Desempenho de transações serializable em cenários de alta concorrência no PostgreSQL

Edison Aguiar de Souza Neto[1], Luiz Fernando Tagliaferro Brito[1], Calebe de Paula Bianchini[1]

[1]Faculdade de Computação e Informática – Universidade Presbiteriana Mackenzie
Rua da consolação, 930 – São Paulo – SP – Brazil

{3181295,31861806}@mackenzista.com.br, calebe.bianchini@mackenzie.br

Abstract. *Currently, many systems receive daily loads of millions of transactions and it is necessary to guarantee the isolation of these transactions with the objective of keeping the data in a consistent state. Thanks to that, different algorithms were developed looking for serializability guarantees and high performance in high concurrency scenarios. The present study aims to analyze the different algorithms implemented on PostgreSQL and perform a performance comparison between them in a high concurrency scenario.*

Resumo. *Atualmente, muitos sistemas recebem cargas de milhões de transações diariamente, e a necessidade de garantir o isolamento das transações com o objetivo de manter os dados em um estado consistente se mantém. Com isso foram desenvolvidos algoritmos buscando garantir serialização de transações com alto desempenho em cenários de alta concorrência. O estudo presente tem como objetivo analisar os diferentes algoritmos de serialização implementados no PostgreSQL e realizar uma comparação de desempenho entre eles em cenários de alta concorrência.*

1. Introdução

Com o crescimento do número de pessoas na internet e o desenvolvimento contínuo de sistemas cada vez mais complexos, nossas vidas se tornam dependentes de sistemas dos quais nem sequer temos contato direto. Bilhões de dados são armazenados e lidos diariamente contendo as mais diversas informações, tais como: saldo de conta bancária, limite do cartão de crédito, documentos sigilosos, endereços, extratos bancários, entre outros. Esse volume continua crescendo e tende a crescer mais e mais com a democratização de serviços de diferentes setores pela internet.

Dado o grande volume de dados armazenados, juntamente com a alta demanda pelo acesso para leitura e modificação dos mesmos, obtém-se um problema relacionado ao desempenho e disponibilidade tanto para leitura como para escrita [Kleppmann 2017, p. 10-18]. Há décadas, já existem técnicas diferentes para lidar com o acesso concorrente à informação, uma das abstrações mais conhecidas para esse problema é o conceito de transação apresentado por Gray [1981] e expandido para o conceito de ACID apresentado por Haerder e Reuter [1983].

Com o objetivo de atingir o isolamento proposto por Haerder e Reuter [1983], foram desenvolvidos diferentes algoritmos ao longo dos anos. Bernstein e Goodman [1981] apresentaram muitos dos conceitos que hoje temos como MVCC (*Multiversion Concurrency Control*), técnicas que são utilizadas em diferentes implementações de bancos de dados. No cenário de crescimento atual, algumas dessas técnicas têm se

provado escaláveis, enquanto outras têm sido refinadas para aumentar sua escalabilidade.

Escalabilidade é definida por Bondi [2000] como a habilidade de um sistema acomodar um número crescente de elementos, para processar volumes crescentes de trabalho graciosamente, e/ou ser suscetível a ampliação. Isso quer dizer que o sistema deve conseguir lidar com o processamento de um alto volume de informações e com um volume ainda maior.

Dado o contexto de alto volume de dados, somado ao problema de utilização de um modelo de dados que se faz necessário um estrito nível de isolamento e alta disponibilidade, este trabalho tem como objetivo apresentar *benchmarks* utilizando o PostgreSQL, em um cenário de alta concorrência com transações nos níveis *serializable* e *repeatable read*, buscando compreender melhor o comportamento do banco de dados nesses cenários, validando assim as implementações de algoritmos de serialização disponibilizados.

Para isso, estressamos ao máximo o banco, a fim de obtermos os dados para descobrimos até que ponto o PostgreSQL consegue garantir consistência nas transações realizadas, juntamente com diversas métricas envolvidas para melhor compreendimento das mesmas, tornando mais ilustrativa e simples o entendimento.

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho tem como objetivo geral avaliar o desempenho do PostgreSQL em cenários de alta concorrência de transações nos níveis *serializable* e *repeatable read*, utilizando as técnicas de serialização disponibilizadas, *two-phase locking* e *serializable snapshot isolation*.

1.1.2 Objetivos Específicos

- Avaliar o comportamento e desempenho em cenários de alta concorrência utilizando a implementação do algoritmo *Serializable Snapshot Isolation* disponibilizado pelo PostgreSQL;
- Avaliar o comportamento e desempenho em cenários de alta concorrência utilizando a implementação do algoritmo *Two-Phase Locking* disponibilizado pelo PostgreSQL;
- Realizar uma análise comparativa dos dados de desempenho obtidos e avaliar qual algoritmo se comportou melhor para cada cenário testado;

1.2 Contribuições da pesquisa para a academia e sociedade

A partir dos dados obtidos, pode-se ver com maior clareza o desempenho dos diferentes algoritmos de serialização implementados no PostgreSQL em situações de alta concorrência.

Esta pesquisa busca obter uma compreensão mais clara de qual algoritmo é mais adequado para diferentes cenários de concorrência. Além de avaliar que as implementações garantem a serialização das transações, este trabalho fornecerá dados

valiosos para outras pesquisas que podem buscar otimizar esses algoritmos para se adaptarem melhor aos cenários avaliados.

2. Referencial Teórico

2.1 Níveis de Isolamento

Isolamento é definido por Haerder e Reuter [1983] como os eventos dentro de uma transação tendo que estar ocultos de outras transações rodando concorrentemente. Isto é, cada transação deve operar como se as outras não estivessem acontecendo. Dessa forma, transações operando são isoladas umas das outras. Essa propriedade é fundamental para evitar problemas de concorrência [Kleppmann 2017, p. 225-226].

Para isso, o ANSI SQL específica alguns níveis de isolamento para as transações. As implementações existentes lidam de formas diferentes sobre como tratar esses níveis como PostgreSQL e MySQL, mas devem sempre evitar as anomalias descritas no padrão [Bereson *et al.* 1995].

No nível *read uncommitted*, as operações dentro da transação conseguem ler dados de transações que ainda não realizaram o *commit*. Com isso, temos o problema de *Dirty Read* [Bereson *et al.* 1995], em que uma transação T1 pode consultar um dado da transação T2 mesmo que T2 não tenha realizado *commit*. Caso T2 tenha *rollback* a T1 ainda assim vai considerar o valor para realizar alguma operação possivelmente deixando o dado em um estado inconsistente.

No nível *read committed*, cada operação dentro da transação leva em consideração apenas o que foi commitado até o ponto em que o comando foi rodado e operações feitas dentro daquela transação que ainda não foram commitadas. Com isso, evita-se o problema de *dirty read,* já que não é possível ler dados de transações que não tiveram o *commit* realizado [Kleppmann 2017, p. 234-235].

O nível de isolamento *repeatable read* é semelhante ao *read committed*, entretanto, nesse nível as operações da transação não leem o que foi commitado até o ponto em que o comando foi executado, e sim até o ponto em que a transação foi iniciada. É como se fosse feita uma cópia do banco no início da transação, com isso todas as operações verificam essa cópia. Desse modo é garantido que não ocorram inconsistências no escopo da transação [Kleppmann 2017, p. 237-239].

O objetivo do nível *serializable* é processar as transações como se todas estivessem executando sequencialmente, sem concorrência nenhuma, mesmo com várias transações executando em paralelo [Kleppmann 2017, p. 252].

Considerando o exemplo apresentado por Ports e Grittner [2012], temos um caso em que uma consulta é realizada para contar quantos médicos estão de plantão, se a quantidade de médicos no plantão for maior ou igual a dois, então um dos médicos pode sair do plantão. Considerando um registro inicial em que apenas Alice e Bob estejam de plantão, neste exemplo teríamos no final um estado inconsistente em que nenhum médico está de plantão, já que as transações leram os dados das *snapshots* criadas no seu início(figura 1).

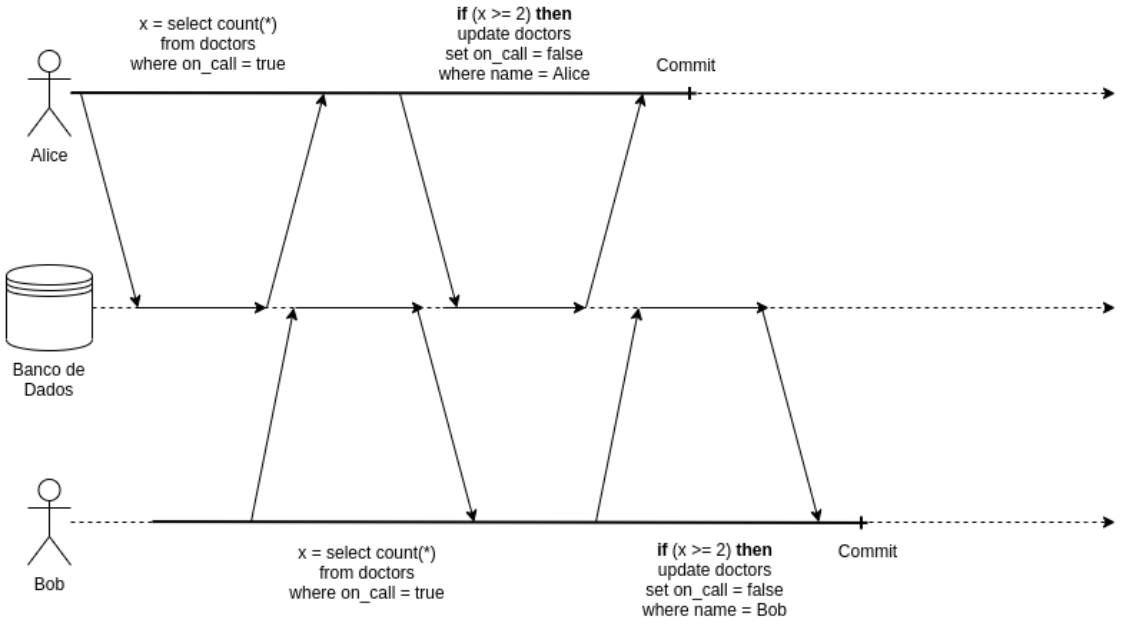

Figura 1. Cenário de *write skew* em que os dados ficam em um estado inconsistente

Realizar essa garantia é algo computacionalmente mais custoso do que os outros níveis de isolamento vistos anteriormente. Por conta disso, as implementações do nível serializable procuram ser o mais escaláveis e performáticas possíveis.

2.2. Snapshot Isolation

Uma das formas de implementar o *repeatable read*, que é a forma utilizada pelo PostgreSQL, é o *snapshot isolation* (SI). O objetivo desse algoritmo é prover uma snapshot do banco logo no início da transação [Kleppmann 2017, p. 239]. Mesmo se outras transações concorrentes escreverem novos valores e realizarem *commit*, a transação atual não irá visualizar esses dados, já que eles foram inseridos após o início da transação.

Esse algoritmo torna operações como um *backup*, em que é necessário ler grande parte dos dados registrados no banco, seguras e mais performáticas, em comparação a algoritmos baseados em *locks* [Bereson *et al.* 1995].

A implementação de SI do PostgreSQL além de prever anomalias como o *read skew*, também prevê anomalias como *lost update*, garantindo que em casos de transações concorrentes que passam pelo ciclo de *read-modify-write* [Kleppmann 2017, p. 243] um update não ficará perdido, com apenas uma transação realizando *commit* e as outras sofrendo *rollback* neste cenário [Ports, Grittner 2012].

Ainda assim, transações concorrentes no SI ainda podem acabar em um estado inconsistente já que o SI não prevê todas as possíveis anomalias causadas por acesso concorrente, como o *write skew*, que pode acontecer quando transações concorrentes lêem os mesmos dados e atualizam alguns desses dados, é uma anomalia similar ao *lost update*, a diferença essencial é que neste cenário múltiplos registros são lidos e atualizados.

2.3. Técnicas de serialização

Uma técnica muito utilizada de serialização nos bancos SQL é a *Two-Phase Locking*(2PL). Essa técnica consiste em realizar o *lock* em duas fases: a fase inicial onde a transação adquire o *lock* dos registros em seu decorrer; E a segunda fase onde todos os *locks* adquiridos na transação são liberados [Gray *et al.* 1976]. Uma característica importante é que os *locks* adquiridos pela transação são apenas liberados no fim da mesma, no *commit* ou *abort*, i.e., uma transação pode adquirir o *lock* de múltiplos registros em seu decorrer, e mesmo que esses registros sejam apenas utilizados em parte da transação, eles só terão o *lock* liberado ao fim da transação e não ao fim de seu uso na transação.

Um dos problemas do 2PL, que impacta diretamente na escalabilidade do processamento das transações, é que escritores bloqueiam leitores. Ou seja, transações que adquirem o *lock* de algum registro que será alterado bloqueiam outras transações que precisem ler esse mesmo registro [Kleppmann 2017, p. 257].

As dificuldades no bom desempenho do 2PL já são bem conhecidas. Autores têm buscado alternativas e melhorias ao 2PL há muitos anos, seja otimizando os algoritmos já existentes pensando em casos de uso específicos[Thomasian, Ryu 1991] [Son, David 1994] ou propondo diferentes algoritmos de *locking* [Buckley, Silberschatz 1985] que também garantam a serialização.

Outra estratégia para serialização é processar as transações de forma ordenada. Ela se baseia em um sistema *single-thread* para executar as transações. Essa estratégia é feita pensando em sistemas OLTP(*Online Transaction Processing*) onde as transações são curtas e rápidas e em geral todo o *dataset* pode ser carregado em RAM, gerando pouco, ou nenhum, uso de disco [Stonebraker *et al.* 2007].

Essa estratégia não era muito utilizada por bancos de dados mais antigos principalmente por dois motivos: a limitação de RAM, já que o *dataset* não poderia ser carregado totalmente em ram era o uso do disco era maior, impactando diretamente o desempenho de forma drástica; E pela implementação desses bancos tentar abranger tantos cenários OLTP quanto OLAP(*Online Analytical Processing*), um cenário com transações curtas e rápidas alterando ou criando muitos registros, e outro cenário com transações longas e lentas [Kleppmann 2017, p. 252-256][Kallman et al 2008].

O modelo de execução *single-thread* traz como principal vantagem a diminuição do *overhead* gerado pela execução *multi-thread* dos algoritmos usados para garantir o isolamento das transações. A diminuição desse *overhead* traz um ganho muito grande, entretanto, faz-se necessário que as transações executadas não sejam muito longas, já que uma transação na fila impactará diretamente o tempo de processamento de todas as outras [Stonebraker *et al.* 2007].

O *Serializable Snapshot Isolation*(SSI) é o algoritmo que foi desenvolvido por Cahill, Rohm e Fekete [2009], com base no algoritmo de *Snapshot Isolation*. No SSI, assim como no SI, as transações são processadas baseando-se na *snapshot* feita no início da transação, e no momento do *commit* são detectados possíveis conflitos na escrita. Dessa forma as transações em que foram detectados conflitos serão abortadas, garantindo assim a serialização das transações [Cahill, Rohm, Fekete 2009].

Fundamentalmente, esse algoritmo busca ter como vantagem o aumento da concorrência entre as transações, já que pelo modelo de MVCC, os escritores não bloqueiam os leitores. Esse é um modelo de controle de concorrência otimista, diferentemente da execução ordenada e do 2PL que são desenvolvidos fazendo com que as transações aguardem a execução das outras, diminuindo a concorrência, o modelo otimista procura deixar as transações executarem de forma concorrente e validar no seu fim se houve algum conflito de serialização [Kleppmann 2017, p. 261-262].

O controle de concorrência otimista tem um bom desempenho em cenários de baixa contenção, i.e. um cenário em que existem poucos possíveis conflitos de serialização, já em cenários com alta contenção o modelo tende a ter um desempenho ruim, dado que muitas transações irão abortar por conflitos de serialização [Zendaoui, Hidouci 2015].

2.4. Serializable Snapshot Isolation no PostgreSQL

O SSI foi o algoritmo escolhido para implementar o nível de isolamento *serializable* do PostgreSQL ao invés de realizar a serialização com 2PL, por questões de desempenho e pelo modelo de MVCC já implementado e bem definido. Um dos objetivos nessa solução era reutilizar a implementação já construída e consolidada do SI e continuar com transações não bloqueantes, i.e. leitores não bloqueiam escritores e escritores não bloqueiam leitores, como os usuários já estavam acostumados [Ports, Grittner 2012].

Como alguns estudam já mostram, em comparação com o SI a implementação do PostgreSQL de SSI tem um desempenho similar em diferentes cenários, tanto de apenas escrita quanto escrita e leitura [Zendaoui, Hidouci 2015]. Outros estudos apontam a degradação do desempenho do SSI quando o número de cores do processador e o número de clientes cresce, devido a contenção de *latch* em estruturas de dados internas do sistema [Han *et al.* 2014].

2.5. Trabalhos Correlatos

No trabalho original de desenvolvimento do SSI no PostgreSQL foram feitos alguns micro benchmarks simples, com o objetivo de realizar uma comparação do *throughput* de transações utilizando SI, SSI e 2PL em cenários de escrita e leitura verificando a quantidade de erros de serialização recebidos para cada implementação [Ports, Grittner 2012].

Em outros trabalhos foram comparadas as implementações de SI e SSI em *workloads* diferentes de escrita e leitura, com uma maior taxa de leitura e baixa taxa de escrita e alta taxa de escrita e baixa taxa de leitura [Zendaoui, Hidouci 2015]. Han *et al.* [2014] exploraram o comportamento de diferentes algoritmos de SSI, incluindo a implementação realizada pelo PostgreSQL, em servidores multicore com múltiplos clientes, obtendo resultados importantes para o objetivo deste trabalho.

Em alguns desses trabalhos o benchmark utilizado foi o *Sibench*. Neste benchmark são gerados múltiplos updates em transações diferentes, atualizando um valor gerado aleatoriamente, e transações de leitura que realizam um scan em toda a tabela buscando pelo registro com o menor valor. Esse benchmark provê bons volumes para representar *workloads* mais próximos do usual, entretanto, não se encaixa bem para simular cenários com muitos conflitos de serialização.

O SSI, assim como 2PL, tende a não performar bem em cenários de alta contenção, isto é, várias transações tentando alterar o mesmo dado [Kleppmann 2017, p. 265-266]. Nestes cenários, tanto os algoritmos de controle de concorrência pessimistas, quanto os otimistas tendem a não performar bem. Tem-se então, uma oportunidade ainda inexplorada.

3. Metodologia

Para avaliar o desempenho da implementação do SSI no PostgreSQL foi usada a ferramenta *Benchbase* [Difallah *et al.* 2013], executando o *benchmark Smallbank* [Alomari *et al.* 2006] nas máquinas do MackCloud.

Os benchmarks foram executados no MackCloud usando a versão estável mais recente PostgreSQL, 14.2. Em uma máquina, foi executado o PostgreSQL e em outra o *benchmark*. A especificação de hardware e software de cada máquina pode ser vista na tabela 1. Cada benchmark foi executado 5 vezes e foi aplicada uma média entre os resultados obtidos, assim como foi feito em outros trabalhos relacionados [Difallah *et al.* 2013][Zendaoui, Hidouci 2015][Han et al 2014].

Tabela 1. Especificações dos nós no MackCloud

Componente	**Especificação**
CPU	Intel Xeon E5-2650 v4 com 48 Cores
RAM	64 GB
Rede	Infiniband 40 Gb/s
Sistema Operacional	CentOS 7.7.1908
Sistema de Arquivos	XFS 4.5

O *benchmark* escolhido para ser executado foi o *Smallbank*, desenvolvido originalmente por Alomari *et al.* [2006]. Esse *benchmark* consiste em gerar várias transações curtas que realizam operações nos saldos de contas bancárias. Ele foi escolhido por ter um cenário onde são geradas transações que geram anomalias que só conseguem ser evitadas com o nível de isolamento *serializable*.

Para a execução do *benchmark* foi usada a ferramenta desenvolvida e mantida pelo grupo de banco de dados da Universidade Carnegie Mellon, o *benchbase*, anteriormente conhecido como OLTP-Bench [Difallah *et al.* 2013]. Essa ferramenta foi desenvolvida com o propósito de ser uma *suite* de diferentes benchmarks com suporte a diferentes bancos de dados. Um dos benchmarks implementados nela é o *smallbank*, que foi extendido além dos trabalhos originais de Alomari *et al.* [2006] e Cahill, Rohm e Fekete (2009).

O *benchbase* foi configurado para usar 96 *threads worker* responsáveis por gerar as transações no banco com uma taxa de limite das transações com 30 mil transações. Cada execução do *benchmark* leva 30 minutos, gerando o máximo de transações possíveis dentro do fator de escala considerado. Os fatores de escala executados foram

0.1, 0.5, 1, 5 e 10. No caso do *smallbank,* o fator de escala controla a quantidade de contas geradas para as transações [Difallah *et al.* 2013]. O fator determina 1 milhão de contas, ou seja, o fator 0.1 gera 100 mil contas e o 10, 10 milhões de contas, tendo assim cenários com maior possibilidade de conflitos e menor possibilidades, como foi realizado em outros trabalhos[Alomari *et al.* 2006].

Além dos testes realizados com a versão mais recente do *benchbase*, para ter-se um comparativo de desempenho, foi realizada uma alteração no código fonte do projeto para rodar o mesmo *benchmark* do *smallbank,* usando o nível de isolamento *repeatable read*(*Snaphot Isolation*) e locks exclusivos nos registros, gerando assim uma serialização sem a necessidade do nível *serializable*.

Assim como foi exposto por Lindblom e Strandin [2015] o *file system* usado pelo sistema operacional pode impactar diretamente no desempenho do PostgreSQL. O *file system* usado nas máquinas usadas foi o XFS que apresentou um bom desempenho no trabalho supracitado.

Para realizar a comparação entre os resultados, foram avaliados o *throughput* das transações por segundo, a quantidade de *rollbacks* realizados e a latência de 90% das transações. Todos os gráficos usados neste trabalho foram gerados com dados gerados pelo próprio *benchbase*.

4. Resultados

Com base nos dados coletados, pode-se observar na figura 2 e 3 que o *throughput* geral das transações com o SSI foi superior ou próximo ao das transações utilizando *lock*. É possível visualizar essa diferença de forma mais clara na figura 4, onde encontra-se o gráfico do cálculo da diferença do *throughput*.

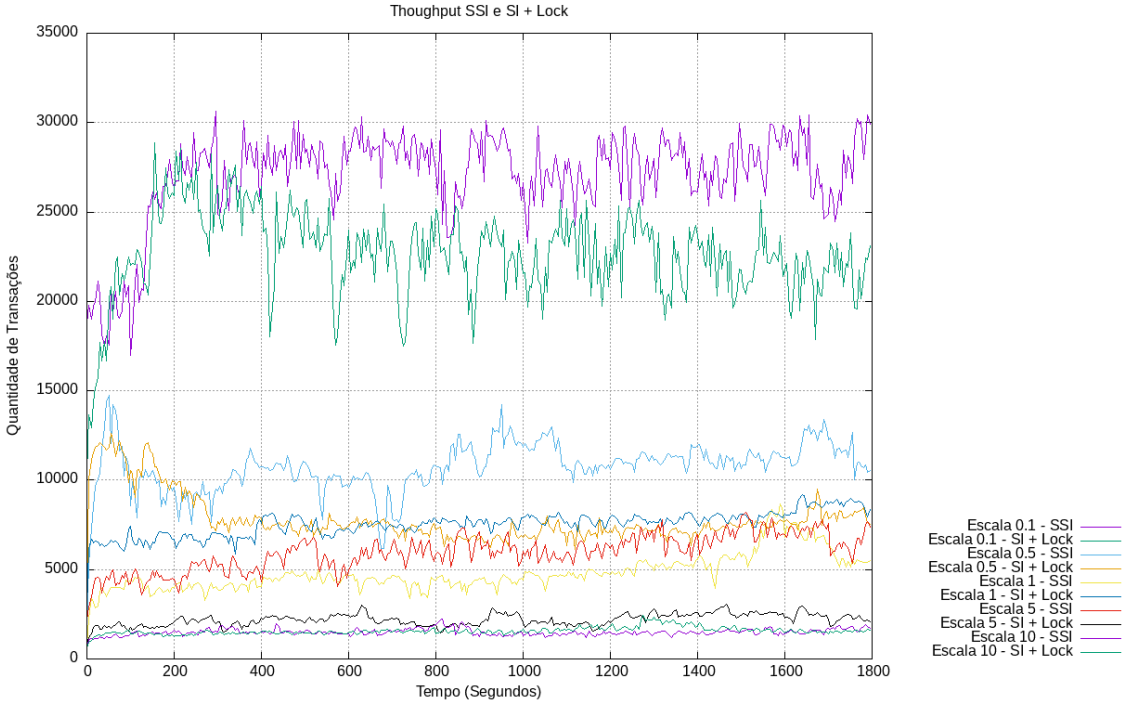

Figura 2. *Throughput* das transações SSI e SI + Lock

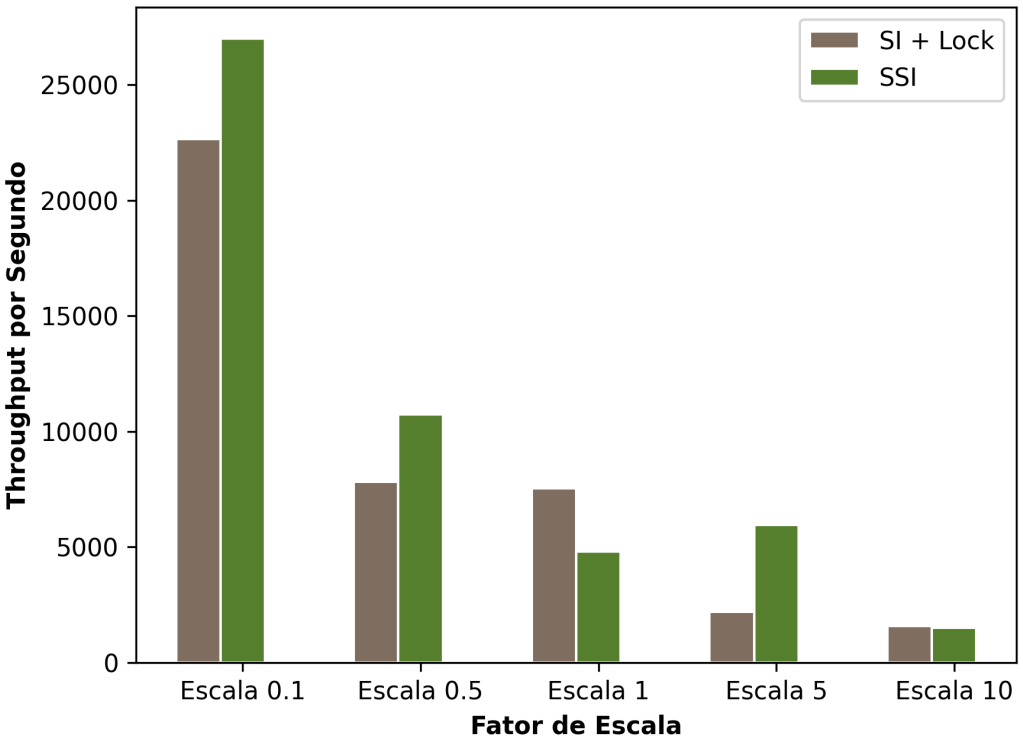

Figura 3. Gráfico de barras do *throughput* das transações SSI e SI + Lock

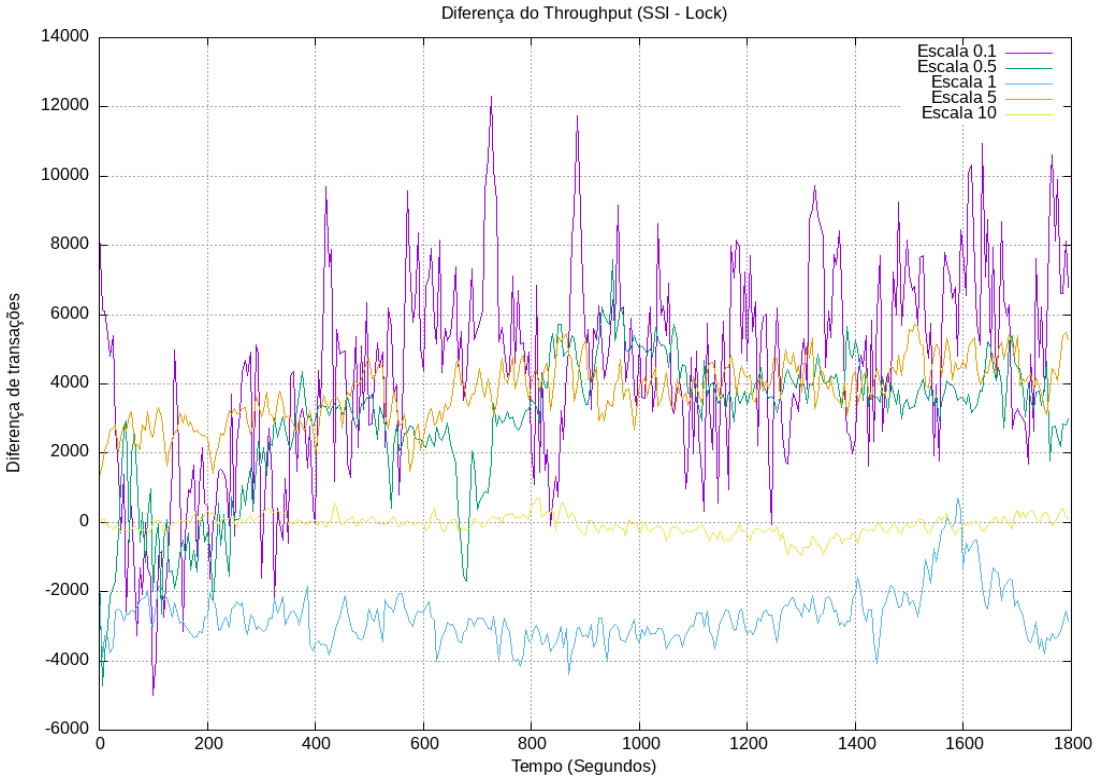

Figura 4. Diferença do *throughput* das transações SSI e SI + Lock

Observa-se que com 90% das transações a latência não apresentou uma grande diferença entre os níveis de isolamento, em alguns casos com a diferença sendo próxima de zero na maior parte da execução do benchmark, como é possível observar na figura 5.

Para analisar a latência foram consideradas 90% das transações já que essa análise é melhor visualizada em percentis. Dessa forma são eliminados outliers tornando possível analisar de forma mais precisa a latência geral do sistema [Kleppmann 2017, p. 36-37].

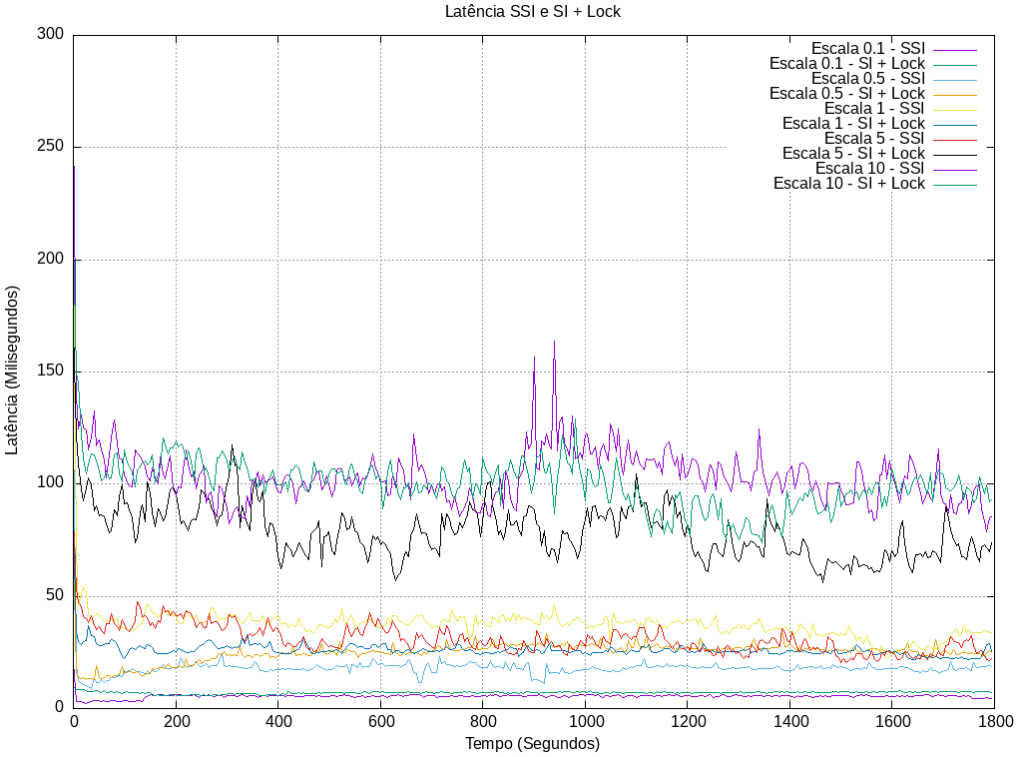

Figura 5. Latência de 90% das transações SSI e SI + Lock

Como é possível observar na figura 6, a quantidade de transações abortadas diminui conforme a escala aumenta, isso é devido ao fato de que com uma escala menor a possibilidade de ocorrer um conflito de serialização se torna muito maior, já que a quantidade de contas geradas é menor [Difallah *et al.* 2013].

Em quase todas as escalas testadas, a estratégia de transações com *lock* obteve menos *aborts* que as transações com SSI. Nos cenários com maior contenção(Escalas 0.1 e 0.5) pode se observar que o Lock obteve uma quantidade consideravelmente menor de *aborts*.

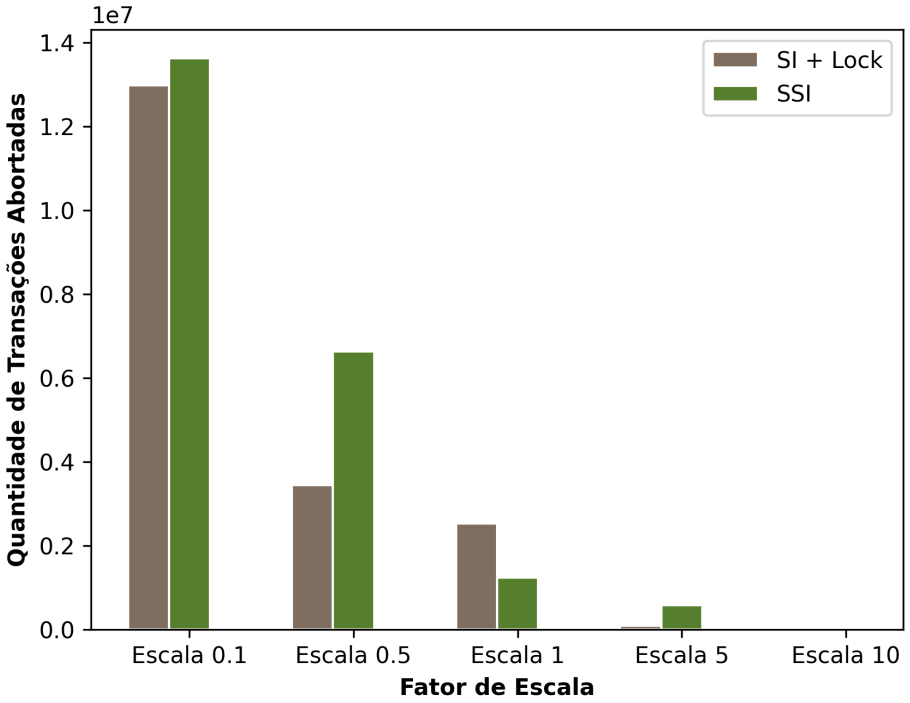

Figura 6. Gráfico de barras com a quantidade de transações abortadas

Deadlocks são eventos que ocorrem quando existem duas(ou mais) transações segurando o *lock* de algum objeto, sendo que cada transação depende do *lock* já obtido pela outra. Eles não ocorrem em transações que rodam no nível serializable, já que a implementação do SSI é realizada por meio de *locks* não bloqueantes. Entretanto, como pode ser observado na figura 7, *deadlocks* podem ocorrer em casos de uso de *lock* explícito, nesses casos o PostgreSQL identifica o *deadlock* e aborta a transação [PostgreSQL 2022].

Os *deadlocks* diminuem conforme a escala aumenta, já que a possibilidade de ter um conflito de *lock* diminui conforme existem mais contas a serem geradas para o benchmark.

O erro "*out of shared memory*" ocorre principalmente durante os testes com grande volume de transações e somente no algoritmo SSI, como pode ser observado na figura 8, devido à maneira de como o mesmo trata o lock [Ports, Grittner 2012]. Quando o *lock* ocorre, diferente de outros algoritmos, ele não é bloqueante, assim outras transações podem ler a mesma página com *lock*.

O *lock* realizado pelo SSI é salvo em uma tabela de locks em memória compartilhada [Ports, Grittner 2012]. Essa tabela mantém uma quantidade máxima de objetos com lock por transação, esse valor máximo é calculado por algumas configurações do PostgreSQL, na caso da configuração padrão o valor máximo é 640 [The PostgreSQL Global Development Group 2022].

O problema ocorre porque com um volume grande de transações acessando múltiplas páginas o limite de *locks* configurado por padrão é ultrapassado facilmente.

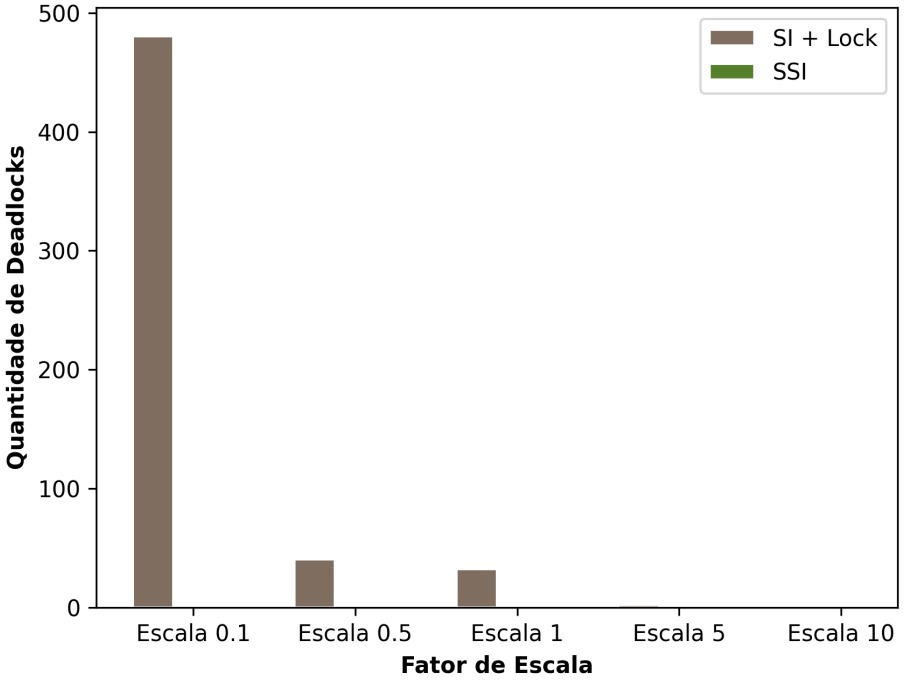

Figura 7. Gráfico de barras com a quantidade de deadlocks

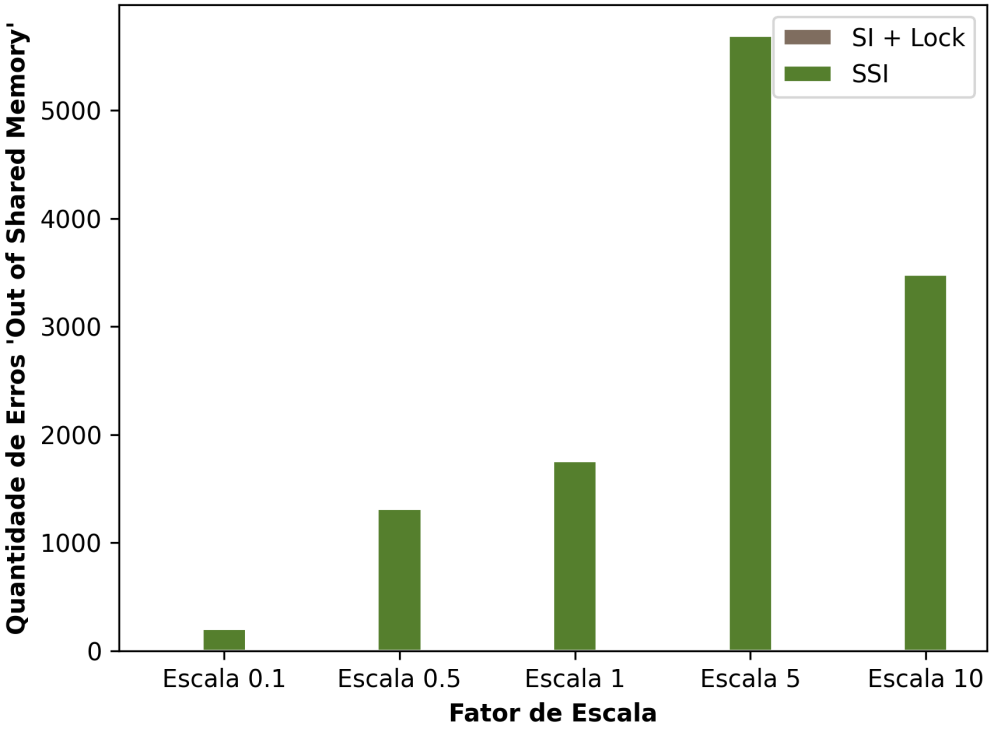

Figura 8. Gráfico de barras com a quantidade de erros 'Out of Shared Memory'

5. Conclusões

Analisando os dados obtidos durante o uso do algoritmo SSI, podemos observar na figura 6 que a quantidade de transações abortadas é maior se comparado com o *lock*. Entretanto, o *throughput* do SSI não foi afetado, isso se justifica devido a maneira como ele lida cenários de alta concorrência, abortando suas transações sem comprometer o desempenho [Ports, Grittner 2012].

O erro "*out of shared memory*" pode ser evitado adaptando a configuração do PostgreSQL para o *workload* executado. Como pôde ser observado, a configuração padrão não funcionou muito bem para os *workloads* que necessitavam manter múltiplos acessos à diferentes páginas, nesses cenários o ideal é aumentar o valor da configuração "max_pred_locks_per_transaction" para que o banco consiga manter mais *locks* em memória.

Em geral, as transações usando o nível *serializable* mostraram um desempenho consideravelmente melhor que as transações usando o nível *repeatable read* com *lock* na maior parte dos cenários testados, como pode ser visualizado na diferença do *throughput* apresentado na figura 4.

Conclui-se que após todos os testes realizados, resultados coletados e estudados, pode-se afirmar que o algoritmo de serialização SSI implementado no PostgreSQL é eficiente em cenários de alta carga e concorrência, oferecendo um ótimo desempenho quando comparado ao outro algoritmo de serialização disponibilizado. Torna-se assim, uma opção valiosa para os cenários descritos e testados, podendo melhorar o desempenho de sistemas que necessitem de transações serializáveis e utilizam a estratégia de *Two-Phase Locking*.

Possíveis trabalhos futuros podem explorar cenários diferentes, por exemplo: gerando *workloads* concorrentes com uma mistura de fluxos com maior contenção e menor contenção; Analisando a degradação do desempenho do PostgreSQL em cenários de *workloads* longos; Realizando uma comparação de desempenho do nível serializable de diferentes bancos de dados; E realizando mudanças na configuração do PostgreSQL para otimizar ao máximo o banco para cenários de alta concorrência.

Agradecimentos

Os autores agradecem ao MackCloud (https://mackcloud.mackenzie.br), Centro Multidisciplinar de Computação Científica e Nuvem da Universidade Presbiteriana Mackenzie, pelo apoio na realização desta pesquisa.

Referências

Alomari, Mohammad; Cahill, Michael; Fekete Alan; Röhm, Uwe (2006) "*The Cost of Serializability on Platforms That Use Snapshot Isolation*". In Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08). IEEE Computer Society, USA, 576–585. Disponível em <https://doi.org/10.1109/ICDE.2008.4497466>.

Bereson, Hal; Bernstein, Phil; Gray, Jim; Melton, Jim; O'neil, Elizabeth; O'neil, Patrick (1995) "A critique of ANSI SQL isolation levels", SIGMOD Rec. 24, 2 (May 1995), 1–10. Disponível em <https://doi.org/10.1145/223784.223785>.

Bernstein, Philip A.; Goodman, Nathan *(1981) "Concurrency Control in Distributed Database Systems"*. Disponível em <https://doi.org/10.1145/356842.356846>.

Bondi, André *(2000) "Characteristics of scalability and their impact on performance", AT&T Labs, Network Design and Performance Analysis Department.* Disponível em <https://doi.org/10.1145/350391.350432>.

Buckley, G. N.; Silberschatz A.; (1985) *"Beyond Two-Phase Locking". University of Texas at Austin. Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 314-326.* Disponível em <https://doi.org/10.1145/3149.3151>.

Cahill, Michael J.; Röhm, Uwe; Fekete, Alan D. (2009) *Serializable Isolation for Snapshot Databases*. Disponível em <https://doi.org/10.1145/1620585.1620587>.

Difallah, D. E.; Pavlo, A.; Curino, C.; Cudré-Mauroux, P. (2013) *"OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases", PVLDB, vol. 7, iss. 4, pp. 277-288,*. Disponível em <https://dl.acm.org/doi/10.14778/2732240.2732246>.

Gray, J.N.; Eswaran K.P.; Lorie, R.A.; Traiger, I.L. (1976) "*The Notions of Consistency and Predicate Locks in a Database System"*. Commun. ACM 19, 11 (Nov. 1976), 624–633. Disponível em <https://doi.org/10.1145/360363.360369>.

Gray, Jim (1981) "The Transaction Concept: Virtues and Limitations", In Proceedings of the seventh international conference on Very Large Data Bases - Volume 7 (VLDB '81). VLDB Endowment, 144–154. Disponível em <https://dl.acm.org/doi/abs/10.5555/48751.48761>.

Haerder, Theo; Reuter, Andreas (1983) "Principles of transaction-oriented database recovery", ACM Comput. Surv. 15, 4 (December 1983), 287–317. Disponível em <https://doi.org/10.1145/289.291>.

Han, Hyuck; Park, SeongJae; Jung, Hyungsoo; Fekete, Alan; Rohm, Uwe; Yeom, Heon Y. *(2014) "Scalable Serializable Snapshot Isolation for Multicore Systems", Dongduk Women's University, Seoul National University and University of Sydney.* Disponível em <https://doi.org/10.1109/ICDE.2014.6816693>.

Stonebraker, Michael; Madden, Samuel; Abadi, Daniel J.; Harizopoulos, Stavros; Hachem, Nabil; Helland, Pat (2007) "*The End of an Architectural Era (It's Time for a Complete Rewrite).*" Disponível em <https://dl.acm.org/doi/10.5555/1325851.1325981>.

Kallman, Robert; Kimura, Hideaki; Natkins, Jonathan; Pavlo, Andrew; Rasin, Alexander; Zdonik, Stanley; Jones, Evan P. C.; Madden, Samuel; Stonebraker, Michael; Zhang, Yang; Hugg, John; Abadi, Daniel J. (2008) *"H-store: a*

high-performance, distributed main memory transaction processing system.” Proc. VLDB Endow. 1, 2 (August 2008), 1496–1499. Disponível em <*https://doi.org/10.14778/1454159.1454211*>.

Kleppmann, Martin *(2017) “Designing data-intensive applications”, O'reilly Media.*

Lindblom, Martin; Strandin, Fredrik (2015) *“Performance Analysis And Improvement of PostgreSQL”. Department of Computer Science, Faculty of Engineering LTH.* Disponível em <http://lup.lub.lu.se/student-papers/record/4939915>.

Ports, Dan R. K.; Grittner, Kevin *(2012) “Serializable snapshot isolation in PostgreSQL”, Proceedings of the VLDB Endowment. Di*sponível em <https://doi.org/10.14778/2367502.2367523>.

Son, S.; David, R. (1994). *“Design and analysis of a secure two-phase locking protocol”. Proceedings of 1993 IEEE 12th Symposium on Reliable Distributed Systems, 1993, pp. 126-135. Disponível em <https://doi.org/10.1109/RELDIS.1993.393466*>.

The PostgreSQL Global Development Group (2022) “*Chapter 13. Concurrency Control*”, Disponível em <https://www.postgresql.org/docs/14/mvcc.html>.

Thomasian, A.; Ryu, I. K. (1991) *“Performance Analysis of Two-Phase Locking”. IEEE Transactions On Software Engineering, Vol. 17, No. 5, May 1991,* Disponível em <https://doi.org/10.1109/32.90443>.

Zendaoui, Fairouz; Hidouci, Walid Kahled *(2015) "Performance Evaluation of Serializable Snapshot Isolation in PostgreSQL", Ecole nationale Supérieure d'Informatique.* Disponível em <https://doi.org/10.1109/ISPS.2015.7244971>.