Replicação Tolerante a Faltas Eficiente em Bancos de Dados Orientados a Grafos

Ray Willy Neiheiser[1], **Lau Cheuk Lung**[1], **Aldelir Fernando Luiz**[2], **Hylson Vescovi Netto**[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina

[2]*Campus* Blumenau – Instituto Federal de Educação, Ciência e Tecnologia Catarinense

{neiheiser.r, hylson.vescovi}@posgrad.ufsc.br, lau.lung@ufsc.br, aldelir.luiz@blumenau.ifc.edu.br

Abstract. *With the growth of social networks and the increase of highly linked data, graph technologies like distributed graph databases gained importance. Replication, widely used in the field of traditional databases is also implemented by various graph database solutions. Unfortunately, existing approaches for other database systems offer weak performance and scalability. We therefore consider deferred update replication using atomic broadcast to offer a scalable fault-tolerant solution for distributed graph databases. Our contribution is the adaption of the deferred update algorithm, offering highly scalable replication with performance advantage of when compared to traditional replication approaches.*

Resumo. *Com o crescimento das redes sociais e o aumento no volume de dados altamente conectados, a tecnologia banco de dados baseada em grafos tem ganhado notória importância. A replicação de dados, que é uma técnica amplamente adotada no âmbito de bancos de dados tradicionais, é também passível de implementação em soluções de bancos de dados baseados em grafos. Porém, as abordagens existentes noutros ambientes de banco de dados oferecem pouca escalabilidade e desempenho. Neste sentido, este trabalho introduz a técnica de atualização tardia (i.e., deferred update) no contexto de sistemas bancos de dados orientados a grafos. A solução proposta consiste numa adaptação do algoritmo clássico desta abordagem, de modo a possibilitar uma replicação bastante escalável, e com desempenho superior quando comparado à replicação tradicional.*

1. Introdução

Os avanços tecnológicos ocorridos nas últimas décadas nas áreas de computação e comunicação têm estimulado o surgimento de novas classes de aplicações que, cada vez mais, têm produzido um imenso volume de dados – principalmente aqueles baseados na Web, como por exemplo, as redes sociais. Estas aplicações demandam por uma gestão eficiente em termos dos dados manipulados, o que não é facilmente provido por um sistema de banco de dados tradicional, tal como o relacional. Tal eficiência é requerida para que seja possível obter escalabilidade e desempenho aceitáveis quanto ao uso destas aplicações. Por conseguinte, tal demanda tem culminado na proposição de novos modelos, paradigmas e tecnologias para a concepção de bancos de dados capazes de prover as necessidades requeridas por estas aplicações [Stonebraker 2010].

É lícito salientar que, ao longo dos anos os sistemas de gerenciamento de banco de dados relacionais (SGBDR) têm se intensificado como um padrão de *facto* como a solução

mais adequada/ideal para a persistência e recuperação de dados em sistemas computacionais de propósito geral [Stonebraker et al. 2007] – o que se deve principalmente pela sua maturidade [Codd 1970]. Todavia, a dinamicidade e a complexidade inerentes à concepção das aplicações mais modernas, associada ao volume de dados produzido/utilizado por elas, podem ser vistos como novos requisitos de aplicações aos sistemas de bancos de dados, o que por sua vez expõem novos desafios para os Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDRs) tradicionais.

Embora os benefícios quanto ao uso dos SGBDRs sejam indiscutíveis para aplicações de propósito geral, aplicações modernas baseadas em modelos de dados não relacionais, e portanto, diferentes daqueles suportados pelos SGBDRs, são passíveis de sofrerem consequências devido à pouca eficiência do SGBDRs, no que tange a realização de operações sobre dados não normalizados/estruturados [Hellerstein et al. 2007]. Em face deste fato, uma nova tecnologia de banco de dados, denominada NoSQL (*Not Only SQL*) [Cattell 2011] surge no intuito de suprir as necessidades de aplicações que processam e armazenam grandes volumes de dados (p. ex.: dados complexos, semi-estruturados ou não estruturados). E por assim dizer, a tecnologia trazida pelo NoSQL visa, sobretudo, melhorar questões relacionadas a escalabilidade, disponibilidade e desempenho em termos do armazenamento e recuperação de dados.

As principais características que diferenciam os bancos de dados NoSQL dos relacionais tradicionais são: (*i*) o relaxamento das propriedades ACID [Haerder and Reuter 1983], uma vez que eles são norteados pelo princípio conhecido por BASE (*Basically*, *Available*, *Soft State*, *Eventually Consistent*) [Pritchett 2008], e; (*ii*) a possibilidade de se utilizar modelos de dados distintos, de acordo com a necessidade das aplicações (p. ex.: chave-valor, orientado a documentos, orientado a colunas e baseado em grafos). Deste modo, as categorias de bancos de dados NoSQL existentes proveem uma flexibilidade em termos da representações de dados, além de dispensar o rigor necessário à definição dos *schemas*, tal como ocorre no modelo relacional [Stonebraker 2010]. Ademais, cabe informar que a tecnologia de banco de dados NoSQL tem sido amplamente adotada por organizações como Facebook, Amazon e Google, Twitter e LinkedIn.

No que diz respeito ao modelo de dados baseado em grafos, tal estrutura de dados tem sido empregada na modelagem de um grande número de aplicações que visam a solução de problemas práticos e reais (p. ex.: mineração de dados nas redes sociais). Alguns destes problemas são, inclusive, resolvidos através do processamento distribuídos do grafo – um exemplo concreto é o Google *Pregel* [Malewicz et al. 2010]. No intuito de prover um ambiente adequado para o armazenamento de dados de aplicações modeladas por grafos é que surgem os bancos de dados baseados em grafos, como uma alternativa aos sistemas de bancos de dados tradicionais, isto é, os relacionais. Note que o armazenamento de grafos consiste num sistema que contém uma sequência de nodos e suas relações que, quando combinadas, dão origem a um grafo [Robinson et al. 2015].

Por outro lado, embora o armazenamento de grafos seja essencialmente construído a partir de uma estrutura de dados simples do tipo nodo-relação-nodo, há uma complexidade inerente à manutenção de alguns atributos requeridos e preconizados pelo movimento NoSQL [Leavitt 2010, Cattell 2011], como é o caso da escalabilidade. No tocante a este atributo, pode-se dizer que o satisfazimento do mesmo em bancos de dados NoSQL baseados em grafos não é algo trivial, o que se deve principalmente pela ligação estreita existente em cada nodo de um grafo. Note que os dados podem ser replicados em diversos servidores

para prover melhorias de desempenho em termos de operações de leitura. Porém, a complexidade associada a operações de escritas de dados em múltiplos servidores e de consultas que se estendem por vários nodos do grafo, podem culminar na inviabilização do uso deste tipo de sistema.

Neste contexto, este artigo apresenta a proposição de um protocolo de replicação baseado na clássica técnica conhecida por *Deferred Update* [Kemme and Alonso 2000], para bancos de dados NoSQL baseados em grafos. Ao nosso conhecimento, trata-se da primeira iniciativa no sentido de especificar/desenvolver um protocolo de replicação para este tipo de ambiente, o que, portanto, consiste numa contribuição em termos de algorítmica distribuída tolerante a faltas, e bancos de dados distribuídos. Na mesma perspectiva, a principal contribuição trazida pelo trabalho consiste na especificação de um protocolo de replicação para bancos de dados NoSQL baseados em grafos.

E por assim dizer, o restante do artigo está organizado da seguinte maneira. A Seção 2 faz uma breve apresentação da tecnologia de banco de dados baseada em grafos; na Seção 3 é descrita a técnica de replicação *deferred update*; a Seção 4 descreve, em detalhes, a especificação do protocolo proposto, e; na Seção 5 é apresentada uma avaliação de desempenho e discussão dos resultados. Por fim, a Seção 6 conclui o trabalho.

2. Tecnologia de Bancos de Dados Baseados em Grafos

De um modo geral, o modelo de dados baseado em grafos contém três elementos básicos, os quais são: (*i*) os nodos, isto é, os vértices do grafo; (*ii*) os relacionamentos, representado pelas arestas, e; (*iii*) as propriedades, que denotam os atributos dos nodos e relacionamentos. Desta maneira, um banco de dados pode ser representado como um multidígrafo rotulado [Rabuske 1992], em que cada par de nodos pode ser conectado por uma ou mais arestas. O armazenamento baseado em grafo consiste numa 3-tupla, onde cada campo desta tem a finalidade de armazenar apenas um dos elementos básicos do grafo, isto é, o(s) nodo(s), o(s) relacionamentos e a(s) propriedade(s). A Figura 1 ilustra de maneira simplificada, a estrutura/organização de um sistema de armazenamento baseado em grafo.

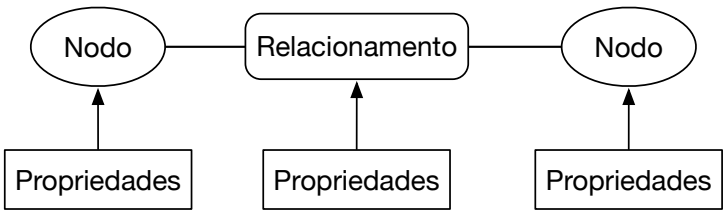

Figura 1. Exemplo de representação do modelo de dados baseado em grafo.

A partir da compreensão do modelo de dados em questão, um banco de dados baseado em grafos pode ser entendido como uma coleção de dados denotada não por um *schema* de dados de dados tradicional, mas por estruturas de dados especificadas a partir de nodos, relacionamentos e propriedades, sob a designação de representar os dados daquela coleção [Robinson et al. 2015] (vide Figura 2). Neste sentido, os nodos são usados para representar entidades como pessoas, negócios, etc., isto é, especificações similares àquelas que visam representar algum objeto de aplicação. Enquanto que as propriedades são usadas para descrever tanto os nodos como os relacionamentos. Por outro lado, os relacionamentos tem a finalidade de ligar (ou relacionar) um nodo a outro, ou a alguma propriedade.

Assim, um sistema de gerência de banco de dados (SGBD) baseado em grafo – tal como o SBGD relacional – é uma componente de software designada a criar, recuperar, atualizar e excluir dados sob a representação de grafos. Todavia, é digno de nota que nem todos os ambientes de bancos de dados baseados em grafos efetuam o armazenamento subjacente de dados de maneira nativa, isto é, na forma de um grafo. Em geral, os sistemas que não proveem o armazenamento nativo em grafos, serializam os dados mapeados no grafo numa representação adequada para os modelos relacional ou de objetos [Robinson et al. 2015]. Alguns exemplos de SGBDs baseados em grafos comerciais suportam armazenamento nativo e merecem destaque: Neo4J, InfoGrid e HyperGraphDB.

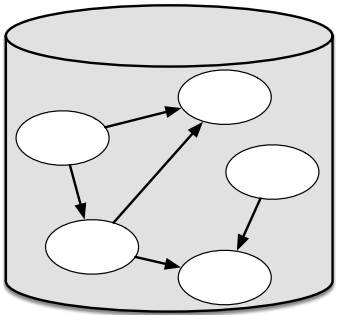

Figura 2. Ilustração de um banco de dados baseado em grafos.

É lícito salientar que as vantagens quanto à utilização do modelo de dados baseado em grafos são mais evidentes quando as aplicações demandam por consultas mais complexas. Neste caso, a opção por um modelo tradicional como o relacional, provavelmente incorrerá num custo demasiadamente grande para a aplicação, já que inúmeras operações de junção serão necessárias para fornecer os dados. Para tais aplicações, os bancos de dados orientados a grafos são uma ótima alternativa, pois são otimizados para efetuar a recuperação de dados em ambientes onde predominam interligações entre os mesmos.

3. Replicação de Dados Baseada na Técnica *Deferred Update*

A literatura descreve a técnica *Deferred Update* como uma das soluções mais promissoras para a replicação de bancos de dados [Kemme and Alonso 2010], num ambiente onde as manipulações de dados ocorrem por meio de transações. Para tanto, a estratégia desta abordagem envolve um conjunto de servidores (ou réplicas), que mantém cópias de um mesmo conjunto de dados (i.e., um banco de dados). A ideia central desta técnica consiste em efetuar a execução de todas as operações de uma transação inicialmente numa única réplica do banco de dados.

Assim, transações que apenas efetuam leituras (i.e., aquelas que não alteram o estado do banco de dados) podem ser validadas localmente naquela réplica em que foram executadas. De outro modo, para transações que alteram o estado do banco de dados (i.e., de escrita), ao término da execução das operações que as compõem, elas devem ser certificadas sobre o ambiente replicado de banco de dados, a fim de verificar se a mesma é passível de validação (i.e., *commit*). Uma vez que uma transação tem êxito em sua certificação, as atualizações por ela efetuadas são propagadas ao ambiente replicado de banco de dados e, após o processamento pelas demais réplicas, a transação é então validada e a partir daí toma efeito no ambiente replicado de banco de dados.

Um aspecto que é digno de nota é que as atualizações podem ser propagadas de maneira consistente às réplicas do ambiente de banco de dados, a partir do uso de uma infi-

nidade de protocolos de sincronização [Pedone et al. 2003]. Tal aspecto, inclusive, tem impelido quanto à adoção da técnica em questão por diversos protocolos de replicação de bancos de dados, e em diferentes contextos [Pedone et al. 2003, Patiño-Martínez et al. 2000, Amir and Tutu 2002]. Ademais, é importante salientar que o uso da mesma tem demonstrado um desempenho bastante aceitável em situações reais/práticas [Pedone et al. 2003, Patiño-Martínez et al. 2000, Amir and Tutu 2002].

Não obstante as benéfices ora elencadas, a respeito da técnica de replicação em lide, outro aspecto que também corrobora para o uso extensivo de tal abordagem em ambientes reais, reside no fato de que nestes ambientes há o predomínio de transações do tipo somente-leitura [Bernstein and Newcomer 2009]. Tal situação permite obter um equilíbrio acerca da carga de trabalho executada pelas réplicas, de modo que elas podem executar as operações de diferentes transações de maneira completamente independente umas das outras; embora ainda seja requerida a sincronização das réplicas para o caso de transações de atualização (i.e., o envio das operações para processamento pelas réplicas). Todavia, em se tratando das escritas, uma otimização bastante recorrente quanto ao uso da técnica é que a maiorias das implementações encapsula todas as operações numa única mensagem. Este encapsulamento permite obter ganhos de escala, bem como evitar sobrecarga do sistema subjacente de comunicação (i.e., quando comparado às técnicas que efetuam a propagação operação a operação).

4. Visão Geral da Solução Proposta

Nesta seção se apresentam os principais aspectos que amparam a solução ora proposta. Neste sentido, conforme já mencionado a solução proposta consiste num protocolo para a replicação de dados em bancos de dados baseados em grafos, a partir da técnica já conhecida, denominada *deferred update*. A escolha da estratégia de replicação se dá por algumas razões como a boa escalabilidade verificada em diversas implementações de tal estratégia de replicação de dados [Patiño-Martínez et al. 2000, Amir and Tutu 2002, Pedone et al. 2003, Kemme and Alonso 2010].

Um aspecto interessante acerca da técnica em questão, é que quando comparada com as abordagens mais clássicas de replicação, tais como a **replicação de máquinas de estados** [Schneider 1990] e a **replicação primário-backup** [Budhiraja et al. 1993]. Por exemplo, a replicação de máquinas de estados não permite um aumento da vazão (i.e., *throughput*) do sistema quando se adicionam novas réplicas – ao contrário do que ocorre com o uso do *deferred update* –, isso porque cada transação é executada por todas as réplicas. Por outro lado, a replicação primário-backup executa a transação sobre a réplica primária, e então propaga as alterações para processamento pelas réplicas de backup. Neste caso, a vazão do sistema é limitada pela capacidade de processamento da réplica primária, e não pelo número de réplicas.

É importante notar que a replicação baseada no *deferred update* apresenta melhor escalabilidade que as demais mencionadas, justamente porque todas as réplicas podem atuar como **primárias**, uma vez que as transações sempre são executadas localmente por alguma réplicas e então propagadas para as demais. Este é o caso do protocolo proposto, conforme ilustrado pela Figura 3. Note que o cliente dá início a uma transação quando submete uma operação para alguma das réplicas (passo 1) – a réplica 2, no caso da Figura 3. Esta réplica é designada como líder da transação, e portanto, tem a função de executar todas as operações submetidas pelo cliente no âmbito daquela transação. No passo 2, ao receber a operação e

processá-la, a réplica líder envia a resposta da operação ao cliente. Esta fase do protocolo conhecida como fase de **execução** perdura até o momento em que o cliente deseja validar sua transação (i.e., efetuar o *commit*) – o que é realizado no passo 3.

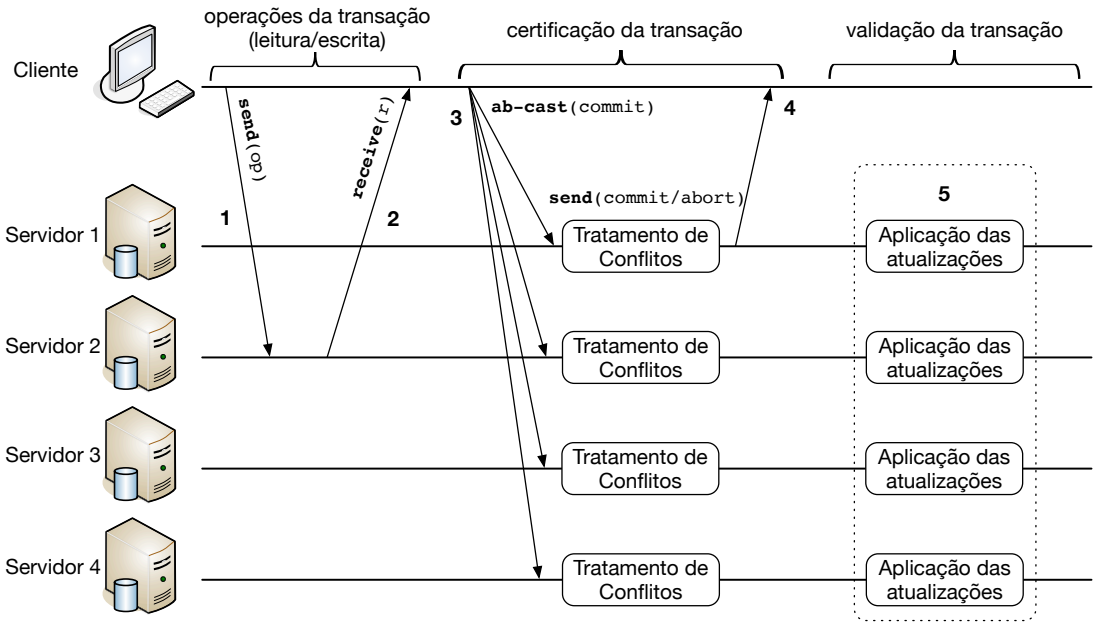

Figura 3. Dinâmica de funcionamento do protocolo.

Quando o cliente conclui a execução da transação, ele envia um pedido de validação (i.e., uma mensagem ***commit***) por meio de difusão atômica (passo 3), o que implica que todas as réplicas terminarão por entregar tal mensagem ao protocolo de replicação. Assim, ao entregar a mensagem em questão, as réplicas dão início a execução de um procedimento para realizar a certificação da transação. Este procedimento verifica se as alterações efetuadas pelas transação estão em consonância com o critério de **serialização** – i.e., se a transação pode ser serializada após a última transação validada e estável. Note que a certificação é efetuada com base na verificação da existência de conflitos entre a transação em processo de validação, em relação as transações já validadas. Uma vez que as réplicas concluem a fase de certificação, iniciada pela entrega da mensagem ***commit***, elas respondem ao cliente acerca do resultado final de transação, o qual será ***commit*** ou ***abort***. E finalmente, no passo 5, após terem enviado suas respostas ao cliente, as réplicas aplicam as atualizações em suas bases de dados locais, a fim de integrar o efeito da transação ao estado do banco de dados. Maiores detalhes a respeito da especificação formal do protocolo serão vistas na Seção 4.2.

4.1. Modelo de Sistema

O ambiente de sistema admitido é composto por um universo de processos $\mathcal{U}$, sendo este dividido em dois subconjuntos. O subconjunto $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ denota as réplicas (ou servidores) do ambiente replicado de banco de dados. Por outro lado, o subconjunto $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ representa os processos dos clientes que interagem com o sistema de banco de dados. O subconjunto $\mathcal{S}$ tem cardinalidade $|\mathcal{S}| \geq 2f + 1$, enquanto que o subconjunto $\mathcal{C}$ pode conter um número arbitrário (não infinito) de processos. Os processos são designados de acordo com seus respectivos comportamentos, isto é, (*i*) um processo é correto se ele segue sua especificação de não falha durante a execução do sistema, ou; (*ii*) é

faltoso, caso contrário. É assumido que até f réplicas dentre as $|\mathcal{S}| \geq 2f + 1$ podem exibir comportamento faltoso, e portanto, parar de executar prematuramente[1]. Em relação aos processos que implementam os clientes, não há um limite para a ocorrência de faltas nestes.

Os processos em $\mathcal{U}$ não compartilham memória, de modo que eles se comunicam apenas por passagem de mensagens. À vista disso, as comunicações ocorrem por meio de primitivas dos tipos **um-para-um** e **um-para-muitos**, sendo que as primitivas *send* e *receive* são empregadas no primeiro caso; e para o segundo caso considera-se o uso de difusão atômica (i.e., com ordem total), cujo envio e entrega de mensagens são denotados ao logo do texto pelas primitivas *A-broadcast* e *A-deliver*, respectivamente. A especificação da primitiva de difusão atômica adotada é baseada no protocolo *Ring-Paxos* [Marandi et al. 2010]. Os canais de comunicação empregados em todos os tipos de comunicações são confiáveis e autenticados [Basu et al. 1996]. Isto implica que, se emissor e receptor são corretos, a mensagem é: (*i*) recebida numa comunicação um-para-um e; (*ii*) entregue numa comunicação um-para-muitos.

Em relação à base de dados, é assumido que cada processo em $\mathcal{S}$ mantém uma cópia completa da mesma, de modo a constituir o ambiente replicado de banco de dados. O sistema de banco de dados adotado é um ambiente nativo de banco de dados baseado em grafo $\mathcal{D} = \{d_i, d_{i+1}, ..., d_n\}$, tal que $i \geq 1 \wedge n < \infty$. Os itens de dados d_i compostos por nodos $\mathcal{N} = \{n_i, n_{i+1}, ..., n_j\}$, relacionamentos $\mathcal{R} = \{r_i, r_{i+1}, ..., r_k\}$ e suas respectivas propriedades. Um relacionamento é sempre orientado e liga exatamente dois nodos n_i e n_j; porém, é importante notar que um relacionamento r_k que liga o nodo n_i com o nodo n_j é, portanto, uma relação diferente daquela que representa uma ligação do nodo n_j com o nodo n_i. É também lícito salientar que um nodo n_i pode ter ou não relacionamentos.

As interações dos clientes com o sistema de banco de dados se dá por meio de transações. O modelo admite o uso de um esquema de travas (i.e., *locks*) durante o processamento de transações, a fim de permitir a execução consistente das operações de leitura e de escrita realizadas no contexto da transação. Ao término da execução das operações, a transação é concluída pela sua validação (*commit*) – que torna permanente as alterações efetuadas pela transações – ou anulação (*abort*) – que descarta todo o efeito da transação. Cada nodo n_i e relacionamento r_j é identificado por um ID único, ou então, pela combinação única de suas propriedades. Ademais, cada transação mantém os conjuntos designados por *read-set* (rs), *write-set* (ws), *delete-set* (ds) e *create-set* (cs), cujos elementos consistem na descrição dos nodos e relacionamentos referenciados pela transação. E finalmente, o critério de consistência adotado é aquele baseado na **serialização**, isto é, aquele cujo efeito da execução concorrente de um conjunto de transações é equivalente ao da execução sequencial do mesmo conjunto de transações (i.e., uma após a outra).

Por fim, a considerar o modelo de interação (i.e., das hipóteses temporais), assumimos o modelo baseado em sincronismo terminal (*eventually synchronous system*) [Dwork et al. 1988] em razão das características realistas presentes em tal modelo – é assíncrono em grande parte do tempo, porém, durante períodos de estabilidade os tempos são limitados mas desconhecidos.

4.2. Base Algorítmica do Protocolo

Nesta seção se apresenta a formalização dos algoritmos que compõem o protocolo proposto. No intuito de facilitar a compreensão por parte do leitor, a especificação do protocolo está

[1]Não são admitidas faltas bizantinas, de modo que os processos falham apenas por parada.

dividida em duas partes, isto é, a parte que trata dos clientes e a parte que trata das réplicas (vide Figuras 1 e 2, respectivamente).

Inicialmente é realizada a declaração das variáveis mantidas pelos clientes para cada transação por eles executada (linhas 1 a 8, Algoritmo 1). Quando um cliente deseja iniciar uma transação, ele o faz por meio do procedimento $begin(t)$, em que t denota o id local da transação atribuído pelo cliente. Tal procedimento simplesmente inicializa as variáveis que serão utilizadas no decurso da transação, RS, WS, CS e DS, que denotam os *read-set*, *write-set*, *create-set* e *delete-set*, respectivamente (linhas 9 a 12, Algoritmo 1).

Uma vez que a transação é iniciada pelo cliente, ele procede com as operações que irão compor a unidade lógica transacional (p. ex.: leituras e/ou escritas) – vide diagrama da Figura 3. Para executar uma operação de leitura o cliente invoca a função $read$ (linhas 19 a 32 – Algoritmo 1). De outro modo, para operações de escrita o cliente invoca a função $write$, especifica nas linhas 33 a 47 do Algoritmo 1. No caso de uma operação de leitura, ao chamar a função $read$ o cliente fornece como argumentos o identificador da transação t, o id do snapshot no qual a transação irá realizar as operação (s_{id}) e o identificador único uid do nodo ou relacionamento sobre o qual a operação será realizada. É importante salientar que o identificador único é obtido por uma composição de dados, de acordo com o tipo de objeto de banco de dados: (*i*) os nodos são identificados por uma composição formada por seus rótulos e suas propriedades; (*ii*) os relacionamentos são identificados pela composição dos rótulos e propriedades dos nodos dos quais faz parte, pelas propriedades do próprio relacionamento, e pelo tipo de relacionamento.

Ao requisitar uma operação de leitura, se tal operação for a primeira executada no âmbito daquela transação, o cliente envia uma mensagem contendo os parâmetros recebidos pela operação $read$ à replica escolhida como líder da transação, e aguarda (*i*) até o recebimento de uma resposta daquela réplica (linhas 19 e 20 do Algoritmo 1), ou; (*ii*) o esgotamento de um temporizador interno. Ao receber a resposta na linha 21, o *snapshot* é inicializado com o que fora recebido na mensagem enviada pela réplica líder (linhas 21 e 22, Algoritmo 1), e o valor v recebido como resultado da réplica é retornado ao cliente como resultado da operação. Por outro lado, se aquela não for a primeira operação da transação, o algoritmo verifica se o item de dados solicitado para leitura (uid) já foi referenciado/manipulado pela transação e está presente nos conjuntos de escrita e criação (linhas 25 e 27, respectivamente) – pela combinação do uid. Neste caso, o item pode ser retornado diretamente dos valores locais do cliente, sem necessidade de interação com a réplica líder. De outro modo, se o item de dados não foi até então manipulado/referenciado pela transação, o cliente envia uma mensagem à replica líder, e aguarda pelo recebimento do resultado da operação (linhas 29 a 32, Algoritmo 1). É digno de nota que, por razões de simplificação, os temporizadores não constam nas especificações dos algoritmos. Entretanto, quando um temporizador é esgotado, uma exceção é lançada ao cliente. Neste caso, o cliente pode reenviar a operação ou então reiniciar a transação numa outra réplica líder.

Ao contrário do que ocorre com as operações de leitura, as operações de escrita não são enviadas à replica líder. Ao observar a especificação da função $write$ (linhas 33 a 47 – Algoritmo 1), se pode verificar que todas as possíveis operações de escrita (i.e., *create*, *update* e *delete*) são efetuadas diretamente sobre os conjuntos locais mantidos pelo cliente (CS, WS e DS). Neste caso, o que ocorre é apenas a manutenção do valor v fornecido como argumento da operação de escrita, ao respectivo conjunto nos termos da operação requisitada (linhas 33, 35, 40 e 46 do Algoritmo 1). Nos mesmos moldes da função $read$,

Algoritmo 1 Tarefa executada pelos clientes.

Declaração de Variáveis:

1: $t.RS = \perp$ /* *ReadSet* */
2: $t.WS = \perp$ /* *WriteSet* */
3: $t.CS = \perp$ /* *CreateSet* */
4: $t.DS = \perp$ /* *DeleteSet* */
5: $s_i = \perp$ /* id do servidor/réplica */
6: $c_i = \perp$ /* id do cliente */
7: $s_{id} = \perp$ /* id do *snapshot* */
8: $it = \perp$ /* id do *snapshot* */

procedure $begin(t)$ /* Procedimento invocado para iniciar a transação */
9: $t.RS \leftarrow \emptyset$ /* inicialização do read-set */
10: $t.WS \leftarrow \emptyset$ /* inicialização do write-set */
11: $t.CS \leftarrow \emptyset$ /* inicialização do create-set */
12: $t.DS \leftarrow \emptyset$ /* inicialização do delete-set */

function $commit(t, s_{id})$
13: **if** $t.CS = \emptyset \wedge t.WS = \emptyset \wedge t.DS = \emptyset$ **then**
14: **return** `committed`
15: **else**
16: $A\text{-}broadcast(c_i, \langle \texttt{commit}, t, s_{id} \rangle)$ to $\forall s_i \in \mathcal{S}$
17: **wait until** $receive(\langle t, \texttt{outcome} \rangle)$ from some $s_i \in \mathcal{S}$
18: **return** `outcome`

function $read(t, s_{id}, uid)$ /* Operação de leitura */
19: **if** $it = \perp$ **then**
20: $send(c_i, \langle \texttt{read}, t, uid, s_{id} \rangle)$ to s_i
21: **wait until** $receive(\langle uid, v, rs_{id} \rangle)$ from s_i
22: $s_{id} \leftarrow rs_{id}$ /* id *snapshot* como a primeira leitura */
23: **return** v
24: **else**
25: **if** $uid \in t.CS$ **then**
26: **return** $t.CS(uid)$ /* retorna o conteúdo do create-set */
27: **else if** $uid \in t.WS$ **then**
28: **return** $t.WS(uid)$ /* retorna o conteúdo do write-set */
29: **else**
30: $send(c_i, \langle \texttt{read}, t, uid, s_{id} \rangle)$ to s_i
31: **wait until** $receive(\langle uid, v, rs_{id} \rangle)$ from s_i
32: **return** v

function $write(op, t, s_{id}, uid, v)$ /* Operação de escrita */
33: **if** $op = \texttt{create}$ **then**
34: $t.CS \leftarrow t.CS \cup \langle uid, v \rangle$
35: **else if** $op = \texttt{update}$ **then**
36: **if** $uid \in t.CS$ **then**
37: $t.CS \leftarrow t.CS \cup \langle uid, v \rangle$
38: **else**
39: $t.WS \leftarrow t.WS \cup \langle uid, v \rangle$
40: **else if** $op = \texttt{delete}$ **then**
41: **if** $uid \in t.WS$ **then**
42: $t.WS \leftarrow t.WS \setminus \langle uid, * \rangle$
43: $t.DS \leftarrow t.DS \cup \langle uid, v \rangle$
44: **else if** $uid \in t.CS$ **then**
45: $t.CS \leftarrow t.CS \setminus \langle uid, * \rangle$
46: **else**
47: $t.DS \leftarrow t.DS \cup \langle uid, v \rangle$

o argumento t fornecido para a função $write$ denota a transação para a qual a operação está sendo requisitada, op é a operação a ser executada, s_{id} corresponde ao *snapshot* do cliente, e uid dizem respeitos ao item de dados e o valor a ser manipulado sobre o item de dados.

Por fim, quando não houver mais operações para serem executadas numa transação, o passo seguinte consiste na validação da transação, isto é, a tentativa de tornar permanente o efeito produzido por ela sobre o estado do banco de dados. No protocolo proposto isto é feito por meio de uma chamada para a função $commit$ (linhas 13 a 18 – Algoritmo 1). Assim, quando a validação para uma transação é solicitada pelo cliente, o algoritmo analisa os conjuntos mantidos no lado cliente durante o decurso da transação. Se for constatado que todos eles estão vazios – o que indica que a transação era de somente-leitura –, a transação é validada imediatamente, pois não nesta condição nenhuma alteração foi realizada sobre os itens de dados do banco de dados. Do contrário, isto é, se algum dos conjuntos não for vazio, o cliente envia por meio de difusão atômica uma mensagem `commit` à todas as réplicas do ambiente de banco de dados, e aguarda até o recebimento da primeira mensagem `outcome` enviada por alguma réplica. Note que, como é pressuposto que não ocorrem faltas bizantinas, apenas uma mensagem é o suficiente para atestar o resultado do pedido de validação processado pelas réplicas. No caso, como as réplicas são inicializadas num mesmo estado e as validações das transações são entregues às réplicas na mesma ordem – devido ao uso da difusão atômica –, o resultado após o processamento dos pedidos de validação é o mesmo em todas das réplicas [Schneider 1990].

Em se tratando da especificação do código das réplicas, o mesmo é formalizado a partir do Algoritmo 2. Note que as interações dos clientes durante a fase de execução da transação, quando apropriado, ocorrem diretamente com a réplica líder da respectiva transação (linha 29, Algoritmo 2). No caso, quando uma réplica recebe uma operação de leitura de algum cliente, ela imediatamente faz uma chamada à função $read$ especificada para a réplica, passando como argumento todos os dados recebidos na mensagem (linhas 6 a 13 do Algoritmo 2). Por sua vez, A função $read$ inicializa o *snapshot* que será usado para as operações daquela transação – se tal operação é a primeira daquela transação –, e então procede com a recuperação do valor v para o item de dados uid contido na operação, a partir de seu *snapshot* local. Em seguida, a réplica envia ao cliente o resultado da operação, isto é, o valor v (linha 13, Algoritmo 2).

Outrossim, quando o protocolo de difusão atômica subjacente entrega uma mensagem `commit` para o protocolo de replicação proposto (linha 30 – Algoritmo 2), todas as réplicas iniciam o procedimento de validação e certificação da transação, o que ocorre através da invocação à função $commit$. Observe que, como a mensagem foi enviada por meio de difusão atômica, isto implica que todas as réplicas não faltosas terminarão por entregar tal mensagem, numa mesma ordem. Então, ao adentrar na função $commit$ (linhas 14 a 23 do Algoritmo 2), o primeiro passo consiste na verificação quanto à existência de conflitos entre a transação em processo de validação, em relação às transações já validadas que não precedem a transação em questão – isto é, cuja validação ocorreu no decurso da transação em lide.

A verificação de conflitos, que é realizada pela função $conflict\_handler$ ocorre conforme descrito a seguir (linhas 24 a 28 do Algoritmo 2). Para cada transação já validada, cujo *snapshot id* é maior que o *snapshot id* da transação que acabara de solicitar a validação, verifica-se se há alguma intersecção entre as escritas das transações já validadas com as leituras efetuadas pela transação em processo de validação. A interseção indica que a transação

Algoritmo 2 Tarefa executada pelas réplicas (servidores).

Declaração de Variáveis:

1: $gid = 0$ /* inicialização do *snapshot* global */
2: $cws\langle it, ws\rangle = \emptyset$ /* *WriteSet* validado */
3: $ctx\langle it, v\rangle = \emptyset$ /* transações validadas */
4: $ltx\langle it, v\rangle = \emptyset$ /* transações locais */
5: $l_{id} = \perp$ /* id local da transação */

function $read(t, s_{id}, uid)$

6: $l_{id} \leftarrow \perp$
7: **if** $s_{id} = \perp$ **then**
8: $l_{id} \leftarrow gid$
9: $ltx \leftarrow t$
10: **else**
11: $l_{id} \leftarrow s_{id}$
12: $v \leftarrow$ `retrieve`(uid, l_{id})
13: $send(s_i, \langle uid, v, l_{id}\rangle)$ to c_i

function $commit(t, s_{id})$

14: $outcome =$ `conflict_handler`(t, s_{id})
15: **if** $outcome =$ `commit` **then**
16: $cws \leftarrow cws \cup \langle s_{id}, t.WS\rangle$
17: $ctx \leftarrow ctx \cup \langle s_{id}, t\rangle$
18: $send(s_i, \langle$`outcome`$\rangle)$ to c_i
19: **for** $t_i \in ltx$ **do**
20: **if** $t_i.RS \cap ctx[0]$ **then**
21: `abort`(t_i)
22: **else**
23: `execute`$(ctx[0])$

function $conflict\_handler(t, s_{id})$

24: **for** $t_i \in ltx : it > s_{id}$ **do**
25: **if** $\neg((t_i.WS \cup t_i.DS) \cap t.RS)$ **then**
26: **return** `commit`
27: **else**
28: **return** `abort`

Tarefa principal:

upon $receive(\langle$`read`$, t, uid, s_{id}\rangle)$ from c_i
29: **call** `read`(t, s_{id}, uid)
upon $A\text{-}deliver(\langle$`commit`$, t, s_{id}\rangle)$
30: **call** `commit`(t, s_{id})

que acabara de solicitar sua validação efetuou uma leitura de um dado que foi alterado em seu decurso, o que denota uma condição de leitura "suja", e portanto, viola o critério de serialização. Se tal condição não é verificada na linha 25 do Algoritmo 2, a transação pode ser serializada no banco de dados, após a última transação validada (linha 26 – Algoritmo 2); ou, do contrário, a transação será anulada (linhas 27 e 28, Algoritmo 2).

5. Aspectos de Implementação e Avaliação

No intuito de validar o protocolo proposto e realizar uma prova de conceito, foi desenvolvido um protótipo do mesmo, o qual foi implementado sobre o SGBD baseado em grafos Neo4J. Para tanto, os testes foram executados em 4 máquinas Intel-core i7-3770k quad-cores, hyper threading com 4GB RAM. O sistema operacional utilizado nos experimentos foi o Linux Debian 7.4 Wheeze 64 bits, kernel 3.2.54-2. O algoritmo foi implementado na linguagem Java, tendo sido executado sob a JVM 1.8.0_31. Para os experimentos, o

Neo4j (`neo4j.com`) foi configurado em modo de alta disponibilidade, para uma melhor comparação com o protocolo proposto baseado no *deferred update*. Ademais, a opção pela tecnologia do Neo4j se deu em razão do mesmo realizar escritas apenas em um único servidor principal.

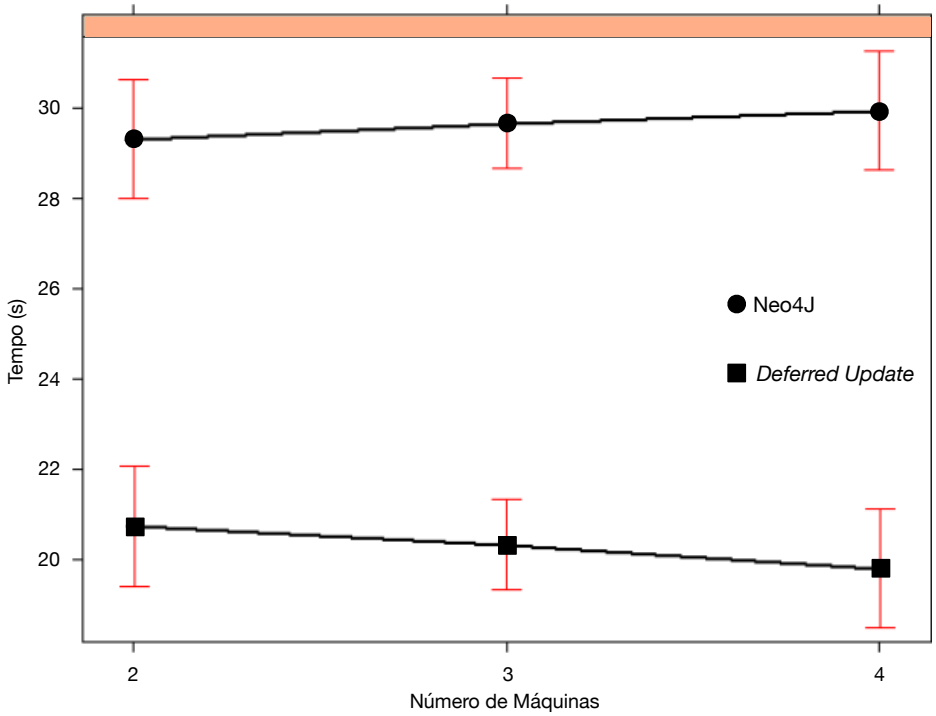

Figura 4. Quatro processos servidores, em duas, três e quatro máquinas físicas.

Foram utilizados bancos de dados inicialmente vazios, e uma função foi projetada no cliente, a fim de enviar 32 transações para os servidores, criando um número aleatório de nós e relacionamentos entre os nós. Foram também criadas operações de atualização dos nós e dos relacionamentos. Durante a execução das transações, podem ocorrer conflitos entre transações concorrentes, que serão resolvidos pelos algoritmos do Neo4J e da solução proposta. Após a execução dos testes, os bancos de dados serão comparados e verificados quanto à consistência. Dois experimentos foram conduzidos para investigar a diferença entre o Neo4J e o protocolo proposto.

O primeiro experimento executou quatro processos servidores sobre duas, três e quatro máquinas físicas. Em cada uma destas três configurações, 10 replicações do experimento foram realizadas, a fim de alcançar resultados estatisticamente relevantes. A seguir, quatro clientes acessaram os quatro processos servidores e exibiram como saída o tempo despendido, assim que os clientes finalizavam a operação. A Figura 4 mostra resultados do Neo4J e do protocolo baseado no *deferred update*. A solução proposta apresenta um desempenho significativamente melhor do que o Neo4J em modo de alta disponibilidade. Neo4J apresenta resultados inferiores quando distribuído em mais servidores, enquanto o algoritmo proposto escala com sucesso, considerando o número de servidores.

O segundo experimento executou de três a oito processos servidores em quatro máquinas físicas. Analogamente ao experimento anterior, de três a oito clientes acessaram os processos servidores em cada configuração, que foi executada dez vezes. Os resultados deste segundo experimento são exibidos na Figura 5. O algoritmo *deferred update* possui

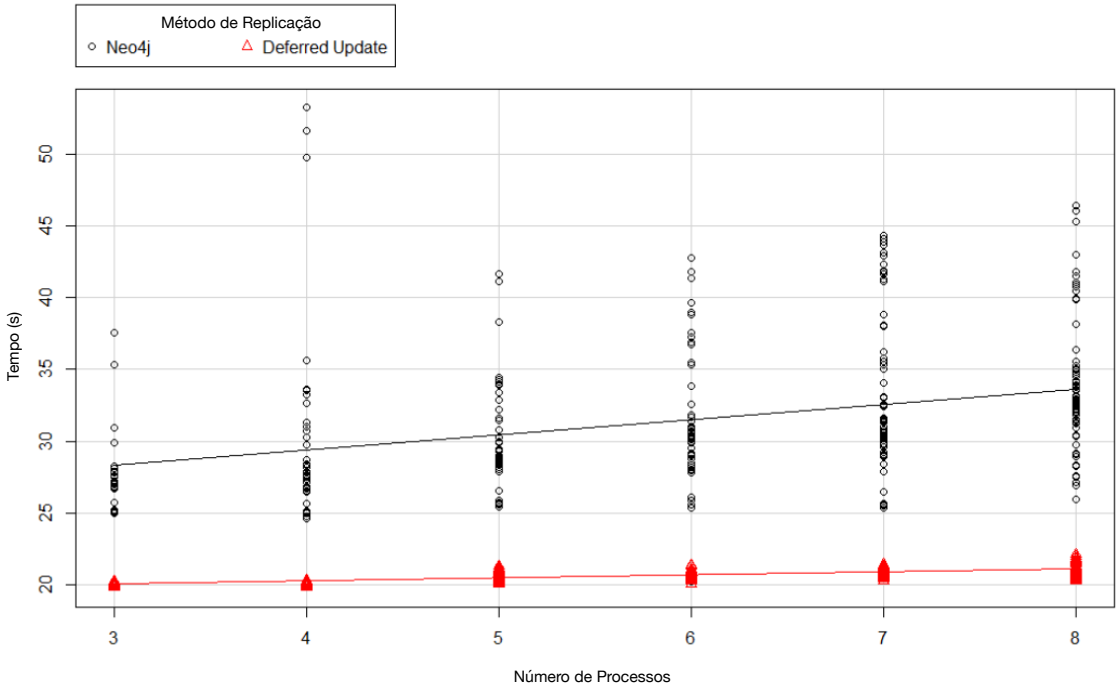

Figura 5. Três a oito processos servidores, em quatro máquinas físicas.

desempenho representado pela linha inferior, enquanto o Neo4j em modo de alta disponibilidade está representado na linha superior. Da mesma maneira que no experimento anterior, o algoritmo proposto demonstrou desempenho superior ao Neo4J. Além disso, a inclinação das linhas mostra que a solução proposta possui uma escalabilidade superior ao Neo4J, em relação ao número de réplicas.

6. Conclusões

O algoritmo apresentado neste artigo, isto é, *deferred update* para bancos de dados baseados em grafos, apresentou desempenho superior à solução desenvolvida para a tecnologia predominante no mercado, o Neo4J. Como possibilidades de expansão das investigações deste trabalho pode-se acessar e utilizar o mecanismo de indexação do Neo4J ou conduzir experimentos em um número maior de réplicas. É digno de nota que a técnica *deferred update* também possui algumas desvantagens. Por exemplo, o algoritmo *deferred update* requer que todas as réplicas contenham o banco de dados inteiro. Embora, na atualidade espaço em disco não seja propriamente um problema, transações de maior duração podem sofrer por inanição. Da mesma maneira, transações longas podem requerer bloqueios globais, o que, portanto, deve ser evitado.

Ademais, a solução proposta pode ser vista como um ponto de partida para que, desenvolvedores de SGBDs baseados em grafos possam implementar soluções escaláveis e ao mesmo tempo, tolerante a faltas em seus respectivos sistemas. Como trabalho futuro, pretende-se estender o protocolo proposto para tolerar também faltas bizantinas.

Agradecimentos

Trabalho parcialmente financiado pela FAPESC/IFC, processo/projeto Nº00001905/2015.

Referências

Amir, Y. and Tutu, C. (2002). From total order to database replication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 494–503.

Basu, A., Charron-Bost, B., and Toueg, S. (1996). Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 105–122.

Bernstein, P. and Newcomer, E. (2009). *Principles of Transaction Processing: for the systems professional*. Morgan Kaufmann Publishers Inc., Burlington, MA, USA, 2 edition.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. In Mullender, S., editor, *Distributed systems (2nd Ed.)*, pages 199–216. Addison-Wesley Publishing Co., New York, NY, USA.

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

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317.

Hellerstein, J. M., Stonebraker, M., and Hamilton, J. (2007). Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259.

Kemme, B. and Alonso, G. (2000). A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379.

Kemme, B. and Alonso, G. (2010). Database replication: A tale of research across communities. *The Proceedings of the VLDB Endowment*, 3(1):5–12.

Leavitt, N. (2010). Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146.

Marandi, P. J., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 527–536.

Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., and Alonso, G. (2000). Scalable replication in database clusters. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 315–329.

Pedone, F., Guerraoui, R., and Schiper, A. (2003). The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98.

Pritchett, D. (2008). BASE: An ACID Alternative. *ACM Queue*, 6(3):48–55.

Rabuske, M. A. (1992). *Introdução à Teoria dos Grafos*. Editora da UFSC, Florianópolis, SC, Brasil.

Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2 edition.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Stonebraker, M. (2010). SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. (2007). The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1150–1160.