Otimização do *HBase* para dados estruturados

Francisco Neves, José Pereira, Ricardo Vilaça e Rui Oliveira

INESC TEC & Universidade do Minho
francisco.t.neves@inesctec.pt, {jop,rmvilaca,rco}@di.uminho.pt

Resumo Os sistemas *NoSQL* escalam melhor que os tradicionais sistemas relacionais, motivando 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*. Neste documento, analisa-se esse custo no *Apache HBase* e apresenta-se o *Prepared Scan*, uma operação adicional de consulta que visa tirar partido do conhecimento da estrutura de dados por parte da aplicação, diminuindo assim o custo associado à consulta de dados estruturados. Recorrendo à ferramenta de *benchmarking* YCSB, verifica-se que esta solução tem um aumento no débito de aproximadamente 29%.

Keywords: NoSQL, HBase, dados estruturados, desempenho

1 Introdução

Ao longo dos últimos anos, a quantidade de informação que tem sido requisitada pelos mais diversos clientes tem crescido de forma exponencial. Este facto levou a que os tradicionais sistemas de bases de dados relacionais (RDBMS) revelassem as suas limitações. Tal acontecimento encorajou a idealização e a 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 que as bases de dados relacionais tornaram cada vez mais evidentes. 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 RDBMS devido à sua impraticabilidade e custos associados [2].

Para atingir uma solução capaz de se enquadrar no processamento analítico em larga escala, foi necessário descartar algumas das propriedades que caraterizam as bases de dados relacionais. Surgiu 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 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 [3] em múltiplas

tabelas, por forma a evitar a necessidade de agregações e operações `join`, e tornar as leituras mais rápidas. 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 RDBMS. 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. Esta migração é facilitada por tecnologias que oferecem uma linguagem de interrogação semelhante a SQL sobre os sistemas *NoSQL*, como *Apache Hive* [4], *Apache Phoenix* [5] e *Cloudera Impala* [6] sobre o *Apache HBase*. Esta linguagem é convertida pelo próprio sistema em chamadas nativas ao sistema de gestão e processamento de dados *NoSQL* [7,8,9,10]. Estas ferramentas evitam assim a maior parte do esforço necessário para a migração para o contexto não-relacional [11].

Por outro lado, ao armazenarem dados que têm uma estrutura regular, nomeadamente, tendo todas as linhas as mesmas colunas, esta migração leva a ineficiências tanto no armazenamento como na consulta. 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*, apesar de ser o mais popular e integrar algumas as maiores aplicações, como Facebook [12].

Contribuição Neste documento, é apresentado o *Prepared Scan* para o *Apache HBase*, que visa otimizar os dados retornados, quando a aplicação conhece efetivamente a estrutura dos dados que lhe está associada, na operação `scan`, sendo esta a operação de leitura mais comum e que torna mais evidente este problema.

Estrutura do documento A Secção 2 deste documento apresenta uma contextualização dos sistemas *NoSQL*, em particular do *Apache HBase*. A Secção 3 foca-se na descrição da implementação e avaliação do desempenho do *Prepared Scan* face à operação `scan`. Na Secção 4, são apresentados os resultados obtidos e, por fim, na Secção 5 as conclusões.

2 Contextualização

NoSQL engloba uma extensa diversidade de tecnologias de bases de dados desenvolvidas para responder ao crescimento do volume de dados relativamente a utilizadores, produtos e objetos, à frequência a que esta informação é acedida e, também, às necessidades de desempenho e processamento desta. Ao longo deste documento o foco é direcionado para o *Apache HBase*, visto que é uma das tecnologias mais usadas e que têm tido mais sucesso.

2.1 *Apache HBase*

Apache HBase [13] é uma base de dados não-relacional, distribuída e escalável. Inspirada na *BigTable* da *Google* [14], é idealizada como um *sorted map* multi-dimensional indexado através do tuplo *(row key, column name, timestamp)*. 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, isto é, *family:qualifier*, sendo a família de colunas um conjunto de caracteres legível e o qualificador um *byte array* 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, a unidade de escalabilidade do *Apache HBase*. Estas regiões são, entretanto, distribuídas por diversos servidores denominados por *RegionServers*.

Cada região é persistida no sistema de ficheiros *Hadoop Distributed File System* (HDFS) [15], inspirado no *Google File System* (GFS) [16], e as instâncias que os gerem são denominadas *DataNodes*. Tipicamente, estas instâncias, com o intuito de maximizar a localidade, são executadas no mesmo nó que o *RegionServer*. No entanto, não invalida a possibilidade de ser atribuída uma região que não está no mesmo nó que o *RegionServer*.

Pares chave-valor O tipo fundamental do *Apache HBase* é o par chave-valor. O formato deste par é visível na Figura 1 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.

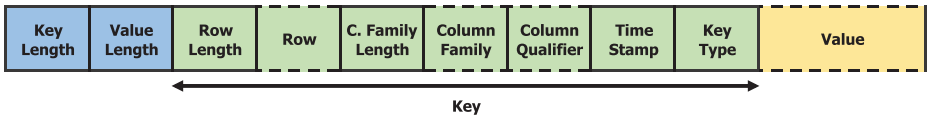


Figura 1: Formato do par chave-valor do *Apache HBase*

Como exemplo, considere que os pares chave-valor na Figura 2, 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 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

```
(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: Exemplo de pares chave-valor.

do identificador de linha com a coluna envolvente. A Tabela 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. O campo *timestamp* é uma marca temporal que pode ser representado como mais uma dimensão na tabela, uma vez que o sistema suporta múltiplas versões de células.

Tabela 1: Representação lógica de pares chave-valor do *Apache HBase*

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

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 *Apache HBase* são imediatamente visíveis em todas as leituras subsequentes, o que lhe confere 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 múltiplas células de todas as linhas ou 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 *Apache HBase* em particular oferece operações *read-modify-write* que aplicam diversas instruções de forma atómica a uma linha:

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

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

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

Funcionamento do *Scan* Aquando da execução da operação `Scan` do *Apache HBase*, todos os seus parâmetros são enviados através de uma chamada RPC ao(s) *RegionServer(s)*. Até obter todos os resultados, várias chamadas podem ser realizadas. A quantidade de chamadas depende do tamanho da *cache* e das linhas a transferir, isto é, será maior no caso em que o valor do primeiro seja muito baixo e o número de linhas a transferir elevado. Os resultados retornados são pares chave-valor transmitidos com toda a informação presente na 1.

Coprocessors O *Apache 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.

3 *Prepared Scan*

A implementação da operação `scan` no *Apache HBase* tem como opção delimitar os resultados às colunas pretendidas, recorrendo ao método `addColumn( byte[] family, byte[] qualifier )`.

O *Prepared Scan* é uma operação de consulta de dados, baseada no `scan` nativo, que estende o protocolo de comunicação entre cliente e servidor do *Apache HBase* por meio da implementação de um *coprocessor* do tipo *endpoint*, ou seja, sem alteração do código-fonte do sistema. Esta 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.

Nesta secção será primeiramente enunciada 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*.

3.1 Interface com o cliente

A operação é representada pelas assinaturas dos métodos presentes na Figura 3, que refletem as três fases: preparação, execução e conclusão. Na fase de preparação, é dado como parâmetro um objeto do tipo *Scan*, uma 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. 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. É possível estender a implementação, numa fase posterior, de modo a suportar filtros entre execuções.

```
public interface PreparedScan extends Closeable {
        public PreparedScan prepare( Scan scan );
        public ResultScanner execute( byte[] startRow, byte[] stopRow,
                int caching );
        public void close();
}
```

Figura 3: Métodos das fases do *Prepared Scan*

Esta interface do cliente permite que sejam efetuadas múltiplas execuções com a mesma instância *Scan*, sendo alterados apenas estes 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 demonstra um exemplo de utilização do *Prepared Scan* com as colunas presentes na Tabela 1. Por motivos de espaço, considere que todos os conjuntos de caracteres são convertidos implicitamente para *byte array*. Note que são realizadas duas execuções com parâmetros diferentes e que, durante estas, não é necessário redefinir as colunas que se pretendem.

3.2 Protocolo

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

A fase de preparação, como foi referido, consiste no envio dos parâmetros, através de chamadas RPC realizadas em paralelo, da instância *Scan*, à exceção do `startRow`, `stopRow` e `caching`, a os *RegionServers* que, por sua vez, guardam esta informação associada ao cliente. É necessário o envio destes dados todos os *RegionServers* devido à inexistência de conhecimento prévio da localização dos resultados da operação. Note que esta chamada RPC inicial não é efetuada na

```
// Declare configuration, table's name and prepared scan
Configuration config = ...;
TableName table = ...;
PreparedScan pScan = ...;

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

// Prepare
pScan.prepare( scan );

// 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: Exemplo de execução de um *Prepared Scan*

operação nativa. Na Figura 5, é possível verificar a definição do tipo *Scan* em PB.

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 na fase de execução, esta é usada para executar a operação. Uma vez obtidos, os resultados são devolvidos ao cliente no formato demonstrado pela Figura 6. Note que o número de chamadas RPC necessárias para obter os resultados, tal como acontece na operação nativa, depende quer do valor usado para `caching` como do número de linhas a retornar.

Este formato contém um campo, `Last Result`, que informa sobre a existência de mais linhas que satisfazem as condições da operação `scan`, prevenindo a execução de uma nova chamada RPC após os últimos resultados. Seguido deste, existe um conjunto de linhas e cada uma é composta por vários campos: o identificador da linha, um conjunto de índices que indicam a coluna a que cada célula pertence e as células. O conteúdo de cada célula é reduzido apenas aos campos, de acordo com a Figura 1, relativos à marca temporal (`Timestamp`), ao tipo da

```
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 5: Definição do *Scan* em PB

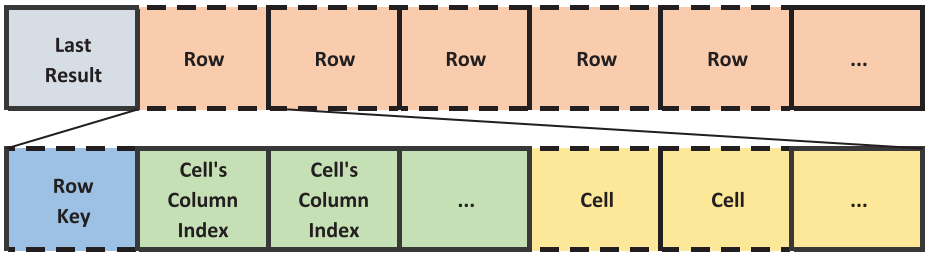


Figura 6: Estrutura dos resultados do *Prepared Scan*

célula (`Type Byte`) e ao valor (`Value`). Note que tanto o identificador de linha, uma vez que está presente no início da linha, como os detalhes das colunas, que são descartáveis devido à presença dos índices, são removidos das células.

3.3 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*, 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. Por cada célula, 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*.

4 Avaliação de desempenho

Para avaliar a solução apresentada ao longo deste documento, recorreu-se à ferramenta de *benchmarking Yahoo! Cloud Serving Benchmark* (YCSB) [17]. Esta ferramenta disponibiliza vários padrões de acesso a dados e permite a personalização de cada um.

4.1 Procedimento

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*. Neste procedimento, foram usados dois *RegionServers*.

Os testes de desempenho foram executados com vários clientes simulados 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 *Apache HBase* é de 1Gbps.

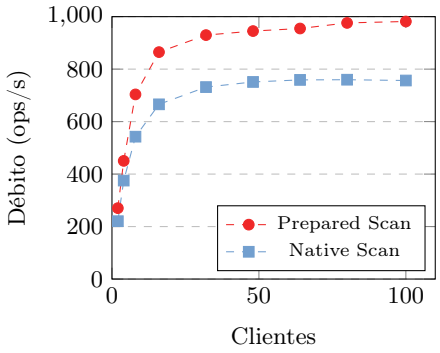
Foi utilizado o *Apache HBase* na versão `1.0.0` e o HDFS na `2.6.0`.

Em termos de dados, povoou-se a base de dados com um milhão de linhas, cada uma com 10 qualificadores, resultando em cerca de dez milhões de pares chave-valor, cujo tamanho do valor é de 100 *bytes*.

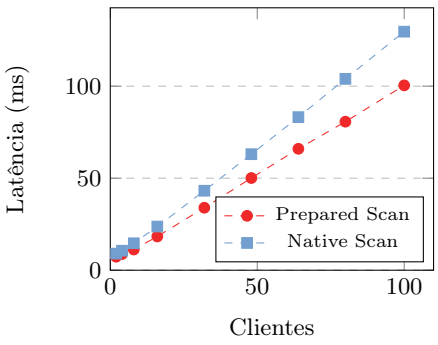
Em cada teste, executou-se um milhão de operações usando unicamente a operação `scan` no padrão de acesso a dados, com tamanho fixo de 100 linhas. Uma vez que o *Prepared Scan* é aplicado a um conjunto bem definido de colunas, o cliente YCSB foi modificado de modo a selecionar explicitamente as colunas 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.

4.2 Resultados

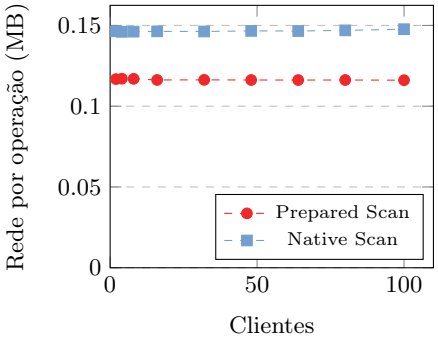
Os resultados são demonstrados na Figura 7, divididos em três métricas: débito, visível na Figura 7a, latência, na Figura 7b e, por fim, a rede usada em cada operação, na Figura 7c.



(a) Débito



(b) Latência



(c) Consumo de rede por operação

Figura 7: Métricas de desempenho para *Prepared Scan* e *Scan*

Focando a Figura 7a, é possível constatar que o débito obtido no *Prepared 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 *Scan* 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 7b, 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 *Scan*, à medida que são adicionados mais clientes. No auge do *Scan*, 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 7c representa a métrica de rede utilizada em cada operação e verifica-se efetivamente uma redução de aproximadamente 20% com esta solução.

De acordo com estes resultados, constata-se que o *Prepared Scan* permite lidar com um maior número de clientes e mantendo o desempenho sempre melhor que o *Scan* nativo, quando a estrutura dos dados é conhecida pela aplicação.

5 Conclusão

Neste documento, focou-se a atenção na otimização da consulta de dados estruturados nas bases de dados não relacionais, mais concretamente no *Apache HBase*. Uma das consequências da migração dos dados do contexto relacional para o não-relacional é a perda de conhecimento da estrutura de dados e que tal resulta num processamento e envio de informação extra.

A solução proposta, a adição de uma nova operação *Prepared Scan* diminui o custo associado a essa informação extra através da sua eliminação nos resultados retornados. Com este procedimento, reduziu-se a quantidade de dados enviados para o cliente em cerca de 20% e, consequentemente, foi possível obter um aumento do débito de 29% e diminuir a latência em cerca de 23% com esta solução. O *Prepared Scan* não exige a alteração do código-fonte do *Apache HBase*, uma vez que segue os padrões de extensão de protocolo, isto é, é implementado como um *coprocessor endpoint*. Esta característica permite que esta operação co-exista com a operação nativa e que seja instalada não só em versões mais antigas como também em distribuições diversas, por exemplo *Cloudera*.

Apesar de ser possível obter um melhor desempenho com esta solução, esta pode não ser tão eficiente com uma quantidade de dados na ordem dos *MegaBytes*. Tal se deve à biblioteca de serialização *Protocol Buffers* (PB), que possui algumas limitações já conhecidas pela comunidade do *Apache HBase*[2]. Desta forma, é ainda possível otimizar esta solução recorrendo a métodos fora dos padrões fornecidos pelo sistema.

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

Agradecimentos

Este trabalho foi parcialmente financiado pelo União Europeia no âmbito do projeto CoherentPaaS – Coherent and Rich PaaS with a Common Programming Model – FP7-611068 (`http://CoherentPaaS.eu`).

Referências

1. Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.
2. Lars George. *HBase: The Definitive Guide*. 2011.
3. 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.
4. Apache Hive. `http://hive.apache.org`.
5. Apache Phoenix. `http://phoenix.apache.org`.
6. Cloudera Impala. `http://impala.io/`.
7. Julian Rith, Philipp S. Lehmayr, and Klaus 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.
8. Ricardo Vilaça, Francisco Cruz, José Pereira, and Rui 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.
9. Liang Lin, Vera Lychagina, Weiran Liu, Younghee Kwon, Sagar Mittal, and Michael Wong. Tenzing: A SQL Implementation On The MapReduce Framework. 2011.
10. A.C. Carniel, A. de Aguiar Sa, V.H.P. Brisighello, M.X. Ribeiro, R. Bueno, R.R. Ciferri, and C.D. 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.
11. Chongxin 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.
12. Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel 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.
13. Apache HBase. `http://hbase.apache.org`.
14. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. 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.
15. Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

16. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.
17. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell 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.