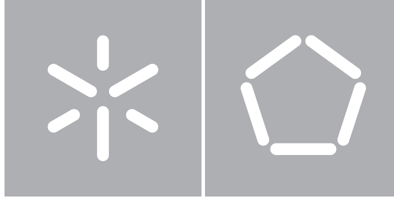
Universidade do Minho
Escola de Engenharia

Francisco Nuno Teixeira Neves

Análise de desempenho e otimização do Apache HBase para dados relacionais

Julho de 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Francisco Nuno Teixeira Neves

Análise de desempenho e otimização do Apache HBase para dados relacionais

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor José Orlando Pereira
Doutor Ricardo Manuel Vilaça

Julho de 2015

Agradecimentos

O meu primeiro agradecimento é dirigido ao meu orientador, Professor José Orlando Pereira, por me ter proposto um novo desafio à um ano atrás e por me ter acompanhado neste percurso sempre com uma disponibilidade ímpar. De seguida, também de forma muito relevante, deixo o meu agradecimento ao meu co-orientador e colega de laboratório, Ricardo Manuel Vilaça, pelo constante acompanhamento e pela compreensão prestados.

Não podia deixar de referir os responsáveis, doutorados, doutorandos, mestres e mestrandos do Grupo de Sistemas Distribuídos por proporcionarem um excelente ambiente de trabalho. Em especial, ao Miguel Matos pelas longas discussões proporcionadas sobre este trabalho e pelo apoio prestado.

Ao projeto CoherentPaaS – CoherentPaaS: A Coherent and Rich PaaS with a Common Programming Model, financiado pela Sétimo Programa Quadro da União Europeia (FP7-611068).

Num âmbito pessoal, quero deixar o meu especial agradecimento à minha família. Aos meus pais, por sempre me incentivarem a ir cada mais longe e pelo esforço que fizeram até hoje para que este momento fosse possível. Ao meu irmão por ser um exemplo na vida académica e, portanto, uma inspiração. Por fim, às minhas duas eternas companheiras de quatro patas, Vitória e Nina, por me proporcionarem momentos de distração e diversão durante a elaboração deste trabalho.

Análise de Desempenho e Otimização do Apache HBase para Dados Relacionais

A popularidade que os sistemas NoSQL têm vindo a conquistar leva a que sejam constantemente submetidos a análises e otimizações de desempenho. A capacidade destes sistemas capazes de escalar melhor que as tradicionais bases de dados relacionais motivou a migração de inúmeras aplicações para sistemas NoSQL mesmo quando não se tira partido da estrutura de dados dinâmica por eles fornecida. Porém, a consulta destes dados estruturados tem um custo adicional que deriva da flexibilidade dos sistemas NoSQL.

Este trabalho demonstra algumas limitações de desempenho do Apache HBase e propõe e avalia o *Prepared Scan*, uma operação que visa tirar partido do conhecimento da estrutura de dados por parte da aplicação, diminuindo assim o custo associado à consulta de dados estruturados.

Performance Evaluation and Optimization of Apache HBase for Relational Data

NoSQL systems are getting more popular and, for this reason, they are constantly the target of performance evaluation and optimization. The capability of these systems to scale better than traditional relational databases motivated a large set of applications to migrate their data to these systems, even without the intention to exploit the flexibility provided by those systems. However, the access to structured data is costly due to such flexibility.

This work characterizes some performance limitations found in Apache HBase using the industry standard benchmark, and proposes and evaluates Prepared Scan, an operation based on native Scan that optimizes the access to data which is structured in a regular manner.

Conteúdo

Lista de Figuras **vii**

Lista de Tabelas **viii**

1 Introdução **1**

1.1 Problema 3

1.2 Contribuição 3

1.3 Estrutura do documento 4

2 Estado da arte **5**

2.1 Bases de dados *NoSQL* 5

2.1.1 Bases de dados chave-valor 7

2.1.2 Linguagens de interrogação sobre NoSQL 7

2.2 Apache HBase 9

2.2.1 Pares chave-valor 10

2.2.2 Arquitetura 11

2.2.3 Interface com o cliente 13

2.2.4 Escrita de dados 13

2.2.5 Leitura de dados 14

2.2.6 *Coprocessors* 15

2.3 Análise de desempenho 16

2.3.1 Métricas relevantes 16
2.3.2 Ferramentas de teste 16
2.4 Metodologias de otimização 18
2.4.1 Armazenamento 19
2.4.2 Balanceamento 20
2.4.3 Comunicação 22
2.5 Sumário 23

3 Análise de desempenho **25**
3.1 Ambiente experimental 25
3.2 *RegionServers* locais 26
3.2.1 Descrição do cenário 26
3.2.2 Resultados 27
3.2.3 Conclusão 28
3.3 Uso de *coprocessors* do tipo *endpoint* 28
3.3.1 Descrição do cenário 29
3.3.2 Resultados 29
3.3.3 Conclusão 30
3.4 Tamanho dos qualificadores 31
3.4.1 Descrição do cenário 31
3.4.2 Resultados 32
3.4.3 Conclusão 33
3.5 Quantidade de famílias de colunas 33
3.5.1 Descrição do cenário 33
3.5.2 Resultados 34
3.5.3 Conclusão 35
3.6 Esquema da tabela 35
3.6.1 Descrição do cenário 36

3.6.2 Resultados . . . 37
3.6.3 Conclusão . . . 37
3.7 Sumário . . . 38

4 Otimização da leitura de tabelas 39
4.1 Interface com a aplicação . . . 40
4.2 Protocolo . . . 41
4.3 Componente servidor . . . 44
4.4 Avaliação de desempenho . . . 45
4.4.1 Procedimento . . . 45
4.4.2 Resultados . . . 47
4.4.3 Conclusão . . . 52
4.5 Sumário . . . 53

5 Conclusões 55
5.1 Trabalho Futuro . . . 56

Bibliografia 58

A Avaliação de desempenho 63
A.1 Configuração 1 . . . 63
A.2 Configuração 2 . . . 64
A.3 Configuração 3 . . . 65

B Mensagens protocolares 67

Lista de Figuras

2.1 Regiões distribuídas pelos *RegionServers* 10
2.3 Exemplo de pares chave-valor. 11
2.4 Componentes principais do HBase 12
2.5 Pedidos *batch* para executar um *endpoint coprocessor* 15

4.1 Instanciação do *Prepared Scan* 40
4.2 Métodos das fases do *Prepared Scan* 41
4.3 Exemplo de execução de um *Prepared Scan* 42
4.4 Definição do *Scan* em PB 43
4.6 Desempenho do *Prepared Scan* e *Scan* na Configuração 1 . . . 48
4.7 Utilização da rede por operação 49
4.8 Desempenho do *Prepared Scan* e *Scan* na Configuração 2 . . . 50
4.9 Desempenho do *Prepared Scan* e *Scan* na Configuração 3 . . . 51

B.1 Definição de mensagens protocolares do *Prepared Scan* 68
B.2 Definição de mensagens protocolares do *Direct Prepared Scan* . 69

Lista de Tabelas

3.1 Denominações e caraterísticas das máquinas usadas nos testes 25

3.2 Resultados relativos as diferentes instâncias de *RegionServers* 27

3.3 Resultados relativos ao impacto do uso de *coprocessor* 30

3.4 Resultados relativos ao custo associado ao tamanho dos qualificadores 32

3.5 Resultados obtidos nas operações sobre variadas famílias de colunas 34

3.6 Resultados relativos ao custo associado ao tamanho dos qualificadores 37

A.1 Resultados do débito (ops/s) na Configuração 1 63

A.2 Resultados da latência (ms) na Configuração 1 64

A.3 Resultados do débito (ops/s) na Configuração 2 64

A.4 Resultados da latência (ms) na Configuração 2 65

A.5 Resultados do débito (ops/s) na Configuração 3 65

A.6 Resultados da latência (ms) na Configuração 3 66

Capítulo 1

Introdução

Ao longo dos últimos anos, a quantidade de informação que tem sido armazenada e processada pelas mais diversas aplicações tem crescido de forma exponencial. Este crescimento revelou as limitações dos tradicionais sistemas de gestão de bases de dados relacionais (SGBDR), levando à conceção de novos sistemas.

As bases de dados NoSQL [1], também denominadas de não-relacionais, são então apresentadas como soluções capazes de acompanhar o crescimento dos dados e atacar os problemas cada vez mais evidentes nas bases de dados relacionais. Estes sistemas são executados numa configuração distribuída com múltiplos nós, pelos quais a informação total é dividida de acordo com uma estratégia bem definida em cada um dos sistemas. À técnica da divisão dos dados dá-se o nome de particionamento e é realizada de forma transparente para a aplicação, contrariamente ao que se verifica nos SGBDR devido à sua impraticabilidade e custos associados [2]. Esta técnica possibilita a escalabilidade horizontal, isto é, acrescentar novos nós à medida que for necessário a custos reduzidos. Desta forma, o sistema é escalável a custos reduzidos, comparativamente com os que estão associados à escalabilidade

vertical que os SGBDR favorecem que implica a aquisição de nós com recursos avançados. Devido a estas características, os sistemas NoSQL são fortes candidatos a serem integrados em ambientes *Web* [3] e também em plataformas *cloud* [4], uma vez que estas se encontram em constante crescimento.

Para atingir uma solução capaz de se enquadrar no processamento analítico em larga escala, foi necessário relaxar algumas propriedades que caraterizam as bases de dados relacionais, tal como os modelos de concorrência e de dados a estas associado. Surgiu também uma nova forma de aceder aos dados que se limita a um conjunto de operações simples: `get`, `put`, `scan` e `delete`, em detrimento da usual linguagem *Structured Query Language* (SQL). Operações adicionais podem ser fornecidas por alguns sistemas, como operações atómicas de incrementos. A desnormalização dos dados traduz-se na duplicação de partes dos mesmos em múltiplas tabelas, por forma a evitar a necessidade de agregações e operações `join` e tornar as leituras mais rápidas [5]. Esta é de facto uma característica dos sistemas NoSQL que implica o redesenho do esquema de dados de forma distinta da que era realizada nos SGBDR. Com o intuito de promover uma maior flexibilidade na forma como os dados são guardados, estas soluções não têm por base uma estrutura dos dados bem definida e permitem que parte desta, como as colunas, seja arbitrária e ajustável à medida que ocorre a manipulação de dados.

Apesar de nem todas precisarem desta flexibilidade, há aplicações que migram os seus dados dos sistemas relacionais para NoSQL, tendo como alvo exclusivamente a sua escalabilidade. No entanto, esta migração não é direta e pode representar alguns desafios, entre estes o redesenho eficiente da estrutura dos dados e a completa adaptação de uma nova interface de interação com o sistema [6, 7]. Existem alguns trabalhos que têm como objetivo tornar a migração dos dados automática [8]. Por outro lado, a migração também

é facilitada por tecnologias que oferecem uma linguagem de interrogação semelhante a SQL sobre os sistemas NoSQL. Esta linguagem é convertida pelo próprio sistema em chamadas nativas ao sistema de gestão e processamento de dados NoSQL [9, 10]. As camadas de linguagem de interrogação são também muito aplicadas em contexto da computação na nuvem, uma vez que reúnem um conjunto de sistemas SGBDR e NoSQL.

1.1 Problema

A migração de dados com uma estrutura regular, nomeadamente tendo todas as linhas as mesmas colunas, implica ineficiência tanto no armazenamento como em consultas posteriores. Em particular, efetua-se repetidamente o armazenamento e transmissão da meta-informação relativa ao nome das colunas, pois o sistema não tem forma de antecipar que essa meta-informação é igual para todas as linhas. Isto é um problema importante no Apache HBase (HBase), sendo este o sistema mais popular e integrante nalgumas das maiores aplicações, como Facebook [11].

1.2 Contribuição

A primeira contribuição deste trabalho é o estudo do impacto de diversas características do HBase no armazenamento e processamento de dados relacionais.

Como segunda e principal contribuição, é apresentado o *Prepared Scan* para o HBase que visa otimizar a consulta dos dados, quando a aplicação conhece efetivamente a estrutura dos dados que lhe está associada, sendo esta a operação de leitura mais comum e que torna mais evidente este problema.

Esta é disponibilizada como uma extensão no protocolo de comunicação entre cliente e servidor e, como tal, não requer alterações no código-fonte sendo, por isso, de fácil integração em qualquer contexto onde o HBase esteja integrado.

1.3 Estrutura do documento

Este documento encontra-se organizado por capítulos. O Capítulo 2 deste documento apresenta uma contextualização dos sistemas NoSQL, em particular do Apache HBase, e o estado da arte relativamente aos trabalhos que têm sido elaborados de modo a obter um melhor desempenho neste sistema. No Capítulo 3, é avaliado o desempenho do HBase, de modo a compreender os pontos de contenção que este apresenta. O Capítulo 4 foca-se na descrição do *Prepared Scan*, quer a nível conceptual quer de implementação, e, de seguida, avalia-se o desempenho do *Prepared Scan* aplicado a três cenários. O Capítulo 5, por último apresenta as conclusões deste trabalho e as metas futuras.

Capítulo 2

Estado da arte

Os sistemas NoSQL têm propósitos, características e modelos de dados muito distintos dos sistemas tradicionais.

Esta secção refere algumas características dos sistemas NoSQL e, mais detalhadamente, do HBase, bem como as metodologias usadas para obter um melhor desempenho na execução de operações.

2.1 Bases de dados *NoSQL*

Entre as definições “Not Only SQL” e “Not Relational”, os sistemas NoSQL foram concebidos para responder aos problemas existentes nas bases de dados tradicionais [1]. O objetivo-chave para esta categoria de sistemas foi a possibilidade de escalar horizontalmente, idealmente, sem qualquer participação da aplicação nesse processo, para operações simples de leitura e escrita em vários nós. Isto não é possível nas bases de dados relacionais, pois, com a adição de novos nós, é essencial intervir na forma como os dados estão distribuídos pelas várias instâncias e no método de acesso a estes dados por parte a aplicação. Em alternativa a esta complexidade, a preferência recai sobre a

escalabilidade vertical, que consiste na aquisição de mais recursos para cada nó, resultando em custos elevados ao longo do tempo.

Contudo, para os sistemas NoSQL serem capazes de suportar a escalabilidade horizontal sem partilha de dados entre nós, foi necessário relaxar algumas propriedades dos sistemas relacionais. Um modelo de concorrência mais fraco, operações simples como CRUD (*Create-Read-Update-Delete*) e o próprio modelo de dados são algumas características destes sistemas que os distinguem dos SGDBR.

A ideia subjacente aos sistemas não relacionais é que é possível obter um melhor desempenho e escalabilidade deixando de parte as restrições provenientes do modelo ACID (*Atomicity, Consistency, Isolation and Durability*).

O modelo de dados destes sistemas favorece aquilo que, nos sistemas tradicionais, se denomina por desnormalização. Este processo resulta na redundância de alguns dados e num melhor desempenho das operações de leitura, uma vez que dispensa agregações e operações `join`. Este modelo de dados fornece uma completa flexibilidade em termos de estruturação e distribuição de dados de uma forma transparente para a aplicação. Com este modelo, também é possível executar operações em paralelo, como sobre um conjunto de dados sem qualquer complexidade, uma vez que não dependem entre si.

Dentro da categoria NoSQL, existem diversos sistemas que se organizam noutras sub-categorias de acordo com o seu modelo de dados específico, tais como bases de dados orientadas a documentos que indexam estruturas de dados complexas (tratadas como documentos), bases de dados orientadas a colunas que persistem informação organizada por colunas pré-definidas e, por fim, bases de dados de grafos que lidam com estruturas de grafos e permitem interrogações semânticas com nós dos mesmos.

Todas estas sub-categorias enquadaram-se, na sua essência, nas bases de dados chave-valor ou seja, tomam o par chave-valor como o seu tipo fundamental.

2.1.1 Bases de dados chave-valor

Bases de dados chave-valor são sistemas cujo modelo se baseia num tipo fundamental: par chave-valor. Isto é, os dados são compostos por chaves únicas que identificam determinados valores, desde os mais simples até ao mais estruturalmente complexos.

As operações suportadas por estes sistemas, essencialmente, são a inserção (`put`), a remoção (`delete`) e procura de chaves (`get` e `scan`). Permitem atingir altos níveis de escalabilidade, uma vez que distribuem os pares chave-valor, possivelmente ordenados pela chave, pelos vários nós e os mantêm em memória RAM para acelerar operações de leitura.

Sistemas como o *MemCached* não têm a capacidade de persistir os dados, mantendo-os indefinidamente em memória. Outros, como *Redis*, possuem as duas capacidades [1]. Ambos são puras bases de dados chave-valor, uma vez que não possuem qualquer tipo de organização dos dados que mantêm. Existem outros sistemas mais complexos, como o HBase, que lidam com pares chave-valor e os organizam, inclusive no armazenamento, por famílias de colunas definidas aquando da criação de uma tabela. Estes sistemas são classificados como bases de dados orientadas a colunas.

2.1.2 Linguagens de interrogação sobre NoSQL

Até ao aparecimento dos sistemas NoSQL, os SGBDR foram então adotados nas aplicações. Estes sistemas disponibilizam a linguagem de interrogações

SQL que se foi tornando bem conhecida entre utilizadores e programadores e, como tal, as aplicações desenvolvidas são ainda dependentes desta.

A necessidade de obter escalabilidade e um melhor desempenho leva à migração de aplicações do contexto relacional para o não-relacional. No entanto, os sistemas NoSQL são diversos e, como não fornecem uma linguagem comum, torna-se difícil a adoção destes. Para além disso, a migração não é direta [6], tipicamente obrigando ao redesenho da estrutura de dados e à dispensa de operações de agregação e `join`, que não são suportadas por estes sistemas, uma vez que não há dependências entre dados em diferentes tabelas, como acontece nos SGBDR. A própria desnormalização dos dados que a migração implica conduz a uma forma de interação distinta da habitual.

Para diminuir este esforço, surgem diversas alternativas à migração completa, isto é, surgem mecanismos de tradução da linguagem de interrogações SQL para as operações nativas dos sistemas NoSQL e que dão suporte ao processamento de um sub-conjunto de interrogações [12]. Como exemplo, tem-se projetos como *BigQuery* [13], *Apache Hive* [14], *Apache Phoenix* [15] e *Tenzing* [16].

Apesar de tornar mais fácil a alteração do sistema de gestão e processamento de dados, esta tradução tem um custo que se reflete num acréscimo no tempo necessário para a execução de uma operação [9, 10]. No entanto, na maioria das aplicações, torna-se muito mais compensador a adoção de uma camada que possibilita o uso de uma linguagem de interrogação semelhante a SQL sobre o sistema não-relacional do que a migração direta e completa dos dados.

2.2 Apache HBase

Apache HBase,[17] é uma base de dados não-relacional, distribuída e escalável. Inspirada na *BigTable* da Google,[18], é idealizada como um *sorted map* multi-dimensional indexado através do tuplo (identificador de linha, coluna, marca temporal). Tanto o identificador de linha como o valor do tuplo são um conjunto de *bytes* não interpretados. O *sorted map* pode ter um número arbitrário e dinâmico de colunas que, por sua vez, se encontram agrupadas em famílias de colunas. A identificação de uma coluna é dada pela concatenação de uma família e de um qualificador de valor, sendo a família de colunas um conjunto de caracteres legível e o qualificador um vetor de *bytes* opcional. Tipicamente as famílias de colunas são definidas de forma estática, contrariamente aos qualificadores que podem ser regularmente criados enquanto o sistema se encontra em execução.

O conjunto de pares chave-valor existentes é repartido horizontalmente, resultando em regiões representadas na Figura 2.1 que são a unidade de escalabilidade do HBase. Estas regiões são, entretanto, distribuídas por diversos servidores denominados por *RegionServers*. Estas regiões podem ser definidas aquando da criação de uma tabela ou criadas automaticamente, uma vez atingido um tamanho dado por parâmetros na configuração.

As famílias de colunas agrupam qualificadores, permitindo uma melhor organização a nível semântico e, quando persistidas, uma maior localidade, uma vez que fazem parte do mesmo ficheiro, denominado *HFile*. O número de famílias de colunas criadas não é limitado por um parâmetro de configuração. No entanto, deve ser reduzido, para não ser necessário aceder a uma quantidade de ficheiros maior do que o necessário durante as operações.

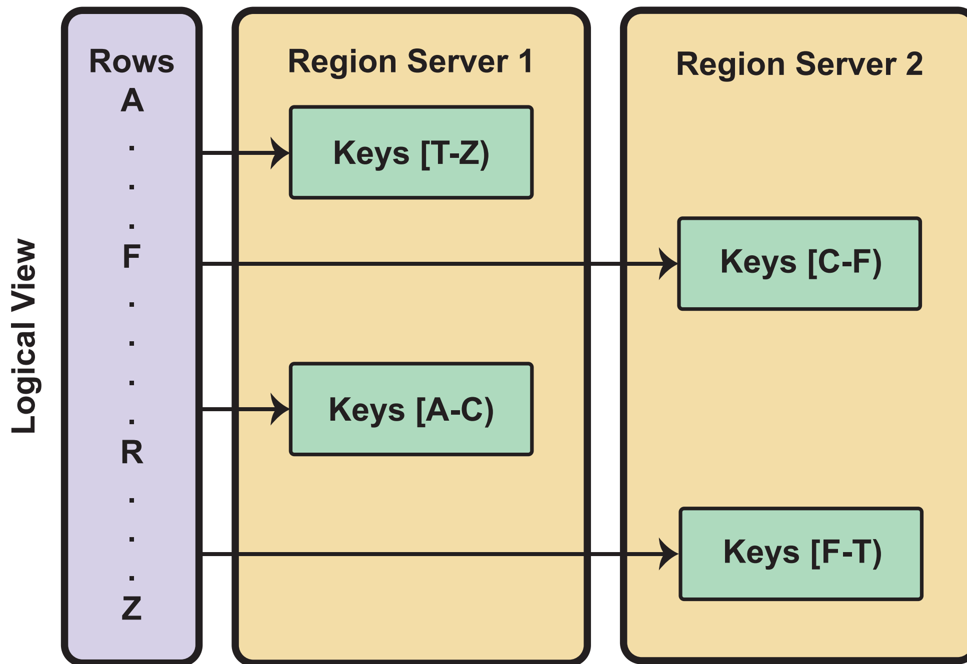


Figura 2.1: Regiões distribuídas pelos *RegionServers*

2.2.1 Pares chave-valor

O tipo fundamental do HBase é o par chave-valor. O formato deste par é visível na Figura 2.2 e corresponde, com alguns campos de controlo, ao tuplo usado para indexar os valores. Quer as operações de escrita quer as de leitura lidam com pares chave-valor.

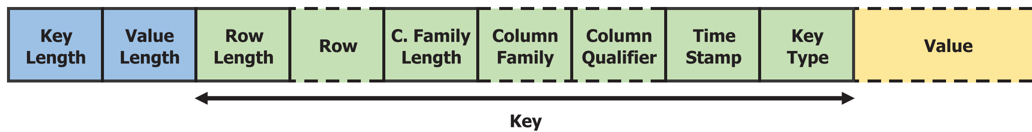


Figura 2.2: Formato do par chave-valor do HBase

Para efeitos de compreensão, considere que os pares chave-valor na Figura 2.3, outrora inseridos através da operação `put`, existem na base de dados.

Estes pares são internamente guardados e mantidos com uma ordem lexicográfica dos identificadores de linha. A ordem total é dada primeiramente

```
(row−1, colfam1:qual1, 1294065304642) −> value
(row−1, colfam1:qual2, 1294065304642) −> value
(row−1, colfam2:qual1, 1294065304642) −> value
(row−2, colfam1:qual1, 1294065308222) −> value
(row−2, colfam2:qual1, 1294065303242) −> value
(row−3, colfam1:qual2, 1294065308222) −> value
```

Figura 2.3: Exemplo de pares chave-valor.

pelos identificadores de linha, seguidos do nome da família de colunas, seguido do qualificador e, por fim, pela marca temporal (em ordem inversa).

É possível obter uma representação destes dados recorrendo a uma tabela, se se assumir que cada par chave-valor é uma célula que resulta da interseção do identificador de linha com a coluna envolvente. A marca temporal pode ser assumida como mais uma dimensão na tabela, uma vez que o sistema suporta múltiplas versões de células. A Tabela 2.1 é um exemplo dessa representação. As células em branco representam a inexistência de qualquer valor que resulta da interseção do identificador de linha com a coluna.

Tabela 2.1: Representação lógica de pares chave-valor do HBase

	colfam1		**colfam2**
	qual1	**qual2**	**qual1**
row-1	value	value	value
row-2	value		value
row-3		value	

2.2.2 Arquitetura

O HBase é composto por três grandes componentes visíveis na Figura 2.4. Estas são a interface com o cliente (API), o *Master* e *RegionServer*.

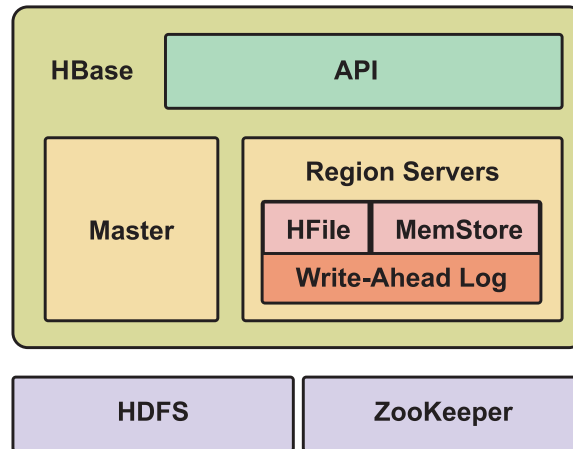


Figura 2.4: Componentes principais do HBase

O *Master* é responsável por lidar com a distribuição de carga, alocando as regiões nos *RegionServers* com menor taxa de utilização juntamente com a meta-informação.

Os *RegionServers* atendem os pedidos de leitura e escrita nas regiões que servem, comunicando diretamente com o cliente. Em cada um dos *RegionServers* existem três componentes, nomeadamente um conjunto de *HFile*, que contém os pares chave-valor das respetivas famílias de colunas, uma *MemStore* onde as operações recém-chegadas se mantém até serem persistidas em disco e, por fim, um WAL que garante a durabilidade dos dados em caso de falta.

O *Apache Zookeeper*, por sua vez, é um serviço de coordenação distribuído que usufrui da propriedade de alta disponibilidade [19].

Por último, o HBase é executado sobre o sistema de ficheiros o *Hadoop Distributed File System* (HDFS) [20], um sistema de ficheiros distribuído baseado no *Google File System* [21].

2.2.3 Interface com o cliente

O acesso aos dados é facultado através de um conjunto de operações que são caraterísticas das estruturas de dados que lidam com pares chave-valor. Todas as alterações efetuadas através destas operações no HBase são imediatamente visíveis nas leituras subsequentes, o que favorece um modelo de consistência forte.

`Get` Obtém pares chave-valor de uma determinada linha;

`Put` Insere um par chave-valor numa linha nova ou existente;

`Scan` Obtém múltiplos pares chave-valor de um intervalo de identificadores de linha;

`Delete` Remove pares chave-valor de uma ou mais linhas.

Estas são as operações existentes geralmente em todos os sistemas NoSQL. No entanto, o HBase em particular oferece operações *read-modify-write* que aplicam diversas instruções de forma atómica:

`IncrementColumnValue` Incrementa um valor numa coluna [1];

`CheckAndPut` Verifica se o valor de uma linha, família ou de um qualificador corresponde ao esperado e, em caso afirmativo, executa a operação `put`.

2.2.4 Escrita de dados

Aquando da chegada de um pedido de escrita, como a operação `put`, a um *RegionServer*, este acrescenta uma entrada no WAL. Uma vez persistida, essa entrada é colocada na *MemStore*.

[1] Um valor que não seja do tipo *big-endian long* pode resultar numa exceção.

A *MemStore* é escrita efetivamente em disco quando atinge um determinado tamanho, originando um novo *HFile*. Após algumas execuções equivalentes a esta, o número de ficheiros *HFile* relativos à mesma família de colunas será maior do que o desejável e, por isso, o sistema despoleta o processo de compactação destes de forma assíncrona.

É durante a compactação que os pares chave-valor existentes nos vários *HFile* da mesma família de colunas são ordenados e os que foram previamente marcados como apagados são efetivamente eliminados. Este processo dá origem a um novo *HFile* que mantém todos os pares chave-valor respetivos a apenas uma família de colunas, o que promove a localidade em pedidos futuros.

2.2.5 Leitura de dados

Quando uma operação de leitura é solicitada, os dados que devem ser retornados como resposta podem não se encontrar no mesmo ficheiro. Isto é, uma determinada linha pode ser transversal a múltiplos ficheiros *HFile* e, simultaneamente, à *MemStore*.

Antes de ler todos os ficheiros de dados que podem conter a informação de uma determinada linha, é realizada uma exclusão rápida usando a marca temporal. Os ficheiros resultantes e a *MemStore* são alvo de procura, de modo a encontrar a chave pretendida.

Note que o facto de ser necessário aceder a múltiplos ficheiros *HFile* para obter os dados de uma linha pode resultar num pior desempenho nas operações de leitura.

2.2.6 *Coprocessors*

O HBase permite a execução de código arbitrário no sistema, através do uso de *coprocessors*. São disponibilizados dois tipos de *coprocessors*: *observers* e *endpoints*. Os *observers* despoletam uma determinada ação quando um comportamento do sistema é verificado. Por sua vez, os *endpoints* permitem a extensão do protocolo e execução de código remoto, através de *Remote Procedure Call* (RPC), abstraindo o cliente da necessidade de ter em conta os característicos problemas dos sistemas distribuídos.

Alojados em cada *RegionServer*, são equivalentes aos *stored procedures* das conhecidas bases de dados relacionais e podem lidar diretamente com cada região. Desde a versão `0.94.2`, existe em cada *RegionServer* uma estrutura partilhada do tipo `Map<String, Object>` entre as várias instâncias de um determinado *coprocessor* que armazena estado que se pretende manter entre chamadas RPC, tal como se pode verificar na Figura 2.5.

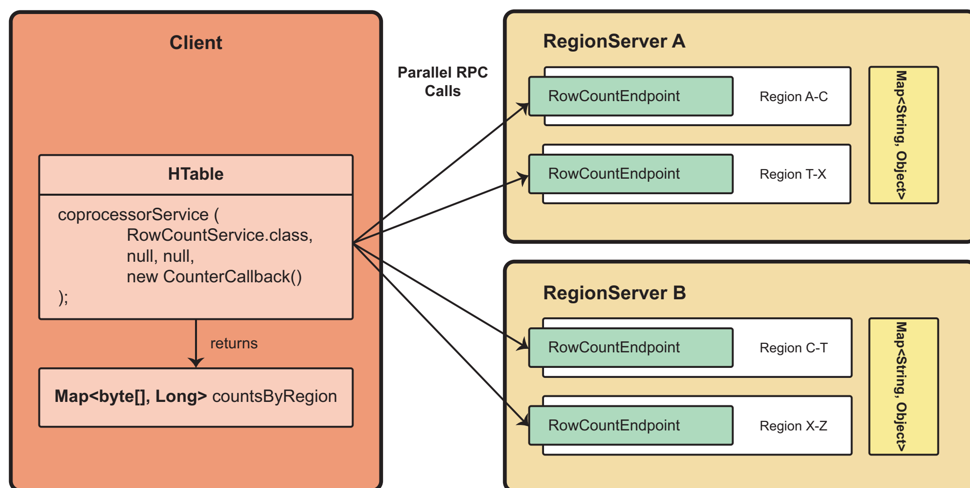


Figura 2.5: Pedidos *batch* para executar um *endpoint coprocessor*

2.3 Análise de desempenho

Para obter as métricas que representam informação sobre o comportamento de uma determinada tecnologia, usam-se ferramentas que simulam clientes. Nos casos que serão referidos de seguida, é originada uma determinada quantidade de carga sobre os serviços alvos de análise e gerados relatórios finais sobre o comportamento do sistema durante o teste.

2.3.1 Métricas relevantes

As métricas que são analisadas são importantes para a deteção da presença de pontos de contenção. Neste tipo de tecnologias, procede-se à recolha das métricas que que se seguem:

- `CPU` Revela o estado do CPU durante a incidência da carga.
- `Memória` Indica o estado da memória, isto é, memória livre, ocupada e em *cache*, durante o teste de desempenho.
- `Disco` Informa sobre o modo de utilização do disco.
- `Rede` Regista o modo como a rede está a ser utilizada.
- `Débito` Demonstra o número de operações que são concluídas por unidade de tempo.
- `Latência` A quantidade de tempo que uma determinada operação demorou a ser executada.

2.3.2 Ferramentas de teste

Para proceder à análise de desempenho do HBase, entre outras, destacam-se duas *frameworks* que serviram de base aos artigos que relatam métodos para

obter um melhor desempenho. Uma delas foi desenhada e desenvolvida pela equipa de investigação da *Yahoo!* e a outra deriva da primeira, desta vez com a autoria do *Parallel Data Lab*.

YCSB - Yahoo! Cloud Serving Benchmark

YCSB é uma popular *framework* extensível de teste de desempenho [22], desenvolvida na linguagem de programação *Java*, compatível com várias tecnologias como HBase, *Cassandra* [23], *PNUTS* [24], e inclusive uma implementação simples que suporta particionamento em *MySQL*. Surge com o intuito de facilitar a comparação entre diferentes tecnologias da nova geração de tecnologias de armazenamento da nuvem.

São disponibilizados múltiplos comandos e padrões parametrizáveis de acesso a dados que simulam diversos tipos de aplicações, isto é, desde a escrita intensiva até à leitura intensiva. O tamanho dos dados e a distribuição dos pedidos são alguns dos parâmetros configuráveis. Além da possibilidade de personalizar os padrões de acesso a dados, torna-se possível implementar operações próprias através de uma interface disponível, tornando esta ferramenta expansível para outros sistemas.

As métricas devolvidas pelo YCSB, no resumo final dos testes, são o débito, em operações por segundo, e a latência, em milissegundos.

A carga é gerada através da instanciação de clientes, cuja quantidade é configurável, não sendo possível distribuir os clientes por múltiplos nós, o que indica que, para testes com milhares de clientes, serão necessários recursos adequados. Para além desta limitação, também não são disponibilizados parâmetros essenciais como o tempo de estabilização do sistema, durante o qual não ocorrem medições, e distribuições constantes.

YCSB++ - Yahoo! Cloud Serving Benchmark++

YCSB++ é uma extensão da ferramenta original que conta com diversas capacidades que haviam sido necessárias durante fases de testes com proporções maiores [25]. Foi desenvolvida com foco nos sistemas HBase e *ACCUMULO*.

Como referido, o YCSB não suporta múltiplos clientes distribuídos por vários nós. Aproveitando a modularidade do YCSB, YCSB++ acrescentou novas extensões com funcionalidades mais avançadas e importantes para o sistema que será usado durante o desenvolvimento da tese, HBase, das quais duas se destacam:

- **Testes com clientes em múltiplos nós.** O *Apache Zookeeper* é usado para sincronizar os clientes, no início e no fim de cada teste.
- **Dados são gerados de forma determinística.** Uma vez que os dados gerados são conhecidos, é possível utilizar filtros adequados na consulta dos dados.

Estas extensões contribuem para um método mais sofisticado de análise de desempenho em *clusters*, algo que não era possível com a versão original do YCSB. No entanto, continuam a ser necessários os parâmetros configuráveis referidos acima, para permitir uma maior flexibilidade e um espectro mais abrangente de testes.

2.4 Metodologias de otimização

Nesta secção, abordam-se técnicas estudadas para obter um melhor desempenho das bases de dados NoSQL, algumas já aplicadas nas maiores aplicações.

2.4.1 Armazenamento

Uma ideia *naïve* para aumentar o desempenho é a aquisição de novo *hardware* de armazenamento. Porém, esta abordagem torna-se muito dispendiosa e, por vezes, os benefícios que se obtêm podem não compensar tal investimento. É necessário, então, identificar o ponto de contenção e ponderar as possíveis soluções e possíveis sistemas resultantes, de modo a compreender se o rácio desempenho/custo compensa.

O serviço de mensagens do Facebook, uma aplicação inserida na categoria de leitura intensiva, é um exemplo de um caso de estudo do desempenho do HDFS sobre HBase e do impacto da adição de uma camada de *hardware* com o objetivo de otimizar o armazenamento de dados [26].

Os pedidos a este serviço foram intercetados individualmente, os detalhes destes colecionados e repetiram o processo simulando diferentes combinações de *hardware* para o armazenamento. O registo dos detalhes foi efetuado com o padrão de acesso a dados real, ou seja, em produção, com o intuito de não colocar carga adicional desnecessária sobre o serviço.

Uma vez que o serviço é uma aplicação de leitura intensiva, cerca de 99% do *core I/O* é devido a leituras. No entanto, evidencia-se que, como o HBase realiza compactações para manter as leituras eficientes, o rácio de leituras/escritas desce para 79/21. Os motivos pelos quais o número de escritas é mais elevado, face ao que é necessário para a aplicação, são a não compressão dos dados nas tarefas de *logging*, responsáveis por manter a durabilidade dos dados, e o processo de compactação de múltiplos ficheiros *HFile* num só. Estima-se que a compressão do *log* reduziria o número de escritas em cerca de quatro vezes. Por sua vez, a compactação tem um impacto excessivamente grande, isto é, dezassete vezes mais escritas do que o que realmente é o essencial no serviço.

A análise dos padrões de acesso a dados pode ajudar a compreender que tipo de solução se pode aplicar por forma a obter um melhor desempenho. No caso de estudo referido, onde se verifica que existem acessos intensivos a dados específicos, tornou-se interessante interpretar os efeitos do uso de *cache*. Os autores propuseram a adição de uma camada *flash* como *cache* imediatamente abaixo da memória RAM, com políticas de promoção de dados, isto é, os dados envolvidos podem ou não ser promovidos para a memória RAM, o que possibilitou um aumento do desempenho num fator de três.

Por outro lado, o uso de *flash* como *cache* requer um estudo dos padrões de acesso aos dados, uma vez que o desempenho eficiente da *cache* depende em grande parte desses padrões. Em alternativa a este procedimento que requer um conhecimento que muda de aplicação para aplicação, é possível obter um melhor desempenho independentemente dos dados. Isto é, o HBase possui várias componentes relevantes nas operações que necessitam de armazenamento, tal como o WAL e tabelas de meta-informação e, por isso, a aquisição de *hardware* de armazenamento mais rápido, como *flash*, para essas componentes pode trazer vantagens no desempenho [27].

No entanto, é importante referir que o armazenamento *flash* tem custos muito superiores e um tempo de vida mais curto que o armazenamento tradicional de discos rígidos. Como tal, a decisão de adicionar *flash* como *cache* deve ser tomada no caso em que o padrão de acesso aos dados não seja muito variável. Caso contrário, torna-se menos dispendioso e mais vantajoso usar *flash* para a persistência de dados de componentes relevantes do HBase.

2.4.2 Balanceamento

A eficiência e um melhor desempenho dependem, entre outros fatores, das configurações adequadas dos sistemas de gestão e processamento de dados

que as aplicações usam. As configurações destes devem ser efetuadas de acordo com os padrões de acesso a dados, embora nem sempre seja possível obtê-los previamente. Para além disso, os próprios padrões podem mudar ao longo do tempo, tornando impraticável a configuração manual. Ainda que atingindo um nível próximo do ótimo em certas circunstâncias, num contexto de computação na nuvem, torna-se muito difícil configurar corretamente de modo a contribuir positivamente para as aplicações, principalmente quando não se conhece efetivamente a aplicação.

Elasticidade é uma propriedade chave das plataformas de computação na nuvem e define-se como a capacidade de um sistema crescer ou diminuir o consumo dos seus recursos de acordo com o que é exigido [28]. As bases de dados NoSQL foram desenhadas para lidar com a adição e remoção de nós de uma forma transparente. Idealmente, num contexto computação na nuvem, a adição de recursos é realizada automaticamente aquando da sobrecarga do sistema, para manter o nível de qualidade dos serviços e, posteriormente, é dada a remoção no relaxamento dessa carga [29]. No entanto, torna-se insuficiente este processo e definitivamente dispendioso, por si só, uma vez que os nós adicionados podem não ser efetivamente necessários.

Uma vez que todos os nós são configurados de uma forma homogénea, é possível que causem um impacto negativo nas aplicações, devido à heterogeneidade existente no acesso aos dados de cada uma. Estas podem levar a uma má gestão de recursos e, por isso, possíveis adições de nós quando não seria necessário se as configurações fossem apropriadas para cada aplicação em questão.

Para atacar este problema, surgem ferramentas que visam dar uma aplicabilidade abrangente no que toca à elasticidade das plataformas de computação na nuvem [30, 31]. Estas não só adicionam e removem nós conforme

necessário, como também os reconfiguram de modo a tornar eficiente o seu uso, favorecendo indiretamente o modelo *pay-as-you-go*. Desta forma, é possível obter uma melhor configuração em cada nó para o padrão de acesso a dados a ocorrer no mesmo, resultando num melhor desempenho e aproveitamento de recursos.

As decisões deste tipo de ferramentas são geralmente tomadas através da recolha e análise de métricas. Embora a reconfiguração heterogénea seja possível de forma automática, este processo de reconfiguração não pode ser realizado com o sistema em execução, uma vez que o HBase não tem essa capacidade, o que requer a reinicialização do mesmo. Este processo incorre num custo que pode ser agravado, no caso em que o padrão de acesso a dados seja frequentemente inconstante. Neste caso, esta reconfiguração acaba por afetar gravemente o desempenho do sistema. No entanto, a partir da versão `1.1.0` do HBase, a configuração *online* encontra-se em desenvolvimento gradual, pelo que pode verificar-se, a curto prazo, uma melhoria do desempenho.

2.4.3 Comunicação

O HBase faz parte das tecnologias NoSQL em que a latência e a execução de operações por uma determinada unidade de tempo são cruciais para o seu sucesso e, por isso, torna-se relevante estudar e otimizar a comunicação por rede, sendo esta uma variável importante na equação do desempenho da tecnologia em questão.

Ao longo dos últimos anos, o domínio da computação de alto desempenho tem vindo a explorar redes de baixa latência e com características avançadas, como *InfiniBand*. Através do uso deste tipo de redes e tendo em conta os dados do HBase é possível usufruir de *Remote Direct Memory Access*

(RDMA) sobre esta tecnologia [32, 33]. A comunicação entre os diversos componentes do HBase é implementada sobre *Java Sockets*, favorecendo a portabilidade entre plataformas em detrimento do desempenho. Isto é, por forma a ser multi-plataforma, é necessário lidar com as diferenças entre estas, com impacto no desempenho.

Recorrendo a implementações nativas através da *Java Native Interface*, têm sido realizados trabalhos no HBase para suportar redes de alto desempenho, tornando-o capaz de lidar com *RDMA*, isto é, ser capaz de ler e modificar conteúdos em memória de outro processo remoto sem necessidade de intermediários. Após a avaliação do comportamento da nova capacidade do HBase, com operações `get` cujas respostas se estendem a 1KB, verifica-se uma melhoria de desempenho.

Desta forma, prevê-se que, usando redes de alto desempenho com suporte a *RDMA*, é possível obter um melhor desempenho reduzindo a latência existente na preparação e realização da comunicação entre nós.

2.5 Sumário

Nesta secção, foi abordado o propósito da conceção dos sistemas NoSQL capazes de lidar com problemas que se tornaram evidentes nos SGBDR. Dentro destes, introduziu-se as bases de dados chave-valor, focando-se no sistema HBase, cujo modelo de dados, comportamento e arquitetura foram descritos com mais detalhe. Verificou-se, também, que é possível adotar este sistema sem implicar um esforço extraordinário durante o processo de migração, através do uso de mecanismos de conversão de interrogações SQL em chamadas nativas destes sistemas.

Referiu-se alguns trabalhos que têm contribuido para a compreensão do

desempenho deste sistema, desde as ferramentas de teste como YCSB, até à melhoria da infraestrutura de rede recorrendo a *InfiniBand*.

Embora todos os trabalhos referidos contribuam para uma melhoria de desempenho em geral, isto é, independentemente do tipo de dados, nenhuma é focada na otimização de operações aplicadas sobre dados estruturados de forma regular, como é sucedido no contexto de aplicações que migram para sistemas NoSQL.

Seguindo a linha destes trabalhos, na próxima secção será analisado o sistema HBase, de modo a compreender quais os pontos de contenção existentes que podem ser alvo de otimização.

Capítulo 3

Análise de desempenho

Como um dos objetivos deste trabalho, pretende-se caracterizar o comportamento do HBase em cenários de utilização de diferentes decisões na estrutura dos dados, configurações distintas e com dados relacionais, isto é, dados que são estruturados de forma regular.

3.1 Ambiente experimental

Para a realização dos testes, recorreu-se a máquinas com as denominações e características enunciadas na Tabela 3.1.

Prefixo do nó	**Características**
Nx	*AMD Opteron(tm) Processor 6172* 120GB RAM
NCx	*Intel i3 CPU @ 3.10GHz* 8GB *RAM* Disco local a 7200 RPM

Tabela 3.1: Denominações e caraterísticas das máquinas usadas nos testes

Em termos de distribuição de nós do HBase, é importante realçar que cada *RegionServer* coexiste, durante os testes, com o respetivo *DataNode* do HDFS, permitindo obter localidade máxima dos dados.

Os testes enunciados de seguida realizam-se com recurso à ferramenta YCSB, devido à sua popularidade como uma solução rápida de teste de carga e recolha de métricas. As configurações do HBase, durante os testes, são as definidas por omissão. Caso contrário, será referido explicitamente, ao longo do capítulo.

3.2 *RegionServers* locais

Os *RegionServers* podem ser instanciados em nós distintos de forma independente, através de um processo *Java Virtual Machine (JVM)* em cada nó, ou localmente, isto é, com processos JVM separados. Como cada JVM contém o seu *Garbage Collector* (GC) e o HBase requer bastante memória, é necessário avaliar a implicância que a gestão do GC tem sobre o desempenho do HBase, no caso em que só existe um *RegionServer* com uma extensa *heap* e em outro caso em que existem quatro *RegionServers* locais com *heap* mais reduzida.

3.2.1 Descrição do cenário

Existem três nós distintos, *N1*, *N2* e *NC1*. *N1* é o nó-cliente, onde são executados os clientes. No nó *NC1*, encontra-se uma instância *Master* e *Apache Zookeeper*, juntamente com um *NameNode*. No nó *N2*, corre um (ou quatro, conforme a configuração) *RegionServer*, com 36GB (ou 9GB no caso dos quatro) de *heap*, com o seu *DataNode*.

Neste cenário, usa-se o HBase na versão `v0.98.7`, com HDFS na versão

`v2.3.0`.

O YCSB cria uma única tabela, com apenas uma família de colunas, dispersa por regiões distribuídas equitativamente, e povoa-a com 12 milhões de linhas, cada uma com dez qualificadores. O valor de cada chave tem 255 *bytes*. São executadas dez milhões de operações `scan`, cuja quantidade de linhas retornadas é variável entre dez e cem.

Seguem, então, as configurações devidamente numeradas:

1. Um *RegionServer* local;

2. Quatro *RegionServers* locais.

3.2.2 Resultados

No fim de cada teste, obteve-se os resultados enunciados na Tabela 3.2.

	Clientes	**Débito (ops/s)**	**Latência (ms)**
1	**50**	728.8	68.6
	100	735.6	135.9
	200	738.3	270.9
	500	734.6	680.6
	800	733.4	1090.6
2	**50**	750.6	66.54
	100	758.5	131.6
	200	761.7	261.826
	500	765.2	651.0
	800	761.6	1043.52

Tabela 3.2: Resultados relativos as diferentes instâncias de *RegionServers*

Na configuração 1, a quebra no débito é visível com 500 clientes. Já na configuração 2, o débito decresce a partir dos 500. No entanto, a diferença não é significativa. Mais importante é verificar que o débito se mantém aproximadamente o mesmo durante o número de clientes, o que já não se verifica com a latência, que sofre aumentos exagerados de acordo com o número de clientes.

Num modo geral, a configuração 1 tem sempre um pior débito do que a configuração 2.

3.2.3 Conclusão

A contenção do débito sugeriu a consulta do estado de sistema de cada nó, durante os testes. O HBase atingiu o limite da rede, pelo que não será útil retirar uma conclusão sobre esta matéria sem recorrer a novos testes com recursos mais avançados.

3.3 Uso de *coprocessors* do tipo *endpoint*

Uma vez que o HBase permite a extensão do protocolo entre cliente e servidor através de *coprocessors*, torna-se relevante analisar a diferença entre efetuar o processamento dos dados no cliente e ou no servidor, evitando a transmissão de dados pela rede neste último caso.

Neste teste, procede-se à contagem de todas as linhas envolvidas no resultado, sem e com *coprocessor*. O *coprocessor* usado é o `RowCountEndpoint` disponível nos pacotes de exemplos do HBase.

3.3.1 Descrição do cenário

Existem três nós distintos, *N1*, *NC1* e *NC2*. *N1* suporta o YCSB, que instancia os clientes. No nó *NC1* encontra-se uma instância *Master* e *Apache Zookeeper*, juntamente com um *NameNode*. No nó *NC2* corre um *RegionServer*, com 4GB de *heap*, com o *DataNode* respetivo.

Neste cenário, usa-se HBase na versão `v0.98.7`, com HDFS na versão `v2.3.0`.

O YCSB cria uma única tabela, com apenas uma família de colunas numa só região e povoa a mesma com um milhão de linhas, com dez qualificadores em cada uma. O valor de cada chave tem 255 *bytes*. Um único cliente executa 500 operações `scan`, quer com *coprocessors* quer sem, por cada configuração que se segue:

1. Obtém todas as linhas, usando o filtro `FirstKeyOnlyFilter`;
2. Obtém todas as linhas apenas com uma determinada coluna;
3. Obtém todas as linhas de uma determinada família de colunas;

O filtro `FirstKeyOnlyFilter` é responsável por retornar apenas o primeiro par chave-valor por cada identificador de linha, sem verificar qualquer condição adicional. Na configuração 2, são procurados os pares chave-valor em que a coluna seja a que se pretende. O mesmo acontece para a configuração 3, mas no contexto da família de colunas.

3.3.2 Resultados

Uma vez terminada a execução dos testes para cada configuração, obteve-se os resultados enunciados na Tabela 3.3. Os valores a negrito correspondem aos que representam o melhor desempenho.

	Sem *coprocessor*		**Com *coprocessor***	
	Débito (ops/s)	**Latência (ms)**	**Débito (ops/s)**	**Latência (ms)**
1	**0.031**	**32689.9**	**0.051**	**19476.7**
2	0.024	41818.6	0.038	26385.8
3	0.011	87424.7	0.047	21142.5

Tabela 3.3: Resultados relativos ao impacto do uso de *coprocessor*

Em ambos os casos, é possível constatar que a configuração 1 — `FirstKeyOnlyFilter` como filtro — é mais eficiente do que as restantes.

No caso particular em que não se usa o *coprocessor*, verifica-se que a obtenção de linhas com uma coluna pré-especificada tem um maior custo. Isto deve-se ao uso de um filtro implícito usado para selecionar a coluna pretendida e, por isso, proporciona uma maior latência. Na última configuração, os valores chegam a um extremo que se traduz em aproximadamente 87 segundos para concluir a contagem de uma família de colunas no lado do cliente, isto é, um acréscimo a rondar os 267% face à configuração 1. Contrariamente, verifica-se que, no caso em que se usa o *coprocessor*, a contagem através da obtenção da coluna de famílias é mais rápida do que através da obtenção de uma coluna por cada linha. Relembre que os *coprocessors* são executados nos *RegionServers*, ou seja, onde se encontram os ficheiros *HFile* que correspondem a famílias de colunas, o que contribui para uma extrema localidade e execução sem qualquer recurso à rede.

3.3.3 Conclusão

O uso da rede traz custos elevados, no que toca a estas operações. Uma contagem de linhas que necessita de uma pesquisa em toda a tabela tem um custo elevado que, quando combinada com a latência de rede, reflete os resultados com a discrepância visível na configuração 3 — obtenção de um

identificador de linha com uma determinada família de colunas.

Sem o uso de *coprocessor*, toda a família de colunas foi transferida para o cliente, incorrendo num acréscimo de 66 segundos na latência. Verifique que, com *coprocessor*, obter toda a família de colunas se tornou mais compensador do que apenas uma coluna específica.

Conclui-se então que, no caso em que se adiciona uma ou mais colunas específicas na procura e se pretende usar um *coprocessor*, se deve proceder à obtenção da família de colunas completa. Deve salientar-se que este teste foi apenas executado com um *RegionServer*, pelo que não se deve concluir que mais *RegionServers*, e possivelmente criação de regiões da família de colunas pretendida, inflacionariam a latência, uma vez que os pedidos são enviados e executados em paralelo por todas estas. No entanto, seria expectável um aumento acentuado da latência na configuração em que não se usam *coprocessors*, uma vez que a procura é sequencial. Neste último caso, é mais compensador a obtenção da coluna.

3.4 Tamanho dos qualificadores

Como visto no Capítulo 2, os qualificadores fazem parte da identificação da coluna, em conjunto com o nome da família de colunas. O tamanho influencia a quantidade de informação enviada para o cliente e é com este teste que se avalia a importância da variação desta porção dos dados.

3.4.1 Descrição do cenário

Neste teste são usados quatro nós, *N1*, *NC1*, *NC2* e *NC3*. *N1* representa a máquina-cliente. No nó *NC1* encontra-se uma instância *Master* e *Apache Zookeeper*, juntamente com um *NameNode*. Em cada um dos nós *NC2* e *NC3*

corre um *RegionServer*, com 4GB de *heap*, com o seu respetivo *DataNode*.

Neste cenário, usa-se o HBase na versão `v1.0.0RC0`, com HDFS na versão `v2.6.0`.

O YCSB cria uma única tabela, com apenas uma família de colunas e povoa a tabela com um milhão de linhas, com dez qualificadores cada uma. O valor de cada chave tem 100 *bytes* (por omissão) de tamanho e são executadas 50 mil operações `scan` de uma coluna aleatória, com probabilidade uniforme. O procedimento é executado para cada uma das seguintes configurações devidamente numeradas:

1. Qualificador com 5 *bytes*;
2. Qualificador com 120 *bytes*.

3.4.2 Resultados

Após a execução do teste nas duas configurações, obteve-se os resultados enunciados na Tabela 3.4.

	Débito (ops/s)			**Latência (ms)**		
	64 Clientes	**128 Clientes**	**256 Clientes**	**64 Clientes**	**128 Clientes**	**256 Clientes**
1	1535.5	1554.8	1557.7	41.5	81.82	162.9
2	790.1	796.0	793.7	80.7	159.9	319.5

Tabela 3.4: Resultados relativos ao custo associado ao tamanho dos qualificadores

A partir dos resultados, observa-se que existe uma discrepância entre as configurações. A configuração 2, que contém o qualificador com 120 *bytes*, é claramente pior em termos de desempenho do que a configuração 1. Note-se

que a latência, em ambas as configurações, duplica aquando da duplicação do número de clientes, embora o número de operações se mantenha o mesmo.

3.4.3 Conclusão

Ao observar os resultados, percebe-se que existe um comportamento linear por parte da latência. Nos detalhes do estado da rede, verificou-se que ambas as configurações atingiram o limite de rede, pelo que se torna essencial estudar as causas provenientes do HBase que levaram a este acontecimento. Ainda assim, percebe-se o aumento do tamanho dos qualificadores tem um impacto direto no número de operações executadas por unidade de tempo.

3.5 Quantidade de famílias de colunas

Cada família de colunas está representada num ficheiro *HFile*. A existência de múltiplas famílias de colunas obriga a um maior esforço por parte do HBase e é efetivamente esse esforço que será analisado neste teste.

3.5.1 Descrição do cenário

Os nós seguem a mesma configuração do teste anterior, ou seja, N1 é o cliente, onde é executado o YCSB. No nó *NC1* encontra-se uma instância *Master* e *Apache Zookeeper*, juntamente com um *NameNode*. Em cada um dos nós *NC2* e *NC3* corre um *RegionServer*, com 4GB de *heap*, com o seu respetivo *DataNode*.

Neste cenário, usa-se o HBase na versão `v1.0.0RC0`, com HDFS na versão `v2.6.0`.

O YCSB cria uma única tabela e povoa a tabela com um milhão de linhas, com dez qualificadores cada uma. Neste cenário pretende-se avaliar o

seguinte comportamento em cada uma das situações:

1. Obter linhas com um qualificador específico, inserido em dez famílias de colunas com um qualificador;
2. Obter linhas com um qualificador específico, inserido numa família de colunas com dez qualificadores;
3. Obter todas as linhas de dez famílias de colunas com um qualificador;
4. Obter todas as linhas de uma família de colunas com dez qualificadores.

A estratégia utilizada para a organização dos dados nas configurações 1 e 3 foi converter os qualificadores que são inseridos pelo YCSB em famílias de colunas. Desta forma, todos os pares chave-valor são inseridos com um qualificador estático.

3.5.2 Resultados

Os resultados que foram obtidos após a conclusão dos testes, estão representados na Tabela 3.5.

	Débito (ops/s)			**Latência (ms)**		
	64 Clientes	**128 Clientes**	**256 Clientes**	**64 Clientes**	**128 Clientes**	**256 Clientes**
1	8613.3	9072.2	9069.2	7.41	14.06	28.10
2	7518.1	8269.9	8753.0	8.49	15.43	29.14
3	1488.7	1488.5	1524.7	42.79	85.46	166.41
4	1531.0	1538.2	1544.0	41.64	82.69	164.35

Tabela 3.5: Resultados obtidos nas operações sobre variadas famílias de colunas

Verifica-se uma maior latência nas duas últimas configurações, uma vez que existe uma maior quantidade de dados transmitidas entre o *RegionServer* e o cliente.

A diferença de débito entre as configurações 1 e 2 indica que a procura de um qualificador dentro de uma família de colunas com vários incluídos é mais dispendioso do que apenas um qualificador por família de colunas. Note que, na configuração 1, os qualificadores do YCSB foram dispersos por famílias de colunas, o que leva a um menor número de chaves por cada uma, reduzindo o conjunto alvo de pesquisa.

Nas configurações 2 e 3, torna-se mais rápida a obtenção de apenas uma família de colunas do que várias em separado. No entanto, a diferença varia apenas entre 1% e 3%.

3.5.3 Conclusão

De acordo com os resultados obtidos, o HBase tem um melhor desempenho com menos pares chave-valor envolvidos em cada família de colunas, no caso em que se pretende um qualificador específico, mesmo que isso implique a existência de mais ficheiros *HFile*, como na configuração 1.

Nas duas últimas configurações, entende-se que, quando se pretende todos os pares chave-valor, quer estejam ou não separados pelos diversos *HFile*, a quantidade de dados transmitida é aproximadamente a mesma. Aqui, o peso de haver vários *HFile* pode levar a uma quebra de desempenho.

3.6 Esquema da tabela

O esquema da tabela pode ser desenhado de muitas formas distintas. As formas mais conhecidas são *tall-narrow*, que consiste na dispersão dos qua-

lificadores em termos de altura, e *flat-wide*, que consiste no agrupamento de qualificadores [2], de modo a conter mais informação numa só linha, tornando-a mais pequena mas mais longa. Neste teste pretende-se avaliar a capacidade do sistema no que toca à resposta à concatenação de qualificadores e dos seus valores num só qualificador, seguindo o modelo *flat-wide*.

3.6.1 Descrição do cenário

Os nós seguem a mesma configuração do teste anterior, ou seja, N1 é o nó-cliente, onde é executado o YCSB. No nó *NC1* encontra-se uma instância *Master* e *Apache Zookeeper*, juntamente com um *NameNode*. Em cada um dos nós *NC2* e *NC3* corre um *RegionServer*, desta vez com 2GB de *heap*, com o seu respetivo *DataNode*, uma vez que as linhas que serão inseridas são em menor quantidade.

Neste cenário, usa-se o HBase na versão `v1.0.0RC0`, com HDFS na versão `v2.6.0`.

O YCSB cria uma única tabela e povoa a tabela com cem mill linhas, com dez qualificadores cada uma. São executadas 50 mil operações `scan` para um qualificador específico com as seguintes configurações:

- Todos os qualificadores e respetivos valores estão concatenados por separadores, como por exemplo `field1_value1-field2_value2`...;
- Cada qualificador tem o seu valor (comportamento por omissão do YCSB).

Note que, para encontrar um qualificador específico na configuração 1, recorreu-se ao filtro `QualifierFilter`.

3.6.2 Resultados

No fim de cada teste, obteve-se os resultados enunciados na Tabela 3.6.

	Débito (ops/s)			**Latência (ms)**		
	64 Clientes	**128 Clientes**	**256 Clientes**	**64 Clientes**	**128 Clientes**	**256 Clientes**
1	114.7	114.6	114.0	540.3	1068.8	2128.7
2	4122.7	4359.9	4321.9	15.2	28.8	58.7

Tabela 3.6: Resultados relativos ao custo associado ao tamanho dos qualificadores

É possível verificar que a concatenação de todos os qualificadores e valores numa só célula torna o sistema incapaz de escalar de uma forma aceitável. Isto deve-se ao facto de ser necessário usar um filtro `QualifierFilter` que recorre a uma expressão regular para encontrar o qualificador pretendido. A configuração 2, que se baseia na organização normal dos qualificadores por parte do YCSB, obtém um débito muito superior comparativamente com a anterior.

3.6.3 Conclusão

A organização dos qualificadores é um fator importante no desenho do esquema a adotar nas tabelas. Neste caso, verificou-se que a concatenação dos qualificadores com os respetivos valores reduz o número de pares chave-valor mas também provoca um custo extraordinário na procura de um determinado padrão. Desta forma, desenhar um esquema seguindo este modelo que requer a procura de um determinado conjunto de dados pode ter consequências devastadoras no desempenho. É importante referir que neste teste, especificamente na configuração 1, é possível obter melhores resultados. Tal meta

pode ser atingida estudando uma forma de acrescentar meta-informação sobre a organização contida no qualificador.

3.7 Sumário

Nesta secção analisou-se o HBase aplicado a vários cenários, de modo a compreender a existência de pontos de contenção que possam ser estudados e otimizados.

Após a análise de desempenho, verificou-se que o sistema atinge facilmente o limite da rede, impedindo a formalização de conclusões em alguns dos testes realizados.

Nos testes passíveis de conclusão, verificou-se que o uso de filtros sobre qualificadores agrupados num só qualificador, tal como acontece no modelo *flat-wide*, tem um custo elevado. Inclusive, o tamanho de cada qualificador não só implica uma perda de desempenho como também evidencia o problema de rede. Constatou-se ainda que a procura de qualificadores específicos é mais eficiente com menos pares chave-valor, mesmo no caso em que a procura tenha de ser efetuada em famílias de colunas e, portanto, diferentes ficheiros. No entanto, note que tornar-se ineficiente no caso de ter de lidar com um número elevado de ficheiros distintos.

Sendo o consumo de rede um problema frequente, no próximo capítulo levar-se-á a cabo o estudo e conceção de uma solução que diminua a quantidade de dados transmitidos pela rede para o caso específico em que a estrutura dos dados é regular.

Capítulo 4

Otimização da leitura de tabelas

Tendo em conta a largura de banda da rede como principal limitação, propõe-se a otimização da leitura de tabelas com uma alternativa à operação *Scan* do HBase, denominada *Prepared Scan*.

Prepared Scan é uma operação de consulta de dados, baseada no `scan` nativo, que estende o protocolo de comunicação entre cliente e servidor do HBase por meio da implementação de um *coprocessor* do tipo *endpoint*, ou seja, sem alteração do código-fonte do sistema. Esta operação visa otimizar a consulta de dados que seguem uma estrutura regular. A operação é dividida em três fases:

Preparação Envio do conhecimento existente por parte da aplicação para os *RegionServers* e preparação do ambiente para a execução da operação;

Execução Obtenção de resultados previamente tratados de acordo com o conhecimento sobre os dados;

Conclusão Finalização da operação e libertação de todos os recursos usados durante as execuções desta.

Neste capítulo, é primeiramente enunciado a interface disponível para o cliente e como, através desta, o conhecimento é transmitido pela aplicação. De seguida, o protocolo usado nesta operação em comparação com o da operação `scan` nativa e, por fim, o funcionamento do *coprocessor*.

4.1 Interface com a aplicação

O *Prepared Scan* é inicializado com a especificação de três parâmetros fundamentais, visíveis na Figura 4.1. O primeiro parâmetro `configuration` diz respeito à configuração do HBase que será utilizada durante a execução. O parâmetro `table` enuncia em que tabela se irá focar, uma vez que a política das outras operações é a mesma, isto é, são aplicáveis a apenas uma tabela em simultâneo. O terceiro e último parâmetro, `scan`, é a instância da operação `scan` nativa que se pretende executar por forma a tirar proveito do conhecimento existe neste e, para tal, a instância desta deve conter as colunas que são pretendidas.

```
PreparedScan( Configuration config, TableName table, Scan scan )
```

Figura 4.1: Instanciação do *Prepared Scan*

Uma vez inicializada a operação, inicia-se então o processo que se divide em três fases — preparação, execução e conclusão — representadas pelas assinaturas dos métodos presentes na Figura 4.2. É importante realçar que os parâmetros usados na fase de execução são usados da mesma forma que o `scan` nativo, isto é, o `startRow` e `stopRow` delimitam o início (inclusive) e fim (exclusive) do intervalo de identificadores de linha onde ocorre a consulta. O parâmetro `caching` indica a quantidade de linhas que serão mantidas em *cache* no cliente.

```
PreparedScan prepare()
ResultScanner execute( byte[] startRow, byte[] stopRow [, int caching] )
void close()
```

Figura 4.2: Métodos das fases do *Prepared Scan*

Esta interface do cliente permite que sejam efetuadas múltiplas execuções com a mesma instância *Scan*, sendo alterados apenas os três parâmetros entre cada execução, o que se traduz num reaproveitamento de todas as opções definidas inicialmente, como filtros, colunas a retornar, entre outros. A Figura 4.3 demonstra um exemplo de utilização do *Prepared Scan* com as colunas presentes na Tabela 2.1. Por motivos de espaço, considere que cada *string* é convertida em *byte array* implicitamente.

Os resultados são retornados através de um *ResultScanner* e podem ser acedidos da mesma forma que a operação nativa. Note que são realizadas duas execuções com parâmetros diferentes e que, durante estas, não é necessário redefinir as colunas que se pretendem.

Desta forma, a transformação da operação nativa para esta solução torna-se simples, uma vez que a interface segue os mesmos padrões.

4.2 Protocolo

O protocolo usado no *Prepared Scan* é implementado recorrendo a *Google Protocol Buffers* (PB), uma vez que é a biblioteca de serialização de estruturas de dados definida pelo HBase para serviços protocolares.

A fase de preparação, como foi referido, consiste no envio dos parâmetros da instância *Scan*, para os *RegionServers* que, por sua vez, guardam esta informação associada ao cliente. Na Figura 4.4, é possível verificar a definição do tipo *Scan* em PB e compreender os dados que são enviados para o sistema

```
// Declare configuration and table's name
Configuration config = ...;
TableName table = ...;

Scan scan = new Scan();
scan.addColumn( "colfam1", "qual1" );
scan.addColumn( "colfam1", "qual2" );
scan.addColumn( "colfam2", "qual1" );

// Prepare
PreparedScan pScan = new PreparedScan( config, table, scan );
pScan.prepare();

// Execute one or more times
ResultScanner scanner = pScan.execute( "row1", "row2" );
for( Result res : scanner ) { ... }

ResultScanner scanner = pScan.execute( "row3", null, 10 );
for( Result res : scanner ) { ... }

// Free resources
pScan.close();
```

Figura 4.3: Exemplo de execução de um *Prepared Scan*

quando se pretende executar uma consulta, entre estes as colunas que se pretendem, no parâmetro `column`, os filtros a usar, dado por `filter`, número máximo de versões, intervalo de marcas temporais e outros. Note que todos os parâmetros são opcionais durante a serialização.

Tendo toda a informação necessária sobre o `scan` que se pretende efetuar, na fase de execução é possível reutilizar os parâmetros sem que tenham de ser enviados novamente. De todos os parâmetros, o que torna esta solução capaz de lidar com dados estruturados de uma forma mais eficiente é o esquema de colunas pretendido, dado pelo nome `column` na especificação em PB.

Após a obtenção da instância do tipo *Scan* que resulta da junção dos parâmetros recebidos na fase de preparação com os parâmetros recebidos

```
message Scan {
  repeated Column column = 1;
  repeated NameBytesPair attribute = 2;
  optional bytes start_row = 3;
  optional bytes stop_row = 4;
  optional Filter filter = 5;
  optional TimeRange time_range = 6;
  optional uint32 max_versions = 7 [default = 1];
  optional bool cache_blocks = 8 [default = true];
  optional uint32 batch_size = 9;
  optional uint64 max_result_size = 10;
  optional uint32 store_limit = 11;
  optional uint32 store_offset = 12;
  optional bool load_column_families_on_demand = 13;
  optional bool small = 14;
  optional bool reversed = 15 [default = false];
  optional Consistency consistency = 16 [default = STRONG];
  optional uint32 caching = 17;
}
```

Figura 4.4: Definição do *Scan* em PB

na fase de execução, isto é `startRow`, `stopRow` e `caching`, esta é usada para executar a operação. Uma vez obtidos, os resultados são retornados ao cliente no formato demonstrado pela Figura 4.5.

Este formato contém um campo, `Last Result`, que indica se não existem mais linhas que satisfazem as condições da operação `scan`, prevenindo a execução de uma nova chamada RPC com esse objetivo. Seguido deste, existe um conjunto de linhas e cada uma é composta por vários campos: o identificador da linha, um conjunto de números que indicam a que colunas (através do seu índice) pertencem as células enviadas. Este último permite a redução de dados redundantes relativos à informação da coluna, isto é, a combinação da família de colunas e do qualificador a que pertencem. Assim sendo, o conteúdo da célula é reduzido apenas aos campos, de acordo com a Figura 2.2, relativos à marca temporal (`Timestamp`), ao tipo da célula (`Type`

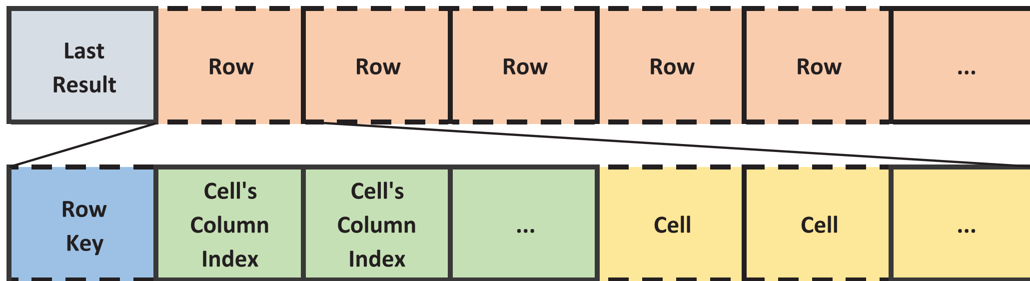


Figura 4.5: Estrutura dos resultados do *Prepared Scan*

`Byte`) e ao valor (`Value`). A definição das mensagens em PB podem ser visualizadas na Figura B.1.

4.3 Componente servidor

As chamadas RPC são realizadas a um módulo carregado no *RegionServer* que se denomina de *coprocessor endpoint*. Este é responsável, nesta operação, por atender os pedidos nas várias fases do processo, da seguinte forma:

Preparação

– Insere os parâmetros da instância *Scan*, associados ao cliente, na estrutura partilhada pelas instâncias do *coprocessor*, obtidos através do protocolo;

– Atribui índices às colunas devidamente ordenadas.

Execução

– Atualiza instância *Scan* associada ao cliente com os parâmetros recebidos através do método RPC;

– Obtém os pares chave-valor que satisfazem o *Scan* através da execução de um *InternalScanner* no *RegionServer*, em cada região, até atingir o número de linhas dado pelo parâmetro `caching` (*default*: 100);

– Por cada linha, remove o identificador de linha em todas as suas células, colocando-o apenas uma vez no início da linha no parâmetro `Row Key`. Por cada célula `Cell`, atribui o índice da coluna a que pertence e remove os dados relativos ao nome da família de colunas e qualificador;

– Constrói os resultados recorrendo a PB. Caso não tenha atingido o valor dado por `caching`, coloca `true` no campo `Last Result`;

– Retorna resultados ao cliente.

Conclusão

– Remove a informação relativa ao pedido de preparação e ao cliente.

Note que os parâmetros são guardados numa estrutura de dados partilhada entre as diversas instâncias deste *coprocessor*.

Com a implementação da solução como um *coprocessor*, não é necessário alterar qualquer código-fonte, uma vez que é uma extensão do protocolo, sendo capaz de acompanhar todas as alterações que não afetem a API do HBase.

4.4 Avaliação de desempenho

Para avaliar a solução apresentada ao longo deste documento, recorreu-se à ferramenta de *benchmarking Yahoo! Cloud Serving Benchmark* (YCSB) [22]. Esta ferramenta disponibiliza vários padrões de acesso a dados e permite a personalização de cada um.

4.4.1 Procedimento

Para avaliar esta solução, foram idealizados três cenários, sendo a Configuração 1 para avaliar o desempenho desta face a uma quantidade de dados

relativamente pequena. A Configuração 2 é o segundo cenário que tem como objetivo compreender o comportamento com uma grande quantidade de dados (MB). Por último, com a Configuração 3 pretende-se representar múltiplas linhas que não têm valores em todas as colunas. Durante a execução, foi utilizado o HBase na versão `1.0.0` e o HDFS na `2.6.0`.

Configuração 1

Os *RegionServers* e o *Master* foram alocados em nós distintos, cada um com um processador Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz, 8GB RAM e um disco 7200 RPM. Os *RegionServers*, instanciados com o *coprocessor* do *Prepared Scan*, têm 4GB de *heap* e co-existiram com as respetivas instâncias *DataNode*, assim como o *Master* com o *NameNode*.

Os testes de desempenho foram executados com vários clientes numa máquina com um processador AMD Opteron(tm) Processor 6172, 120GB RAM e um disco de 7200 RPM. A largura de banda existente entre os clientes e o sistema HBase é de 1Gbps.

Em cada teste, executaram-se um milhão de operações usando unicamente a operação `scan` no padrão de acesso a dados, com tamanho fixo de 100 linhas. Uma vez que o *Prepared Scan* é aplicado a um conjunto bem definido de colunas, o cliente YCSB foi modificado de modo a selecionar explicitamente as colunas pretendidas, neste caso todas, no *Scan*, recorrendo ao método `addColumn`. A avaliação foi realizada para 2, 4, 8, 16, 32, 48, 64, 80 e 100 clientes.

Em termos de dados, povoou-se a base de dados com um milhão de linhas, cada uma com dez qualificadores, resultando em cerca de dez milhões de pares chave-valor, cujo tamanho do valor é de 100 *bytes*.

Configuração 2

Nesta configuração, todo o sistema anteriormente descrito co-existiu, neste caso com um só *RegionServer* com 1/4 da memória RAM como *heap* e o respetivo *DataNode*, num nó com um processador AMD Opteron(tm) Processor 6172, 120GB RAM e um disco de 7200 RPM. Desta forma, é possível aumentar a quantidade de dados sem ter em conta o limite da largura de banda.

A base de dados foi povoada com um milhão de linhas, com dez qualificadores, cujo valor se fixa nos 500 *bytes*. Em cada operação de consulta, foi solicitado um conjunto de 2500 linhas. A execução ocorreu para 2, 4, 6, 8, 16 e 32 clientes.

Configuração 3

Esta configuração segue o mesmo procedimento da Configuração 1, com a exceção de que, nesta, cada linha tem apenas um valor relativo a um qualificador aleatoriamente escolhido num conjunto de dez.

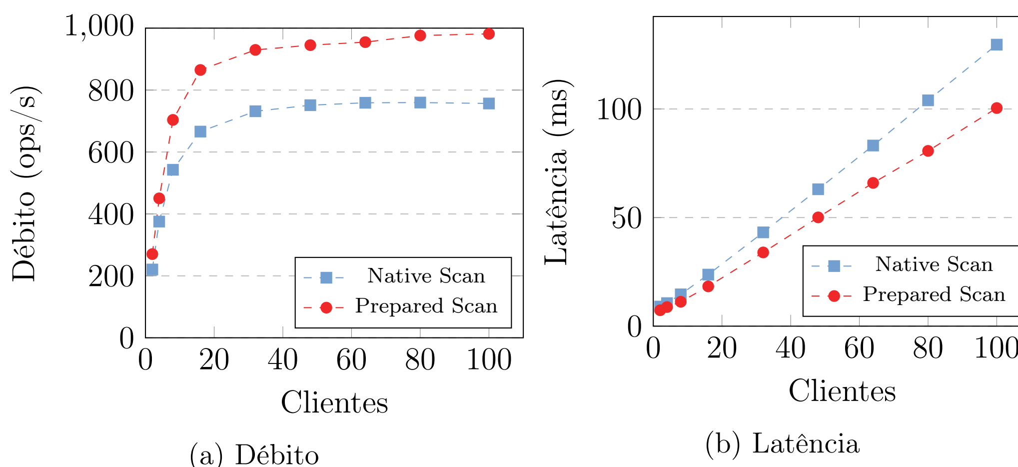
Uma vez que o *Prepared Scan* exige a computação das colunas por cada célula, o objetivo desta configuração é compreender se o bom desempenho desta operação se verifica também em dados menos estruturados.

4.4.2 Resultados

Configuração 1

Os resultados são demonstrados em três métricas: débito, visível na Figura 4.6a, latência, na Figura 4.6b e, por fim, a rede usada em cada operação, na Figura 4.7.

Focando a Figura 4.6a, é possível constatar que o débito obtido no *Pre-*



(a) Débito

(b) Latência

Figura 4.6: Desempenho do *Prepared Scan* e *Scan* na Configuração 1

pared Scan é sempre superior ao que se obtém por parte da implementação nativa, algo que se evidencia cada vez que o número de clientes é aumentado. Para 2 clientes, a diferença situa-se nas 50 operações por segundo, o que equivale a um aumento de cerca 23%, relativamente ao nativo. A diferença no número de operações, nos 80 clientes, apresenta uma diferença de 216 operações por segundo, o equivalente a um acréscimo de cerca 29% no débito. É entre os 80 e os 100 clientes que o *Scan* inicia a sua ligeira quebra no débito, passando de 759 para 756 operações por segundo. Já o débito no *Prepared Scan* se mantém num ligeiro crescimento, atingindo as 981 operações por segundo para 100 clientes.

Em termos de latência, na Figura 4.6b, verifica-se um afastamento das linhas que representam as duas operações, dando vantagem ao *Prepared Scan*, uma vez que a sua latência aumenta de forma menos acentuada que o nativo, à medida que são adicionados mais clientes. No auge do nativo, isto é, nos 80 clientes, a diferença entre a latência das duas operações é de 22,41%. Este valor aumenta nos 100 clientes, atingindo os 22,53%.

Um aspeto que era expectável está relacionado com a rede. Previa-se uma

redução de dados que se propagam pela mesma, uma vez que a redundância de dados como as colunas e os identificadores de linha é eliminada com o *Prepared Scan*. A Figura 4.7 representa a métrica de rede utilizada em cada operação e verifica-se efetivamente uma redução de aproximadamente 20% com esta solução.

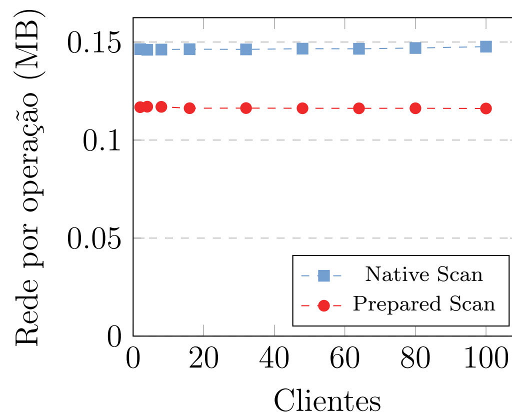
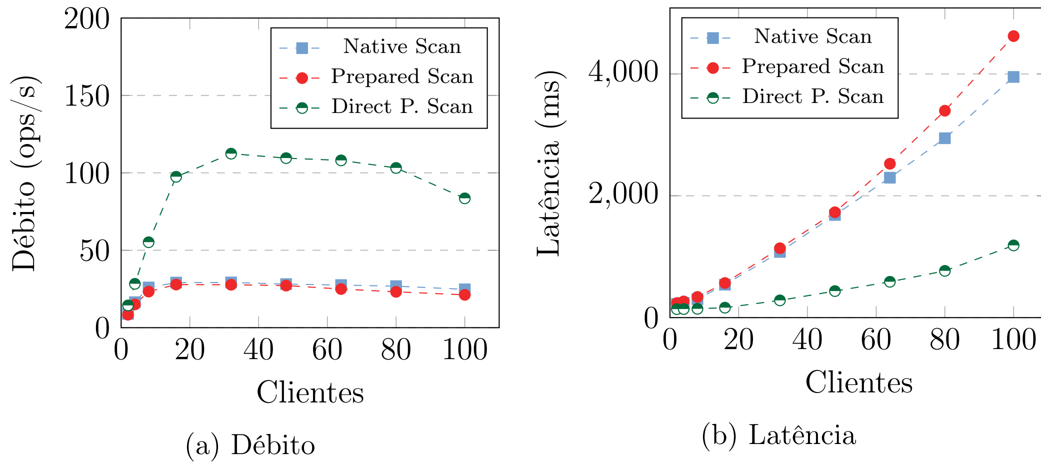


Figura 4.7: Utilização da rede por operação

Configuração 2

Os resultados do comportamento do *Prepared Scan* face a uma grande quantidade de dados entre o sistema e o cliente são demonstrados na Figura 4.8, sob a forma de débito, na Figura 4.8a, e latência, na Figura 4.8b. Note que a medição de rede não é relevante, uma vez que tanto os clientes como o HBase foram executados no mesmo nó e, por acréscimo, já se verificou a redução no consumo de rede na configuração anterior.

De acordo com a Figura 4.8a, verifica-se que o *Prepared Scan*, comparativamente com o nativo, se torna ineficiente nesta configuração, mantendo o seu débito abaixo do pretendido. Como também é possível verificar na Figura 4.8b, a latência aumenta de forma mais acentuada do que a operação nativa.



(a) Débito

(b) Latência

Figura 4.8: Desempenho do *Prepared Scan* e *Scan* na Configuração 2

Estes resultados refletem o custo da utilização de PB que, como é conhecido pela comunidade do HBase[1], não lida com a serialização de *megabytes* de dados de forma eficiente. De facto, é isso que se constata.

A operação nativa não recorre a PB para enviar as células para o cliente, usando a conexão direta com o cliente. Apenas os meta-dados são realmente serializados com PB.

Para efetivamente compreender o impacto do uso de PB, procedeu-se a uma implementação experimental, *Direct Prepared Scan* (DPS). Esta implementação, em alternativa a enviar as células para o cliente com PB, transmite-as por conexão direta entre o cliente e os *RegionServers*, seguindo o exemplo da operação nativa. A definição das mensagens protocolares desta implementação podem ser visualizadas na Figura B.2.

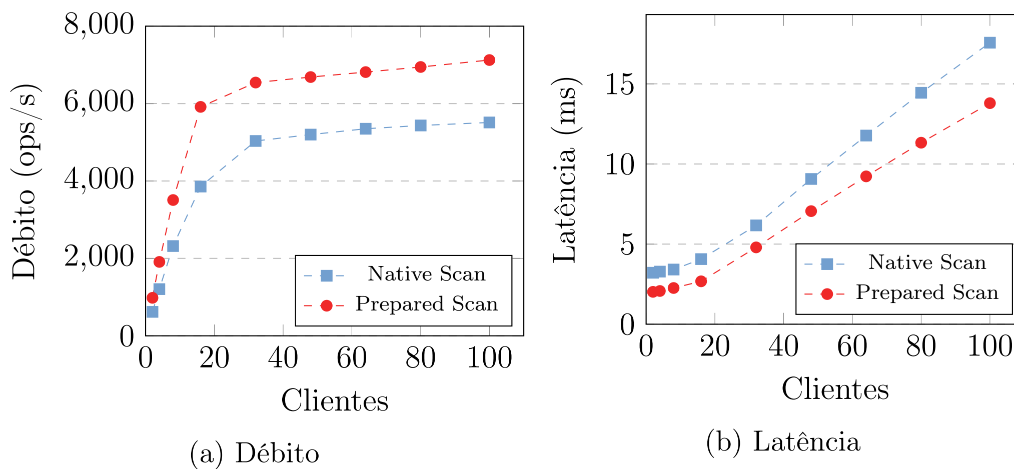
Como resultado desta implementação, em relação ao `scan` nativo, verifica-se um débito de aproximadamente 384% face ao do nativo, dos 32 aos 80 clientes. Em termos de latência, obteve-se uma melhoria de aproximadamente 75%, também dos 32 a 80 clientes. Note que, nos 100 clientes, o DPS inicia a

[1] `https://issues.apache.org/jira/browse/HBASE-7233`

perda de desempenho devido à saturação no CPU, levando a uma quebra de débito e aumento de latência mais acentuada.

Configuração 3

Os resultados obtidos nesta configuração encontram-se na Figura 4.9, sob a forma de débito, na Figura 4.9a, e latência, na Figura 4.9b.



(a) Débito

(b) Latência

Figura 4.9: Desempenho do *Prepared Scan* e *Scan* na Configuração 3

Constata-se que, na Figura 4.9a, o *Prepared Scan* tem um melhor desempenho do que a operação nativa, ao longo das execuções. Nos 2 clientes, obtém-se um débito de 981 operações por segundo, ou seja, um acréscimo de aproximadamente 58%. Nos 100 clientes, a diferença ronda os 29%, com 7124 operações executadas por segundo, face às 5512 operações por segundo do nativo. Note que, nos 100 clientes, ambas as operações se encontram numa tendência crescente.

Na latência, na Figura 4.9b é possível verificar também um melhor comportamento nesta solução ao longo das execuções, que se traduz num crescimento menos acentuado. A redução na latência atinge, nos 100 clientes, os 21%.

4.4.3 Conclusão

Os resultados apresentados caraterizam o comportamento do *Prepared Scan* aplicado a vários cenários.

No primeiro cenário, na Configuração 1, que representa uma tabela totalmente preenchida, como acontece no contexto relacional, constatou-se que o desempenho do *Prepared Scan* é claramente superior, isto é, obteve-se um aumento significativo de 29% no débito. Tal indica que a remoção de dados redudantes implica uma redução de dados transmitidos na rede e, como tal, favorece o desempenho do sistema.

Na Configuração 2, representou-se um caso de uso em que a operação requer uma larga quantidade de informação, atingindo vários *megabytes*. O comportamento do *Prepared Scan* foi severamente prejudicado, chegando a ser pior do que o `scan` nativo em todas as execuções. Isto deve-se à biblioteca de serialização usada que possui algumas limitações já conhecidas pela comunidade e que são perfeitamente visíveis nos resultados. Tal levou a uma implementação experimental, o DPS, que mostrou ser melhor, tornando evidente o impacto de serializar *megabytes* de dados com PB. Esta última implementação atingiu um débito quatro vezes superior ao conseguido com o nativo. Note que, apesar de o desempenho ser visivelmente melhor, esta foi uma implementação experimental que recorreu a métodos que não se encontram nos padrões do *coprocessor endpoint*, pelo que é necessário um esforço adicional para lidar com tolerância a faltas, segurança e outros problemas de sistemas distribuídos.

Por último, na Configuração 3, simulou-se o equivalente a uma tabela com um conjunto de valores nulos no contexto relacional. O comportamento do *Prepared Scan* nesta situação, mesmo com a computação necessária para indexar a única célula e remover os dados redundantes desta, foi também

melhor do que o nativo em aproximadamente 29%.

Após esta avaliação de desempenho, conclui-se que, com a redução de dados na rede, a consulta de dados pode tornar-se mais eficiente e que a biblioteca de serialização tem um impaco forte quando a quantidade destes atinge dimensões que a tornam ineficiente. No caso em que, através de métodos que não se enquadram nos padrões das ferramentas que permitem a extensão do protocolo, não se usa PB para serializar a informação, é possível obter uma solução, como o DPS, que supera a operação nativa em todos estes cenários.

4.5 Sumário

Nesta secção, apresentou-se uma otimização da operação `scan`, *Prepared Scan*, para dados que são estruturados de uma forma regular e, portanto, conhecida pela aplicação. Esta solução ajusta-se definitivamente a aplicações que migram do contexto relacional para o não-relacional, usando ou não mecanismos que fornecem uma linguagem de interrogação semelhante a SQL ou que têm uma estrutura de dados conhecida à partida.

A interface com a aplicação conta com essencialmente três métodos que refletem as fases necessárias para a realização da operação. A forma como os dados são tratados após a receção dos resultados não necessita de ser alterada aquando da adoção desta solução, tornando a adoção desta operação acessível.

A operação faz uso de um protocolo que favorece a remoção de dados redundantes e tal é possível através da implementação de um *coprocessor endpoint* como foi visto no longo da secção.

Constatou-se que, em termos de desempenho, esta operação consegue um

débito maior do que a operação nativa, uma vez que reduz o consumo de rede por operação. No entanto, para resultados extensos, é necessário sair dos padrões de extensão de protocolo fornecidos pelo HBase, aumentando a complexidade da implementação da solução, como foi visto com o *Direct Prepared Scan*.

Capítulo 5

Conclusões

Com os sistemas NoSQL a apresentarem características de escalabilidade distintas dos tradicionais SGBDR, muitas aplicações iniciaram o seu processo de migração de dados do contexto relacional para o contexto não-relacional. Desde então, estes sistemas têm sido alvo de análise de desempenho e otimização.

O processo de migração, que requeria uma transformação completa tanto dos dados como da linguagem de interação, tornou-se menos dispendioso devido à disponibilização de mecanismos de tradução da linguagem de interrogações SQL, comum nos sistemas relacionais, para operações nativas dos sistemas NoSQL.

Embora os sistemas NoSQL permitam também uma grande flexibilidade na estrutura de dados, nem sempre é o objetivo principal da migração das aplicações, principalmente daquelas que adotam os mecanismos que fornecem uma linguagem de interrogação semelhante a SQL sobre estes sistemas. Apenas têm o intuito de atingir níveis de escalabilidade superiores sem afetar de maneira nenhuma a forma como a aplicação lida com o sistema de gestão e processamento de dados. No entanto, independentemente disso, não existe a

opção de descartar esta flexibilidade conseguida através da desnormalização dos dados. A duplicação de dados característica da desnormalização resulta num processamento e envio de informação extra, uma vez que o sistema não é capaz de prever qual a estrutura dos dados que armazena.

Neste documento, apresentou-se uma solução capaz de otimizar a consulta de dados sem alteração do código-fonte do sistema, o *Prepared Scan*. Esta tem como princípio usar a informação, mais propriamente a estrutura dos dados, fornecida pela aplicação para reorganizar os dados pretendidos por forma a não enviar informação já conhecida pela aplicação. O procedimento descrito permitiu a redução do consumo de rede no envio dos resultados por cada operação, obtendo-se assim um desempenho superior em cerca de 29%.

Desta forma, as aplicações que mantêm uma estrutura de dados bem definida podem usufruir de um melhor desempenho do que as aplicações com uma estrutura dinâmica.

5.1 Trabalho Futuro

Mesmo com a obtenção de um melhor desempenho reduzindo a quantidade de informação transmitida pela rede, é possível traçar uma linha de investigação interessante de seguir no mesmo contexto deste trabalho.

Primeiramente, de acordo com os testes realizados, denotou-se uma quebra de desempenho quando se recorreu a um filtro sobre um conjunto de qualificadores aglomerado num só, de modo a encontrar um valor de um qualificador pretendido. Uma alternativa que pode melhorar o acesso a qualificadores é a adição de meta-informação sobre esses dados aglomerados, promovendo um acesso mais rápido sem recorrer a filtros. Tal implicaria a averiguação da viabilidade de realizar implementações experimentais no

HBase.

Seguidamente, como resultado da análise de desempenho do número de famílais de colunas, seria interessante encontrar um balanço entre o número de famílias de colunas e o número de pares chave-valor existentes em cada família de colunas, isto é, encontrar o ponto em que se torna mais compensador ter mais famílias de colunas com menos pares chave-valor do que o inverso, para aplicações que recorrem a consultas de qualificadores específicos.

Por fim, de acordo com o *Prepared Scan* que está idealizado apenas para dados que são previamente conhecidos, pode ser possível obter um maior desempenho num contexto mais geral. Isto é, recorrendo a alterações no código-fonte do HBase, seria interessante analisar os efeitos da computação do esquema de dados resultante dos dados a ser retornados ao cliente, enviando as células sem qualquer informação sobre as colunas, estando esta contida na meta-informação. De um modo grosseiro, seria realizar o processo de tratamento de dados do *Prepared Scan* em cada execução da operação `scan`. Desta forma, tal como o *Prepared Scan*, a quantidade de dados enviados seria reduzida e a reconstrução de células seria transparente para o cliente.

Uma vez analisados estes pontos, então seria necessário voltar a executar os testes que se encontram limitados pela rede, de modo a perceber se novos pontos de contenção se podem sobressair.

Bibliografia

[1] R. Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[2] L. George. *HBase: The Definitive Guide*. 2011.

[3] J. Pokorny. NoSQL Databases: A Step to Database Scalability in Web Environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[4] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010.

[5] D. Salmen. Cloud Data Structure Diagramming Techniques and Design Patterns. `https://www.data-tactics-corp.com/index.php/component/jdownloads/finish/22-white-papers/68-cloud-data-structure-diagramming`, 2009.

[6] C. Li. Transforming relational database into HBase: A case study. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pages 683–687, July 2010.

[7] A. Schram and K. Anderson. MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 191–202, New York, NY, USA, 2012. ACM.

[8] M. Mior. Automated Schema Design for NoSQL Databases. In *Proceedings of the 2014 SIGMOD PhD Symposium*, SIGMOD'14 PhD Symposium, pages 41–45, New York, NY, USA, 2014. ACM.

[9] J. Rith, P. Lehmayr, and K. Meyer-Wegener. Speaking in Tongues: SQL Access to NoSQL Systems. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 855–857, New York, NY, USA, 2014. ACM.

[10] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira. An Effective Scalable SQL Engine for NoSQL Databases. In Jim Dowling and François Taïani, editors, *Distributed Applications and Interoperable Systems*, volume 7891 of *Lecture Notes in Computer Science*, pages 155–168. Springer Berlin Heidelberg, 2013.

[11] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[12] A. Carniel, A. de Aguiar Sa, V. Brisighello, M. Ribeiro, R. Bueno, R. Ciferri, and C. de Aguiar Ciferri. Query processing over data warehouse using relational databases and NoSQL. In *Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En*, pages 1–9, Oct 2012.

[13] Google BigQuery Fully Managed Big Data Analytics Service. https://cloud.google.com/bigquery/.

[14] Apache Hive. http://hive.apache.org.

[15] Apache Phoenix. http://phoenix.apache.org.

[16] L. Lin, V. Lychagina, W. Liu, Y. Kwon, S. Mittal, and M. Wong. Tenzing: A SQL Implementation On The MapReduce Framework. 2011.

[17] Apache HBase. http://hbase.apache.org.

[18] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[19] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[21] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[22] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[23] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[24] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[25] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM.

[26] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 199–212, Berkeley, CA, USA, 2014. USENIX Association.

[27] A. Awasthi, A. Nandini, A. Bhattacharya, and P. Sehgal. Hybrid HBase: Leveraging Flash SSDs to Improve Cost Per Throughput of HBase. In *Proceedings of the 18th International Conference on Management of Data*, COMAD '12, pages 68–79, Mumbai, India, India, 2012. Computer Society of India.

[28] L. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically Scaling Applications in the Cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52, January 2011.

[29] I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas. TIRAMOLA: Elastic Nosql Provisioning Through a Cloud Management Platform. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 725–728, New York, NY, USA, 2012. ACM.

[30] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça. MeT: Workload Aware Elasticity for NoSQL. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 183–196, New York, NY, USA, 2013. ACM.

[31] E. Kassela, C. Boumpouka, I. Konstantinou, and N. Koziris. Automated workload-aware elasticity of NoSQL clusters in the cloud. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 195–200, Oct 2014.

[32] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 774–785, Washington, DC, USA, 2012. IEEE Computer Society.

[33] N. Islam, M. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. Panda. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 35. IEEE Computer Society Press, 2012.

Apêndice A

Avaliação de desempenho

A.1 Configuração 1

Tabela A.1: Resultados do débito (ops/s) na Configuração 1

Clientes	**Scan**	**Prepared Scan**
2	220.51	270.27
4	375.27	450.33
8	542.54	703.65
16	665.88	864.85
32	731.56	929.67
48	751.13	945.03
64	758.93	954.8
80	759.65	976.1
100	756.62	981.56

Tabela A.2: Resultados da latência (ms) na Configuração 1

Clientes	**Scan**	**Prepared Scan**
2	9.01	7.33
4	10.6	8.79
8	14.64	11.23
16	23.73	18.32
32	43.18	33.94
48	63.03	50.08
64	83.16	65.95
80	103.97	80.69
100	129.6	100.42

A.2 Configuração 2

Tabela A.3: Resultados do débito (ops/s) na Configuração 2

Clientes	**Scan**	**Prepared Scan**	**Direct P. Scan**
2	8.92	8.37	14.37
4	16.45	15.01	28.32
8	25.97	23.31	55.18
16	29.12	27.85	97.46
32	29.21	27.72	112.39
48	28.14	27.18	109.51
64	27.52	24.96	108.05
80	26.82	23.16	103.17
100	24.77	21.21	83.53

Tabela A.4: Resultados da latência (ms) na Configuração 2

Clientes	Scan	Prepared Scan	Direct P. Scan
2	223.45	238.42	138.91
4	241.41	265.42	141.02
8	303.86	339.48	144.54
16	541.06	567.81	163.17
32	1,078.04	1,138.98	282.79
48	1,685.08	1,729.62	435.5
64	2,296.76	2,525.39	588.15
80	2,944.75	3,396.45	767.28
100	3,950.97	4,621.57	1,185.26

A.3 Configuração 3

Tabela A.5: Resultados do débito (ops/s) na Configuração 3

Clientes	Scan	Prepared Scan
2	618.63	981.7
4	1,206.23	1,909.41
8	2,318.93	3,508.31
16	3,856.52	5,914.8
32	5,033.72	6,543.38
48	5,198.69	6,685.2
64	5,349.62	6,813.65
80	5,434.63	6,945.52
100	5,512.05	7,124.98

Tabela A.6: Resultados da latência (ms) na Configuração 3

Clientes	**Scan**	**Prepared Scan**
2	3.21	2.02
4	3.28	2.07
8	3.41	2.26
16	4.06	2.68
32	6.17	4.79
48	9.06	7.06
64	11.77	9.23
80	14.44	11.33
100	17.57	13.8

Apêndice B

Mensagens protocolares

```
option java_generic_services = true;
option java_generate_equals_and_hash = true;
option optimize_for = SPEED;

import "Client.proto";
import "Cell.proto";

message Row {
    optional bytes row_key = 1;
    repeated int32 cell_position_in_schema = 2;
    repeated Cell cell = 3;
}
message ResultSet {
    repeated Row row = 1;
    optional bool end_of_region = 2;
}
message PrepareScanRequest {
    required int64 client_id = 1;
    required Scan scan = 2;
}
message PrepareScanResponse {
}
message ResultSetRequest {
    required int64 client_id = 1;
    optional bytes startRow = 2;
    optional bytes stopRow = 3;
    optional int32 caching = 4;
}
message ClosePreparedScanRequest {
    required int64 client_id = 1;
}
message ClosePreparedScanResponse {
}
service PreparedScanService {
    rpc prepare (PrepareScanRequest) returns (PrepareScanResponse);
    rpc execute (ResultSetRequest) returns (ResultSet);
    rpc close (ClosePreparedScanRequest) returns (ClosePreparedScanResponse);
}
```

Figura B.1: Definição de mensagens protocolares do *Prepared Scan*

```
option java_generic_services = true;
option java_generate_equals_and_hash = true;
option optimize_for = SPEED;

import "HBase.proto";
import "Client.proto";
import "Cell.proto";

message RowMetaData {
    optional bytes row_key = 1;
    optional uint32 cells_count = 2;
    repeated uint32 cells_position_in_schema = 3;
}
message ResultSet {
    repeated RowMetaData rows_meta_data = 1;
    optional bool more_results = 2;
}
message PrepareScanRequest {
    required uint64 client_id = 1;
    required Scan scan = 2;
}
message PrepareScanResponse {
    optional ServerName server_info = 1;
}
message ResultSetRequest {
    required uint64 client_id = 1;
    optional bytes startRow = 2;
    optional bytes stopRow = 3;
    optional uint32 caching = 4;
}
message ClosePreparedScanRequest {
    required uint64 client_id = 1;
}
message ClosePreparedScanResponse {
}
service PreparedScanService {
    rpc prepare (PrepareScanRequest) returns (PrepareScanResponse);
    rpc execute (ResultSetRequest) returns (ResultSet);
    rpc close (ClosePreparedScanRequest) returns (ClosePreparedScanResponse);
}
```

Figura B.2: Definição de mensagens protocolares do *Direct Prepared Scan*