Jean Souto Galvão Moreira

Connection pooling em PostgreSQL: caracterização empírica do ponto de equilíbrio entre sobrecarga e contenção

Uberlândia, Minas Gerais
2026

Jean Souto Galvão Moreira

Connection pooling em PostgreSQL: caracterização empírica do ponto de equilíbrio entre sobrecarga e contenção

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Uberlândia, como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação.

Universidade Federal de Uberlândia
Faculdade de Computação
Curso de Bacharelado em Ciência da Computação

Orientadora: Profa. Dra. Maria Adriana Vidigal de Lima

Uberlândia, Minas Gerais
2026

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

M838 Moreira, Jean Souto Galvão, 2003-
2026 Connection pooling em PostgreSQL [recurso eletrônico] : caracterização empírica do ponto de equilíbrio entre sobrecarga e contenção / Jean Souto Galvão Moreira. - 2026.

Orientadora: Maria Adriana Vidigal de Lima.
Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Uberlândia, Graduação em Ciência da Computação.
Modo de acesso: Internet.
Inclui bibliografia.

1. Computação. I. Lima, Maria Adriana Vidigal de,1971-, (Orient.). II. Universidade Federal de Uberlândia. Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

À minha mãe e à minha avó,
que nunca deixaram de acreditar em mim.

AGRADECIMENTOS

Sou o primeiro do meu núcleo familiar a concluir uma graduação, e por isso esta conquista não é só minha.

À minha mãe, Fabiana Souto, e à minha avó, Celeida Souto, que me criaram e me fizeram ser quem eu sou. Minha mãe sai de casa de manhã e só volta à noite, dividida entre dois empregos e sem descanso, para que o meu caminho seja mais leve do que o dela; minha avó me acolheu, me criou como a um filho e nunca parou de cuidar, da mensagem de todos os dias à comida que chega sempre que ela pode. Cada página deste trabalho existe por causa do sacrifício das duas, e devo a elas mais do que caberia nestas linhas.

Ao meu padrasto, Alef Oliveira, que nunca precisou de laço de sangue para cuidar de mim como um filho, e à minha irmã, Layra Souto, minha maior torcida em cada conquista.

Às professoras Vilse Arantes, do ensino fundamental, e Maria Regina Zanetti, do ensino médio, que despertaram em mim o desejo pelo conhecimento e foram as primeiras a me apontar o caminho da UFU.

Aos meus amigos, porto seguro de quem chegou sozinho a uma cidade nova. Estiveram comigo nos momentos de caos e também nos de tranquilidade, e esses anos não teriam sido os mesmos sem eles.

À minha orientadora, Profa. Dra. Maria Adriana Vidigal de Lima, que acreditou no meu potencial desde os tempos em que eu era vice-presidente do Diretório Acadêmico da Computação e apoiou cada projeto que propus, inclusive os mais ambiciosos. A ela, minha gratidão pela orientação, pela paciência e pela confiança ao longo de toda a graduação.

À Universidade Federal de Uberlândia e à Faculdade de Computação, pela formação pública, gratuita e de qualidade que tornou este estudo possível.

A todos que, cada um à sua maneira, seguraram uma ponta deste trabalho, o meu obrigado.

DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL

Em conformidade com o Código de Conduta para Autores em Publicações da Sociedade Brasileira de Computação (SBC), declara-se o uso de ferramentas de inteligência artificial generativa na preparação deste trabalho.

A ferramenta Claude (Anthropic) foi empregada como apoio em: (i) revisão linguística e estilística do texto (clareza, naturalidade, padronização de terminologia e pontuação); (ii) assistência à redação de trechos a partir de conteúdo, dados e resultados definidos pelo autor; e (iii) geração de scripts auxiliares para a produção das figuras a partir dos dados experimentais.

A concepção do estudo, o desenho experimental, a execução da bateria de ensaios, a coleta e a análise dos dados, a interpretação dos resultados e as conclusões são de autoria exclusiva do autor. Nenhum conteúdo gerado por inteligência artificial foi utilizado como fonte primária; o autor revisou integralmente o texto e assume total responsabilidade por seu conteúdo.

"No meio do inverno, aprendi por fim que havia em mim um verão invencível."

— Albert Camus, *L'Été* (1954)

RESUMO

Sistemas de banco de dados PostgreSQL sofrem degradação severa de desempenho sob alta concorrência de conexões, em razão do modelo *process-per-connection* e da contenção em estruturas internas de *locking*. Os *connection poolers* são a mitigação padrão da indústria, mas a decisão de adotá-los costuma se apoiar em heurísticas, e não em evidência quantitativa, pois falta uma caracterização pública do ponto de concorrência a partir do qual o *pooler* compensa a sobrecarga que introduz. Este trabalho investiga essa transição por meio de uma análise empírica controlada de seis configurações de conexão sobre PostgreSQL 18 em nuvem AWS: conexão direta (*baseline*), PgBouncer e PgCat nos modos *transaction* e *session*, e o AWS RDS Proxy. Ao longo de 1.332 ensaios, variaram-se a carga de trabalho, o modo de *query*, a concorrência e o tamanho do *pool*, com instrumentação não apenas do cliente, mas também do interior do servidor. Os resultados indicam a existência de um ponto de equilíbrio: em baixa concorrência, o *pooler* apenas acrescenta latência; acima de algumas dezenas de clientes (cerca de 30 na carga de escrita), a conexão direta colapsa por contenção, enquanto o *pooler* mantém a vazão estável. A instrumentação do servidor atribui esse ganho à redução da contenção interna de *locks*, e não a um custo de conexão menor. A arquitetura multi-thread (PgCat) só supera a single-thread (PgBouncer) quando o próprio *pooler* se torna o recurso crítico, e o *pooling* gerenciado (RDS Proxy) apresenta teto de vazão inferior ao das alternativas auto-hospedadas. O estudo oferece uma caracterização aberta e reprodutível desse ponto de equilíbrio para PostgreSQL, sustentada por evidência causal coletada no servidor.

Palavras-chave: PostgreSQL. *Connection pooling*. PgBouncer. PgCat. Contenção de *locks*. Computação em nuvem.

ABSTRACT

PostgreSQL database systems suffer severe performance degradation under high connection concurrency, owing to their *process-per-connection* model and contention in internal locking structures. *Connection poolers* are the industry-standard mitigation, but the decision to adopt one usually rests on heuristics rather than quantitative evidence, since there is no public characterization of the concurrency point beyond which a pooler offsets the overhead it introduces. This work investigates that transition through a controlled empirical analysis of six connection configurations over PostgreSQL 18 in an AWS cloud environment: a direct connection (*baseline*), PgBouncer and PgCat in *transaction* and *session* modes, and AWS RDS Proxy. Across 1.332 trials, the workload, query mode, concurrency, and pool size were varied, with instrumentation of both the client and the server internals. The results indicate a *break-even point*: at low concurrency the pooler merely adds latency; beyond a few dozen clients (about 30 under the write workload), the direct connection collapses under contention while the pooler keeps throughput stable. Server-side instrumentation attributes this gain to reduced internal lock contention rather than to a lower connection cost. The multi-threaded pooler (PgCat) outperforms the single-threaded one (PgBouncer) only when the pooler itself becomes the critical resource, and managed pooling (RDS Proxy) shows a lower throughput ceiling than the self-hosted alternatives. The study provides an open, reproducible characterization of this break-even for PostgreSQL, grounded in causal evidence collected at the server.

Keywords: PostgreSQL. Connection pooling. PgBouncer. PgCat. Lock contention. Cloud computing.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AWS	*Amazon Web Services*
AZ	*Availability Zone* (zona de disponibilidade)
CPU	*Central Processing Unit*
CSV	*Comma-Separated Values*
EC2	*Elastic Compute Cloud*
ECS	*Elastic Container Service*
FIFO	*First In, First Out*
IaC	*Infrastructure as Code*
MD5	*Message-Digest Algorithm 5*
NAT	*Network Address Translation*
OLTP	*Online Transaction Processing*
PK	*Primary Key* (chave primária)
RDS	*Relational Database Service*
SCRAM	*Salted Challenge Response Authentication Mechanism*
TLS	*Transport Layer Security*
TPC-B	*TPC Benchmark B* (Transaction Processing Performance Council)
TPS	*Transactions Per Second* (transações por segundo)
vCPU	*virtual CPU*
VPC	*Virtual Private Cloud*
WAL	*Write-Ahead Log*

LISTA DE SÍMBOLOS

c^*	*break-even point*: concorrência a partir da qual o *pooler* compensa sua sobrecarga
c	número de clientes simultâneos (concorrência)

SUMÁRIO

1 INTRODUÇÃO

Bancos de dados relacionais são a espinha dorsal da maioria das aplicações *online*, e o PostgreSQL consolidou-se como um dos sistemas de código aberto mais adotados nesse cenário. À medida que uma aplicação cresce, cresce também o número de clientes que disputam o banco simultaneamente, e é justamente sob alta concorrência de conexões que o PostgreSQL revela uma de suas limitações arquiteturais mais conhecidas: o modelo *process-per-connection*, no qual cada conexão de cliente é atendida por um processo dedicado do sistema operacional, denominado *backend*. Esse modelo, simples e sólido em baixa escala, torna-se um gargalo em escala alta, pois cada novo *backend* consome memória, agrava a contenção em estruturas internas de *locking* e amplia o custo de coordenação entre processos (POSTGRESQL WIKI, 2014; FREUND, 2020b).

Para tornar o problema concreto, considere uma loja virtual em um dia de pico como a *Black Friday*: dezenas de milhares de clientes navegam e finalizam compras ao mesmo tempo, e cada requisição da aplicação precisa falar com o banco de dados. Se cada uma abrir a própria conexão, o PostgreSQL passa a gastar mais recursos coordenando processos do que respondendo consultas, e o tempo de resposta se degrada justamente no momento de maior receita. É esse tipo de cenário que exige conter o número de conexões abertas ao banco.

A resposta padrão da indústria a esse problema é o *connection pooling*: interpor entre a aplicação e o banco um intermediário (o *connection pooler*) que multiplexa um grande número de conexões de cliente sobre um conjunto pequeno e estável de conexões reais ao servidor. Ferramentas como PgBouncer e PgCat são amplamente recomendadas, e provedores de nuvem oferecem soluções gerenciadas como o AWS RDS (*Relational Database Service*) Proxy. Há, no entanto, uma lacuna prática persistente: **a decisão de adotar um *pooler*, e de como configurá-lo, costuma ser tomada por heurística e convenções de engenharia, não por evidência quantitativa controlada**. Camargos (2018), em uma fonte industrial de referência, reconhece ter pulado justamente a faixa de concorrência onde o *trade-off* se decide.

Este trabalho aborda essa lacuna. Por meio de uma bateria experimental controlada e reprodutível, executada em ambiente de nuvem com instrumentação tanto do lado do cliente quanto do lado do servidor, investiga-se em quais condições, e por quê, um *connection pooler* é recomendável para melhorar o desempenho de um sistema PostgreSQL.

1.1 Contextualização e problema

A sobrecarga de uma conexão PostgreSQL não é desprezível. Cada conexão ociosa custa memória da ordem de alguns megabytes (FREUND, 2020c), e, mais importante para este trabalho, cada *backend* ativo participa da coordenação interna de *locks* (bloqueios). O PostgreSQL utiliza um mecanismo de *fast-path locking* para travas de relação: cada *backend* dispõe de um conjunto próprio de entradas rápidas que, no PostgreSQL 18, escala com `max_locks_per_transaction` (64 por padrão), ante o limite fixo de dezesseis das versões anteriores. Quando esse conjunto se esgota, ou quando muitos processos disputam as mesmas partições da tabela de *locks*, o sistema recai sobre um caminho lento que se manifesta no *wait event* `LWLock:LockManager` (KUMAR; SINGH; VENKATRAMAN, 2025; FITTL, 2023). Esse *wait event* cresce com o número de *backends* físicos e é um indicador direto de que o banco está gastando tempo coordenando, e não executando trabalho útil. Demonstrações experimentais em serviços gerenciados e casos reais de produção, como o do GitLab, documentam esse mesmo padrão de degradação (OTS, 2024; FITTL, 2023).

Uma percepção comum é que "conexão direta é sempre mais rápida, pois o *pooler* é apenas um passo a mais no caminho". Essa percepção é válida apenas em um regime, o de baixa concorrência, e inválida no outro, onde a conexão direta colapsa por contenção. O problema central, então, não é *se* um *pooler* ajuda, mas **a partir de qual ponto de concorrência ele passa a ajudar**, e como esse ponto varia conforme a carga, o modo de *query* e o tamanho do *pool*. Esse ponto de equilíbrio, denominado neste trabalho *break-even point* e denotado c^*, é o objeto central deste estudo.

1.2 Pergunta de pesquisa e hipóteses

Este trabalho investiga:

> *A partir de qual nível de concorrência (número de clientes simultâneos) a sobrecarga introduzida por um connection pooler é compensada pela redução de contenção de recursos que ele provoca no servidor PostgreSQL? E como esse ponto de equilíbrio (c^*) varia entre diferentes poolers, modos de operação e perfis de carga?*

A partir dessa pergunta, formulam-se as hipóteses que o experimento testa. A hipótese geral é que *existe* um ponto de concorrência, o *break-even*, abaixo do qual o *pooler* degrada o desempenho percebido e acima do qual se torna necessário para evitar o colapso por contenção, com a posição desse ponto variando conforme o *pooler* e o perfil de carga. Dela derivam cinco subhipóteses verificáveis:

H1 (*break-even* por carga). c^* é menor para cargas dominadas por escrita do que para cargas somente leitura, pois a contenção em bloqueios como `Lock:tuple` e `Lock:transactionid` ocorre mais cedo na escrita.

H2 (*transaction* vs. *session*). O modo *transaction* apresenta c^* menor que o modo *session*, porque a multiplexação por transação compensa a sobrecarga já com poucos clientes a mais e é elástica a concorrências muito superiores ao tamanho do *pool*.

H3 (*multi-thread*, hipótese central). Um *pooler* de múltiplas *threads* (*multi-threaded*), como o PgCat, em Rust sobre o ambiente de execução *tokio*, sustenta vazão maior que um *pooler* de *thread* única (*single-threaded*), como o PgBouncer, em C e orientado a eventos, sob concorrência alta, *quando o host do pooler tem múltiplos núcleos*, porque o *pooler* de *thread* única satura um núcleo e passa a ser ele próprio o gargalo (BINIDXABA, 2024).

H4 (gerenciado vs. auto-hospedado). O *pooling* gerenciado (AWS RDS Proxy) adiciona sobrecarga de rede (um salto extra dentro da AWS) e *pinning* conservador, ou seja, a fixação de um cliente a uma mesma conexão de servidor, em troca de remover a operação; espera-se vazão/latência abaixo dos *poolers* auto-hospedados (RAWAT, 2025; AWS RE:POST COMMUNITY, 2021).

H5 (modo *prepared*). O modo `prepared` reduz c^* para todos os *poolers*, com ganho relativo maior em PgBouncer-*transaction* 1.21+, onde *prepared statements* eram historicamente inacessíveis (PGBOUNCER AUTHORS, 2023).

1.3 Objetivos

A partir do problema exposto, o objetivo geral deste trabalho é caracterizar empiricamente o *break-even point* c^* de *connection poolers* para PostgreSQL e identificar, a partir de evidência *server-side*, o mecanismo causal da transição. Para alcançá-lo, foram definidos os seguintes objetivos específicos:

1. Estimar c^* para cada combinação de *pooler*, carga e modo, com base em repetições e intervalos de confiança.
2. Atribuir o colapso de vazão à contenção interna, com base em evidência do servidor (distribuição de *wait events* e contenção de *locks*), distinguindo "cliente esperando na fila do *pooler*" de "cliente esperando *lock* no PostgreSQL".
3. Comparar diretamente a arquitetura *single-thread* (PgBouncer) e *multi-thread* (PgCat) sob carga, em *host* multi-core.
4. Posicionar o *pooling* gerenciado (RDS Proxy) frente aos auto-hospedados.

5. Documentar ao menos um regime em que o *pooler piora* o desempenho, comprovando o *break-even* e não o benefício genérico do *pooling*.

6. Entregar um experimento integralmente reprodutível, com a infraestrutura descrita em código, a orquestração automatizada e os dados regeneráveis pela reexecução da bateria.

1.4 Contribuições

Do desenho experimental e dos resultados obtidos derivam quatro contribuições principais:

1. **Uma caracterização aberta e reprodutível do c^* para PostgreSQL com atribuição causal *server-side***, cobrindo a faixa de concorrência que, conforme Camargos (2018), não foi delimitada pela literatura industrial.

2. **A evidência de que o *pooler realoca* a contenção, em vez de eliminá-la.** Ao coletar amostras simultâneas da fila do *pooler* (`cl_waiting`) e dos *locks* internos do banco (`pg_locks`, *wait events*), mostra-se que a espera migra do gerenciador de *locks* para a fila ordenada do *pooler*, um enquadramento que os *benchmarks client-side* usuais não conseguem expor.

3. **A demonstração de que a vantagem *multi-thread* é *condicional*.** Contra a leitura corrente de que "*multi-thread* é mais rápido", evidencia-se que o ganho do PgCat sobre o PgBouncer só emerge quando o regime de *query* coloca o *pooler* no caminho crítico, sendo nulo caso contrário.

4. **Um artefato experimental reutilizável**: o ambiente em nuvem descrito na linguagem de infraestrutura Terraform e o orquestrador em Python, versionados, mais o conjunto de 1.332 ensaios, regenerável pela nova execução da bateria, que permitem executar novamente ou estender o estudo.

1.5 Organização do trabalho

O restante desta monografia está organizado como segue. O Capítulo 2 apresenta a fundamentação teórica: o modelo de conexões do PostgreSQL, os mecanismos de contenção interna e as arquiteturas dos *poolers* comparados. O Capítulo 3 revisa os trabalhos relacionados, situando a contribuição deste estudo frente à literatura acadêmica e industrial. O Capítulo 4 descreve a metodologia: o ambiente experimental em nuvem, o desenho fatorial da bateria, as métricas coletadas e o tratamento dado aos colapsos. O Capítulo 5 apresenta os resultados, organizados por hipótese. O Capítulo 6 discute as implicações e, em especial, as ameaças à validade. O Capítulo 7 conclui e aponta trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo estabelece os conceitos necessários para interpretar os resultados. A Seção 2.1 revisa os fundamentos de sistemas de gerenciamento de bancos de dados. A Seção 2.2 descreve o modelo de conexões do PostgreSQL e por que ele limita a escalabilidade. A Seção 2.3 detalha os mecanismos internos de contenção, em particular o *wait event* que serve de métrica-chave deste trabalho. A Seção 2.4 explica o que é *connection pooling* e seus modos. A Seção 2.5 apresenta os *poolers* comparados e a distinção arquitetural entre *thread* única e múltiplas *threads* que motiva a hipótese central. A Seção 2.6 trata de autenticação e cifragem, relevantes para as ameaças à validade.

2.1 Fundamentos de sistemas de gerenciamento de bancos de dados

Antes de tratar do modelo de conexões do PostgreSQL, convém fixar os conceitos sobre os quais o restante do capítulo se apoia. Um *sistema de gerenciamento de banco de dados* (SGBD) é o *software* que medeia o acesso a um conjunto organizado de dados: oferece às aplicações uma interface declarativa, tipicamente SQL, e encarrega-se, internamente, do armazenamento, da recuperação, do controle de integridade, da concorrência e da segurança dos dados (ELMASRI; NAVATHE, 2016; SILBERSCHATZ; KORTH; SUDARSHAN, 2020). Ao concentrar essas responsabilidades, isola a aplicação dos detalhes físicos de armazenamento e assegura que múltiplos usuários vejam um estado consistente da base. Três noções desse arcabouço são suficientes para acompanhar os argumentos deste capítulo.

Paradigma cliente-servidor. Os SGBD relacionais modernos operam segundo o modelo cliente-servidor: um processo servidor, residente em uma máquina dedicada, mantém os dados e atende às requisições de vários processos cliente, que se comunicam com ele por meio de uma conexão de rede (ELMASRI; NAVATHE, 2016). Cada cliente abre uma conexão, submete comandos e recebe resultados sem conhecer a organização interna do servidor. É essa separação que torna a conexão, e o seu custo, um objeto de estudo próprio: ela é o canal por onde toda a carga de trabalho trafega.

Concorrência. Um servidor de banco de dados atende muitos clientes ao mesmo tempo, e disso decorrem duas formas distintas de concorrência. A primeira é a concorrência *de dados*: transações simultâneas que acessam os mesmos registros precisam ser coordenadas para preservar a consistência, o que o SGBD realiza por meio de mecanismos de controle de concorrência, como bloqueios (*locks*) e níveis de isolamento (SILBERSCHATZ; KORTH; SUDARSHAN, 2020). A segunda é a concorrência *de execução*: o servidor precisa de alguma estratégia para processar as requisições de

vários clientes em paralelo, seja dedicando um *processo* do sistema operacional a cada conexão, seja multiplexando as conexões sobre um conjunto de *threads*. Um programa de *thread* única (*single-threaded*) executa em um só fluxo de controle e, portanto, aproveita no máximo um núcleo de CPU; um programa de múltiplas *threads* (*multi-threaded*) reparte o trabalho por vários fluxos e pode ocupar vários núcleos ao mesmo tempo. Essa distinção reaparece tanto no modelo de conexões do PostgreSQL (Seção 2.2) quanto na comparação entre os *poolers* (Seção 2.5).

Segurança. Cabe ainda ao SGBD controlar quem acessa os dados e proteger o tráfego entre cliente e servidor. Isso envolve *autenticação*, a verificação da identidade de quem se conecta, em geral por senha submetida a um protocolo de desafio-resposta; *autorização*, a determinação de quais operações cada usuário pode executar; e *cifragem* do tráfego em trânsito, que impede a leitura, por terceiros, dos dados transmitidos pela rede (SILBERSCHATZ; KORTH; SUDARSHAN, 2020). Esses mecanismos têm custo, e parte dele incide sobre o estabelecimento da conexão, ponto retomado na Seção 2.6.

2.2 O modelo de conexões do PostgreSQL

O PostgreSQL adota o modelo *process-per-connection*: para cada conexão de cliente, o processo supervisor (*postmaster*) cria um processo *backend* dedicado, que vive enquanto a conexão existir. Esse desenho favorece o isolamento (a falha de um *backend* não derruba os demais), mas impõe três custos que crescem com a concorrência (POSTGRESQL WIKI, 2014; FREUND, 2020b):

1. **Memória.** Cada *backend* reserva estruturas próprias (*cache* de catálogo, *work_mem*, *buffers* locais). A medição empírica do custo residente situa-o na ordem de poucos megabytes por conexão ociosa (FREUND, 2020c), valor que, multiplicado por centenas ou milhares de conexões, deixa de ser desprezível.

2. **Custo de escalonamento.** Centenas de processos competindo pelos núcleos disponíveis aumentam a troca de contexto e a pressão sobre o escalonador do sistema operacional.

3. **Contenção interna.** Mais *backends* significam mais participantes na coordenação de estruturas compartilhadas do PostgreSQL, o ponto central deste trabalho, detalhado a seguir.

A recomendação histórica da comunidade é manter o número de conexões ativas próximo a um pequeno múltiplo do número de núcleos disponíveis (POSTGRESQL WIKI, 2014; WOOLDRIDGE, 2021). Quando a aplicação precisa de muito mais conexões do

que isso (cenário comum em arquiteturas web com muitos processos ou *serverless*), o *connection pooling* torna-se necessário.

2.3 Contenção interna e o *wait event* `LWLock:LockManager`

Quando um *backend* não está executando trabalho útil, ele está *esperando* por algum recurso, e o PostgreSQL registra a natureza dessa espera em `pg_stat_activity` sob a forma de *wait events* (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2025c; YEN, 2026). Os relevantes para cargas de escrita concorrente são:

`Lock:transactionid` / `Lock:tuple`. Backends aguardando outra transação liberar uma linha. É a contenção *aplicacional* inerente à escrita concorrente sobre os mesmos dados, inevitável quando muitos clientes atualizam o mesmo conjunto de linhas.

`LWLock:LockManager`. Contenção *interna* do gerenciador de *locks*. Cada *backend* mantém um conjunto de *fast-path locks*, um atalho que dispensa a trava global enquanto a transação adquire poucas travas fracas de relação. Esse conjunto era fixo em dezesseis até o PostgreSQL 17; no PostgreSQL 18 (versão deste estudo) ele passou a escalar com `max_locks_per_transaction` (64 por padrão) (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2025b). Esgotado o atalho, seja por muitas travas numa única transação, seja por muitos *backends* concorrendo pelas mesmas partições da tabela de *locks*, a aquisição cai no caminho global e serializado, que aflora como este *wait event* (KUMAR; SINGH; VENKATRAMAN, 2025). A intensidade desse *wait event* **cresce com o número de backends físicos**, o que faz dele um indicador direto do custo de coordenação (FITTL, 2023; OTS, 2024).

`IO:WalSync`. Espera pela persistência em disco do *Write-Ahead Log* (WAL), o registro sequencial em que o PostgreSQL grava cada alteração antes de aplicá-la para garantir durabilidade; é um limite físico de escrita, ortogonal ao número de conexões.

Sem *wait event* (`wait_event` nulo). Trabalho útil: o *backend* está executando em CPU, sem esperar por nenhum recurso.

A escolha de `LWLock:LockManager` como métrica-chave deste estudo é proposital: por crescer com a contagem de *backends* e não com a carga de trabalho em si, ele isola justamente o efeito que o *pooler* deveria mitigar. Se um *pooler* reduz o número de *backends* físicos de centenas para poucas unidades, espera-se uma queda acentuada nesse *wait event*, e essa queda é a evidência causal de *por que* o *pooler* compensa, e não apenas *que* ele compensa.

Mesmo em cargas somente leitura, a contenção pode emergir em estruturas como a `ProcArrayLock`, formalmente caracterizada como gargalo dominante em alta concorrência e

parcialmente mitigada a partir do PostgreSQL 14 pela reformulação do `GetSnapshotData()` (FREUND, 2020b,a). Isso justifica por que a contenção (e o benefício do *pooling*) não é exclusiva da escrita, ainda que apareça mais cedo nela.

2.4 *Connection pooling*: conceito e modos

Um *connection pooler* é um intermediário que mantém um conjunto pequeno e estável de conexões reais ao PostgreSQL e as compartilha entre um número muito maior de clientes. A aplicação conecta-se ao *pooler* como se fosse o banco; o *pooler* arbitra qual cliente usa qual conexão de servidor. Há dois modos principais de operação, com implicações importantes para este estudo, conforme PgBouncer Authors (2026a) e Moppel (2016):

Modo *session*. A conexão de servidor é "alugada" ao cliente pela duração inteira de sua sessão e só retorna ao *pool* quando o cliente desconecta. É o modo mais compatível, pois preserva os recursos ligados à sessão (*session-scoped*): variáveis de `SET`, *cursores* que percorrem o resultado de uma consulta e canais de `LISTEN/NOTIFY`. É, porém, o menos elástico. Se há mais clientes ativos que conexões no *pool*, os excedentes ficam *bloqueados* aguardando. Em termos experimentais, isso significa que combinações com *quantidade de clientes* > *tamanho do pool* são arquiteturalmente inviáveis nesse modo.

Modo *transaction*. A conexão de servidor retorna ao *pool* ao final de *cada transação*. É muito mais elástico (mil clientes podem revezar sobre duas conexões de servidor), ao custo de incompatibilidade com recursos que atravessam transações (PGBOUNCER AUTHORS, 2026b). É o modo que viabiliza concorrências muito superiores ao tamanho do *pool* e, portanto, o regime onde o ganho do *pooling* é maior.

O modo *transaction* tem um ponto de atenção relacionado aos *prepared statements*, comandos que o banco analisa e planeja uma única vez para reutilizar nas execuções seguintes, poupando esse custo. Como, nesse modo, cada transação pode cair em uma conexão de servidor diferente, um *statement* preparado em uma transação podia não existir na conexão usada pela transação seguinte, o que historicamente tornava os dois recursos incompatíveis. O PgBouncer 1.21 (outubro de 2023) passou a rastrear os *prepared statements* e a prepará-los sob demanda em cada conexão, removendo essa limitação. É essa mudança recente que a hipótese H5 investiga, ao medir se, e quanto, o modo `prepared` eleva o desempenho quando combinado com o *pooling* por transação (PGBOUNCER AUTHORS, 2023; MULLANE, 2023).

2.5 Os *poolers* comparados

Foram medidos três *poolers*, escolhidos por serem *poolers puros* (sua única função é multiplexar conexões), o que os torna diretamente comparáveis:

PgBouncer. *Pooler* leve e maduro, escrito em C e de *thread* única (*single-threaded*): processa todas as conexões em um só fluxo de execução, via laço de eventos. Suporta os modos *session* e *transaction* e, desde a versão 1.21, rastreia *prepared statements* em *transaction mode*. Por usar uma só *thread*, sua vazão é limitada pela capacidade de um único núcleo de CPU; sob carga alta, o próprio PgBouncer pode tornar-se o gargalo (PGBOUNCER AUTHORS, 2026a; CAMARA, 2023). É notoriamente econômico em memória.

PgCat. *Pooler* escrito em Rust sobre o ambiente de execução assíncrono *tokio*, de múltiplas *threads* (*multi-threaded*): o número de *worker threads* é configurável e permite distribuir o processamento por vários núcleos. Suporta os modos *session* e *transaction* e *prepared statements* em *transaction mode*. Mantido pela PostgresML, posiciona-se como alternativa ao PgBouncer com escalabilidade *multi-core*. Recursos opcionais de *sharding* (o particionamento horizontal dos dados entre múltiplos servidores de banco) e balanceamento de carga existem, mas não foram exercidos neste trabalho para manter a paridade de "*pooler* puro". É a peça que viabiliza a hipótese H3 (POSTGRESML, 2026; BINIDXABA, 2024).

AWS RDS Proxy. *Pooling* gerenciado pela AWS (*transaction pooling*). Escala automaticamente, não exige *host* dedicado, conecta ao *backend* sempre via TLS (*Transport Layer Security*) e é cobrado por vCPU (*virtual CPU*). Em troca, adiciona um salto de rede dentro da AWS (a "perna gerenciada") e aplica *pinning* conservador (a fixação de um cliente a uma mesma conexão de servidor, suspendendo a multiplexação) diante de recursos *session-scoped* (AMAZON WEB SERVICES, 2026b). Permite testar a hipótese H4, que prevê vazão e latência inferiores às dos *poolers* auto-hospedados em troca da conveniência operacional.

A diferença de arquitetura entre os dois *poolers* auto-hospedados é central para este estudo. Um *pooler* é, no fundo, um *proxy* que copia bytes entre o *socket* do cliente e o do *backend* e arbitra qual transação usa qual conexão de servidor. Esse trabalho é leve por transação, mas sob concorrência alta o volume agregado de *I/O* e o custo de *parsing* e de roteamento podem saturar uma CPU. Como o PgBouncer executa tudo em uma *thread*, num *host* com vários núcleos ele usa no máximo um deles; quando esse núcleo atinge 100%, sua vazão estaciona, independentemente de quanta CPU sobra na máquina, e o *pooler* torna-se o gargalo, não o banco. O PgCat, ao repartir o trabalho por várias *threads*, escala com os núcleos disponíveis e empurra o gargalo de volta para o PostgreSQL,

onde a contenção de *locks* é o limite real. Daí a hipótese H3: em concorrência alta e *host multi-core*, espera-se que o PgCat sustente vazão superior à do PgBouncer. Em um *host* de poucos núcleos o efeito tende a desaparecer, e por isso o desenho experimental em nuvem (Capítulo 4) aloca intencionalmente uma instância de 4 vCPU ao *pooler*, dando folga de núcleos para o *multi-threading* do PgCat se manifestar.

2.6 Autenticação e cifragem

Dois eixos são relevantes para as ameaças à validade:

Autenticação: MD5 vs. SCRAM. O PostgreSQL suporta autenticação por desafio-resposta baseada em MD5 (legado) e em SCRAM-SHA-256 (padrão desde o PG 10, controlado por `password_encryption`). SCRAM é criptograficamente mais forte. O método de autenticação afeta o *handshake* (a negociação inicial) da conexão, **não** a vazão nem a latência das transações em regime, que são precisamente as variáveis medidas neste estudo; o método de autenticação é, portanto, ortogonal ao que se quer medir.

Cifragem: TLS. O tráfego pode ser cifrado com TLS. No RDS, `rds.force_ssl=1` (padrão para PostgreSQL 15 e posteriores) *obriga* TLS em todas as conexões de *backend*, e o RDS Proxy o impõe na perna gerenciada. Isso adiciona custo de *handshake* e cifragem, mas, se aplicado *uniformemente* a todas as configurações de conexão comparadas, não afeta a comparação *relativa* entre elas.

As escolhas adotadas (MD5 global e TLS uniforme no *backend*) são impostas por limitações de paridade entre as seis configurações de conexão do experimento e estão detalhadas no Capítulo 6.

3 TRABALHOS RELACIONADOS

A literatura sobre *connection pooling* para PostgreSQL distribui-se em três frentes: trabalhos acadêmicos sobre *proxies* de banco de dados e escalabilidade de conexões; estudos empíricos industriais comparando *poolers*; e a documentação técnica das ferramentas. Esta seção percorre as três e, ao final, posiciona a contribuição deste trabalho frente à lacuna identificada por Camargos (2018).

3.1 Trabalhos acadêmicos

O estudo acadêmico mais próximo é o de Butrovich et al. (2023), que mede PgBouncer, Pgpool-II e Odyssey lado a lado sobre PostgreSQL. Trata-se, porém, de um *baseline* para propor um *proxy* novo (Tigger), e não de uma caracterização do *break-even point*. O objetivo é demonstrar o ganho de uma técnica de *user-bypass*, não investigar *quando* um *pooler* convencional compensa. O presente trabalho ocupa justamente essa lacuna, usando as mesmas ferramentas, mas com a pergunta invertida.

Do lado da escalabilidade interna do PostgreSQL, Freund (2020b) caracteriza formalmente o custo de varredura do `ProcArray` (na rotina `GetSnapshotData()`) como limite de escalabilidade de conexões em alta concorrência e documenta sua reformulação no PostgreSQL 14. Esse trabalho fundamenta *por que* a contenção emerge sob alta concorrência mesmo em cargas somente leitura, sustentando a premissa central deste estudo. Já Markwort (2024), em palestra recente da comunidade, avalia diferentes opções de *pooler*, evidenciando que o tema é ativo, mas sem produzir a caracterização quantitativa do *trade-off*.

3.2 Estudos empíricos industriais

Um estudo industrial de referência é o de Camargos (2018), da Percona, cujo próprio título, "*you may need a connection pooler sooner than you expect*", indica que o *pooler* compensa antes do que a intuição sugere. É também a fonte que **deixou de medir a faixa de 56 a 150 clientes** ("*the jump was too big*"), criando a oportunidade direta deste TCC: preencher empiricamente a faixa de concorrência onde o *trade-off* se decide.

O benchmark da Binidxaba (2024) mede PgBouncer, PgCat e Supavisor em TPS, latência e CPU, e reporta que o **PgBouncer satura por ser *single-thread*** acima de um certo número de conexões de servidor, enquanto o PgCat (*multi-thread*) escala melhor. Esse resultado é a motivação direta da hipótese H3 e da inclusão do PgCat na matriz com um *pool* super-dimensionado. Por tratar-se de *blog* corporativo de um provedor com *pooler*

próprio, este trabalho interpreta esse resultado tendo o viés em mente e mede novamente o efeito de forma independente.

Comparações qualitativas amplamente citadas, como as de Anderson (2020) e Mannem (2019) sobre PgBouncer *vs.* Pgpool-II, fundamentam as diferenças arquiteturais entre *pooler* puro e *middleware*. Análises críticas de produção feitas por Camara (2023) documentam falhas de recursos com escopo de sessão sob *pooling* em modo *transaction*, aceitas pelo PgBouncer sem erro ou aviso (*prepared statements* nomeados, *advisory locks* e `LISTEN/NOTIFY`), úteis para a discussão de limitações. No eixo de contenção, o *blog* de banco de dados da AWS (KUMAR; SINGH; VENKATRAMAN, 2025) explica o mecanismo de *fast-path locking* e quando ele degrada para o caminho lento; demonstrações experimentais em Aurora (OTS, 2024) e o *post-mortem* do GitLab (FITTL, 2023) mostram que a contenção em `LWLock:LockManager` não é um problema acadêmico, mas operacional e em escala real.

Quanto ao *pooling* gerenciado, relatos independentes de produção apontam que o RDS Proxy adiciona latência e impõe limitações operacionais (RAWAT, 2025), e um relato da comunidade AWS reporta tempos de resposta duas a três vezes maiores em cargas de baixíssima latência (AWS RE:POST COMMUNITY, 2021), justamente o tipo de sobrecarga de rede que a hipótese H4 busca quantificar de forma controlada.

3.3 Documentação técnica das ferramentas

A documentação oficial do PgBouncer (PGBOUNCER AUTHORS, 2026a,b) fornece a referência de `pool_mode`, `default_pool_size` e `max_prepared_statements`, parâmetros diretamente usados na configuração do experimento, além de especificar quais recursos são incompatíveis com cada modo. O suporte a *prepared statements* em *transaction mode* (1.21+) é documentado tanto pela comunidade (PGBOUNCER AUTHORS, 2023) quanto em análises técnicas com *benchmark* (MULLANE, 2023). Para o PgCat, a referência é o repositório oficial (POSTGRESML, 2026); a ausência de suporte a SCRAM *client-side*, registrada na *issue* #255 (o relato público de defeito ou limitação que os projetos de código aberto mantêm em seu repositório), é relevante para a decisão de usar MD5 global, discutida no Capítulo 6. A documentação do RDS Proxy (AMAZON WEB SERVICES, 2026b) e do PostgreSQL (em particular o sistema de estatísticas cumulativas e o `pg_stat_io` reformulado no PG 18 (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2025c,d; EMMETT, 2025)) embasa a instrumentação *server-side*.

3.4 Posicionamento e lacuna

A Tabela 1 sintetiza o posicionamento. A literatura acadêmica reconhece a importância do tema (o custo da escalabilidade de conexões e o papel do *pooling* em PostgreSQL), mas

avalia os *poolers* existentes apenas como referência de comparação ao propor novos *proxies*, como o Tigger, sem caracterizar o ponto de equilíbrio; a literatura industrial mede TPS/latência/CPU, porém de forma *client-side* e frequentemente saltando a faixa de concorrência crítica. Em nenhuma das frentes encontra-se, de forma conjunta: (i) caracterização explícita do c^*, (ii) com instrumentação *server-side* que *explique* o colapso, (iii) comparando *single-thread* e *multi-thread* como variável isolada, e (iv) com reprodutibilidade integral. É essa interseção que o presente trabalho preenche.

Tabela 1 – Posicionamento frente à literatura.

Fonte	**Mede c^***	***Server-side***	**single vs. multi**	**Reprod.**
Butrovich et al. (2023) (acad.)	não	parcial	não	parcial
Camargos (2018) (ind.)	parcial*	não	não	não
Binidxaba (2024) (ind.)	não	não	sim	não
Este trabalho	**sim**	**sim**	**sim**	**sim**

* Reconhece o *break-even*, mas omite a faixa de concorrência onde ele se decide.

Fonte: Elaborado pelo autor.

4 METODOLOGIA

Este capítulo descreve o desenho experimental. A Seção 4.1 apresenta o ambiente de nuvem; a Seção 4.2, o desenho fatorial e a contagem de ensaios; a Seção 4.3, as cargas de trabalho; a Seção 4.4, as métricas e a instrumentação *server-side*; a Seção 4.5, o tratamento dado aos colapsos; e a Seção 4.6, os mecanismos de reprodutibilidade.

4.1 Ambiente experimental

O estudo principal foi conduzido na nuvem AWS, com **três *hosts* separados**, numa única zona de disponibilidade, de modo a medir a sobrecarga de rede real entre cliente, *pooler* e banco, um elemento ausente de experimentos que rodam todos os componentes num mesmo *host*. A topologia está resumida na Tabela 2: o *host* cliente executa o orquestrador da bateria, o gerador de carga *pgbench* e o *collector* de métricas; o *host* intermediário hospeda, um por vez, os *poolers* sob teste; e o banco é uma instância RDS PostgreSQL 18. Na sexta configuração de conexão, o *pooler* auto-hospedado dá lugar ao AWS RDS Proxy, serviço gerenciado que dispensa um *host* próprio.

Tabela 2 – Topologia do ambiente experimental em nuvem.

Host	**Instância**	**Papel**
Cliente	EC2 `m5.large` (2 vCPU)	orquestrador + *pgbench* + *collector*
Pooler	EC2 `m5.xlarge` (4 vCPU)	*poolers* em Docker (*host network*), um por vez
Banco	RDS PostgreSQL 18 `db.m5.large`	servidor sob teste
AWS RDS Proxy (gerenciado)		alternativa ao *pooler* EC2

Fonte: Elaborado pelo autor.

As decisões de configuração foram norteadas pela necessidade de não contaminar a medição:

Família `m5` (CPU dedicada). As instâncias do AWS EC2 são organizadas em famílias, cada uma otimizada para um tipo de carga de trabalho: as da família T (como a `t3`) destinam-se a aplicações com uso variável de CPU e operam sob um sistema de créditos que permite picos de desempenho a baixo custo, enquanto as da família M (como a `m5`) são de uso geral e oferecem equilíbrio entre processamento, memória e rede (AMAZON WEB SERVICES, 2026a). Optou-se por `m5` em vez de `t3` porque, sob dezenas de horas de carga sustentada, o esgotamento dos créditos de CPU da

`t3` reduziria a CPU disponível e contaminaria os dados *CPU-bound*; a família `m5` oferece desempenho de CPU constante.

Parâmetros do RDS fixados. O AWS RDS administra a configuração do banco por meio de *grupos de parâmetros*, que definem opções do motor como o tamanho dos *buffers*, o número máximo de conexões e as opções de *log*; alguns desses parâmetros exigem reinicialização da instância para ter efeito, e outros são gerenciados pela própria AWS e não podem ser alterados (AMAZON WEB SERVICES, 2026c). Neste experimento, fixou-se `max_connections=415` e `shared_buffers=1 GB` no grupo de parâmetros, para emular o consumo de memória de uma instância menor, preservando os pontos de colapso do experimento sem reintroduzir os créditos de CPU do `t3`. O limite de 415 conexões define, por construção, o ponto em que a conexão direta satura.

***Pooler* em 4 vCPU.** Uma vCPU (*virtual CPU*) é a unidade de processamento virtual alocada a uma instância de nuvem; quanto mais vCPUs disponíveis para o *pooler*, maior sua capacidade de processar conexões simultâneas, o que influencia diretamente os resultados observados. A instância do *pooler* (`m5.xlarge`, 4 vCPU) tem o dobro de núcleos do cliente e do banco. Essa folga de núcleos é condição necessária para a hipótese H3. Sem ela, o *multi-threading* do PgCat não teria onde se manifestar.

Custo. Em experimentos com serviços de nuvem, o custo corresponde ao valor financeiro dos recursos consumidos durante a execução: o tipo e o tamanho das instâncias (vCPUs e memória), o tempo em que as máquinas permanecem ligadas, o armazenamento, as operações de entrada e saída, o tráfego de rede e os serviços gerenciados utilizados. Somados esses componentes, o custo medido foi de US$ 0,505 por hora; a bateria completa (incluindo o reprocessamento descrito na Seção 4.5) totalizou aproximadamente US$ 21, correspondentes a cerca de 42 horas de execução ininterrupta.

Um ambiente local equivalente em Docker Compose foi mantido para desenvolvimento e verificação de sanidade do *pipeline*, mas não constitui fonte primária de dados. A topologia em nuvem pode ser vista na Figura 1. Nela, o *pooler* ocupa a posição central do caminho: multiplexa as muitas conexões de cliente sobre poucas conexões reais ao banco, absorvendo o excesso em sua fila interna (`cl_waiting`). A perna *client* → *pooler* trafega em texto claro, enquanto a perna *pooler* → RDS usa TLS (`rds.force_ssl`); o *collector*, instalado no *host* cliente, coleta amostras do estado interno do banco durante toda a medição. Na sexta configuração de conexão, o RDS Proxy substitui o *pooler* auto-hospedado.

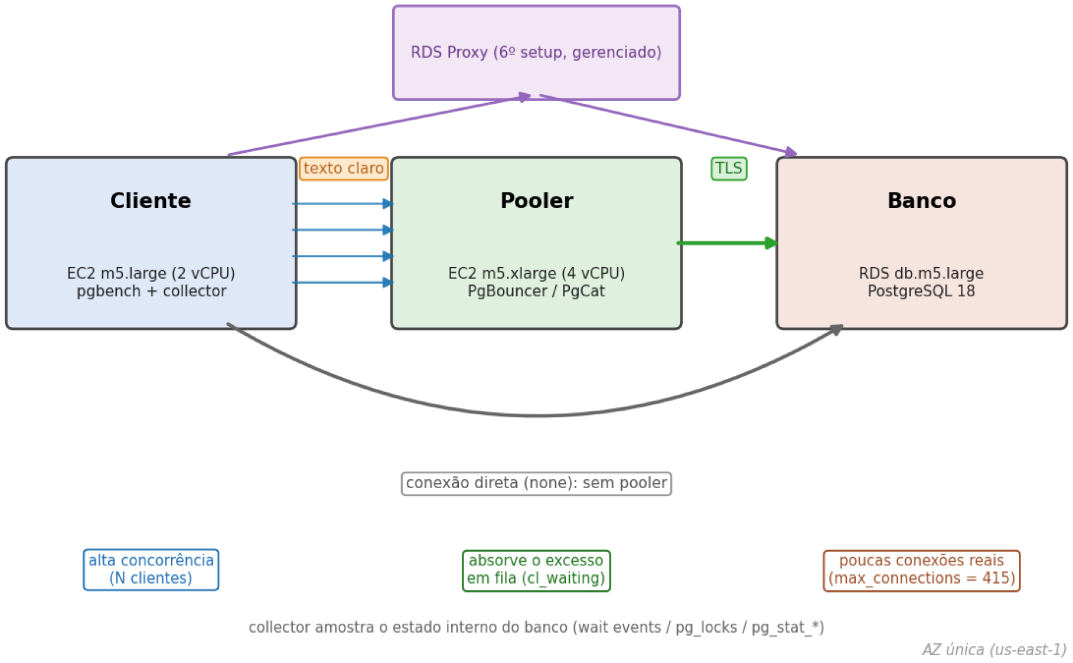

Figura 1 – Topologia do experimento em nuvem: três *hosts* dedicados em uma única zona de disponibilidade.

Fonte: Elaborado pelo autor.

4.2 Desenho da bateria

Uma bateria de testes é um conjunto organizado de testes executados de forma sistemática para avaliar o comportamento, o desempenho ou a qualidade de um sistema sob diferentes condições. A bateria deste estudo seguiu um desenho fatorial completo, cruzando seis eixos de variação (Tabela 3 e Figura 2): o *setup* de conexão (a conexão direta e os cinco arranjos de *pooler*), a carga de trabalho (leitura, escrita ou mista), o modo de *query* (`simple` ou `prepared`), o nível de concorrência (de 1 a 1.000 clientes), o tamanho do *pool* (subdimensionado, adequado ou superdimensionado) e a repetição de cada combinação, para atenuar a variância. Cada combinação executada constitui um *ensaio*: uma execução completa e independente da carga contra o sistema, da qual se extraem as métricas descritas na Seção 4.4.

Na Figura 2, cada linha corresponde a um eixo e cada caixa a um valor possível; o caminho em destaque, do *setup* à repetição, representa um ensaio completo. O produto dos seis eixos, descontadas as combinações inviáveis discutidas adiante, resulta nos 1.332 ensaios do estudo.

Tabela 3 – Eixos do desenho fatorial (predefinição “expandida”).

Eixo	**Valores**
Setups (6)	`none`, `pgbouncer-tx`, `pgbouncer-session`, `pgcat-tx`, `pgcat-session`, `rds-proxy`
Cargas (3)	*read* (SELECT por PK), *write* (UPDATE contencioso), *mixed* (TPC-B)
Modos (2)	`simple`, `prepared`
Concorrências (8)	1, 30, 50, 100, 200, 300, 500, 1.000 clientes
Tamanhos de *pool* (3)	2 (sub), 8 (adequado), 100 (super)
Repetições	3

Fonte: Elaborado pelo autor.

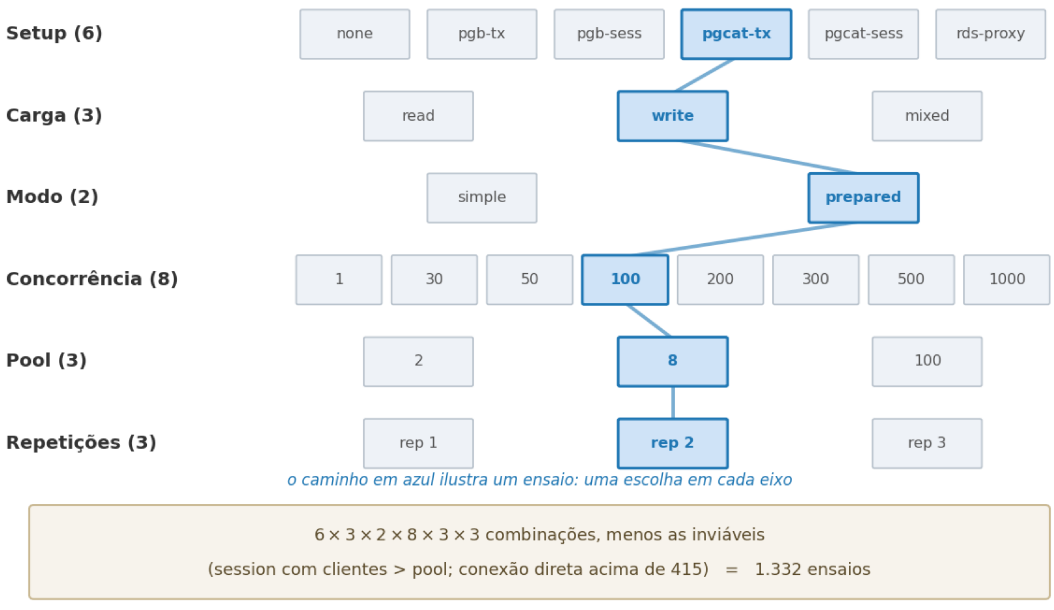

Figura 2 – Desenho fatorial da bateria: seis eixos de variação e um exemplo de combinação.

Fonte: Elaborado pelo autor.

A grade de concorrência combina amplitude e resolução: parte de 1 cliente (*baseline* sem contenção), avança por saltos aproximadamente geométricos até 1.000 clientes (regime de estresse) e adensa os pontos intermediários (50, 200 e 500) na faixa em que o *break-even* costuma emergir, para localizá-lo com precisão. Os dois maiores níveis (500 e 1.000) superam o limite de `max_connections` = 415, o que garante observar a saturação da conexão direta; e a faixa em torno de 50 a 150 clientes cobre justamente o intervalo que Camargos (2018) declarou ter omitido.

Nem toda combinação é viável. Em modo *session*, combinações com *quantidade de clientes* > *tamanho do pool* são arquiteturalmente impossíveis (os clientes excedentes ficariam bloqueados indefinidamente); no *baseline* `none`, concorrências acima de `max_connections` também são inviáveis. Tais casos são pré-marcados como `expected_failure` e não executados. Após essa filtragem, restaram 1.332 **ensaios viáveis** (e 684 combinações `expected_failure`), executados em aproximadamente 40 horas.

Os ensaios foram embaralhados dentro de cada bloco de *setup*, em vez de executados em ordem fixa. O sorteio da ordem usa uma semente fixa (42). Como um gerador de números pseudoaleatórios produz sempre a mesma sequência quando inicializado com a mesma semente, o embaralhamento é idêntico em qualquer reexecução da bateria, o que preserva a reprodutibilidade do experimento. O objetivo do embaralhamento é evitar que fatores que variam com o tempo se confundam com o efeito medido. Se as concorrências fossem executadas sempre em ordem crescente e o ambiente degradasse gradualmente ao longo das horas, os níveis mais altos seriam medidos sistematicamente no período mais degradado, e parte da queda de vazão seria atribuída à concorrência quando, na verdade, viria do ambiente.

4.3 Cargas de trabalho

Em um experimento de desempenho, as cargas de trabalho são o conjunto de operações e requisições que o sistema deve executar para simular seu uso, e distinguem-se pelo tipo de operação predominante (leituras, escritas, atualizações ou consultas complexas). Para este trabalho foram geradas três cargas com a ferramenta *pgbench* (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2025a), escolhidas por estressarem dimensões distintas do sistema:

read. `SELECT` por chave primária. Estressa CPU e rede, com contenção mínima, já que múltiplos clientes podem ler a mesma página simultaneamente. Serve para isolar o efeito do *pooler* como *proxy* (relevante para H3), sem o ruído da contenção de escrita.

write. `UPDATE` simples sobre uma tabela contenciosa. Estressa a contenção de escrita: *locks* de linha, de transação e o *Write-Ahead Log*. É a carga *central* do estudo, por

ser onde a contenção (e, portanto, o benefício do *pooler*) se manifesta de forma mais expressiva.

mixed. Carga do tipo TPC-B (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2025a), mistura realista de leituras e escritas. Representa um perfil OLTP típico, intermediário entre os extremos.

Cada ensaio compreendeu uma fase de aquecimento de *cache*, cujos dados foram descartados, seguida de uma fase de medição de 60 segundos, durante a qual o *collector* coletou amostras do estado do servidor em paralelo.

4.4 Métricas e instrumentação

Medir um sistema sob carga exige decidir o que observar e de onde observar. Métricas coletadas do lado do cliente descrevem o que a aplicação experimenta (quantas transações por segundo completa e quanto tempo cada uma leva), mas não revelam o que se passa dentro do servidor; métricas coletadas do lado do servidor expõem em que o banco gasta o tempo, mas não dizem, sozinhas, se o serviço percebido pelo cliente é bom. A coleta deste trabalho combinou os dois lados: métricas tradicionais *client-side* e instrumentação interna *server-side*, esta última o diferencial metodológico do estudo. As subseções a seguir detalham cada frente.

4.4.1 Métricas *client-side*

As métricas *client-side* são as reportadas pelo próprio gerador de carga, o *pgbench*, e refletem o desempenho tal como visto pela aplicação. Duas delas sustentam a análise:

TPS (*transactions per second*). Vazão reportada pelo *pgbench*. Métrica primária de desempenho: quanto maior, melhor.

Latência (média e máximo por janela). Tempo por transação, em milissegundos, agregado pelo *pgbench* em janelas de 1 segundo (média, mínimo e máximo). O *log* agregado não fornece percentis verdadeiros (p99); o máximo por janela é, portanto, a medida de cauda disponível, e sua explosão indica saturação.

4.4.2 Métricas *server-side*

As métricas *server-side* descrevem o estado interno do servidor sob carga, como o número de processos ativos, os eventos em que eles esperam e os *locks* que mantêm. O PostgreSQL expõe esse estado em visões de sistema consultáveis por SQL, e o *collector* desenvolvido para este trabalho coletou amostras delas periodicamente, ao longo de cada ensaio, registrando tanto os contadores cumulativos quanto os instantâneos:

`pg_stat_activity`. Distribuição de *wait events* (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2025c; YEN, 2026), a base para identificar *no que* o banco gastou o tempo, com destaque para `LWLock:LockManager` (Seção 2.3).

`pg_locks`. Contagem absoluta de *locks* adquiridos por tipo e modo, confirmando quantitativamente a contenção observada nos *wait events*.

`pg_stat_statements` (v1.12) e `pg_stat_io`. Estatísticas de *parse*/*plan* e de I/O por tipo de *backend*, recursos enriquecidos no PostgreSQL 18.

Console administrativo do *pooler*. `SHOW POOLS` / `SHOW STATS` (PgBouncer e PgCat), permitindo distinguir "cliente esperando na fila do *pooler*" de "cliente esperando *lock* no PostgreSQL".

A combinação cliente + servidor é o que viabiliza a *atribuição causal*: não basta observar que o TPS caiu; é preciso mostrar que ele caiu *porque* a contenção interna cresceu, e que o *pooler* a reduz. O casamento das duas frentes de amostragem no tempo pode ser visto na Figura 3. Cada ensaio inicia com 30 segundos de aquecimento, cujos dados são descartados, seguidos de 60 segundos de medição; durante a medição, o *pgbench* (no cliente) e o *collector* (no servidor) coletam amostras em paralelo, à razão de uma por segundo, o que permite casar as métricas *client-side* e *server-side* instante a instante.

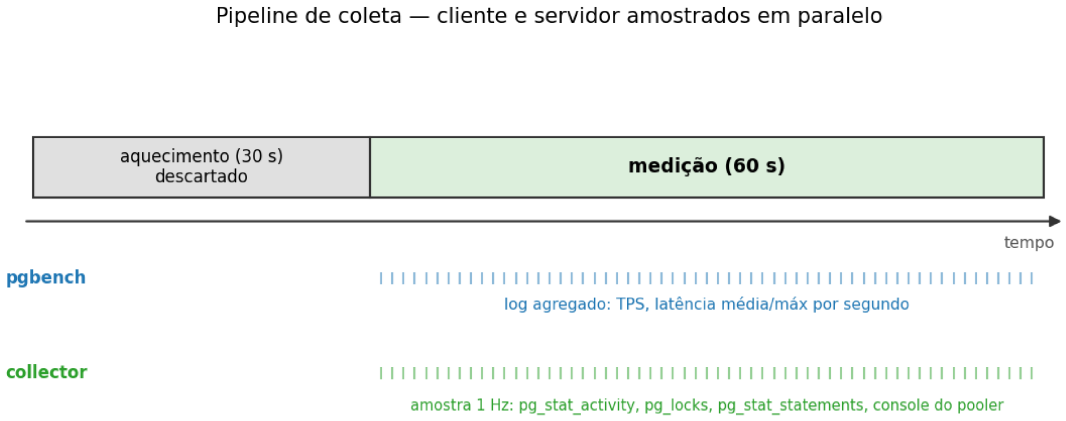

Figura 3 – Linha do tempo de um ensaio: aquecimento e medição.

Fonte: Elaborado pelo autor.

4.5 Classificação de resultados e tratamento de colapsos

Em um experimento de estresse, nem todo ensaio termina com o sistema respondendo normalmente. Denomina-se *colapso* a situação em que o caminho de conexão deixa de produzir vazão mensurável porque algum componente atingiu seu limite interno. Isso

ocorre quando a fila do *pooler* transborda ou quando o servidor esgota `max_connections` e passa a recusar novas conexões. Um aspecto metodológico importante deste trabalho é tratar o colapso como *resultado válido*, e não como falha de medição, pois é o próprio efeito que as hipóteses querem localizar. Para isso, cada ensaio foi classificado em um de três estados:

`done`. O ensaio completou e mediu TPS normalmente.

`saturated`. O caminho **colapsou sob a carga**: a fila do *pooler* ou o servidor estouraram seus limites internos. Este é o "lado de cima" do *break-even*, capturado como dado. O colapso é justamente o fenômeno que a hipótese deseja medir, e não foi descartado.

`expected_failure`. Combinação arquiteturalmente impossível (*session* com quantidade de clientes > tamanho do *pool*), não executada.

Dos 1.332 ensaios viáveis, 1.081 (81,2 %) completaram como `done` e 251 (18,8 %) saturaram; estes últimos, todos em cargas *write*/*mixed* de alta concorrência, justamente onde a teoria prevê o colapso. A distinção entre saturação (resultado válido) e falha real de instrumentação foi feita por assinaturas no *stderr* do *pgbench* (mensagens específicas de cada *pooler* indicando fila esgotada) combinadas com a verificação de que a coleta *server-side* ocorreu normalmente, isto é, o banco permaneceu vivo respondendo enquanto o *pgbench* ficou preso na fila saturada.[1]

4.6 Reprodutibilidade

Reprodutibilidade é a capacidade de um terceiro repetir um experimento, com os mesmos métodos e materiais, e chegar aos mesmos resultados; em estudos experimentais de desempenho, ela é o que separa uma medição verificável de um relato pontual. Neste trabalho, a reprodutibilidade foi tratada como requisito de projeto, e não como preocupação posterior: cada camada do experimento é versionada e automatizada, de modo que um terceiro possa recriar o ambiente, reexecutar a bateria e reobter os resultados sem depender de conhecimento tácito do autor. Quatro mecanismos sustentam essa garantia:

- a infraestrutura de nuvem descrita na Seção 4.1 (os três *hosts* da Tabela 2) é definida em Terraform versionado (*Infrastructure as Code*) e recriável com um único comando, o que elimina a montagem manual das máquinas e a mantém idêntica entre execuções;

[1] Durante a execução foi identificado e corrigido um artefato de instrumentação em que, em concorrências intermediárias da carga *write* com PgBouncer, a fase de aquecimento era encerrada por tempo limite do orquestrador antes de o *pooler* emitir a assinatura de saturação, mascarando o colapso como falha. A correção fez a fase de aquecimento tolerar o tempo limite e prosseguir para a medição, onde a saturação se manifesta corretamente. Os 72 ensaios afetados foram reprocessados, convertendo-se em `saturated` (71) ou `done` (1), sem nenhuma regressão.

- o orquestrador, o *collector* e as configurações dos *poolers* são versionados em Python e em arquivos de configuração, de forma que a matriz de ensaios e os parâmetros de cada *setup* ficam explícitos e auditáveis;
- um *manifest* CSV idempotente registra o estado de cada ensaio (`done`, `saturated` ou pendente), o que permite interromper e retomar baterias longas sem reexecutar o que já foi medido;
- os *notebooks* que produzem as tabelas e figuras deste texto são versionados, mas os dados brutos coletados (cerca de 4 GB) não, por serem integralmente regeneráveis pela reexecução da bateria.

Na prática, reproduzir o estudo resume-se a aplicar o Terraform, apontar o orquestrador para a predefinição *expandida* e aguardar o processamento; o passo a passo, o endereço do repositório público e o custo estimado (cerca de US$ 21 por bateria completa) estão detalhados no Apêndice B.

5 RESULTADOS

Este capítulo apresenta os resultados da bateria, organizados em torno das hipóteses. Todos os valores numéricos foram extraídos diretamente dos 1.332 ensaios viáveis. Cada número reportado é a **média das três repetições** de cada combinação, no modo de *query* `simple`; as únicas exceções são as Seções 5.5 e 5.7, que comparam os modos `simple` e `prepared` e identificam cada um explicitamente. A dispersão entre repetições é pequena (desvio-padrão típico abaixo de 2 % da média) e está representada como banda sombreada nas figuras. A Seção 5.1 caracteriza o *baseline*; as Seções 5.2 a 5.7 tratam das hipóteses H1 a H5; a Seção 5.8 reporta um resultado adicional sobre o tamanho do *pool*.

5.1 Caracterização do *baseline* (conexão direta)

Em experimentos de desempenho, um *baseline* é a configuração de referência utilizada como ponto de comparação para avaliar o impacto de uma nova técnica, ferramenta ou arquitetura. Em bancos de dados, a conexão direta é normalmente adotada nesse papel, com a aplicação estabelecendo conexões diretamente com o servidor, sem componentes intermediários como *pooler* ou *proxy*, e os resultados dessa configuração servem de referência para comparar métricas como vazão e tempo de resposta. Aqui, o *baseline* é o *setup* `none` (conexão direta, sem *pooler*), que estabelece o pano de fundo contra o qual o ganho do *pooler* é medido.

Na carga *write*, seu TPS degrada *monotonicamente* com a concorrência, como se vê na Figura 4: 643 TPS em 1 cliente, 633 em 30, 449 em 100 e 213 em 300 clientes, uma queda de 67 %. Acima de 415 clientes (o `max_connections` configurado), o *baseline* **satura completamente** e deixa de produzir vazão mensurável.

A leitura (*read*) é mais resiliente, porém *não imune*. Parte de um pico de ≈16.560 TPS em 30 clientes e cai para ≈11.456 em 300 (queda de 31 %, contra 67 % na escrita) e, como a escrita, satura ao ultrapassar `max_connections`. Ou seja, a degradação por concorrência atinge ambas as cargas; o que muda é a *inclinação* da curva, mais suave na leitura.

Essa degradação não decorre de falta de recursos brutos da máquina, mas da contenção interna que cresce com o número de *backends*, como a Seção 5.2.1 demonstra.

5.2 H1: *break-even* e a divergência por carga

A hipótese H1 prevê que o *break-even* c^* existe e aparece mais cedo nas cargas intensivas em escrita do que nas de leitura, porque a contenção por bloqueios emerge primeiro quando

há escrita concorrente. Esta seção testa a previsão na carga em que ela é mais visível, a *write*; a Seção 5.2.3 estende o resultado às demais cargas.

Os TPS medidos na carga *write* com *pool* = 2, para o *baseline* e para os dois *poolers* auto-hospedados em modo *transaction*, estão reunidos na Tabela 4 e, em forma gráfica, na Figura 4. Na figura, a conexão direta (cinza) parte de um pico e despenca conforme a concorrência cresce, enquanto os *poolers* partem de um patamar inferior, mas sustentam vazão constante; o cruzamento das curvas, em torno de 30 clientes, é o *break-even point* c^*, e a banda sombreada corresponde a ±1 desvio-padrão das três repetições. A leitura conjunta confirma H1 e revela o formato da transição, resumido nos quatro regimes a seguir.

Tabela 4 – TPS médio na carga *write*, *pool* = 2: o *break-even* entre conexão direta e *pooler*.

Clientes	`none`	`pgbouncer-tx`	`pgcat-tx`	**Vencedor**
1	643	530	527	`none` (pedágio do *pooler*)
30	633	634	632	empate (*break-even*)
50	566	639	645	*pooler*
100	449	635	626	*pooler*
200	297	626	626	*pooler*
300	213	633	624	*pooler* (≈ 3×)
500	*sat.*	632	636	só o *pooler* sobrevive
1.000	*sat.*	629	618	só o *pooler* sobrevive

Média de 3 repetições; desvio-padrão típico <15 TPS (ver banda na Figura 4). "*sat.*" indica saturação total do *baseline* (quantidade de clientes > `max_connections` = 415).

Fonte: Elaborado pelo autor.

1. **Em 1 cliente, o *baseline* vence** (643 vs. 530 TPS). Sem concorrência não há contenção, e o *pooler* é apenas um passo adicional no caminho, o "pedágio" de ≈18 %. Este é o regime em que o *pooler piora* o desempenho, cumprindo o objetivo de documentar tal caso.

2. **Em ≈30 clientes ocorre o empate**, o *break-even* c^*. A vazão com *pooler* iguala a da conexão direta (633 vs. 634 TPS).

3. **De 100 a 300 clientes, a transição é abrupta.** O *baseline* despenca (de 633 para 213 TPS), enquanto os *poolers* permanecem em ≈630 TPS. Em 300 clientes, o *pooler* entrega quase *três vezes* a vazão da conexão direta.

4. **Acima de 500 clientes, a vantagem torna-se qualitativa.** O *baseline* não roda; o *pooler* mantém o mesmo patamar.

A divergência por carga prevista em H1 confirma-se na *inclinação*. A escrita cruza c^* cedo, como visto na Figura 4 (queda de 67 % até 300 clientes), enquanto a leitura, com

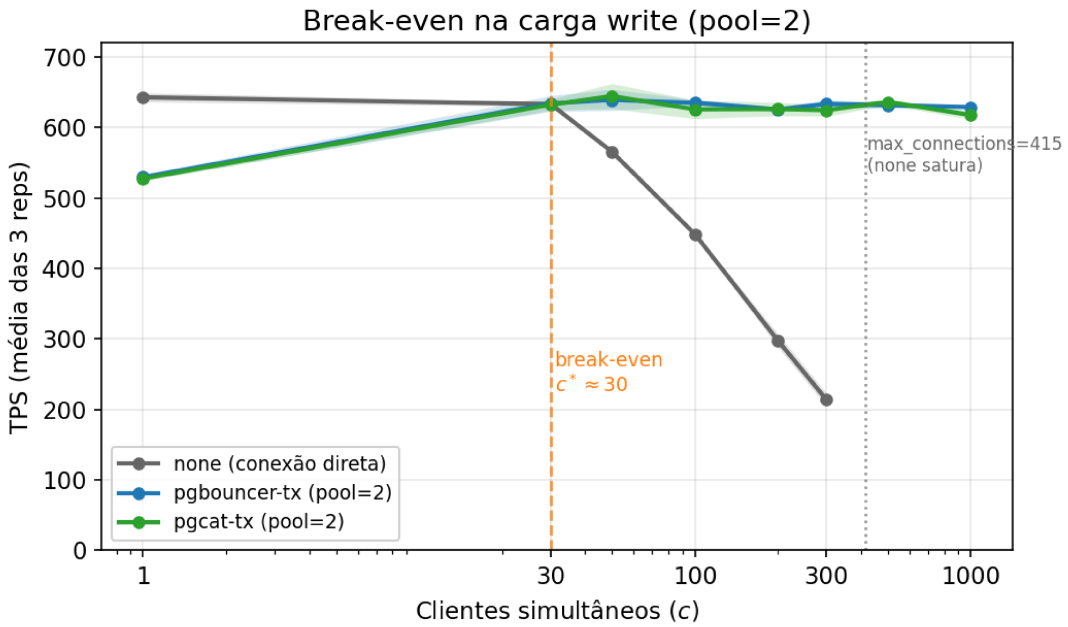

Figura 4 – *Break-even* na carga *write* (*pool* = 2): conexão direta vs. *poolers* em modo *transaction*.

Fonte: Elaborado pelo autor.

contenção mais branda, cruza tarde (queda de 31 % na mesma faixa, caracterizada na Seção 5.1).

5.2.1 O mecanismo causal do *break-even*

A contribuição central deste trabalho é *explicar* a Tabela 4, e não apenas exibi-la. A fração do tempo ativo que o servidor gastou no *wait event* `LWLock:LockManager` (a contenção interna de coordenação) na carga *write* está quantificada na Tabela 5 e na Figura 5. Na tabela, o desvio-padrão elevado do `none` reflete a natureza amostral da medida (instantâneos de `pg_stat_activity`): a fração instantânea oscila, mas a separação frente ao *pooler* é de uma ordem de grandeza em todos os instantes.

Tabela 5 – Fração do tempo ativo em `LWLock:LockManager` (carga *write*; *poolers* com *pool* = 8; média ± desvio de 3 repetições).

Clientes	`none`	`pgbouncer-tx`	`pgcat-tx`
30	1,5 % ± 0.5	0,3 %	0,3 %
100	8,0 % ± 4.0	0,6 %	0,2 %
300	14,9 % ± 5.4	0,3 %	0,1 %

Fonte: Elaborado pelo autor.

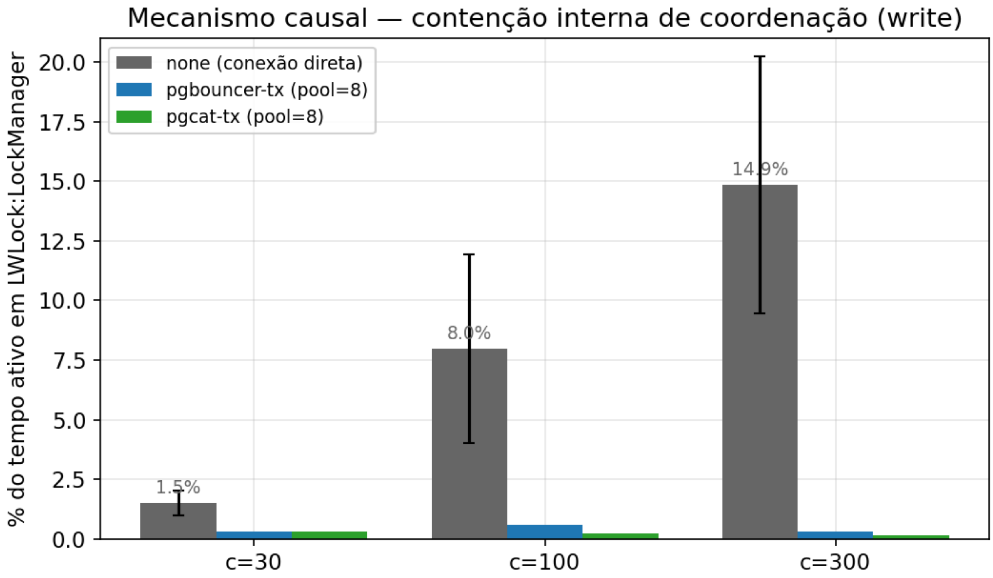

Figura 5 – Fração do tempo ativo em `LWLock:LockManager` na carga *write*. Barras de erro: ±1 desvio-padrão.

Fonte: Elaborado pelo autor.

Concretamente, com conexão direta e 300 clientes, o PostgreSQL gasta cerca de 15 % do seu tempo ativo *apenas coordenando quem pode adquirir qual lock*, trabalho puramente burocrático, sem produzir transações. Com *pooler*, esse custo cai para ≈0,3 %, uma redução de aproximadamente 98 %. O *pooler* não acelera a execução de cada transação; ele *elimina a coordenação supérflua* ao reduzir o número de *backends* físicos de centenas para poucas unidades.

A confirmação quantitativa vem da contagem de *locks* simultaneamente mantidos (Tabela 6): para a mesma carga de trabalho (os mesmos `UPDATE`s), o *baseline* mantém, a cada instante, cerca de *seiscentos locks* de relação, enquanto o *pooler* mantém cerca de *dezessete*, uma redução de 97 %, proporcional à redução no número de *backends*. A coluna `rds-proxy` antecipa o resultado de H4 (Seção 5.6): o *pooling* gerenciado *não* reduz essa contagem.

Tabela 6 – *Locks* mantidos simultaneamente, em média por amostra (carga *write*, 300 clientes; *poolers* com *pool* = 8).

Tipo de *lock*	`none`	`pgbouncer`	`pgcat`	`rds-proxy`	**Red. *pooler***
`relation / RowExclusiveLock`	589	17	17	599	−97%
`transactionid / ExclusiveLock`	294	8	8	299	−97%
`transactionid / ShareLock`	286	7	7	290	−98%
`virtualxid / ExclusiveLock`	295	8	8	300	−97%

A coluna `rds-proxy` acompanha o `none`: o *pooler* gerenciado não reduz a contenção interna.

Fonte: Elaborado pelo autor.

A cadeia causal fica completa, e cada elo dela foi medido neste capítulo:

1. mais *backends* físicos geram mais entradas na tabela de *locks* (com 300 clientes em conexão direta, o servidor mantém 589 *locks* de relação por amostra, contra 17 sob *pooler*, Tabela 6);
2. mais entradas exigem mais tempo de coordenação em `LWLock:LockManager` (a fração do tempo ativo sobe de 1,5 % com 30 clientes para 14,9 % com 300, Tabela 5);
3. mais tempo coordenando significa menos tempo executando transações (nesse regime, cerca de 15 % do tempo ativo é consumido sem produzir transação alguma);
4. com menos tempo útil, o TPS despenca (de 633 para 213, queda de 67 %, Tabela 4).

O *pooler* rompe a cadeia no primeiro elo, e o efeito se propaga pelos demais. Ao limitar os *backends* físicos a poucas unidades, as entradas na tabela de *locks* caem 97 %, o tempo de coordenação recua para 0,3 % e a vazão se mantém em ≈630 TPS até 1.000 clientes.

5.2.2 Onde a espera acontece: fila do *pooler* vs. *lock* no banco

Sob concorrência alta, nem toda requisição encontra o caminho livre. Quando há mais clientes do que capacidade de atendimento, alguém espera, e essa espera tem custo direto para a aplicação, pois se acumula na latência percebida pelo usuário final. O ponto sutil é que a espera pode acontecer em lugares diferentes do caminho, com consequências distintas para o servidor. Um diferencial da instrumentação *server-side* adotada é poder distinguir *dois tipos de espera* que, do ponto de vista do cliente, são indistinguíveis: aguardar uma conexão livre na fila do *pooler* ou aguardar um *lock* dentro do PostgreSQL. Para o `pgbouncer-tx` na carga *write* com *pool* = 2, o cruzamento entre o número de conexões ativas *no banco* (`sv_active`) e o número de clientes *enfileirados no pooler* (`cl_waiting`) é apresentado na Figura 6, na qual a faixa verde marca as conexões ativas no banco, a linha azul a fila do *pooler* e o eixo direito o tempo máximo de espera (`maxwait`).

As duas séries contam histórias opostas. O número de *backends* no banco permanece travado em **2**, enquanto a fila no *pooler* cresce linearmente com a carga (28 clientes esperando em $c = 30$; 298 em $c = 300$; 998 em $c = 1.000$), com `maxwait` subindo de 44 para ≈575 ms. A leitura conjunta com a Tabela 5 fecha o argumento: no `none`, a espera está *dentro do banco* (`LWLock:LockManager` em 15 %); no *pooler*, ela **migra** para a fila FIFO do *pooler*, deixando o banco descontendido (`LWLock` em 0,3 %). A espera não desaparece, apenas é *realocada* de uma estrutura interna sob contenção para uma fila ordenada externa, e é justamente essa realocação que preserva a vazão.

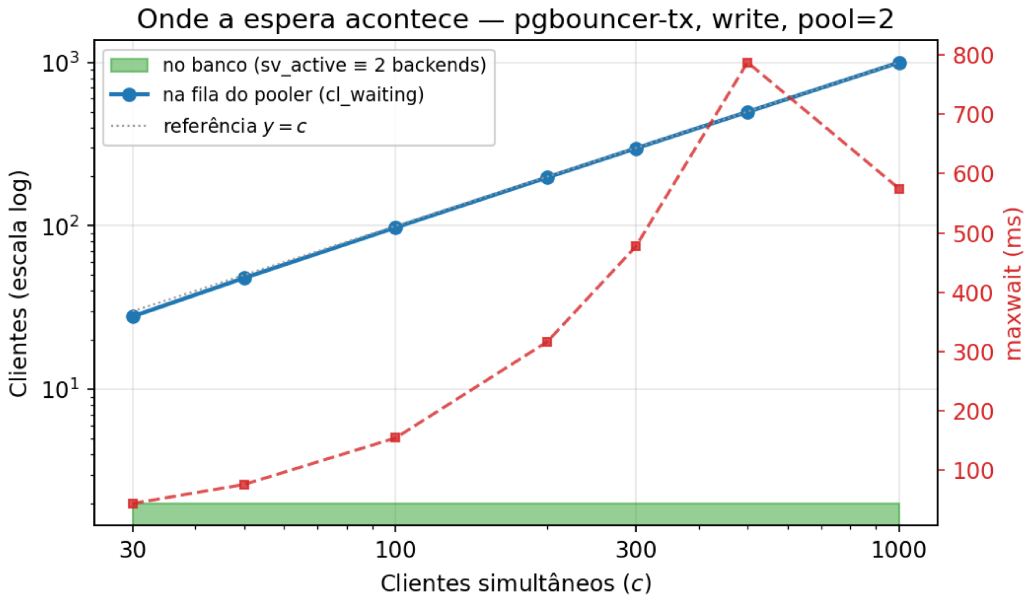

Figura 6 – Localização da espera (`pgbouncer-tx`, carga *write*, *pool* = 2).

Fonte: Elaborado pelo autor.

5.2.3 A divergência completa: c^* por carga

Até aqui, a análise fixou a carga *write*. Para verificar se o padrão se repete nos demais perfis, os valores de c^* medidos em cada uma das três cargas estão reunidos na Tabela 7. O padrão é claro. c^* é governado pelo *perfil de carga*, não pelo *pooler*. PgBouncer e PgCat exibem o mesmo ponto de virada.

Tabela 7 – c^* por carga (modo `simple`); PgBouncer e PgCat coincidem.

Carga	c^* **(clientes)**	**Natureza da vantagem**
write	≈ 30	vazão (o *baseline* colapsa cedo)
mixed	≈ 100	vazão (colapso intermediário)
read	≈ 415	sobrevivência (o *baseline* satura)

Fonte: Elaborado pelo autor.

Na carga *mixed* (TPC-B), intermediária, o *baseline* degrada de 638 TPS (30 clientes) para 464 (300) e satura acima de 415, enquanto o *pooler* sustenta $\approx$590 TPS; o c^* fica em $\approx$100 clientes, entre os extremos.

A carga *read* é o caso mais sutil. Com *pool* = 2, o *pooler* *não* supera a conexão direta em vazão (limitado a $\approx$10.600 TPS, contra $\approx$16.500 do *baseline*), pois duas conexões de servidor estrangulam a leitura paralela; com *pool* = 8, ele *iguala* o *baseline* ($\approx$16.700). Em ambos os casos, o *baseline* satura acima de 415 clientes e some, enquanto o *pooler* sobrevive. O c^* da leitura, portanto, não é um cruzamento de vazão, mas o ponto de

saturação. A vantagem do *pooler* em leitura é de *sobrevivência*, não de desempenho bruto.

A instrumentação *server-side* explica *por que a leitura é mais resiliente*. Na carga *read* com 300 clientes, o *baseline* passa 85,8 % do tempo ativo em `CPU:CPU` (trabalho útil) e 14,2 % em `Client:ClientRead`, *sem* contenção interna mensurável: nem `LWLock:LockManager`, nem `ProcArrayLock`. A leitura é *CPU-bound*, e não limitada por coordenação; por isso degrada suavemente (perde vazão por disputa de CPU) em vez de colapsar como a escrita (presa em *locks*). A `ProcArrayLock`, apontada pela literatura (FREUND, 2020b,a) como possível gargalo de leitura em concorrência muito alta, *não* se materializou na escala deste experimento (*scale* 10): o mecanismo teórico existe, mas não foi exercido neste experimento.

5.3 Latência: o pedágio na média, o controle na cauda

A latência é uma das principais métricas utilizadas na avaliação de sistemas de banco de dados, pois mede o tempo que cada transação leva para ser concluída e, portanto, o atraso que o usuário final percebe. Duas visões se complementam na análise: a média descreve o comportamento típico, e a cauda (os valores máximos) captura os piores casos, os que mais degradam a experiência de uso.

A vazão conta apenas metade da história do *trade-off*; a outra metade é a latência. Na Figura 7, pode-se ver, na carga *write* com *pool* = 2 e em escala logarítmica, a latência *média* e o *máximo por janela de 1 segundo* (uma medida de cauda) para o *baseline* e o *pooler*; a banda sombreada separa a média do máximo de cada *setup*.

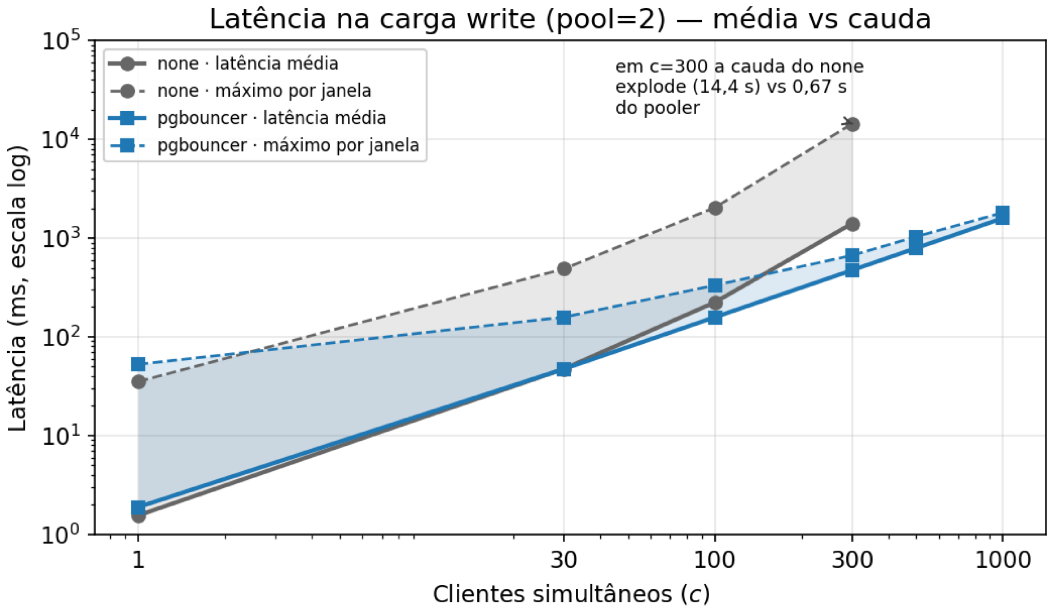

Figura 7 – Latência média e máxima na carga *write* (*pool* = 2), escala logarítmica.

Fonte: Elaborado pelo autor.

O padrão confirma o *trade-off* também no eixo da latência:

- **Em baixa concorrência, o *pooler* cobra um pedágio.** Em 1 cliente, a latência média sobe de 1,6 para 1,9 ms, a mesma sobrecarga de $\approx 18\,\%$ vista na vazão.
- **Em alta concorrência, o *pooler* controla a cauda.** Em 300 clientes, o máximo do *baseline* explode para ≈ 14.400 ms (14 segundos, transações presas na contenção), enquanto o do *pooler* fica em ≈ 668 ms, uma cauda *vinte vezes* menor. A média acompanha: 1.407 ms (direto) contra 474 ms (*pooler*).

Acima de 300 clientes o *baseline* satura e deixa de reportar latência; o *pooler* segue, com a média subindo de forma controlada (reflexo do tempo de espera na fila, Seção 5.2.2) e a cauda contida. A latência, portanto, reforça a tese central. O *pooler* troca um pedágio pequeno e previsível em baixa carga pelo controle da cauda em alta carga.

5.4 H2: *transaction* vs. *session*

A hipótese H2 prevê que o modo *transaction* alcança o *break-even* antes do modo *session*, por multiplexar conexões no limite de cada transação e, com isso, atender concorrências muito superiores ao tamanho do *pool*. Os dados confirmam a previsão de forma ainda mais forte do que o enunciado sugere, pois a diferença se revela estrutural, e não apenas quantitativa. Os setups *session* (`pgbouncer-session`, `pgcat-session`) só produzem ensaios `done` enquanto *quantidade de clientes* $\leq$ *tamanho do pool*: com *pool* $= 100$, eles aparecem até 100 clientes e desaparecem (`expected_failure`) acima disso. Os setups *transaction* sustentam até 1.000 clientes sobre *pools* de apenas 2 ou 8 conexões. Onde ambos coexistem (baixa concorrência), o desempenho é equivalente, pois nesse regime a multiplexação ainda não foi exercida. A consequência prática é que c^* do modo *session* é, em rigor, *inalcançável* para concorrências altas (o modo simplesmente não opera nesse regime), confirmando H2 na direção esperada: o modo *transaction* é o que viabiliza o *pooling* no regime onde ele importa.

5.5 H3: multi-thread (PgCat) vs. single-thread (PgBouncer)

A hipótese H3, central neste trabalho, prevê que um *pooler* de múltiplas *threads*, como o PgCat, sustenta vazão maior que um de *thread* única, como o PgBouncer, quando o *host* do *pooler* dispõe de vários núcleos, pois o processo de *thread* única satura um núcleo e passa a ser ele próprio o gargalo. O teste usa a carga *read* (*CPU-bound*, sem contenção de escrita a mascarar o efeito do *pooler*), com *pool* $= 8$. O dado, porém, conta algo mais sutil do que "*multi-thread* é mais rápido": a vantagem do PgCat *depende do regime de query*. Os dados apresentados na Figura 8 e na Tabela 8 permitem comparar os dois *poolers* nos

modos `simple` e `prepared`. Na figura, as curvas tracejadas correspondem ao modo `simple` e as contínuas ao modo `prepared`; o verde identifica o PgCat e o azul, o PgBouncer. A separação vertical entre as curvas contínuas, ausente nas tracejadas, é a manifestação gráfica da vantagem condicional.

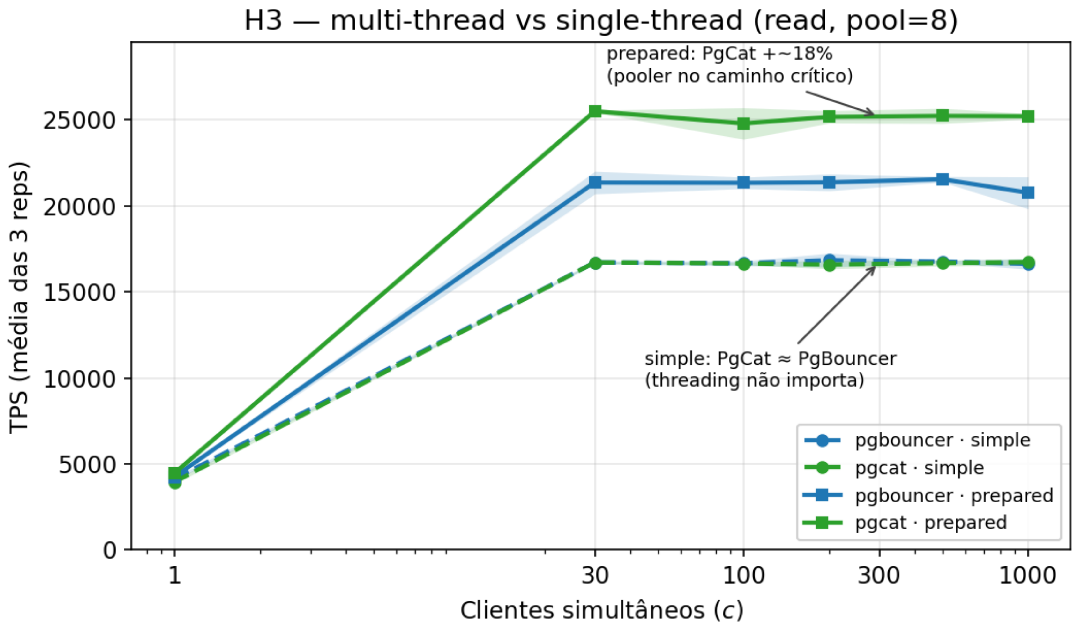

Figura 8 – H3 na carga *read* (*pool* = 8): PgCat vs. PgBouncer nos modos `simple` e `prepared`.

Fonte: Elaborado pelo autor.

Tabela 8 – TPS médio na carga *read*, *pool* = 8: PgCat (*multi-thread*) vs. PgBouncer (*single-thread*), por modo de *query*.

	modo `simple`			**modo `prepared`**		
Cli.	`pgb`	`pgcat`	Δ	`pgb`	`pgcat`	Δ
30	16.708	16.707	−0%	21.354	25.501	+19%
100	16.659	16.651	−0%	21.339	24.788	+16%
200	16.842	16.581	−2%	21.367	25.173	+18%
500	16.758	16.671	−1%	21.547	25.229	+17%
1.000	16.627	16.727	+1%	20.761	25.210	+21%

Fonte: Elaborado pelo autor.

O resultado é, ao mesmo tempo, uma confirmação e uma qualificação de H3:

- **No modo `simple`, não há vantagem *multi-thread*.** PgCat e PgBouncer entregam vazão estatisticamente *idêntica* (≈ 16.700 TPS, Δ entre −2 e +1%, dentro da dispersão). Quando o gargalo não é o *pooler*, o número de *threads* dele é irrelevante.

- **No modo `prepared`, a vantagem emerge e é consistente.** O PgCat sustenta de 16 a 21 % mais que o PgBouncer (≈ 25.200 vs. ≈ 21.300 TPS). A diferença (≈4.000 TPS) é uma ordem de grandeza maior que o desvio-padrão das repetições (≈200 TPS). É um efeito real, não ruído.

A leitura conjunta dos dois modos revela o resultado central: `prepared` remove o custo de *parse/plan* do lado do banco (H5), deslocando o gargalo para o processamento de requisições do próprio *pooler*, e é *aí* que a arquitetura *multi-thread* do PgCat passa a importar. O modo `simple` funciona como *controle*, pois, ao manter o gargalo fora do *pooler*, demonstra que a igualdade observada não é coincidência, mas a ausência da condição que ativa o ganho. H3, portanto, confirma-se **condicionalmente**. A vantagem *multi-thread* é real, mas circunscrita ao regime em que o *pooler* é o recurso crítico. A interpretação arquitetural é desenvolvida no Capítulo 6.

5.6 H4: *pooling* gerenciado (RDS Proxy)

A hipótese H4 prevê que o *pooling* gerenciado troca desempenho por conveniência operacional, pois o RDS Proxy adiciona um salto de rede extra e um modelo de multiplexação mais conservador em relação aos *poolers* auto-hospedados. Os resultados confirmam a previsão em dois aspectos, ambos medidos no modo `simple` para comparação justa:

1. **Teto de vazão inferior e degradante.** Na carga *read*, o RDS Proxy atinge no máximo ≈14.900 TPS (em 100 clientes) e *degrada* com a carga, para ≈11.400 em 1.000 clientes, enquanto os auto-hospedados mantêm ≈16.700 TPS *constantes* na mesma faixa. Na carga *write* a diferença é ainda maior. O RDS Proxy não passa de ≈250 TPS, contra ≈630 dos auto-hospedados. O salto de rede adicional (cliente → *proxy* → RDS) e o modelo de multiplexação cobram esse preço.

2. **Não reduz a contenção interna.** Como mostra a coluna `rds-proxy` da Tabela 6, a contagem de *locks* mantidos pelo RDS Proxy (599 *locks* de relação) é *indistinguível* da do *baseline* `none` (589), ao contrário dos auto-hospedados, que a reduzem para 17. O RDS Proxy multiplexa conexões de cliente, mas mantém muitos *backends* ativos no servidor, trocando a redução de contenção pela conveniência operacional de não exigir um *host* a administrar.

O RDS Proxy, portanto, oferece o benefício de *sobrevivência* a alta concorrência (não satura como o `none`), mas *não* o benefício de desempenho que vem de descontender o banco, justamente porque não reduz o número de *backends*.

5.7 H5: modo `prepared`

A hipótese H5 prevê que o modo `prepared` beneficia todos os *poolers*, por reaproveitar o plano de execução entre chamadas e eliminar o custo de *parse*/*plan* repetido a cada *query* (MULLANE, 2023), com expectativa de ganho relativo maior no PgBouncer, onde *prepared statements* em modo *transaction* só se tornaram possíveis a partir da versão 1.21 (PGBOUNCER AUTHORS, 2023). O ganho na carga *read* (*pool* = 8), onde o efeito é mais visível, está quantificado na Figura 9: as barras correspondem à média sobre as concorrências $c \geq 30$, comparando cada *pooler* consigo mesmo nos dois modos.

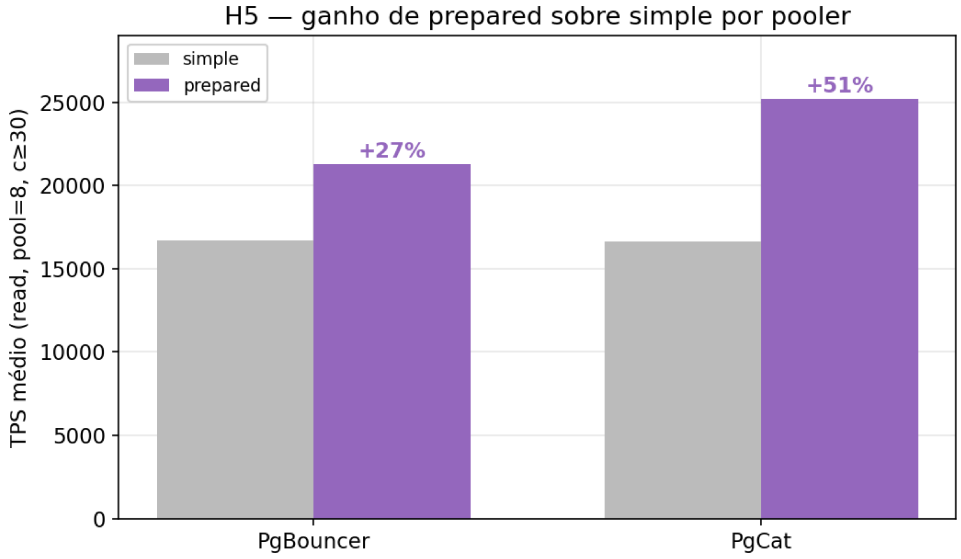

Figura 9 – Ganho do modo `prepared` sobre `simple` (carga *read*, *pool* = 8, média sobre $c \geq 30$).

Fonte: Elaborado pelo autor.

O ganho é expressivo e *assimétrico*. O `prepared` eleva a vazão do PgBouncer em 27 % e a do PgCat em 51 %. A assimetria conecta H5 a H3. O `prepared` diminui o custo de cada requisição, mas só o *pooler multi-thread* (PgCat) consegue *converter* essa redução em vazão adicional, pois não esbarra no limite de um único *thread* de processamento. É a mesma física vista por outro ângulo na Seção 5.5.

Há, porém, uma limitação de instrumentação a registrar. A prova *direta* do mecanismo de H5 (a queda do tempo de *planning* por chamada) não pôde ser extraída. A coluna `total_plan_time` de `pg_stat_statements` registrou valor zero em todos os ensaios, pois o parâmetro `pg_stat_statements.track_planning` (desabilitado por padrão no PostgreSQL) não foi ativado na coleta. A evidência apresentada neste estudo é, portanto, do *efeito* (ganho de vazão), não da medição direta do tempo de *planning*; este último fica como recomendação de instrumentação para trabalhos futuros (Seção 7.1).

5.8 Resultado adicional: o efeito do tamanho do *pool*

Uma constatação não prevista nas hipóteses, mas relevante, emerge ao variar o tamanho do *pool* na carga *write* com `pgbouncer-tx`, como pode ser visto na Figura 10, na qual o *pool* = 2 aparece em verde, o *pool* = 8 em azul e o *pool* = 100 em vermelho:

- *pool* = 2: sustenta ≈630 TPS até 1.000 clientes (curva plana);
- *pool* = 8: levemente inferior, ≈600–626 TPS, também estável;
- *pool* = 100: degrada visivelmente, de ≈574 em 30 clientes para ≈425 a partir de 100, e satura em 1.000 clientes.

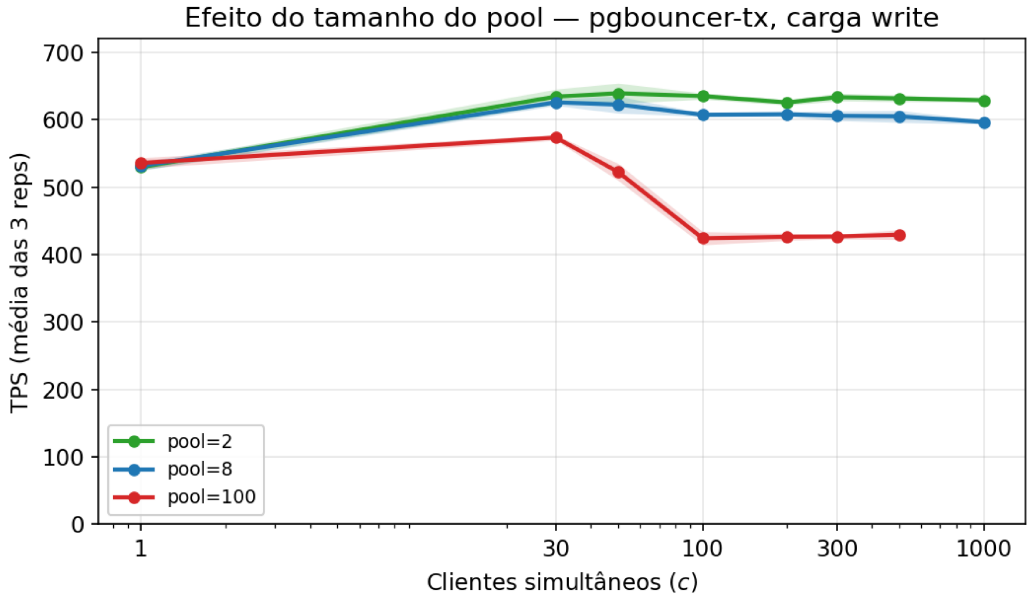

Figura 10 – Efeito do tamanho do *pool* na carga *write* (`pgbouncer-tx`).

Fonte: Elaborado pelo autor.

O efeito é modesto em magnitude (não um colapso, mas uma degradação de ≈33 % do *pool* = 100 frente ao *pool* = 2) e, principalmente, *contraintuitivo*: *menos* conexões resultam em *mais* estabilidade na escrita. A explicação é coerente com o mecanismo de H1. Em escrita, cada conexão adicional é mais um *backend* disputando os mesmos *locks* de linha e o WAL. Um *pool* mínimo (2) serializa a escrita e *protege o banco de si mesmo*; um *pool* grande (100) reintroduz parte da contenção que o *pooler* deveria evitar. A implicação prática (de que não existe um `pool_size` ótimo universal, e que ele depende do perfil de carga) é discutida no Capítulo 6.

6 DISCUSSÃO

Este capítulo interpreta os resultados do Capítulo 5 à luz das hipóteses (Seção 6.1), extrai implicações práticas (Seção 6.2) e, principalmente, expõe as ameaças à validade (Seção 6.3). A credibilidade de uma comparação experimental depende de expor abertamente as decisões que afetam a interpretação.

6.1 Síntese frente às hipóteses

Antes da síntese, convém relembrar o que cada hipótese afirmava. H1 previa um *break-even* dependente do perfil de carga, surgindo mais cedo na escrita; H2, que o modo *transaction* alcança esse ponto antes do modo *session*; H3, a central, que o *pooler multi-thread* supera o *single-thread* quando o *host* tem núcleos disponíveis; H4, que o *pooling* gerenciado paga um preço de desempenho pela conveniência; e H5, que o modo `prepared` beneficia todos os *poolers*. O confronto de cada uma com os resultados segue abaixo.

H1 (*break-even* por carga): confirmada. Existe um regime de baixa concorrência em que o *pooler* piora o desempenho (documentado em 1 cliente: $-18\,\%$) e um *break-even* em ≈ 30 clientes na carga *write*, acima do qual o *pooler* passa de vantajoso a indispensável para evitar o colapso. A divergência por carga prevista confirma-se: c^* aparece cedo na escrita e tardiamente (ou fora da faixa medida) na leitura. O trabalho vai além de confirmar a hipótese e *identifica* o mecanismo: a redução de $98\,\%$ no tempo de `LWLock:LockManager` e de $97\,\%$ na contagem de *locks* mantidos explica *por que* o colapso ocorre e por que o *pooler* o evita.

H2 (*transaction* vs. *session*): confirmada estruturalmente. O modo *transaction* é o único que opera no regime de alta concorrência; o modo *session* é, por construção, limitado a concorrências menores ou iguais ao *pool*. Onde ambos coexistem, o desempenho se equivale.

H3 (*multi-thread*): confirmada condicionalmente. A vantagem do PgCat sobre o PgBouncer *depende do regime de query*: é nula no modo `simple` (vazão idêntica, ≈ 16.700 TPS em leitura de alta concorrência) e consistente no modo `prepared` ($\approx 18\,\%$, com diferença de ≈ 4.000 TPS, uma ordem de grandeza acima da dispersão das repetições). Arquiteturalmente, o `prepared` remove o custo de *parse*/*plan* do lado do banco e desloca o gargalo para o processamento de requisições do *pooler*, e é aí que o *único thread* do PgBouncer passa a limitar, enquanto os múltiplos *threads* do PgCat compensam o custo. O modo `simple` funciona como *controle*: sem o *pooler* no caminho crítico, o número de *threads* é irrelevante. A vantagem *multi-thread*,

portanto, não é universal. É circunscrita ao regime em que o *pooler* é o recurso crítico. Esse resultado dialoga diretamente com o *benchmark* da Tembo (BINIDXABA, 2024), que motivou H3 ao reportar a saturação *single-thread* do PgBouncer: confirma-se o fenômeno, mas qualifica-se seu alcance.

H4 (gerenciado vs. auto-hospedado): confirmada. O RDS Proxy apresenta teto de vazão inferior e, como constatação adicional, *não reduz* a contenção interna como os auto-hospedados. Troca-se desempenho e redução de contenção por conveniência operacional.

H5 (`prepared`): confirmada. O modo `prepared` eleva a vazão de leitura em 27 % no PgBouncer e 51 % no PgCat. A assimetria reforça H3. Só o *pooler multi-thread* converte a redução do custo por requisição em vazão adicional. A prova *direta* do mecanismo (queda do tempo de *planning*) não pôde ser medida, pois `pg_stat_statements.track_planning` não estava ativo na coleta (ver Seção 5.7).

6.2 Implicações práticas

Três recomendações de engenharia decorrem dos resultados:

1. **Adote *pooler* antes do que a intuição sugere.** Em cargas de escrita, o *break-even* aparece já em poucas dezenas de clientes, consistente com o alerta da literatura industrial (CAMARGOS, 2018). A penalidade em baixa concorrência ($\approx 18\,\%$) é pequena frente ao risco de colapso total em alta concorrência.

2. **Não existe `pool_size` ótimo universal.** A constatação da Seção 5.8 (*pools* pequenos serem mais estáveis em escrita) mostra que o dimensionamento deve seguir o perfil de carga: *pools* enxutos para escrita pesada (serializam e protegem o banco), *pools* maiores para leitura (aproveitam o paralelismo). Um valor único, copiado de um *tutorial*, pode introduzir a contenção que o *pooler* deveria evitar.

3. **Avalie o *pooling* gerenciado com olhos abertos.** O RDS Proxy entrega operação simplificada, mas a um custo de teto de vazão e sem o benefício de redução de contenção. Para cargas onde a contenção interna é o gargalo, um *pooler* auto-hospedado bem dimensionado é mensuravelmente superior.

6.3 Ameaças à validade

Ameaças à validade são fatores do desenho experimental que podem enviesar os resultados ou limitar o alcance das conclusões; expô-las abertamente é parte do método, pois permite ao leitor julgar o peso de cada decisão. As decisões a seguir afetam a interpretação dos resultados e são expostas integralmente.

Autenticação MD5 global. O PgCat 1.2 não suporta SCRAM *client-side* (*issue* #255 do repositório `postgresml/pgcat`). Para manter paridade entre os seis setups, configurou-se `password_encryption=md5` em todos. *Mitigação:* (i) o método de autenticação é ortogonal à variável medida (afeta apenas o *handshake*, não a vazão/-latência em regime); (ii) a cifragem do tráfego é garantida por TLS na perna *backend* e pela rede privada da VPC; MD5 enfraquece o *hashing* da senha, não o tráfego em si.

TLS uniforme no *backend*. Como `rds.force_ssl=1` é padrão no RDS para PostgreSQL 15 e posteriores, o que inclui o PG 18 deste estudo (AMAZON WEB SERVICES, 2026d), *todas* as conexões de *backend* (pgbench, PgBouncer, PgCat, RDS Proxy → RDS) usam TLS. O RDS Proxy impõe TLS na perna gerenciada de qualquer forma. Portanto, o TLS uniforme *preserva a paridade* entre setups, em vez de comprometê-la. As pernas *client* → *pooler* usam texto claro em todos os setups, e `verify_server_certificate` está desabilitado nos *poolers* auto-hospedados, mesma postura entre eles. Consequentemente, este trabalho *não* mede o custo do TLS como variável isolada: ele é constante entre os setups.

Zona de disponibilidade única. A bateria roda em uma única AZ. O foco é a sobrecarga de *pooling*, não alta disponibilidade ou *failover*; a AZ única reduz a variabilidade de rede e mantém a comparação limpa. Uma *subnet* privada secundária existe apenas para satisfazer o requisito de duas AZs do *DB Subnet Group* do RDS.

Poolers em Docker (*host network*). Os *poolers* rodam em *containers* Docker com *host network*. A sobrecarga de *container* nessa configuração (sem NAT/*bridge*) é praticamente nula e representa fielmente uma implantação moderna em orquestradores como Kubernetes ou ECS.

***CPU steal* residual.** As instâncias `m5` compartilham *hypervisor* (sem *dedicated tenancy*), de modo que há *CPU steal* residual, muito menor que o de instâncias `t3`, mas não nulo, e fonte de ruído de poucos pontos percentuais. No mesmo sentido, a utilização de CPU *por núcleo* no *host* do *pooler* não foi coletada nesta bateria; assim, o mecanismo proposto para H3 (saturação do núcleo único do PgBouncer sob `prepared`) apoia-se na evidência *indireta* da comparação `simple` vs. `prepared` (Seção 5.5), e não em medição direta de CPU por núcleo (ver Seção 7.1).

`auth_query` desabilitado no PgCat. Por um *bug* do PgCat 1.2 (envia o usuário sem aspas no `auth_query`), optou-se por senha *inline* em vez de *lookup* dinâmico. Isso não afeta as medições. Altera apenas como o *pooler* obtém a credencial.

RDS Proxy `require_tls=false` na perna cliente. O parâmetro foi desabilitado na perna *client* → *proxy* para manter paridade com os *poolers* auto-hospedados (em texto claro no *client-side*). A perna *proxy* → RDS permanece sob TLS obrigatório.

Latências de saturação. Nos ensaios classificados como `saturated`, a latência reportada pelo *pgbench* assume valores muito elevados (da ordem de centenas de milhares de milissegundos). Esses valores *não* são erro de medição: são a latência real de uma transação presa em fila que excedeu o tempo limite interno do *pooler*. Em agregações e gráficos de latência, tais pontos devem ser marcados como saturados para não distorcer médias; este é um cuidado de apresentação a observar na consolidação final.

Generalização. Os resultados referem-se a um PostgreSQL 18 autônomo (sem réplicas), em *hosts* e parâmetros específicos. Cargas, *hardware* e versões distintas deslocarão os valores absolutos de c^*; a contribuição reside nas *tendências* e no mecanismo causal, que se espera generalizáveis, não nos números exatos.

7 CONCLUSÃO

Este trabalho investigou empiricamente o *trade-off* central do *connection pooling* em PostgreSQL: a partir de qual nível de concorrência a sobrecarga de um *pooler* é compensada pela redução de contenção que ele provoca no servidor. Por meio de uma bateria controlada de 1.332 ensaios sobre seis configurações de conexão, em ambiente de nuvem com instrumentação *server-side*, caracterizou-se o *break-even point* c^* e, principalmente, identificou-se o seu mecanismo causal.

Os resultados sustentam que:

- **O *break-even* existe e é precoce na escrita** (H1 confirmada): abaixo de $\approx$ 30 clientes o *pooler* cobra um pedágio de $\approx$18 %; acima, a conexão direta colapsa (de 633 a 213 TPS, e saturação total acima de 415 clientes) enquanto o *pooler* sustenta vazão constante.
- **O mecanismo é a redução de contenção interna**: o tempo gasto em `LWLock:LockManager` cai de 14,9 % para 0,3 %, e a contagem de *locks* mantidos reduz em 97 %. Esta atribuição causal, viabilizada pela instrumentação *server-side*, é a principal contribuição do trabalho frente à literatura existente, que mede majoritariamente do lado do cliente.
- **A arquitetura *multi-thread* ajuda condicionalmente** (H3): o PgCat supera o PgBouncer em $\approx$18 % na leitura *com* ***prepared***, mas os dois empatam no modo `simple`: a vantagem só emerge quando o *pooler* é o recurso crítico.
- **O *pooling* gerenciado tem teto inferior** (H4) e não reduz contenção; **o modo `prepared`** eleva a vazão de leitura em 27 % (PgBouncer) a 51 % (PgCat) (H5); e **não há `pool_size` ótimo universal** (resultado adicional).

Em conjunto, os resultados apontam para uma conclusão simples: o ganho de um *connection pooler* não está em elevar o teto de desempenho, mas em *impedir o colapso* por contenção, uma proteção cujo custo, mensurável e pequeno em baixa concorrência, se paga à medida que a carga cresce.

Este trabalho oferece mais que números: um método reprodutível e uma explicação causal para um problema de engenharia cotidiano, em geral resolvido por intuição. Duas constatações, em particular, contrariaram a expectativa inicial e só emergiram da instrumentação *server-side*: a espera sob alta concorrência não desaparece com o *pooler*, apenas *muda de lugar*, do gerenciador de *locks* do banco para a fila ordenada do *pooler*; e, na escrita, um *pool* mínimo protege o banco melhor que um *pool* generoso. Ao tornar visível

por que o *pooler* funciona, e não apenas *que* ele funciona, espera-se que as decisões de arquitetura de banco de dados possam apoiar-se em evidência, e não em intuição.

7.1 Trabalhos futuros

O recorte experimental adotado, necessário para isolar o efeito de interesse, deixa abertas extensões naturais, capazes de aprofundar o mecanismo identificado ou de testar sua generalização em outros contextos. Cinco direções destacam-se:

- **Medição direta do mecanismo de H3** (a hipótese de que o *pooler multi-thread* supera o *single-thread* quando o *pooler* é o recurso crítico): instrumentar a utilização de CPU *por núcleo* no *host* do *pooler* para confirmar diretamente que o ganho condicional do PgCat decorre da saturação do núcleo único do PgBouncer no modo `prepared`, evidência que neste trabalho é indireta.
- **Instrumentação do tempo de *planning* (H5):** ativar `pg_stat_statements.track_planning` para medir diretamente a queda do custo de *parse*/*plan* sob `prepared`, complementando a evidência de vazão apresentada neste trabalho.
- **Outras alternativas de *pooling*:** Odyssey (YANDEX, 2026), Supavisor e o *pooling* aplicacional (HikariCP, `pgxpool`), deixados fora desta comparação por escopo.
- **Cenários de alta disponibilidade:** réplicas de leitura, *failover* e múltiplas zonas de disponibilidade, intencionalmente fora do escopo deste estudo para isolar a sobrecarga de *pooling*.
- **Versões e *hardware* diversos:** replicar a bateria em outras famílias de instância e versões do PostgreSQL para testar a generalização das tendências.

REFERÊNCIAS

AMAZON WEB SERVICES. **Amazon EC2 instance types**. 2026. Documentação AWS. Disponível em: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html>. Acesso em: 21 jul. 2026. Citado na p. 28.

_____. **Amazon RDS Proxy — User Guide**. 2026. Documentação AWS. Disponível em: <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/rds-proxy.html>. Acesso em: 5 jun. 2026. Citado nas pp. 23, 26.

_____. **Parameter groups for Amazon RDS**. 2026. Documentação AWS. Disponível em: <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithParamGroups.html>. Acesso em: 21 jul. 2026. Citado na p. 29.

_____. **Using SSL with a PostgreSQL DB instance**. 2026. Documentação AWS (parâmetro `rds.force_ssl`). Disponível em: <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/PostgreSQL.Concepts.General.SSL.html>. Acesso em: 5 jun. 2026. Citado na p. 51.

ANDERSON, Kristi. **PostgreSQL Connection Pooling: Part 4 — PgBouncer vs. Pgpool-II**. 2020. ScaleGrid Blog. Disponível em: <https://scalegrid.io/blog/postgresql-connection-pooling-part-4-pgbouncer-vs-pgpool/>. Acesso em: 27 mai. 2026. Citado na p. 26.

AWS RE:POST COMMUNITY. **Using RDS Proxy doubles or triples average response times**. 2021. AWS re:Post — thread da comunidade. Disponível em: <https://repost.aws/questions/QUVquqTlcFS62sOiJfGp4snA/using-rds-proxy-doubles-or-triples-average-response-times>. Acesso em: 2 jun. 2026. Citado nas pp. 17, 26.

BINIDXABA. **Benchmarking PostgreSQL connection poolers: PgBouncer, PgCat and Supavisor**. 2024. Tembo Blog, colaborador da comunidade (versão arquivada no Internet Archive). Disponível em: <https://web.archive.org/web/20240527193244/https://tembo.io/blog/postgres-connection-poolers>. Acesso em: 26 mai. 2026. Citado nas pp. 17, 23, 25, 27, 50.

BUTROVICH, Matthew et al. Tigger: A Database Proxy That Bounces With User-Bypass. **Proceedings of the VLDB Endowment**, v. 16, n. 11, p. 3335–3348, 2023. DOI: 10.14778/3611479.3611530. Disponível em:

<https://www.vldb.org/pvldb/vol16/p3335-butrovich.pdf>. Acesso em: 19 mai. 2026. Citado nas pp. 25, 27.

CAMARA, JP. **PgBouncer is useful, important, and fraught with peril**. 2023. Blog pessoal. Disponível em: <https://jpcamara.com/2023/04/12/pgbouncer-is-useful.html>. Acesso em: 28 mai. 2026. Citado nas pp. 23, 26.

CAMARGOS, Fernando Laudares. **Scaling PostgreSQL with PgBouncer: You May Need a Connection Pooler Sooner Than You Expect**. 2018. Percona Database Performance Blog. Disponível em: <https://www.percona.com/blog/scaling-postgresql-with-pgbouncer-you-may-need-a-connection-pooler-sooner-than-you-expect/>. Acesso em: 26 mai. 2026. Citado nas pp. 15, 18, 25, 27, 32, 50.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Fundamentals of Database Systems**. 7. ed. Hoboken, NJ: Pearson, 2016. Citado na p. 19.

EMMETT, Ben. **Per-backend IO stats in PostgreSQL 18**. 2025. Redgate Blog (*guest post*). Disponível em: <https://www.red-gate.com/blog/per-backend-io-stats-in-postgresql-18>. Acesso em: 10 jun. 2026. Citado na p. 26.

FITTL, Lukas. **GitLab's challenge with Postgres LWLock lock_manager contention**. 2023. pganalyze — 5mins of Postgres. Disponível em: <https://pganalyze.com/blog/5mins-postgres-LWLock-lock-manager-contention>. Acesso em: 30 mai. 2026. Citado nas pp. 16, 21, 26.

FREUND, Andres. **Analyzing the Limits of Connection Scalability in Postgres**. 2020. Citus Data Blog. Disponível em: <https://www.citusdata.com/blog/2020/10/08/analyzing-connection-scalability/>. Acesso em: 21 mai. 2026. Citado nas pp. 22, 43.

_____. **Improving connection scalability: GetSnapshotData()**. 2020. Lista de discussão pgsql-hackers; análise no blog da Citus Data. Disponível em: <https://www.citusdata.com/blog/2020/10/25/improving-postgres-connection-scalability-snapshots/>. Acesso em: 21 mai. 2026. Citado nas pp. 15, 20, 22, 25, 43.

_____. **Measuring the Memory Overhead of a Postgres Connection**. 2020. Blog pessoal (anarazel.de). Disponível em: <https://blog.anarazel.de/2020/10/07/measuring-the-memory-overhead-of-a-postgres-connection/>. Acesso em: 7 jun. 2026. Citado nas pp. 16, 20.

KUMAR, Sumit; SINGH, Devinder; VENKATRAMAN, Ramesh Kumar. **Improve PostgreSQL performance: Diagnose and mitigate lock manager contention**. 2025. AWS Database Blog. Disponível em: <https://aws.amazon.com/blogs/database/improve-postgresql-performance-diagnose-and-mitigate-lock-manager-contention/>. Acesso em: 29 mai. 2026. Citado nas pp. 16, 21, 26.

MANNEM, Sebastiaan. **Pgpool vs PgBouncer**. 2019. EnterpriseDB Blog. Disponível em: <https://www.enterprisedb.com/blog/pgpool-vs-pgbouncer>. Acesso em: 27 mai. 2026. Citado na p. 26.

MARKWORT, Julian. **Comparing Connection Poolers for PostgreSQL**. 2024. PGConf.EU 2024 (palestra). Disponível em: <https://www.postgresql.eu/events/pgconfeu2024/schedule/session/5846-comparing-connection-poolers-for-postgresql/>. Acesso em: 24 mai. 2026. Citado na p. 25.

MOPPEL, Kaarel. **Connection Pooling Intro: PgBouncer and Pgpool-II**. 2016. Cybertec PostgreSQL Blog. Disponível em: <https://www.cybertec-postgresql.com/en/connection-pooling-intro-pgbouncer-and-pgpool-ii/>. Acesso em: 28 mai. 2026. Citado na p. 22.

MULLANE, Greg Sabino. **Prepared Statements in Transaction Mode for PgBouncer**. 2023. Crunchy Data Blog. Disponível em: <https://www.crunchydata.com/blog/prepared-statements-in-transaction-mode-for-pgbouncer>. Acesso em: 4 jun. 2026. Citado nas pp. 22, 26, 47.

OTS, Indrek. **Dealing With LWLock:LockManager Wait Events in Aurora Postgres**. 2024. Blog pessoal. Disponível em: <https://blog.indrek.io/articles/dealing-with-lwlock-lockmanager-wait-events-in-aurora-postgres/>. Acesso em: 29 mai. 2026. Citado nas pp. 16, 21, 26.

PGBOUNCER AUTHORS. **PgBouncer changelog**. 2023. Notas de versão (1.21.0, de 2023, adiciona *prepared statements* em modo *transaction*). Disponível em: <https://www.pgbouncer.org/changelog.html>. Acesso em: 3 jun. 2026. Citado nas pp. 17, 22, 26, 47.

_____. **PgBouncer configuration reference**. 2026. Documentação oficial. Disponível em: <https://www.pgbouncer.org/config.html>. Acesso em: 3 jun. 2026. Citado nas pp. 22, 23, 26.

_____. **PgBouncer FAQ**. 2026. Documentação oficial. Disponível em: <https://www.pgbouncer.org/faq.html>. Acesso em: 3 jun. 2026. Citado nas pp. 22, 26.

POSTGRESML. **PgCat — PostgreSQL pooler with sharding, load balancing and failover**. 2026. Repositório oficial (GitHub). Disponível em: <https://github.com/postgresml/pgcat>. Acesso em: 4 jun. 2026. Citado nas pp. 23, 26.

POSTGRESQL GLOBAL DEVELOPMENT GROUP. **pgbench — run a benchmark test on PostgreSQL**. 2025. Documentação oficial, versão 18. Disponível em: <https://www.postgresql.org/docs/18/pgbench.html>. Acesso em: 10 jun. 2026. Citado nas pp. 32, 33.

_____. **PostgreSQL 18.0 Release Notes**. 2025. Documentação oficial. Item de desempenho de *locking* para consultas que acessam muitas relações (*commit* `c4d5cb71d`). Disponível em: <https://www.postgresql.org/docs/release/18.0/>. Acesso em: 21 jul. 2026. Citado na p. 21.

_____. **PostgreSQL Documentation — The Cumulative Statistics System**. 2025. Documentação oficial, versão 18. Disponível em: <https://www.postgresql.org/docs/18/monitoring-stats.html>. Acesso em: 9 jun. 2026. Citado nas pp. 21, 26, 34.

_____. **The Cumulative Statistics System — visão `pg_stat_io`**. 2025. Documentação oficial, versão 18 (seção 27.2). Disponível em: <https://www.postgresql.org/docs/18/monitoring-stats.html#MONITORING-PG-STAT-IO-VIEW>. Acesso em: 9 jun. 2026. Citado na p. 26.

POSTGRESQL WIKI. **Number Of Database Connections**. 2014. Wiki comunitária do projeto PostgreSQL (última modificação em 2014). Disponível em: <https://wiki.postgresql.org/wiki/Number_Of_Database_Connections>. Acesso em: 6 jun. 2026. Citado nas pp. 15, 20.

RAWAT, Meghna. **RDS Proxy in Production: Real-World Lessons, Limitations, and Why We Use It**. 2025. TO THE NEW Engineering Blog. Disponível em: <https://www.tothenew.com/blog/rds-proxy-in-production-real-world-lessons-limitations-and-why-we-use-it/>. Acesso em: 2 jun. 2026. Citado nas pp. 17, 26.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Database System Concepts**. 7. ed. New York, NY: McGraw-Hill Education, 2020. Citado nas pp. 19, 20.

WOOLDRIDGE, Brett. **HikariCP — About Pool Sizing**. 2021. Wiki do projeto HikariCP (GitHub). Disponível em: <https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing>. Acesso em: 6 jun. 2026. Citado na p. 20.

YANDEX. **Odyssey: Scalable PostgreSQL connection pooler**. 2026. Repositório oficial (GitHub). Disponível em: <https://github.com/yandex/odyssey>. Acesso em: 4 jun. 2026. Citado na p. 54.

YEN, Richard. **Understanding PostgreSQL Wait Events**. 2026. Blog pessoal (richyen.com). Disponível em: <https://richyen.com/postgres/2026/04/13/wait_events.html>. Acesso em: 8 jun. 2026. Citado nas pp. 21, 34.

Apêndices

APÊNDICE A – MATRIZ EXPERIMENTAL COMPLETA

A Tabela 9 consolida a distribuição de estados por setup, somando todas as cargas, modos, concorrências e *pools*.

Tabela 9 – Distribuição de estados por setup (todos os ensaios viáveis).

Setup	`done`	`saturated`	`failed`	**total**
`none`	108	0	0	108
`pgbouncer-tx`	313	119	0	432
`pgbouncer-session`	108	0	0	108
`pgcat-tx`	312	120	0	432
`pgcat-session`	108	0	0	108
`rds-proxy`	132	12	0	144
Total	1.081	251	0	1.332

Não inclui as 684 combinações `expected_failure` (arquiteturalmente inviáveis, não executadas). A ausência de `failed` reflete o reprocessamento descrito na Seção 4.5.

Fonte: Elaborado pelo autor.

APÊNDICE B – REPRODUTIBILIDADE

O código-fonte do experimento está publicado em https://github.com/jean-souto/postgres-pooler-benchmark-tcc, sob licença MIT. O repositório reúne o que é necessário para reproduzir o estudo a partir do zero: a infraestrutura em Terraform (provisionamento de RDS, RDS Proxy e instâncias EC2), o orquestrador, o *collector* e as configurações dos *poolers* em Python, a matriz de experimentos declarada em `experiments.yaml` e os *notebooks* de análise.

Os dados brutos coletados, cerca de 4 GB em arquivos CSV, não integram o repositório, pois são regeneráveis pela própria reexecução da bateria, que os *notebooks* então convertem nas figuras deste texto. Cada ensaio é registrado em um *manifest* CSV idempotente, de modo que a execução pode ser interrompida e retomada sem repetir o trabalho já concluído.

A reexecução se dá por um único ponto de entrada, parametrizado pelo *preset* de ensaios e pelo ambiente de destino:

```
python -m orchestrator --preset <nome> --mode <local|cloud>
```

No modo `local`, a pilha de PostgreSQL e *poolers* auto-hospedados sobe via Docker Compose, adequada à validação e às baterias sem custo; no modo `cloud`, o orquestrador executa contra a infraestrutura provisionada na AWS. O estudo principal corresponde ao *preset* `expandida` em modo `cloud`, o único que contempla o RDS Proxy, enquanto o *preset* `smoke` valida o fluxo de ponta a ponta em poucos minutos.