FLEXMVCC: Uma abordagem flexível para protocolos de controle de concorrência multi-versão

Eder C. M. Gomes[1], **J. Filipe L. de Sousa**[1],
Paulo R. P. Amora[1], **Javam C. Machado**[1]

[1]Laboratório de Sistemas e Bancos de Dados (LSBD)
DC/UFC – CEP 60440-900 – Fortaleza – CE – Brazil

{eder.clayton,filipe.lobo,paulo.amora,javam.machado}@lsbd.ufc.br

Abstract. *Different concurrency control protocols impact the performance of database systems depending on the workload profile. Without prior knowledge of this workload and its changes, the decision on which protocol to use becomes challenging. To alleviate this impact, we created FLEXMVCC, which integrates two compatible multi-version concurrency control protocols, an optimistic one and a pessimistic one, able to adapt as the workload changes. Preliminary experiments show that the exchange between protocols is feasible and results in a performance gain over static protocols.*

Resumo. *Diferentes protocolos de controle de concorrência impactam o desempenho do banco de dados, dependendo do perfil da carga de trabalho. Sem conhecimento prévio desta carga e suas mudanças, a decisão na escolha de um protocolo se torna desafiadora. Para tanto, criamos o FLEXMVCC, que integra dois protocolos de controle de concorrência multi-versão compatíveis, um otimista e um pessimista, capaz de se adaptar conforme as mudanças na carga de trabalho. Experimentos preliminares mostram que a troca entre protocolos é viável e representa um ganho de desempenho em relação aos protocolos estáticos.*

1. Introdução

Com o aumento do número de núcleos nos processadores atuais e a queda de preço das memórias RAM no mercado é cada vez mais viável um banco de dados ser armazenado inteiramente em memória e consequentemente o controle de concorrência em um sistema de banco de dados se torna um gargalo por que o processamento de consulta em memória é tão rápido quanto a execução do protocolo [Harizopoulos et al. 2008, Yu 2015].

Existem diversos controles de concorrência para inúmeras situações de acesso ao dado, por exemplo, cargas de trabalho OLTP e OLAP, sendo que cada controle de concorrência vai ter um melhor desempenho para um conjunto dessas situações e infelizmente não há um controle de concorrência que tenha o melhor desempenho para todas as situações de acesso ao dado. Cabe então ao administrador do banco de dados escolher, de acordo com o acesso aos dados, a melhor alternativa dentre os controles de concorrência disponíveis. Mas nem sempre o acesso aos dados segue um padrão único, podendo diversificar ao decorrer do tempo do banco de dados.

Tenhamos como exemplo um supermercado, no qual corriqueiramente há atualizações em seus dados (por exemplo venda de mercadoria) e leitura dos dados (por exemplo

verificação de estoque) e o administrador do banco de dados escolhe com base nessa carga de trabalho padronizada a melhor configuração do banco de dados. Sabemos que o mundo é dinâmico, há mudanças no ambiente que geram novos acessos aos dados, por exemplo, o evento *Black Friday* que aumenta as atualizações nos dados e auditoria na empresa que aumenta as leituras nos dados para geração de relatórios e em nenhuma dessas situações o banco de dados pode ter um desempenho abaixo do esperado, mas como diverge do acesso padrão estabelecido pelo administrador do banco de dados, acaba tendo perda de desempenho.

Pensando nisso, criamos um gerenciador de transações para controle de concorrência multi-versão (MVCC) chamado FLEXMVCC que permite uma execução flexível e correta entre os protocolos otimista para multi-versão (MVOCC) [Larson et al. 2011] e *two-phase locking* (MV2PL) [Larson et al. 2011]. Também desenvolvemos um método geral de reconfiguração de protocolo para suportar as mudanças online entre os protocolos. A essência desta abordagem é que no período de transição de um protocolo ao outro é criado um protocolo compatível com os protocolos antigo e novo, de modo que o processo de troca não precise interromper todos as transações, ao mesmo tempo em que minimiza o impacto na taxa de transferência e latência.

2. Trabalhos Relacionados

Alguns projetos propõem novos protocolos de controles de concorrência otimizados para bancos de dados de memória principal, como Doppel [Narula et al. 2014] propõe um protocolo escalável de controle de concorrência para operações comutativas sob cargas de trabalho de alta contenção. Tictoc [Yu et al. 2016] extrai mais concorrência ao avaliar os *timestamps* das transações. Hekaton [Diaconu et al. 2013, Larson et al. 2011] melhora o desempenho dos protocolos 2PL e OCC usando tabelas *hash* sem múltiplas versões para bancos de dados de memória principal. Silo [Tu et al. 2013] desenvolve um protocolo OCC que atinge alto desempenho evitando todos os pontos de contenção centralizados entre núcleos de CPU. Nesses trabalhos, o foco principal é no desempenho quando há um hardware com um grande número de núcleos de processamento e novos hardwares com uma alta taxa de I/O, mas não se atentam em ter uma boa performance em situações mistas.

Outros trabalhos exploram a combinação de protocolos de controle de concorrência em um sistema de banco de dados, como no trabalho [Xie et al. 2015] e na continuação do seu trabalho [Su et al. 2017] em que é fornecido uma modularização apresentando um protocolo de controle de concorrência para cada grupo com base em uma análise da carga de trabalho off-line. Para resolução de conflitos transacionais entre grupos é apresentado um protocolo base para resolve-lo. Embora a atribuição de protocolo orientada ao procedimento armazenado possa processar conflitos dentro do mesmo grupo de forma mais eficiente, a sobrecarga de controle de concorrência adicional de executar protocolos entre grupos pode se tornar um gargalo de desempenho para um banco de dados de memória principal. Além disso, o agrupamento baseado na análise off-line assume que os conflitos de carga de trabalho são conhecidos por conta própria, o que pode não ser verdade em aplicativos reais.

Um outro trabalho que mistura protocolos é o [Wang and Kimura 2016]. Esse trabalho consiste na criação do protocolo *Mostly-optimistic concurrency control* (MOCC)

com base no OCC e com a adição de bloqueios similares ao 2PL, mas com modificações capazes de melhorar o desempenho do protocolo para equipamentos com milhares de núcleos, mas para cargas de trabalho de baixo conflito, o MOCC mantém o desempenho padrão do protocolo OCC não se preocupando na diversidade da carga de trabalho abordada no nosso trabalho. O [Shang et al. 2016] também mistura os protocolos OCC e 2PL, mas se restringe a melhorar o desempenho de conjuntos de dados gráficos.

3. MVCC - Controle de concorrência multi-versão

De acordo com [Yu 2015] o esquema mais popular usado nos SGBDs desenvolvidos na última década é o controle de concorrência multi-versão (MVCC). A principal diferença desse esquema para esquemas com uma versão é que o SGBD mantém várias versões físicas de cada objeto lógico no banco de dados para permitir que as operações no mesmo objeto continuem em paralelo, dessa forma, as transações somente leitura acessam versões mais antigas de tuplas, sem impedir que as transações de leitura e gravação gerem versões mais novas simultaneamente. [Yu 2015] continua expondo que para se obter as multi-versões é necessário que para cada escrita em uma tupla gere uma nova versão daquela tupla.

Independente do protocolo implementado, há metadados comuns, tanto para tuplas como para transações. Nas transações, é atribuído um *timestamp*, como identificador único (Tid) no SGBD, quando a transação é iniciada no sistema. Os protocolos de controle de concorrência usam esse identificador para marcar quais versões de tupla essa transação acessa. As tuplas são compostas por uma ou mais versões e em cada versão há 4 campos de metadados além do conteúdo da tupla que as diferencia.

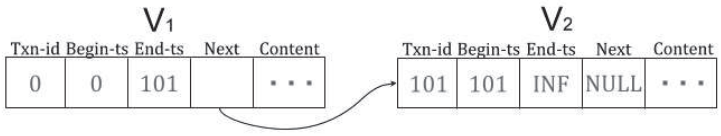

Figura 1. Exemplo de versões de uma tupla e seus metadados

O primeiro campo é o *txn-id* que tem como objetivo ser o bloqueio de escrita para a versão. As versões tem o valor zero nesse campo quando não há bloqueios de escrita. Se uma transação com identificador T_{id} quiser atualizar uma versão de uma tupla X, o SGBD verifica se o campo *txn-id* é zero, caso positivo o SGBD irá definir o valor de *txn-id* para T_{id}. Na Figura 1 temos um exemplo de duas versões, a V1 sem bloqueio e a V2 com bloqueio de escrita. Os próximos dois campos são os *timestamps* de começo e final que representam o tempo de vida da versão da tupla. O SGBD define quais tuplas são visíveis a uma transação com identificador T_{id} verificando se o T_{id} está entre esses campos. As tuplas são inicializadas com zero no *begin-ts* e *INF* no *end-ts*. Na Figura 1 temos um exemplo de duas versões, a V1 visível para transações com *timestamp* entre 0 e 100 e V2 visível para transações a partir de 101. O último campo chamado de *Next* armazena o endereço para a próxima versão da tupla, formando assim uma lista encadeada de versões.

3.1. MVOCC - Otimista

O MVOCC [Larson et al. 2011] é um protocolo otimista para multi-versão baseado no protocolo otimista para uma versão proposto em 1981 [Kung and Robinson 1981]. A

motivação por trás do protocolo otimista é que assume-se que a probabilidade de que as transações conflitem é miníma, dessa forma as transações são liberadas para efetuar suas operações, sem se importar em adquirir bloqueios de leitura e escrita e na etapa de confirmação é verificado se as transações podem confirmar suas operações, caso elas não possam, a transação é abortada.

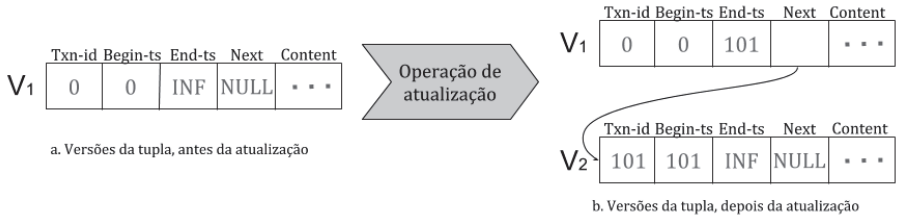

Figura 2. Operação de atualização em um tupla usando o protocolo MVOCC

Esse protocolo não precisa de adição de metadados nas e divide uma transação em três fases, a execução, a validação e a confirmação. Na fase de execução é onde acontece as operações de leitura e escrita da transação. A operação de leitura consiste em procurar a versão visível mais recente com base nos campos *begin-ts* e *end-ts*, e efetua a leitura do conteúdo dessa tupla. A operação de escrita consiste em procurar a versão visível mais recente e criar uma nova versão com base nela, mas só é permitido a escrita caso o *Txn-id* dessa versão seja igual a zero. Quando é necessário confirmar, entra na fase de validação e é atribuído a um novo *timestamp* chamado T_{commit} para determinar a ordem das transações. O SGBD então determina se as tuplas no conjunto de leitura da transação foram atualizadas por uma transação já confirmada, caso positivo a transação é abortada. Passando por essas verificações, a transação entra na fase de gravação na qual o SGBD instala todas as novas versões e define os campos *begin-ts* para T_{commit} e *end-ts* para *INF*.

3.2. MV2PL - Bloqueio em duas fases

O MV2PL [Larson et al. 2011] é um protocolo pessimista para multi-versão baseado no protocolo *two-phase lock* para uma versão [Bernstein et al. 1987]. Diferente do protocolo MVOCC, a motivação para esse protocolo é que assume-se que a probabilidade de que as transações conflitem é alta, dessa forma para se efetuar uma operação é necessário adquirir o bloqueio pela versão requisitada. Nesse protocolo existem dois tipos de bloqueios, o bloqueio de escrita e o bloqueio de leitura compartilhada. Já existe um campo nos metadados já explicados para o bloqueio de escrita, o campo *txn-id*, mas para leitura não há, portanto é necessário criar um campo extra, com nome de *read-cnt*, como visto na Figura 3. Esse novo campo serve para ser um contador de leituras para uma versão especifica.

Esse protocolo divide a transação em duas fases, a fase de expansão e a fase de recolhimento. Na fase de expansão é onde acontece a obtenção dos bloqueios de leitura e escrita e na fase de recolhimento acontece a liberação dos bloqueios e a confirmação das operações feitas pela transação. Na operação de leitura, o SGBD busca a versão da tupla visível mais recente e verifica se não há bloqueios de escrita verificando se o campo *txn-id* é igual a zero, depois é incrementado o valor do campo *read-cnt*, assim como está representado na transição das Figuras 3a para 3c. Na operação de escrita, o SGBD busca

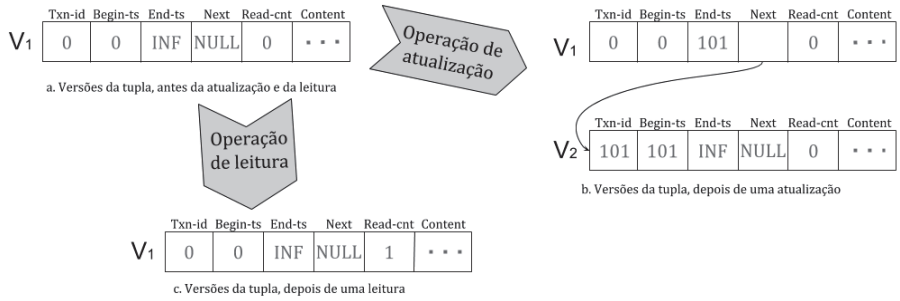

Figura 3. Operação de atualização e leitura em um tupla usando o protocolo MV2PL

a versão da tupla visível mais recente e verifica se há bloqueios de escrita ou leitura, verificando se os campos *txn-id* e *read-cnt* estão com valor zero, se estiverem livres de bloqueio é criado uma nova versão atualizada, assim como está representado na transição das Figuras 3a para 3b. Na confirmação da transação, é gerado um T_{commit} que é usado para atualizar o campo *begin-ts* para as versões criadas por essa transação e, em seguida, libera todos os bloqueios da transação.

4. Arquitetura para gerenciadores de transações

Quando se tem mais de um protocolo de controle de concorrência para ser gerenciado, a arquitetura padrão de um gerenciador de transações não é mais satisfatório, ou seja, todas as transações usarem apenas um protocolo para acessar todos os registros não é mais o suficiente. Em [Tanger 2017] são criadas 4 alternativas de arquitetura para SGBDs que usam mais de um controle de concorrência, que podem ser visualizados na Figura 4. Discutiremos agora as desvantagens com relação ao nosso trabalho dentre as possibilidades expostas.

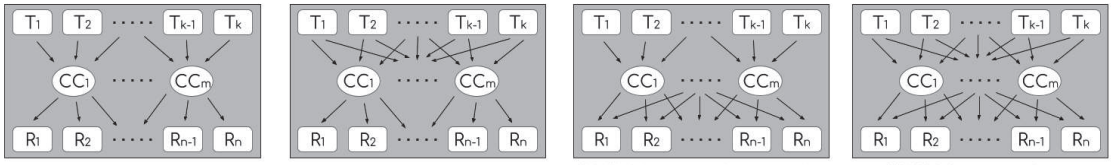

Figura 4. Arquiteturas para controle de concorrência misto. Fonte (adaptado): [Tanger 2017]

A primeira arquitetura é a mais simples e pode ser encontrado na Figura 4a. Abordamos aqui uma arquitetura em que cada transação (denotado por T) só pode escolher um protocolo de controle de concorrência (denotado por CC) e cada registro (denotado por R) pode ser acessado através de apenas um protocolo. Na Figura 4b temos uma outra arquitetura que consiste em qualquer transação pode escolher um ou mais protocolos ao decorrer de sua existência e cada registro pode ser acessado apenas por um protocolo. Continuando na Figura 4c, cada transação pode escolher apenas um protocolo de controle de concorrência e os registros podem ser acessados por qualquer controle de concorrência. A última arquitetura, encontrado na Figura 4d, é a abordagem que temos que qualquer transação

pode escolher um ou mais protocolos ao decorrer de sua existência e cada protocolo pode acessar qualquer registro.

Nas arquiteturas das Figuras 4a e 4b precisamos de um estudo sobre os dados antes da execução do banco de dados, pois para usarmos essas arquiteturas, com tuplas sendo acessadas por apenas um protocolo, seria necessário manter dados separados de acordo com o protocolo de controle de concorrência e para se atingir um bom desempenho, cada protocolo deve ser o melhor para o seu conjunto de dados. Como o nosso trabalho pretende não ter a necessidade de se ter um estudo prévio dos dados, essas duas arquiteturas foram descartadas.

Nas arquiteturas das Figuras 4b e 4d, temos que as transações podem escolher um ou mais protocolos ao decorrer de suas existências. Essa troca de protocolos abre possibilidades de tratamento para um maior número de casos específicos durante a troca de protocolos em uma mesma transação, aumentando assim a sobrecarga no banco. Portanto, essa sobrecarga pode se tornar grande o suficiente e consequentemente penalizando nossa estratégia.

Por fim, ficamos satisfeitos com a arquitetura da Figura 4c, pois os dados estão disponíveis para qualquer protocolo de controle de concorrência e as transações iniciam e finalizam sem mudanças de protocolo. Com alguns ajustes no gerenciamento dos metadados existentes em protocolos MVCC, podemos conseguir com que duas ou mais transações usando protocolos diferentes possam ser gerenciadas pelo SGBD.

5. FLEXMVCC

O FLEXMVCC tem como proposta ser um controle de concorrência multi-versão flexível a mudanças entre os protocolos MVOCC e MV2PL.

O principal objetivo da nossa proposta é que quando a carga de trabalho for melhor para o protocolo MVOCC o FLEXMVCC escolha-o como protocolo ativo no SGBD, e quando a carga de trabalho for melhor para o protocolo MV2PL o FLEXMVCC escolha-o. Com isso a tendência é que o banco de dados ganhe desempenho em cargas de trabalho mista, ganhando as vantagens que cada protocolo pode oferecer para cargas de trabalho especificas.

Para que não haja perda no desempenho entre a troca de protocolos, ou seja, para que o SGBD não espere todas as transações de um protocolo finalizarem para começar as transações do outro protocolo iniciarem, foi necessário criar um período de transição entre a troca de protocolo, para que os dois protocolos convivam no SGBD ao mesmo tempo até que todas as transações que estão executando no protocolo antigo finalizem.

Estudando os protocolos, percebemos que o bloqueio de escrita do MV2PL já está presente no MVOCC, por meio do campo *txn-id*, mas o bloqueio de leitura não é implementado no MVOCC. Levando isso em consideração adicionamos o campo *read-cnt* no protocolo MVOCC e pequenas alterações na operação de leitura e confirmação do protocolo MVOCC.

Na operação de leitura, como no MV2PL, adicionamos o incremento no campo *read-cnt* com intuito de que o SGBD saiba esse valor atualizado quando as transações MV2PL precisem verificar se certa versão está livre para escrita ou não. Diferente do

MV2PL não é necessário que transações MVOCC verifiquem esse campo, apenas escrevam.

Na operação de confirmação do protocolo MVOCC se torna necessário decrementar o campo *read-cnt* de todas as versões no conjunto de leituras da transação que necessita confirmar, ainda pelo motivo da compatibilidade com o MV2PL. O protocolo MVOCC já mantém esse conjunto de leituras e na operação de confirmação já verifica todas as versões lidas por uma transação para ter a certeza que não seja executada uma leitura suja pelas transações.

A sobrecarga do incremento e do decremento nesse campo no protocolo MVOCC é miníma se comparada ao ganho em ter melhor desempenho unindo os dois protocolos em uma carga de trabalho mista.

Para níveis mais baixos de isolamento é seguido a implementação discutida no trabalho [Larson et al. 2011] que tivemos como base de implementação dos protocolos MVOCC e MV2PL.

5.1. Corretude

Vamos agora provar que o *scheduler* do FLEXMVCC só admite históricos multi-versão 1SR. Usamos como base a abordagem descrita em [Larson et al. 2011] que prova a corretude do MVOCC, bem como a seção 5.5.2 do [Weikum and Vossen 2001] que descreve o MV2PL de maneira mais detalhada e prova que admite apenas históricos multi-versão 1SR. Na nossa prova são usadas as notações e os teoremas da Seção 5.2 de [Bernstein et al. 1987] que são definidos para protocolos multi-versão.

O *scheduler* do FLEXMVCC se comporta como um *scheduler* MVOCC, MV2PL e um período de transição entre os dois protocolos. A corretude do MVOCC e do MV2PL já é abordado no trabalho [Larson et al. 2011].

Em [Larson et al. 2011] é definido que a transação T_x é uma transação confirmada com um timestamp inicial chamado T_{xBegin} e um timestamp final chamado T_{xEnd} e para transações do MVOCC são expostas as seguintes propriedades:

Propriedade 1: Os *timestamps* são designados em ordem crescente e monotonicamente, e cada transação possui um *timestamp* exclusivo de início e fim, de forma que $T_{xBegin} < T_{xEnd}$.

Propriedade 2: Uma determinada versão é válida para o intervalo especificado pelos *timestamps* de início e fim. Existe uma ordem total de versões, denotado por $<<$, para uma determinada tupla, conforme determinado pela ordem dos *timestamps* dos intervalos de validade da versão não sobreposta.

Propriedade 3: A transação T_x lê a última versão confirmada chamada T_{xRead} (onde $T_{xBegin} <= T_{xRead} < T_{xEnd}$) e valida (ou seja, repete) a leitura da última versão confirmada a partir de T_{xEnd}. A transação falhará se as duas leituras retornarem versões diferentes.

Propriedade 4: Primeiro verificar a visibilidade de uma versão V, para atualizar ou apagar V. Verificar a visibilidade de V é equivalente a leitura de V. Portanto, uma gravação é sempre precedida por uma leitura: se a transação T_x gravar V_{new}, a transação T_x terá que ler primeiro o V_{old}, onde V_{old} « V_{new}. Além disso, não existe nenhuma versão

V tal que $V_{old} << V << V_{new}$, caso contrário T_x nunca teria confirmado.

Propriedade 5: A transação T_x grava logicamente no *timestamp* T_{xEnd}, porque a versão é invisível para outras transações até o *timestamp* T_{xEnd}.

Com base em [Weikum and Vossen 2001] expomos as propriedades para transações MV2PL:

Propriedade 6: Dado uma transação T_x e uma T_y, T_x só poderá ler uma versão V, caso a transação T_y que criou a versão V já tenha confirmado.

Propriedade 7: Verificar a visibilidade de uma versão V primeiro, para atualizar ou apagar V. Verificar a visibilidade de V é equivalente a leitura de V. Portanto, uma gravação é sempre precedida por uma leitura: se a transação T_x gravar V_{new}, a transação T_x terá que ler primeiro o V_{old}, onde $V_{old} << V_{new}$. Além disso, não existe nenhuma versão V tal que $V_{old} << V << V_{new}$, caso contrário T_x nunca teria escrito V_{new}.

Propriedade 8: Operações de leitura são sempre direcionadas à versão atual, e bloqueio de confirmação de escritores concorrentes, que são incompatíveis com os bloqueios de leitura, servem para determinar situações na qual uma nova versão confirmada é produzida enquanto um leitor ainda está em andamento. Nesse caso, a confirmação do escritor é postergada até que o leitor tenha terminado.

As propriedades **1**, **2** e **5** também são propriedades das transações MV2PL.

O grafo de serialização multi-versão MVSG $(H, <<)$ é um grafo definido em um histórico multi-versão H e uma ordem total da versão $<<$. O MVSG possui nós para as transações confirmadas em H e, por definição, existe uma aresta $T_i \rightarrow T_j$ no MVSG (onde i, j, k são distintos), se e somente se:

A) T_i **escreve** V_i **e** T_j **lê** V_i

ou

B) T_i **escreve** V_i **e** T_k **lê** V_j**, onde** $V_i << V_j$

ou

C) T_i **lê** V_k **e** T_j **escreve** V_j**, onde** $V_k << V_j$

Vamos provar que todas as arestas do MVSG são ordenadas com relação à ordem do *timestamp* final das transações MVOCC e MV2PL misturadas. Ou seja, provaremos que qualquer aresta direcionada $T_i \rightarrow T_j$ sempre apontará de uma transação T_i para uma transação T_j tal que $T_{iEnd} < T_{jEnd}$.

Seja, T_o uma transação MVOCC que gera versões V_o e T_p uma transação MV2PL que gera versão V_p. Temos que:

A.1) T_p **escreve** V_p **e** T_o **lê** V_p**:**

Para (A.1), deixe T_o ler V_p em $T_{oRead} < T_{oEnd}$. Se $T_{oRead} < T_{pEnd}$, das propriedades 3 e 5, teria sido impossível ler V_p, como não é confirmado. Portanto, $T_{pEnd} < T_{oRead} < T_{oEnd}$, então, a aresta $T_p \rightarrow T_o$ é ordenada com relação ao *timestamp* final, como $T_{pEnd} < T_{oEnd}$.

A.2) T_o **escreve** V_o **e** T_p **lê** V_o

Para (A.2), a mesma discussão em (A.1) é valida, com a troca da propriedade 3 exclusiva do MVOCC pela propriedade 6 exclusiva do MV2PL.

B.1) T_o **escreve** V_o **e** T_k **lê** V_p**, onde** $V_o << V_p$

B.1.1) se T_k **for uma transação MVOCC:**

Para (B.1.1), T_k lê V_p, portanto, a leitura é precedida por T_p escrevendo V_p e confirmando. Além disso, da propriedade 3, $T_{pEnd} < T_{kRead}$ ou V_p não seria visível no T_{kRead}. Da propriedade 7, desde que T_p escreveu V_p, isso significa que T_p leu V_o (ou qualquer versão posterior) em T_{pRead}, onde $T_{pRead} < T_{pEnd}$. Portanto, a partir de (A.2), $T_{oEnd} < T_{pRead} < T_{pEnd}$, a aresta $T_o \rightarrow T_p$ é ordenada com relação ao *timestamp* final, como $T_{oEnd} < T_{pEnd}$.

B.1.2) se T_k **for uma transação MV2PL:**

Para (B.1.2), a mesma discussão em (B.1.1) é valida, com a troca da propriedade 3 exclusiva do MVOCC pela propriedade 6 exclusiva do MV2PL.

B.2) T_p **escreve** V_p **e** T_k **lê** V_o**, onde** $V_p << V_o$

B.2.1) se T_k **for uma transação MVOCC:**

Para (B.2.1), a mesma discussão em (B.1.1) é valida, com a troca da propriedade 7 exclusiva do MV2PL pela propriedade 4 exclusiva do MVOCC.

B.2.2) se T_k **for uma transação MV2PL:**

Para (B.2.2), a mesma discussão em (B.1.2) é valida, com a troca da propriedade 7 exclusiva do MV2PL pela propriedade 4 exclusiva do MVOCC.

C.1) T_o **lê** V_k **e** T_p **escreve** V_p**, onde** $V_k << V_p$

Para (C.1), deixe a T_o ler V_k no T_{oRead}, onde $T_{oRead} < T_{oEnd}$. T_p escreve V_p no T_{pEnd}. Existem três casos:

C.1.1) $T_{pEnd} < T_{oRead} < T_{oEnd}$. Isso viola a propriedade 3, já que V_p era uma versão confirmada no T_{oRead}, portanto, T_o teria lido V_p, não V_k.

C.1.2) $T_{oRead} < T_{pEnd} < T_{oEnd}$. Isso viola a propriedade 3, pois a T_o repetiria a leitura no T_{oEnd} durante a validação e leria V_p. No entanto, T_o leu V_k na T_{oRead}, portanto, T_o falharia na validação e nunca participaria do MVSG.

C.1.3) $T_{oRead} < T_{oEnd} < T_{pEnd}$. T_o validaria a leitura em T_{oEnd}, leria V_k novamente e confirmaria. Portanto, a aresta $T_o \rightarrow T_p$ é ordenada com relação ao *timestamp* final, como $T_{oEnd} < T_{pEnd}$.

C.2) T_p **lê** V_k **e** T_o **escreve** V_o**, onde** $V_k << V_o$

Para (C.2), deixe a T_p ler V_k no T_{pRead}, onde $T_{pRead} < T_{pEnd}$. T_o escreve V_o no T_{oEnd}. Existem três casos:

C.2.1) a mesma discussão em (C.1.1) é valida, com a troca da propriedade 3 exclusiva do MVOCC pela propriedade 6 exclusiva do MV2PL.

C.2.2) $T_{pRead} < T_{oEnd} < T_{pEnd}$. Isso viola a propriedade 6 e 8, pois A T_p leria a versão V_k e pela a propriedade 8 T_o não poderia confirmar suas alterações até que o T_p finalizasse, portanto T_o falharia na confirmação e nunca participaria do MVSG.

C.2.3) a mesma discussão em (C.1.3) é valida aqui.

Portanto, de (A) e (B), todas as arestas no MVSG são ordenadas com relação à ordem do *timestamp* final da transação e, portanto (da Propriedade 1), elas não podem ser envolvidas em um ciclo. Daí se conclui que os históricos de MV que são aceitos pelo nosso *scheduler* de controle de concorrência FLEXMVCC são 1SR.

6. Experimentação

Apresentamos agora nossa análise experimental do protocolo de controle de concorrência discutido neste artigo. O protocolo proposto foi implementado no SGBD Peloton [Pavlo et al. 2017], que armazena tuplas em heaps em memória não ordenados, orientados a linhas. O Peloton, como um SGBD em memória de código aberto, foi escolhido por utilizar controle de concorrência multi-versão. Nele, implementamos o FLEXMVCC, além dos protocolos MVOCC e MV2PL. As transações foram executadas sob o nível de isolamento SERIALIZABLE.

6.1. Configuração

O build do Peloton foi feito em uma máquina Intel Core i7-7800X com 6 núcleos e 12 threads rodando a uma frequência base de 3.5GHz e 128GB de memória SDRAM DDR4, com sistema operacional Ubuntu 16.04.3.

Utilizamos o benchmark YCSB [Cooper et al. 2010] em nossa experimentação por sua flexibilidade ao modelar diferentes configurações de workloads de aplicações OLTP. O framework OLTPBench [Difallah et al. 2013] foi utilizado na experimentação por sua facilidade na configuração e execução do YCSB, além da coleta de informações como latência e vazão por tipo de transação. O banco de dados contém uma única tabela com 50 mil tuplas, cada uma com 11 atributos inteiros de 64 bits.

6.2. Cargas de trabalho

Selecionamos quatro cargas de trabalho para variar o número de transações com operações de read e update: (1) 80% leituras, 20% updates, (2) 60% leituras, 40% updates, (3) 40% leituras, 60% updates e (4) 20% leituras, 80% updates. As cargas de trabalho citadas foram selecionadas devido ao favorecimento do protocolo MVOCC em relação à operações de leitura e ao MV2PL em relação a operações de escrita. Estas cargas foram executadas em sequência, em cada protocolo: MVOCC, MV2PL e FLEXMVCC. O benchmark foi configurado para utilizar 50 threads concorrentes onde cada transação executa 10 operações. Esta modificação foi feita para aumentar o tempo de duração de cada transação, consequentemente, auxiliando a avaliação dos respectivos protocolos de controle de concorrência.

Nos nossos experimentos, a carga (1) apresenta maior vazão ao ser executada pelo MVOCC, a carga (2) pelo MV2PL e assim sucessivamente. O FLEXMVCC, durante essa execução sequencial, foi modificado para trocar o protocolo dependendo da configuração da carga, de forma que as transações executadas utilizariam o protocolo mais otimizado para o tipo de carga no momento.

6.3. Resultados Experimentais

A Figura 5 mostra o resultado da execução do experimento, contendo as médias de vazão das quatro cargas de trabalho em cada protocolo. Observamos que o FLEXMVCC

obtém maior vazão do que os outros protocolos, devido à sua flexibilidade em alternar os protocolos conforme a mudança no perfil da carga de trabalho.

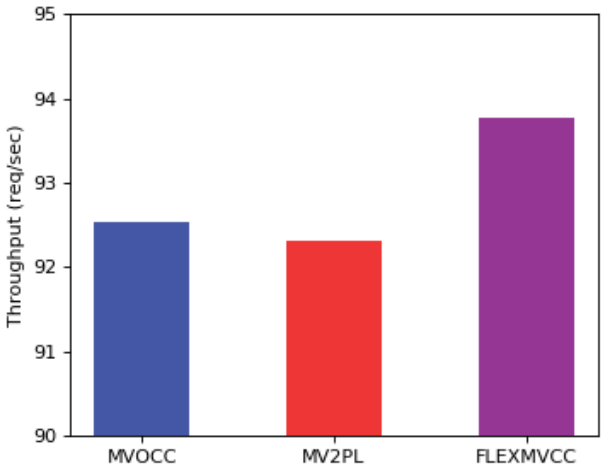

Figura 5. Média da vazão dos protocolos executando as cargas (1), (2), (3) e (4) sequencialmente

7. Conclusão

Este trabalho apresenta o FLEXMVCC, um protocolo de controle de concorrência que une dois protocolos de controle de concorrência multi-versão compatíveis entre si, o MVOCC e o MV2PL. Alternando entre os dois, o FLEXMVCC consegue se adaptar às mudanças de perfil na carga de trabalho, conferindo um aumento no desempenho do banco de dados, evidenciado pelo acréscimo da vazão observado por meio de benchmarks. Também provamos a corretude do FLEXMVCC, principalmente no período de transição entre o MVOCC e o MV2PL, característica essencial quando é proposto um novo mecanismo de controle de concorrência.

Agradecimentos

Esta pesquisa foi apoiada pelo Laboratório de Sistemas e Banco de Dados e FUNCAP (Bolsa BMD-0008-01237.01.09/17).

Referências

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). Concurrency control and recovery in database systems.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). 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. ACM.

Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. (2013). Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM.

Difallah, D. E., Pavlo, A., Curino, C., and Cudre-Mauroux, P. (2013). Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288.

Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. (2008). Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM.

Kung, H.-T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226.

Larson, P.-Å., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., and Zwilling, M. (2011). High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309.

Narula, N., Cutler, C., Kohler, E., and Morris, R. (2014). Phase reconciliation for contended in-memory transactions. In *OSDI*, volume 14, pages 511–524.

Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T. C., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Aken, D. V., Wang, Z., Wu, Y., Xian, R., and Zhang, T. (2017). Self-driving database management systems. In *CIDR*. www.cidrdb.org.

Shang, Z., Li, F., Yu, J. X., Zhang, Z., and Cheng, H. (2016). Graph analytics through fine-grained parallelism. In *Proceedings of the 2016 International Conference on Management of Data*, pages 463–478. ACM.

Su, C., Crooks, N., Ding, C., Alvisi, L., and Xie, C. (2017). Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 283–297. ACM.

Tanger, D. (2017). Toward coordination-free and reconfigurable mixed concorrency control. Master's thesis, University of Chicago.

Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. (2013). Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM.

Wang, T. and Kimura, H. (2016). Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60.

Weikum, G. and Vossen, G. (2001). *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.

Xie, C., Su, C., Littley, C., Alvisi, L., Kapritsos, M., and Wang, Y. (2015). High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 279–294. ACM.

Yu, X. (2015). *An evaluation of concurrency control with one thousand cores*. PhD thesis, Massachusetts Institute of Technology.

Yu, X., Pavlo, A., Sanchez, D., and Devadas, S. (2016). Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642. ACM.