MRV*: Uma biblioteca de tipos de dados para aplicações concorrentes*

Adriano Maior[0009−0006−0245−8551], Nuno Faria[0000−0003−4691−0440] e José Pereira[0000−0002−3341−9217]

Universidade do Minho & INESCTEC

Resumo A existência de itens que são acedidos e modificados com grande frequência em sistemas de gestão de dados é um obstáculo significativo à obtenção de elevado desempenho. Assim, torna-se cada vez mais relevante encontrar métodos de lidar com a execução simultânea de operações que manipulam dados de forma eficiente, proporcionando às aplicações melhores tempos de resposta. No entanto, as abordagens existentes não permitem que se garantam invariantes sobre os dados, têm pressupostos estritos sobre a concorrência, ou aplicam-se apenas a valores numéricos. Este trabalho tem como objetivo a construção de uma biblioteca de estruturas de dados que toleram elevada concorrência através da partição e aleatoriedade, tal como proposto para os MRVs e adequado a sistemas distribuídos, mas oferecendo uma gama de estruturas mais diversa, para suportar diferentes aplicações. Esta proposta é avaliada com uma implementação em sistemas de bases de dados SQL.

Palavras chave: Programação concorrente · Sistemas de bases de dados.

1 Introdução

O isolamento transacional é essencial para assegurar integridade de dados nos sistemas de bases de dados. No entanto, estes sistemas podem necessitar de sacrificar desempenho de modo a garantir o isolamento quando processos concorrentes manipulam os dados armazenados. Assim, a existência de registos que são acedidos e modificados com muita frequência tornam-se num obstáculo significativo à obtenção de alto desempenho nas aplicações, tendo sido propostas múltiplas técnicas ao longo do tempo [9,10,6,7,1,16]. No contexto de sistemas distribuídos, em que há elevada concorrência e é necessário garantir a coerência entre múltiplos nós, este problema é ainda mais grave, uma vez que muitos dos métodos tradicionais não são aplicáveis ou eficientes.

* Este trabalho é cofinanciado pela Componente 5 - Capitalização e Inovação Empresarial, integrada na Dimensão Resiliência do Plano de Recuperação e Resiliência no âmbito do Mecanismo de Recuperação e Resiliência (MRR) da União Europeia (EU), enquadrado no Next Generation UE, para o período de 2021 - 2026, no âmbito do projeto ATE, com a referência 56.

É pois relevante encontrar métodos que permitam lidar com concorrência de forma eficiente de modo a garantir bom desempenho dos sistemas. Uma proposta recente, os Multi-Record Values (MRVs)[4], resolvem este problema em bases de dados SQL: Em resumo, esta técnica divide um valor por várias variáveis e usa aleatoriedade para atribuir cada operação a uma delas, de forma a que possam executar concorrentemente sem interferência. Esta técnica foi no entanto proposta apenas para valores numéricos, explorando a associatividade da adição e da subtração. Outras técnicas, como Phase Reconciliation [8], suportam um conjunto alargado de estruturas de dados, mas não se adequam a sistemas de bases de dados SQL. Nomeadamente, exigem o conhecimento prévio e atribuição de uma variável a cada uma das atividades concorrentes (que se assume serem núcleos do processador) existentes, que num sistema SQL variam com as ligações de clientes. Além disso, exigem a transição entre diferentes representações físicas dos dados, o que tem impacto na escrita e otimização das interrogações.

Neste trabalho, propomos uma biblioteca de estruturas de dados que exploram o mesmo princípio dos MRVs, no contexto de sistemas de gestão de bases de dados SQL, mas que oferecem uma variedade de estruturas de dados que permitem suportar outras aplicações [8].

O resto deste texto está organizado da seguinte forma: na Secção 2, descrevemos o funcionamento dos MRVs; Na Secção 3, introduzimos a biblioteca MRV* proposta; Na Secção 4, fazemos uma avaliação experimental das estruturas propostas, em comparação com os dados em tabelas relacionais simples; E finalmente, discutimos o trabalho relacionado na Secção 5 e apresentamos as conclusões na Secção 6.

2 Contexto

2.1 Estrutura de Dados

MRVs [4] são uma técnica que permite maior paralelismo em sistemas SQL através da aleatoriedade e da partição de dados. MRVs fazem uso da partição de um valor de uma dada coluna em múltiplos registos, utilizando uma tabela auxiliar e intercetando operações feitas na tabela original para que sejam aplicadas na nova tabela. Durante a execução, o número de registos desta tabela poderá ter de ser ajustado e balanceado consoante a carga atual. Deste modo, MRVs fazem uso de tarefas em segundo plano que ficam responsáveis por este processo. Atualmente, os MRVs são utilizados exclusivamente para valores numéricos.

Para se poder utilizar MRVs, por cada coluna que precisa de ser transformada, é necessário criar uma nova tabela com uma relação de 1 para n com a tabela original, com as chaves primárias da tabela original como chave estrangeira para cada registo correspondente. De seguida, o valor original é dividido pelos múltiplos registos na nova tabela identificados por um inteiro único entre $[0, N-1]$, onde N é o número máximo de registos considerado por um dado valor. Deste modo, o valor original deixa de ser armazenado na tabela original, mas calculado com base nos registos da tabela auxiliar com a chave estrangeira correspondente, através de uma agregação.

2.2 Operações

MRVs dispõem de operações de leitura, escrita, teste de condição (garante um limite inferior), adição e subtração para valores numéricos. A principal vantagem dos MRVs é que as operações de adição, subtração, e teste de condição não causam conflitos umas com as outras se consultarem registos diferentes, o que significa que elas podem ser combinadas ou executadas de forma concorrente.

Para ler um valor, a tabela auxiliar precisa de ser percorrida e todos os registos com a chave estrangeira correspondente têm que ser somados. Esta operação causa conflitos com todas as outras que estiverem a atualizar o mesmo valor, a não ser que o nível de isolamento utilizado seja *Snapshot Isolation* ou semelhante, ou seja, que não adquira *locks* exclusivos para a leitura [2].

Para escrever um valor, todos os registos na nova tabela precisam de ser modificados de modo a que a soma reflita o novo valor. Por exemplo, cada registo pode ser modificado para ficar com a mesma quantidade parcial. Uma vez que é necessário modificar todos os registos, esta operação irá causar conflitos com todas as operações que estejam a modificar o mesmo MRV.

De modo a adicionar uma quantidade q a um dado MRV, é primeiro selecionada uma chave aleatória rk' no intervalo [0, N-1]. De seguida, procura-se o registo com $rk = rk'$ e adiciona-se q ao valor existente. Se não houver nenhum registo com $rk = rk'$, seleciona-se o registo válido que vem imediatamente a seguir a rk'. No caso de não existir nenhum registo após rk', escolhe-se o rk com menor índice, o que significa que os MRVs são representados logicamente por estruturas circulares.

No que toca à subtração de um δ a algum valor v, considerando $v \geq 0$, seleciona-se primeiramente uma chave aleatória rk' e o respetivo registo, tal como para as adições. Se o valor v_i deste registo for maior que δ, então basta substituir v_i por $v_i - \delta$. Caso contrário v_i é substituído por 0 e δ é atualizado para o resultado de $\delta - v_i$, que será depois utilizado para fazer a mesma operação no próximo registo após rk, até $\delta = 0$. Se $\delta > v$ então a tabela será percorrida por completo e a transação acabará por ser revertida no fim, dado que não existe valor necessário para a satisfazer.

A operação de teste de condição verifica um limite inferior para um dado registo. Para executar esta operação, os registos com a chave estrangeira correspondente serão visitados até que a quantidade desejada seja verificada. Esta operação pode ser combinada com a subtração, permitindo que a subtração ocorra apenas se um limite inferior for verificado.

2.3 Exemplo

Consideremos de seguida um produto X, com chave 15, e 20 unidades iniciais, representado pelo MRV da Figura 1a. Em detalhe, este MRV contém 3 registos, com sub-chaves 1, 6, e 8, e valores parciais 5, 5, e 10, respetivamente. Este produto será atualizado por duas transações distintas: T_1 irá comprar 5 unidades a X, enquanto que T_2 irá comprar 10 unidades.

v_x	f_key	rk
5	15	1
5	15	6
10	15	8

(a) Antes de aplicar as operações.

v_x	f_key	rk
0	15	1
0	15	6
5	15	8

(b) Depois de executar as operações.

Figura 1: Exemplo do MRV.

Para realizar uma subtração, cada transação irá selecionar uma chave aleatória para decidir a que registo aceder. Assumindo que T_1 seleciona $rk' = 7$, o registo selecionado corresponderá a $rk = 8$. Dado que o registo escolhido tem 10 unidades e T_1 quer decrementar apenas 5, é apenas necessário calcular $10 - 5$ e atualizar o valor de v_x na linha onde $rk = 8$. Relativamente a T_2, assumindo que este seleciona $rk' = 9$, dado que não existe nenhum $rk \geq rk'$ valido, é selecionado o registo com $rk = 1$. Visto que este registo não tem a quantidade suficiente para satisfazer a operação por completo, subtrai-se a quantidade possível e avança-se para o próximo registo. Neste caso, coloca-se o valor de $rk = 1$ a zero e avança-se para $rk = 6$ com o resto (5 unidades). Visto que agora o registo tem a quantidade necessária, a operação termina depois da atualização. A tabela final ficará com os valores presentes na Figura 1b. É importante salientar que dado que T_1 e T_2 modificam registos diferentes, ambas podem executar concorrentemente sem conflito. No caso de ambas as transações acabarem por modificar o mesmo registo, cabe ao sistema de gestão de base de dados tratar dos conflitos de acordo com o nível de isolamento selecionado.

Para obter o valor total de v_x, basta somar todas as linhas da coluna v_x correspondente. Neste caso, com *Snapshot Isolation*, não há conflito com qualquer outra operação concorrente pelo que é executada eficientemente.

3 Proposta

Com a biblioteca MRV* propomos um conjunto de estruturas de dados comparáveis às disponíveis na proposta de *Phase Reconciliation* [8], de forma a que possam suportar a mesma gama de aplicações concorrentes. A disponibilização destas estruturas de dados num sistema SQL de forma transparente para as aplicações usa a mesma estratégia genérica proposta para os MRVs [4]: Primeiro, propomos um esquema físico para cada estutura, que normalmente recorre a tabelas adicionais. Segundo, definimos vistas que expõem o valor atual de cada estrutura como tabelas lógicas, que podem ser usadas diretamente em interrogações. Terceiro, definimos também regras que permitem intercetar operações de modificação feitas sobres as vistas e as traduzem na atualização das tabelas subjacentes. Finalmente, propomos um utilitário que opera sobre um esquema existente e que cria todas estas tabelas, vistas, e regras de forma semi-automática. O resto desta secção descreve os detalhes para cada uma das estruturas consideradas.

3.1 Max/Min

A estrutura Max/Min permite manter a cada instante o valor máximo/mínimo de todos os que foram inseridos na estrutura. De modo a permitir maior concorrência, em vez de se guardar apenas um registo com o valor máximo/mínimo, temos k registos, sendo k um valor variável. Quando se tenta fazer uma escrita na tabela que contém o valor, a escrita é intercetada e passa a ser aplicada a um dos k registos escolhido de forma aleatória. Quando se pretende fazer uma leitura do valor, cria-se uma vista que calcula o máximo/mínimo dos k registos e apresenta o valor resultante como resposta.

Exemplo de Utilização Num leilão online com múltiplos utilizadores a tentar fazer licitações no mesmo produto, é necessário permitir que os utilizadores possam licitar concorrentemente e, além disso, também é necessário manter o valor da licitação mais alta disponível para consulta. Neste cenário, a estrutura Max seria ideal, uma vez que permite que vários utilizadores façam licitações em simultâneo e também mantém a tabela disponível para consulta.

Consideremos agora um exemplo concreto em que 2 utilizadores, U_1 e U_2, tentam licitar num veículo que está a ser leiloado em simultâneo, num cenário em que o valor inicial para as licitações é de 1500 unidades monetárias, U_1 pretende fazer uma licitação no valor de 2000 unidades monetárias e U_2 pretende fazer uma no valor de 1900 unidades monetárias.

Se não se utilizar esta estrutura e U_1 receber primeiro o *lock* do registo, U_1 vai ler o valor atual, verificar que $2000 > 1500$, atualizar o registo e libertar o *lock*. No entanto, U_2 que também pretende fazer uma licitação só a poderá fazer quando U_1 libertar o *lock* e irá concluir que o valor da licitação atual alterou. Imaginando agora que temos milhares de utilizadores a tentar licitar em simultâneo no veículo, este método irá levar a períodos de espera muito elevados.

Utilizando a estrutura Max, no mesmo cenário, tanto U_1 como U_2 recebem uma chave aleatória rk'_1, rk'_2, respetivamente, e cada um deles irá interagir com registos rk_1 e rk_2 com valores v_1 e v_2, respetivamente. Assim, cada utilizador irá requisitar o *lock* do seu registo e ambos podem fazer as modificações sem existir um período de espera. Se $2000 > v_1$ e $1900 > v_2$, então $v_1 = 2000$ e $v_2 = 1900$. Para fazer a leitura da licitação mais alta atual, basta percorrer todos os registos e selecionar o valor mais alto, sendo que neste caso seria 2000.

3.2 OPut

A estrutura OPut permite guardar o valor mais recente de um dado registo de acordo com o número de ordem. Cada registo é composto por um valor v e por um número de ordem o.

Utilizando MRVs, o funcionamento do OPut é semelhante ao Max, em que temos k registos com o valor e o número de ordem. Para atualizar um valor na tabela, escolhe-se inicialmente um registo aleatório. Se o número de ordem novo for superior ao que estava armazenado, atualiza-se o valor e o número de ordem. Para fazer uma leitura, escolhe-se o valor com o maior número de ordem.

$ordem$	$valor$	f_key	rk
1	8	15	0
2	5	15	1
9	7	15	2
7	12	15	3

(a) Antes de executar as operações.

$ordem$	$valor$	f_key	rk
10	2	15	0
2	5	15	1
9	7	15	2
7	12	15	3

(b) Depois de executar as operações.

Figura 2: Exemplo de MRV OPut.

Exemplo de Utilização Em sistemas distribuídos, existem frequentemente vários processos a fazer escritas concorrentes. Esta estrutura é ideal em cenários em que é necessário guardar apenas o valor mais recente de acordo com alguma ordem.

Consideremos um cenário com 2 processos, P_1 e P_2, a tentar escrever concorrentemente no mesmo registo os pares ordem-valor $(10, 2)$ e $(5, 5)$, respetivamente. O MRV é inicialmente composto por 4 registos, tal como apresentado na Figura 2a. Se P_1 receber a chave aleatória $rk' = 0$ e P_2 $rk' = 2$, ambos os processos poderão prosseguir com a transação sem causar conflitos, visto que irão modificar os registos distintos $rk = 0$ e $rk = 2$, respetivamente.

No caso de P_1, dado que o seu número de ordem é maior que a ordem associada ao registo escolhido $(10 > 1)$, o registo será atualizado para $ordem = 10$ e $valor = 2$. Por sua vez, P_2 tem número de ordem 5 e pretende escrever na linha com $rk = 2$. Neste caso, dado que o número de ordem é menor que o existente $(5 < 9)$, a escrita não irá ocorrer e consequentemente o registo manter-se-á igual. Os dados finais estão representados na Figura 2b. Caso uma transação tente fazer uma leitura, o MRV será percorrido e o valor final será (10, 2), uma vez que é a linha com maior número de ordem.

3.3 Top-K

Uma estrutura de dados Top-K permite guardar os k maiores/menores elementos inseridos de forma ordenada. Para implementar esta estrutura, considerou-se que cada registo deste tipo é composto por um *array* com tamanho k em que o índice de cada elemento representa a sua posição no Top-K. Considerando 2 elementos e_1 e e_2, com índices i_1 e i_2, respetivamente, $e_2 > e_1 \equiv i_2 > i_1$.

Tal como nos casos anteriores, em vez de se manter apenas um registo para o Top-K, existem vários disponíveis e as inserções podem afetar qualquer um. Cada um destes registos mantém um *array* ordenado com até k elementos. Para inserir um elemento e, é necessário descobrir em que posição do Top-K será inserido. Sabendo que o *array* está ordenado, basta percorrer os elementos enquanto o valor de e for superior ao que está guardado na posição do Top-K em questão. No entanto, existem outros algoritmos que permitem determinar a posição onde se deve inserir o novo valor, como através de procura binária.

Quando se tenta inserir um valor v num Top-K que não esteja cheio, o valor será sempre inserido e a ordem de todos os elementos com valor v_e, tal que $v_e \geq v$,

valor	*f_key*	*rk*
[1,2,7,12]	15	0
[1,4,9,14,17]	15	1

(a) Antes de executar as operações.

valor	*f_key*	*rk*
[1,2,7,8,12]	15	0
[1,4,9,14,17]	15	1

(b) Depois de executar T1.

valor	*f_key*	*rk*
[1,2,7,8,12]	15	0
[4,5,9,14,17]	15	1

(c) Depois de executar T1 e T2.

valor	*f_key*
[8,9,12,14,17]	15

(d) Leitura do Top-K.

Figura 3: Exemplo e MRV Top-K.

será incrementada. Se o Top-K estiver cheio, v só será inserido se for superior a pelo menos um dos elementos do Top-K. Caso esta condição se verifique, a inserção será feita na posição correspondente e a ordem dos elementos com valor v_e, tal que $ve \leq v$, será decrementada, exceto para o valor de ordem 1 uma vez que este será removido. Para fazer a leitura do registo, é necessário percorrer todos os registos Top-K com a rk correspondente e selecionar os K maiores elementos.

Exemplo de Utilização Um possível cenário onde esta estrutura demonstra a sua utilidade é no registo do top de produtos mais vendidos de uma loja online. Consideremos o exemplo da Figura 3a em que se pretende manter o top 5. Em detalhe, este MRV apresenta 2 registos, com sub-chaves 0 e 1.

Consideremos ainda três transações, T_1, T_2 e T_3, que pretendem inserir os valores 8, 5, e 3, respetivamente. T_1 seleciona primeiramente a chave aleatória $rk' = 0$, acabando no registo com $rk = 0$. Para inserir 8 neste registo, terá que determinar a posição de inserção, que neste caso será aquela com índice 4, dado que $8 > 7$ e $8 < 12$. Como o Top-K não está cheio, nenhum valor precisa de ser removido, logo basta deslocar o valor 12 para a posição seguinte e inserir 8. O resultado final é apresentado na Figura 3b. Em detalhe, o array $[1, 2, 7, 12]$ foi convertido em $[1, 2, 7, \underline{8}, 12]$.

T_2 seleciona a chave aleatória $rk' = 1$ e tenta inserir o valor 5. Primeiro, irá determinar que a posição de inserção é aquela correspondente ao índice 2. Como o Top-K está cheio, é necessário primeiro remover o valor com menor índice e deslocar os valores menores que 5 uma posição para a esquerda. De seguida, basta inserir 5 na posição de índice 2, tal como exemplificado na Figura 3c. Em detalhe, o array $[1, 4, 9, 14, 17]$ foi convertido em $[4, \underline{5}, 9, 14, 17]$.

Por último, após T_2, T_3 seleciona a chave aleatória $rk' = 1$ e pretende inserir o valor 3. Irá concluir que não o poderá fazer dado que o Top-K está cheio e o valor na primeira posição é maior que 3. Assim, a transação irá concluir sem fazer alterações na tabela.

Para efetuar a leitura do Top-K, selecionam-se os maiores valores de todos os registos do MRV, o que resulta no valor final representado na Figura 3d. Em

Tabela 1: Especificações das máquinas de teste.

CPU	RAM	Armazenamento	SO	Base de Dados
8 vCPU (E2)	16GB	40GB SSD	Ubuntu 22.04 LTS	PostgreSQL 15.1

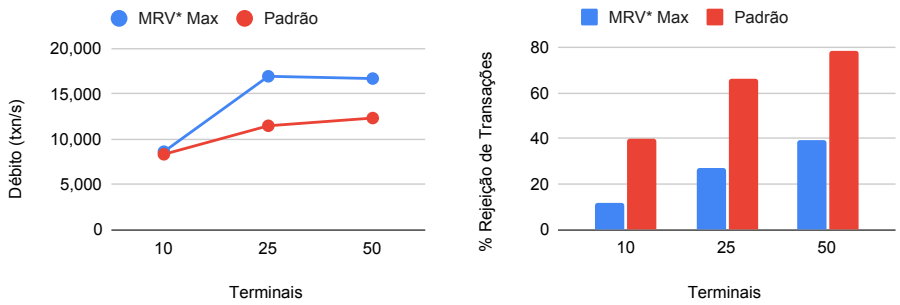
detalhe, o top final é composto por $[8, 9, 12, 14, 17]$. Os valores menores que 8 armazenados no MRV serão excluídos do resultado.

4 Avaliação

Todos os testes nesta secção foram executados em máquinas virtuais no Compute Engine da Google Cloud, sendo o PostgreSQL escolhido como sistema de gestão de base de dados onde os vários MRVs* foram implementados. Para os testes efetuados, usamos o nível de isolamento *REPEATABLE_READ* que no PostgreSQL resulta em Snapshot Isolation. Com este isolamento, as operações de escrita adquirem primeiro um lock exclusivo consoante a linha a modificar. Em escritas concorrentes à mesma linha, a primeira escrita a adquirir o lock poderá concluir com sucesso, enquanto que as outras terão que ser repetidas. Os detalhes do hardware e software usados podem ser consultados na Tabela 1. Consideramos como padrão o PostgreSQL 15.1 sem qualquer modificação.

Estes testes foram realizados 5 vezes com 10, 25, e 50 terminais a executar operações em 100 MRVs*. Nos casos em que a composição da carga de trabalho não é indicada explicitamente, 65% correspondem a operações de escrita e 35% a operações de leitura de forma a reproduzir o cenário alvo com contenção. Cada teste executou durante 90 segundos. A não ser que seja indicado, cada MRV* contém um número fixo de registos (8). No entanto, utilizando *background workers*, este valor seria adaptativo consoante a carga.

Max/Min. Dado que as estruturas Max e Min são semelhantes, os próximos testes foram realizados apenas para a estrutura Max. Com base na Figura 4a, pode-se constatar que com MRVs*, o débito é mais elevado do que a versão base. Para além disso, na Figura 5, verificamos que o aumento no número de registos por MRV* resulta num aumento do débito. Com 50 terminais, o débito dos MRVs* é cerca de $1.35\times$ superior à versão padrão. Este aumento no débito é consequência da menor probabilidade de colisão nas escritas, evidente pelos resultados da Figura 4b. Em detalhe, na versão padrão com 10 terminais, 40% das transações são rejeitadas, ao invés de cerca de 10% no caso em que se utiliza MRVs*. À medida que a concorrência aumenta, o número de rejeições também aumenta, pelo que com 50 terminais na versão padrão apenas 22% das transações completam. Por outro lado, na versão com MRVs*, mais de 60% das transações são executadas. Para *workloads* com mais leituras seria de esperar que o desempenho da estrutura fosse pior, no entanto, as estruturas continuam a ter um bom desempenho nestes cenários. Efetivamente, o desempenho com a



(a) Comparação de débito.

(b) Comparação do número de transações rejeitadas.

Figura 4: Comparação de desempenho na operação de Max entre a versão com e sem MRV*.

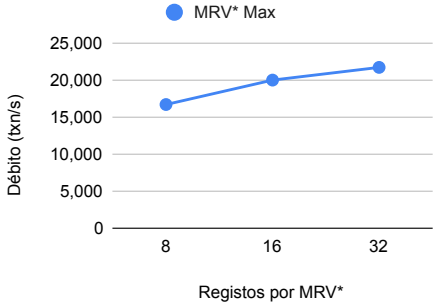
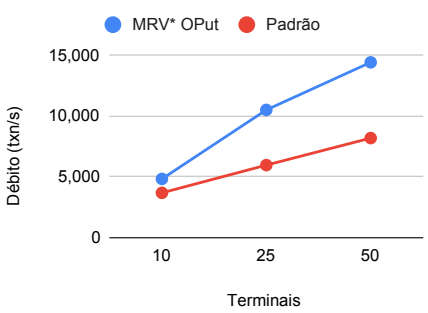


Figura 5: Comparação do débito na operação de Max consoante o número de registos por MRV*.

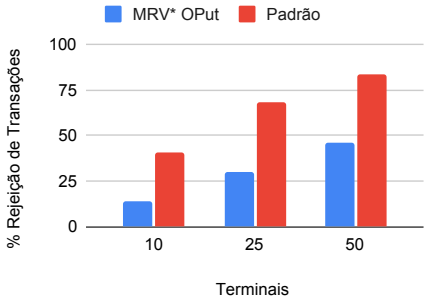
estrutura MRV* Max é 1.1× melhor do que a versão padrão em cenários com 35% escritas e 65% leituras com 50 terminais.

OPut. Para a estrutura OPut, cujo os resultados são visíveis na Figura 6a, a versão com MRVs* tem novamente um maior débito que a versão padrão para todos os cenários testados, resultante da menor taxa de colisão (Figura 6b). Para 50 terminais, a versão com MRVs* é 1.76× superior à versão padrão. Dado que esta estrutura apresenta o mesmo padrão de acesso aos registos, o seu desempenho é semelhante ao da anterior em cenários com maior volume de leituras.

Top-K. Relativamente à estrutura Top-K, tal como nas estruturas anteriores, o débito é maior do que a versão padrão em todos os testes o que se verifica na Figura 7a. Na Figura 7b, constata-se que a utilização da estrutura MRV* Top-K leva a uma menor taxa de colisão do que a versão padrão para os diferentes níveis

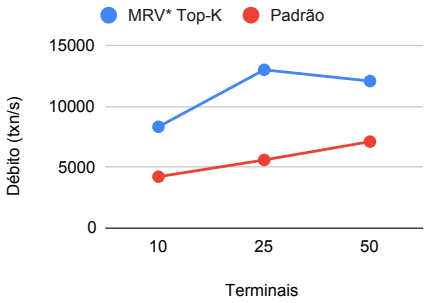


(a) Comparação de débito.

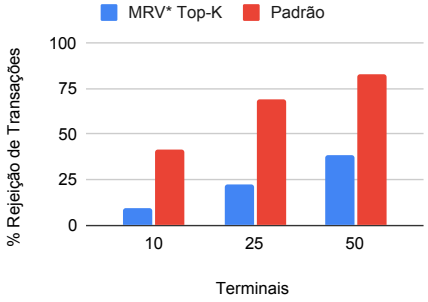


(b) Comparação do número de transações rejeitadas.

Figura 6: Comparação de desempenho na operação de OPut entre a versão com e sem MRV* para 10/25/50 terminais.



(a) Comparação de débito.



(b) Comparação do número de transações rejeitadas.

Figura 7: Comparação de desempenho na operação de Top-K entre a versão com e sem MRV* para 10/25/50 terminais.

de concorrência testados. Com 50 terminais, o débito é 1.70× superior à versão padrão quando se utiliza a estrutura MRV* Top-K. Tal como na estrutura MRV* Max, para um maior volume de leituras, o desempenho da estrutura MRV* Top-K também se apresenta semelhante.

5 Trabalho relacionado

O problema de conflitos transacionais no mesmo campo/estrutura tem levado à investigação e desenvolvimento de diversas técnicas para a sua mitigação ao longo dos anos. Um Bounded Counter[1] é um CRDT[11] *state-based* construído tendo por base uma *key/value store* que mantém informação para impor invariantes

numéricos. *Bounded Counters* usam reservas de forma a que réplicas diferentes recebam direitos para fazer mudanças ao contador, ou seja cada réplica pode alterar o contador desde que respeite os direitos que lhe foram atribuídos. Esta estrutura de dados é especialmente desenhada para evitar a latência inerente da sincronização distribuída e não para aumentar o paralelismo dentro de cada nó, ao contrário de MRVs.

Phase Reconciliation[8] é um método de controlo de concorrência para bases de dados *multicore* que é divido em três fases. Existem duas fases que permitem executar transações, sendo que uma delas aceita qualquer tipo de transação, *joined phase*, enquanto a outra, *split phase*, permite apenas que adições e subtrações sejam executadas. Na *split phase*, um valor é dividido em vários registos, tal como os MRVs. Já na *joined phase*, apenas existe um registo para cada valor, tal como na versão padrão. A última fase é chamada de *reconciliation phase* e é responsável por juntar registos mudados durante uma *split phase* ao armazenamento global. A mudança de fase ocorre dependendo da carga atual. Ao contrário dos MRVs, leituras no Phase Reconciliation precisam de bloquear até que um valor esteja na *joined phase*. No entanto, *Phase Reconciliation* não foi pensado tendo em conta bases de dados SQL, pelo que se torna difícil implementar este mecanismo neste contexto devido às mudanças de fase de que este método tira partido.

Os contadores implementados no Cassandra [16] também permitem a execução de operações de atualização de forma concorrente, ao dividi-los por múltiplos *shards*. Contudo, ao contrário de MRVs, não definem qualquer tipo invariante de limite inferior.

Tal como MRVs*, diversas outras soluções foram propostas no contexto de estruturas mais genéricas. Os Conflict-free Replicated Data Types (CRDTs) [12] oferecem diversas estruturas que permitem a atualização concorrente e assíncrona de operações sobre as mesmas, como conjuntos, listas, e mapas. Eventualmente, as modificações são combinadas de forma a que todas as réplicas apresentem o mesmo resultado final. Apesar de serem semelhantes com as soluções aqui propostas, os MRVs* focam-se especialmente em operações com fortes garantias. Para além dos CRDTs, os Counting Sets [13] também permitem acessos assíncronos a conjuntos.

Timestamp Splitting [5] permite a execução de transações concorrentes à mesma linha numa base de dados desde que estas modifiquem diferentes campos, ao fazer a partição dos *timestamps* em vários subconjuntos de colunas. De certa forma, ao separar os valores modelados da tabela original, os MRVs também permitem que acessos concorrentes ao MRV e à linha original executem em paralelo.

Por último, certas técnicas permitem executar operações inteiras em paralelo sem qualquer conflito, independentemente dos campos modificados [14,15,3]. Estas técnicas são particularmente populares em base de dados distribuídas que tenham como prioridade a disponibilidade ao invés da consistência, como DynamoDB ou Cassandra. Depois da confirmação, as várias operações são combinadas e, em caso de conflito, algumas poderão ser simplesmente descartas. Ao contrário

dos MRVs, estes mecanismos não são recomendados para casos de uso onde se pretendam fortes garantias de execução.

6 Conclusões

Neste trabalho, expandimos as utilizações de Multi-Record Values (MRVs) através da implementação de novas operações para estruturas de dados que ainda não tinham sido abordadas para este método de controlo de concorrência. Todas as estruturas MRVs* introduzidas apresentam resultados superiores às versões padrão, o que significa que a sua utilização em sistemas distribuídos poderá levar a um aumento de desempenho, mantendo garantias de coerência entre os vários nós. Dado que estas estruturas assentam nos princípios dos Multi-Record Values, podem ser aplicadas a sistemas de base de dados transacionais já existentes sem modificações ao código do seu motor, o que é especialmente útil em sistemas que não disponibilizam o seu código fonte.

Referências

1. Balegas, V., Serra, D., Duarte, S., Ferreira, C., Shapiro, M., Rodrigues, R., Preguiça, N.: Extending eventually consistent cloud databases for enforcing numeric invariants. In: 2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS). pp. 31–36 (2015). https://doi.org/10.1109/SRDS.2015.32
2. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels (1995). https://doi.org/10.1145/568271.223785, http://dx.doi.org/10.1145/568271.223785
3. Ellis, C., Gibbs, S.: Concurrency control in groupware systems. In: Proc. 1989 ACM SIGMOD Intl. Conf. on Management of data. pp. 399–407 (1989)
4. Faria, N., Pereira, J.: MRVs: Enforcing numeric invariants in parallel updates to hotspots with randomized splitting. Proc. ACM Manag. Data (SIGMOD) **1**(1) (2023). https://doi.org/10.1145/3588723
5. Huang, Y., Qian, W., Kohler, E., Liskov, B., Shrira, L.: Opportunities for optimism in contended main-memory multicore transactions. Proc. VLDB Endowment **13**(5), 629–642 (2020)
6. Liu, J., Magrino, T., Arden, O., George, M., Myers, A.: Warranties for faster strong consistency. In: 11th USENIX Symp. on Networked Systems Design and Implementation (NSDI 14). pp. 503–517 (2014)
7. Magrino, T., Liu, J., Foster, N., Gehrke, J., Myers, A.: Efficient, consistent distributed computation with predictive treaties. In: Proc. 14th EuroSys Conf. 2019. pp. 1–16 (2019)
8. Narula, N., Cutler, C., Kohler, E., Morris, R.: Phase Reconciliation for Contended In-Memory Transactions p. 15 (2014)
9. O'Neil, P.E.: The escrow transactional method. ACM Trans. on Database Systems (TODS) **11**(4), 405–430 (1986)
10. Preguiça, N., Martins, J., Cunha, M., Domingos, H.: Reservations for conflict avoidance in a mobile database system. In: Proc. 1st Intl. Conf. on Mobile systems, applications and services. pp. 43–56 (2003)
11. Preguiça, N.: Conflict-free replicated data types: An overview (2018)

12. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Symp. on Self-Stabilizing Systems. pp. 386–400. Springer (2011)
13. Sovran, Y., Power, R., Aguilera, M., Li, J.: Transactional storage for geo-replicated systems. In: Proc. 23rd ACM Symp. on Operating Systems Principles (SOSP). pp. 385–400 (10 2011). https://doi.org/10.1145/2043556.2043592
14. Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proc. 15th ACM Symp. on Operating Systems Principles (1995). https://doi.org/10.1145/224056.224070, https://doi.org/10.1145/224056.224070
15. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. on Database Systems (TODS) **4**(2), 180–209 (1979)
16. Yeschenko, A.: The DataStax blog - what's new in Cassandra 2.1: Better implementation of counters. https://www.datastax.com/blog/whats-new-cassandra-21-better-implementation-counters (2014)