Anonimização de Dumps SQL com Memória Constante via Arquitetura em Streaming

Carlos Eduardo Pinheiro da Silva, Tarsis Marinho de Souza, Anderson Felinto Barbosa

[1] Núcleo de Desenvolvimento e Inovação Tecnológica (DIT)
Instituto Federal de Alagoas (IFAL) – Arapiraca, AL – Brazil

ceps1@aluno.ifal.edu.br, tarsis.souza@ifal.edu.br,
anderson.barbosa@ifal.edu.br

Abstract. *Data privacy regulations, such as the Brazilian LGPD, mandate the anonymization of sensitive information. However, masking massive database dumps for testing and staging environments often imposes severe memory and processing bottlenecks. This paper presents a database connection-agnostic command-line tool developed in Rust for anonymizing SQL dumps using a streaming architecture. The system employs a finite state machine (FSM) parser to process SQL statements in a single pass, avoiding the need for full in-memory parsing. Masking rules are defined declaratively via configuration files, enabling flexible strategies such as deterministic hashing (HMAC), synthetic data generation, suppression, and local differential privacy. Experimental results demonstrate that the proposed tool maintains a constant memory footprint (approximately 4.7 MB) while processing large datasets, confirming its suitability for integration into CI/CD pipelines without performance degradation.*

Resumo. *Regulamentações de privacidade, como a LGPD, exigem a anonimização de informações sensíveis. Contudo, o mascaramento de dumps massivos de banco de dados para ambientes de teste frequentemente impõe severos gargalos de memória e processamento. Este artigo apresenta uma ferramenta de linha de comando agnóstica a conexões com banco de dados desenvolvida em Rust para anonimização de dumps SQL por meio de uma arquitetura em streaming. O sistema utiliza um parser baseado em Máquina de Estados Finita (FSM) para processar comandos SQL em uma única passagem, evitando a necessidade de carregamento completo em memória. As regras de mascaramento são definidas de forma declarativa via arquivos de configuração, suportando estratégias como hashing determinístico (HMAC), geração de dados sintéticos, supressão e privacidade diferencial local. Resultados experimentais demonstram que a ferramenta desenvolvida mantém consumo de memória constante (aproximadamente 4,7 MB) mesmo com grandes volumes de dados, evidenciando sua viabilidade para integração em pipelines de CI/CD sem degradação de desempenho.*

1. Introdução

A transição de dados de produção para ambientes de desenvolvimento e ecossistemas de análise (*Analytics*/BI) é uma prática vital na engenharia de software; por exemplo,

equipes frequentemente replicam bases reais para testes e validação de funcionalidades. Contudo, esse processo esbarra em severas restrições legais. Com a vigência da Lei Geral de Proteção de Dados Pessoais (LGPD) [Brasil 2018], a cópia direta de Informações Pessoalmente Identificáveis (PII) tornou-se uma vulnerabilidade crítica, potencialmente violando os princípios da finalidade, necessidade e segurança.

Apesar dessa urgência, as abordagens tradicionais de mascaramento apresentam gargalos operacionais crônicos. Processos em lote exigem a restauração do banco em ambientes isolados, consumindo tempo e armazenamento excessivos, o que inviabiliza sua integração em esteiras ágeis de Integração Contínua/Entrega Contínua (CI/CD). Por outro lado, ferramentas que atuam em trânsito frequentemente sofrem com estouro de memória (*Out-Of-Memory*, isto é, quando a aplicação consome toda a memória RAM disponível) ou exigem conexões diretas via SSH e *proxy*. Tais requisitos aumentam a complexidade de rede e auditoria, tornando-se indesejáveis em cenários de terceirização ou sistemas legados.

Para solucionar este gargalo, propõe-se, neste trabalho, uma ferramenta de linha de comando (CLI) projetada para operar estritamente por meio de fluxos de dados (*streaming*). A solução adota um estilo arquitetural orientado a fluxo de dados (*data flow*) para anonimização de *dumps* SQL estáticos, fundamentada em um parser baseado em Máquina de Estados Finita (FSM) [Hopcroft et al. 2001] para processamento incremental de dados.

Ao desacoplar as regras de anonimização e aplicar uma leitura bufferizada, a abordagem permite o processamento e a desidentificação de volumes massivos de dados relacionais com uso mínimo de recursos de hardware, dispensando a necessidade de conexões diretas, como SSH e *proxy*, e reduzindo a complexidade de integração com a infraestrutura do banco de dados. Dessa forma, opera de forma agnóstica à conexão com SGBDs, atuando exclusivamente sobre *dumps* SQL estáticos, o que elimina a necessidade de acesso direto às instâncias de banco de dados.

As principais contribuições deste trabalho incluem a proposta de uma arquitetura em *streaming* para anonimização de *dumps* SQL com consumo de memória constante, o desenvolvimento de um parser baseado em Máquina de Estados Finita (FSM) [Hopcroft et al. 2001] para processamento incremental de dados, a definição de um mecanismo desacoplado de regras de anonimização por meio de configuração declarativa e uma avaliação empírica que demonstra a estabilidade do consumo de memória em diferentes ambientes de execução.

O restante deste trabalho está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica e os trabalhos relacionados. A Seção 3 descreve a arquitetura proposta, detalhando seus componentes e o mecanismo de configuração das regras de anonimização. A Seção 4 apresenta a avaliação experimental, analisando o desempenho e o consumo de recursos em diferentes cenários. Por fim, a Seção 5 discute as considerações finais e direções para trabalhos futuros.

2. Fundamentação Teórica e Trabalhos Relacionados

A relevância do mascaramento de dados em ambientes de engenharia de software é fundamentada pela própria base legal brasileira. A LGPD estabelece o limite jurídico para dados que passam por ofuscação irreversível, definindo em seu Artigo 12:

> "Os dados anonimizados não serão considerados dados pessoais para os fins desta Lei, salvo quando o processo de anonimização ao qual foram submetidos for revertido, utilizando exclusivamente meios próprios, ou quando, com esforços razoáveis, puder ser revertido." [Brasil 2018]

Para atingir o grau de irreversibilidade exigido pela legislação sem destruir o valor analítico e relacional da informação, estudos recentes têm explorado abordagens práticas para anonimização em bases relacionais em conformidade com a LGPD, discutindo desafios técnicos e limitações de ferramentas atuais [Oliveira et al. 2024]. Nesse contexto, diversas técnicas computacionais são empregadas, tais como: a Privacidade Diferencial, que adiciona ruído matemático aos dados numéricos preservando suas propriedades estatísticas; a supressão sumária de dados; a substituição de campos sensíveis por dados fictícios; a atribuição de valores padrão genéricos; e a aplicação de algoritmos de *hash* criptográfico com chaves secretas (HMAC)[Krawczyk et al. 1997] para garantir a consistência referencial (determinismo) em chaves estrangeiras. Em bancos com modelagem relacional, o uso de chaves substitutas (*surrogate keys*), como inteiros auto-incrementais ou UUIDs, é recomendado exatamente porque estas não possuem "significado de negócio", existindo apenas para a estruturação técnica do banco de dados [Codd 1979].

Dentre essas técnicas, a Privacidade Diferencial destaca-se por fornecer garantias matemáticas formais de proteção, limitando o impacto da presença ou ausência de um indivíduo nos dados [Dwork et al. 2006, Mironov et al. 2009]. Na prática, isso é alcançado por meio da injeção de ruído aleatório calibrado de acordo com a sensibilidade da função (Δf).

Formalmente, um algoritmo aleatorizado $\mathcal{M}$ satisfaz a ϵ-Privacidade Diferencial se, para quaisquer conjuntos de dados adjacentes D e D' (que diferem em apenas um registro) e qualquer subconjunto de saídas possíveis S, a seguinte inequação é válida:

$$\Pr[\mathcal{M}(D) \in S] \leq e^{\epsilon} \cdot \Pr[\mathcal{M}(D') \in S] \tag{1}$$

Neste trabalho, adota-se a *Privacidade Diferencial Local* (LDP) [Duchi et al. 2013], na qual a perturbação é aplicada diretamente a cada registro individual antes de qualquer agregação. Tal abordagem é particularmente adequada ao paradigma de processamento em fluxo (*streaming*) adotado, pois elimina a necessidade de acesso global ao conjunto de dados e viabiliza a anonimização incremental com garantias formais de privacidade.

Formalmente, um mecanismo $\mathcal{M}$ satisfaz ϵ-LDP se, para quaisquer valores possíveis x e x' de um mesmo indivíduo e qualquer subconjunto de saídas S, vale:

$$\Pr[\mathcal{M}(x) \in S] \leq e^{\epsilon} \cdot \Pr[\mathcal{M}(x') \in S] \tag{2}$$

Apesar dos claros incentivos legais e teóricos, a adoção prática de técnicas avançadas de desidentificação permanece marginal na indústria. Conforme evidenciado por [Garrido et al. 2022], existe um hiato significativo entre a pesquisa acadêmica em privacidade e sua utilização no ambiente corporativo. Devido à complexidade de

implementação e à escassez de ferramentas de anonimização de fácil integração, muitas organizações acabam por negligenciar a ofuscação estrutural dos dados.

Em vez disso, as empresas recorrem a processos burocráticos de controle de acesso e segurança de perímetro. Contudo, essa abordagem baseada estritamente em permissões frequentemente resulta em políticas de privacidade fragilmente aplicadas, deixando os dados brutos expostos a usos indevidos assim que o acesso ao ambiente é concedido aos analistas ou desenvolvedores.

A tentativa de implementar técnicas de anonimização em arquivos de exportação (*dumps*) esbarra diretamente em limitações estruturais relevantes das ferramentas atuais. Apesar da vasta disponibilidade de bibliotecas para análise sintática (tais como o *sqlparse* em Python ou *node-sql-parser*), a aplicação destas em arquivos volumosos é proibitiva. Tais ferramentas operam construindo uma Árvore de Sintaxe Abstrata (AST) [Grune et al. 2012] do arquivo SQL inteiro, carregando os *tokens* em memória. O *overhead* estrutural de uma AST frequentemente exige múltiplas vezes mais memória RAM do que o tamanho do arquivo original em disco, resultando em falhas por esgotamento de memória (*out-of-memory*).

Alternativas baseadas em *DataFrames* analíticos (como Pandas) também herdam o paradigma *in-memory*, no qual os dados são integralmente carregados em memória antes do processamento. Esse modelo reduz a escalabilidade para grandes volumes de dados e pode resultar em falhas por esgotamento de memória em ambientes com recursos limitados.

Uma exceção notável ao processamento *in-memory* é o *myanon* [Pomès 2026], utilitário de linha de comando que mascara arquivos *mysqldump* via fluxo de dados (*streaming*). Embora elimine o esgotamento de memória e preserve chaves estrangeiras via *hashing* determinístico (HMAC-SHA256), a ferramenta apresenta limitações analíticas e de compatibilidade. O *myanon* restringe-se à pseudonimização básica e substituição de *strings* (via integração com bibliotecas como Faker), não oferecendo mecanismos nativos de Privacidade Diferencial Local (LDP) [Kairouz et al. 2014] para ofuscação estatística de dados numéricos, por exemplo. Ademais, seu forte acoplamento ao dialeto do MySQL e a complexidade de manutenção de sua arquitetura baseada em C dificultam sua adoção em ecossistemas de dados heterogêneos. Assim, o sistema proposto diferencia-se por combinar suporte a múltiplos dialetos SQL, integração com Privacidade Diferencial Local e uma arquitetura modular baseada em configuração declarativa.

3. Arquitetura Proposta

O sistema foi projetado seguindo o estilo arquitetural *pipes and filters*, adotando o modelo de *pipes* do ecossistema Unix [Ritchie 1984], com leitura da entrada padrão (*stdin*) e escrita na saída padrão (*stdout*).

Portanto, a arquitetura segue um modelo de processamento em fluxo (*streaming*), no qual os dados são transformados incrementalmente ao longo de um pipeline linear. Nesse modelo, cada componente do sistema é responsável por uma etapa específica da análise e anonimização do *dump* SQL, permitindo o processamento em única passagem e com consumo constante de memória. Essa abordagem favorece a escalabilidade do sistema e reduz a necessidade de alocação de memória proporcional ao volume de dados processado.

De forma geral, o fluxo ilustrado na Figura 1 inicia-se com a leitura incremental do *dump* SQL por meio do componente *Stream Reader*, que alimenta o parser baseado em FSM responsável por identificar estruturas sintáticas relevantes. Em seguida, metadados como nomes de tabelas e colunas são extraídos para a construção de um mapeamento de estratégias de anonimização a partir do arquivo de configuração. Na etapa seguinte, os dados são tokenizados em nível de linha e coluna, permitindo a aplicação das regras definidas pelo *Masking Engine*. Por fim, os dados transformados são escritos de forma contínua na saída, preservando a estrutura original do *dump*.

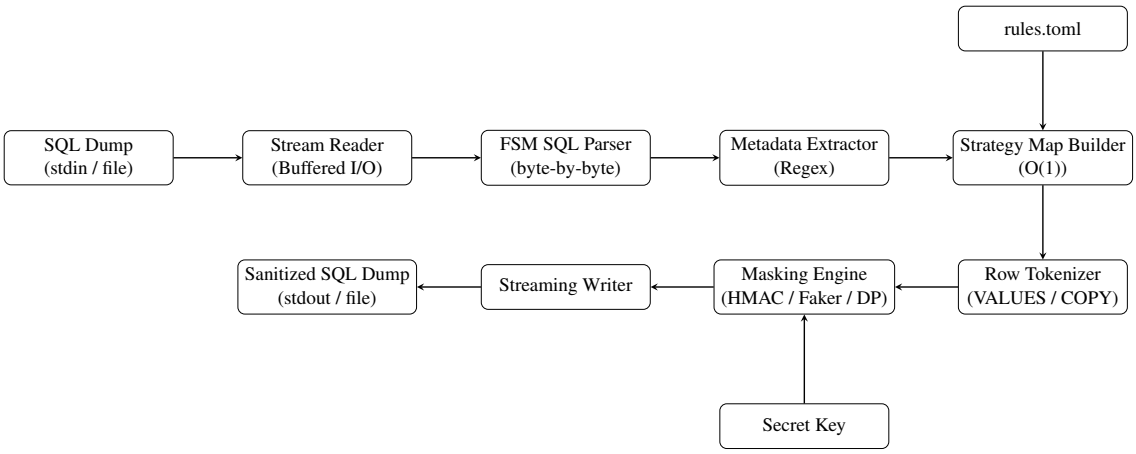

Figura 1. Arquitetura da solução baseada no estilo *pipes and filters*.

Desenvolvida em Rust, a solução se beneficia de um modelo de gerenciamento de memória baseado em posse (*ownership*), no qual a alocação e a liberação de recursos são determinadas em tempo de compilação. Nesse modelo, quando um valor sai de escopo, seus recursos associados são automaticamente liberados, garantindo a segurança de memória (*memory safety*) sem a necessidade da sobrecarga de execução de um *Garbage Collector* [Jung et al. 2021].

Esse mecanismo permite que o sistema mantenha baixo consumo de memória e latência previsível, características fundamentais para aplicações de processamento em fluxo (*streaming*) de alto desempenho. Além disso, o modelo de *ownership* se estende para além da memória, garantindo o gerenciamento seguro de recursos do sistema operacional, como descritores de arquivos e conexões de rede, o que reduz consideravelmente a probabilidade de vazamentos e inconsistências durante a execução contínua do *pipeline*.

A arquitetura é dividida em dois componentes principais: o **Motor de Regras** e o **Processador de Fluxo** (*Streaming Processor*). O Motor de Regras é alimentado por um arquivo de configuração no formato TOML (Tom's Obvious, Minimal Language), uma linguagem de configuração simples e legível, permitindo que equipes de Segurança da Informação definam as políticas de ofuscação (ex: mascarar e-mails, aplicar HMAC em UUIDs) sem necessidade de alterar o código-fonte.

O Processador de Fluxo utiliza *buffers* de tamanho fixo para leitura incremental dos dados, realizando todo o processamento em uma única passagem (*single-pass*) sobre a entrada. Em vez de construir uma AST completa, o sistema utiliza uma Máquina de Estados Finita (FSM) para detectar padrões estruturais de comandos SQL relevantes

mesmo em sintaxes provenientes de diferentes sistemas gerenciadores de banco de dados, incluindo MySQL, PostgreSQL e SQLite. A partir dessa identificação, as regras definidas no arquivo TOML são aplicadas estritamente nas colunas mapeadas. A FSM opera diretamente sobre o fluxo de bytes, identificando delimitadores sintáticos (como parênteses, vírgulas e aspas), o que permite a segmentação incremental de tuplas e colunas sem a necessidade de materializar estruturas intermediárias em memória.

Atualmente, o escopo de blocos sintáticos do parser contempla de forma completa comandos *INSERT*, que representam o padrão estrutural na maioria dos SGBDs relacionais. Contudo, reconhece-se uma limitação prática importante: o comando *COPY*, amplamente utilizado no ecossistema PostgreSQL (dialeto de forte adoção no cenário brasileiro), possui suporte apenas parcial nesta versão experimental. Embora a arquitetura em fluxo seja agnóstica, o tratamento completo da sintaxe tabular do *COPY* exige extensões na FSM que estão além do escopo inicial deste trabalho.

O restante do conteúdo é propagado sem modificações, preservando a estrutura original do *dump*. Para identificação inicial de metadados (como nomes de tabelas e colunas), são utilizadas expressões regulares compiladas uma única vez e reutilizadas ao longo da execução (por exemplo, via inicialização estática), evitando overhead durante o processamento contínuo.

Para evitar comparações repetidas de *strings* durante o processamento, as regras são pré-compiladas em uma estrutura indexada por posição de coluna, permitindo acesso em tempo constante ($\mathcal{O}(1)$) durante a transformação de cada linha, conforme ilustrado pela Figura 1.

A flexibilidade do Motor de Regras e seu desacoplamento do código-fonte, evidenciados na arquitetura, podem ser observados na prática através da Listagem 1. Por meio de uma sintaxe puramente declarativa, as equipes de governança de dados podem definir granularmente as estratégias de desidentificação para cada entidade e coluna, o que favorece a modularidade e a extensibilidade do sistema.

Listagem 1. Exemplo de configuração de mascaramento em formato TOML.

```toml
[tables.users]
columns = [
    # Deterministic Masking
    { name = "id", strategy = "hmac" },
    { name = "company_id", strategy = "hmac" },

    # Faker
    { name = "name", strategy = "faker_name" },
    { name = "email", strategy = "faker_email" },

    # Fixed
    { name = "password_hash", strategy = "fixed",
      value = "\$2a\$12\$R9eb8igRomIQaZAq" },

    # Suppression
    { name = "session_token", strategy = "nullify" },

    # Differential Privacy
    { name = "salary", strategy = "dp_laplace",
      epsilon = 0.5, sensitivity = 15000.0 }
]
```

O exemplo ilustra de forma pragmática a aplicação simultânea das múltiplas

técnicas abordadas na fundamentação teórica. A configuração resolve desafios práticos da engenharia de software, como a substituição de *hashes* de senha por um valor estático conhecido (*fixed*), o que viabiliza a autenticação de desenvolvedores em ambientes de homologação sem expor credenciais reais. Simultaneamente, garante a integridade referencial do banco de dados ao aplicar *hashing* criptográfico determinístico (*hmac*) [Krawczyk et al. 1997] nas chaves primárias e estrangeiras (`id` e `company_id`), preservando a capacidade de realizar operações de *JOIN* nas análises de dados.

Destaca-se ainda a materialização da Privacidade Diferencial Local (LDP) [Kairouz et al. 2014] na coluna `salary` por meio do mecanismo de Laplace. Como a arquitetura orientada a fluxo (*streaming*) [Abadi et al. 2003] inviabiliza o cálculo global da variância dos dados, o sistema delega ao usuário a injeção do orçamento de privacidade ($\epsilon = 0.5$) e da sensibilidade ($\Delta f = 15000.0$). Essa abordagem permite a adição de ruído matemático calibrado registro a registro, preservando o valor estatístico do conjunto de dados numérico enquanto mitiga ataques de reidentificação.

Apesar da flexibilidade provida pela configuração declarativa, a implementação interna do mecanismo de Laplace no sistema assume um compromisso consciente entre desempenho de fluxo (*streaming*) e honestidade no rigor criptográfico. Na implementação atual, a injeção de ruído baseia-se em Amostragem por Transformada Inversa (*Inverse Transform Sampling*) [Kroese and Rubinstein 2012] utilizando tipos de ponto flutuante padrão do Rust (`f64`). Contudo, conforme demonstrado por Mironov [Mironov 2012], implementações ingênuas de privacidade diferencial apoiadas na precisão da IEEE 754 [IEEE 2019] são suscetíveis a ataques baseados nos bits menos significativos, o que pode comprometer a garantia teórica do ϵ.

A decisão arquitetural de não mitigar esse risco por meio da integração de bibliotecas consolidadas de Privacidade Diferencial — como as bibliotecas disponibilizadas pelo Google [Google 2023], implementadas em linguagens como C++, Go e Java — fundamenta-se nas restrições de desempenho do sistema. Tais bibliotecas seguem uma abordagem baseada em *building blocks*, isto é, fornecem primitivas matemáticas fundamentais (como mecanismos de adição de ruído e controle de orçamento de privacidade) que devem ser explicitamente combinadas pelo desenvolvedor para a construção de soluções completas de anonimização.

Embora ofereçam garantias formais robustas, essas bibliotecas não são projetadas para integração direta em pipelines de processamento estruturado em fluxo, como o proposto neste trabalho. A incorporação dessas implementações externas ao ecossistema Rust exigiria o uso extensivo de *Foreign Function Interfaces* (FFI) [Klabnik and Nichols 2023], mecanismo que permite a interoperabilidade entre linguagens distintas. O *overhead* de transição de contexto associado a chamadas FFI frequentes — potencialmente realizadas a cada registro processado — degradaria significativamente o rendimento (*throughput*) da Máquina de Estados Finita, comprometendo a principal vantagem do processamento em única passagem (*single-pass*).

Para versões futuras da ferramenta, prevê-se o desenvolvimento de uma implementação nativa em Rust do Mecanismo de Arredondamento (*Snapping Mechanism*), capaz de prover imunidade a ataques baseados em precisão de ponto flutuante sem sacrificar a latência do fluxo de dados.

4. Avaliação Experimental

Para validar empiricamente as propriedades de consumo de memória e vazão da ferramenta frente a volumes massivos, os experimentos foram divididos em duas etapas: uma análise comparativa de arquiteturas (macOS vs. Windows/WSL2) com 1 milhão de registros, e um teste de estresse extremo processando 100 milhões de registros.

Os experimentos foram conduzidos em dois ambientes físicos e lógicos distintos: (A) um macOS executando em um processador Apple M2 (ARM64) com 8 GB de memória unificada e armazenamento SSD nativo; e (B) um *desktop* equipado com processador Intel Core i5-10400F (2.90 GHz) e 16 GB de memória RAM, executando o sistema operacional Windows 11 Pro através do Windows Subsystem for Linux (WSL2) sobre a arquitetura x86_64.

Como detalhado na Tabela 1, a primeira etapa confirmou a premissa de portabilidade e estabilidade de memória do projeto. Em ambos os ecossistemas, o consumo máximo de RAM manteve-se estagnado na faixa de 4,7 MB. A variação na vazão reflete os limites físicos de I/O de disco de cada sistema operacional, com destaque para a sobrecarga imposta pela transição de sistemas de arquivos (*cross-OS file system*) entre o Linux e o NTFS no WSL2.

Tabela 1. Análise comparativa de arquiteturas (Volume: 1 Milhão de registros)

Ambiente	I/O (Escrita)	Memória (Pico)	Vazão (Linhas/s)
macOS / Apple M2 (8 GB)	/dev/null	4,70 MB	~ 301.634
Windows 11 / i5-10400F (16 GB)	/dev/null	4,56 MB	~ 174.216

Para demonstrar empiricamente o comportamento assintótico constante ($\mathcal{O}(1)$) do consumo de memória em larga escala, a segunda etapa consistiu em um teste de estresse massivo no ambiente macOS, processando **100 milhões de registros relacionais** de forma ininterrupta.

Os resultados, sumarizados na Tabela 2, indicam que o lote foi completamente processado em 325,03 segundos (aproximadamente 5,4 minutos), mantendo um rendimento sustentado de **307.661 linhas por segundo**. O tamanho máximo do conjunto residente (*Maximum Resident Set Size* - RSS) atingiu 4.669.440 bytes ($\sim$4,67 MB), corroborando a estabilidade do consumo de memória independentemente do volume processado.

Tabela 2. Teste de estresse (100 milhões de registros) – macOS / Apple M2

Métrica	Valor
Registros processados	100.000.000
Tempo total	325,03 s
Vazão média	$\sim$ 307.661 linhas/s
Memória (RSS máximo)	4,67 MB
Page faults	13
Operações de swap	0
Instruções executadas	$\sim 4{,}94 \times 10^{12}$

A instrumentação do sistema operacional durante o teste de estresse reforça a eficiência da arquitetura em fluxo. Foram observadas zero operações de *swap* e apenas 13 falhas de página (*page faults*), evidenciando a ausência de pressão significativa sobre a memória primária. O consumo máximo de memória residente manteve-se em aproximadamente 4,67 MB ao longo de todo o processamento, mesmo diante de um volume de dados que excede a capacidade de memória física disponível.

Adicionalmente, a execução envolveu aproximadamente $4{,}94 \times 10^{12}$ instruções na CPU, refletindo a intensidade computacional das operações de anonimização (incluindo hashing criptográfico e injeção de ruído diferencial) realizadas em tempo real, sem impacto na estabilidade do sistema.

Esse resultado contrasta com abordagens tradicionais baseadas em parsing sintático completo (AST), cujo consumo de memória cresce proporcionalmente ao tamanho da entrada, frequentemente resultando em falhas por esgotamento de memória (*out-of-memory*) em cenários de grande escala.

A estabilidade observada da métrica de memória atesta a viabilidade e robustez da ferramenta para integração em esteiras de CI/CD submetidas a cargas intensivas de dados.

5. Considerações Finais

A conformidade contínua com diretrizes como a LGPD requer ferramentas que não imponham atrito ao ciclo de desenvolvimento. A arquitetura em *streaming* proposta demonstrou ser uma alternativa promissora aos gargalos de memória tradicionais na ofuscação de bancos de dados volumosos. Os resultados obtidos trazem indícios de que arquiteturas baseadas em fluxo contínuo possuem potencial para atender cenários de anonimização de dados em larga escala sob restrições estritas de memória.

Trabalhos futuros estenderão esta pesquisa com a implementação de processamento de dialetos nativos de alta performance, como o comando *COPY* do PostgreSQL, além de validações experimentais mais abrangentes que incluam *benchmarks* comparativos e análise de escalabilidade da engenharia de software sobre a máquina de estados. Adicionalmente, prevê-se a investigação sobre a viabilidade de paralelização do processamento do fluxo — visando maximizar o rendimento em processadores *multicore* —, bem como o aprimoramento matemático do mecanismo de Laplace, introduzindo proteções formais contra vulnerabilidades de precisão em ponto flutuante na aplicação da Privacidade Diferencial Local.

Referências

Abadi, D. J. et al. (2003). Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139.

Brasil (2018). Lei nº 13.709, de 14 de agosto de 2018. lei geral de proteção de dados pessoais (lgpd). `http://www.planalto.gov.br`. Brasília, DF: Presidência da República. Acesso em: 18 mar. 2026.

Codd, E. F. (1979). Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434.

Duchi, J. C., Jordan, M. I., and Wainwright, M. J. (2013). Local privacy and statistical minimax rates. In *IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 429–438. IEEE.

Dwork, C., McSherry, F., Nissim, K., and Smith, A. (2006). Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. Springer.

Garrido, G. M., Liu, X., Matthes, F., and Song, D. (2022). Lessons learned: Surveying the practicality of differential privacy in the industry. *arXiv preprint arXiv:2211.03898*.

Google (2023). Google differential privacy library. `https://github.com/google/differential-privacy`. Acesso em: 29 mar. 2026.

Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J., and Langendoen, K. (2012). *Modern Compiler Design*. Springer, 2 edition.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, 2 edition.

IEEE (2019). Ieee standard for floating-point arithmetic (ieee 754-2019). `https://ieeexplore.ieee.org/document/8766229`. IEEE Std 754-2019.

Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. (2021). Safe systems programming in rust: The promise and the challenge. *Communications of the ACM*, 64(4):144–152.

Kairouz, P., Oh, S., and Viswanath, P. (2014). Extremal mechanisms for local differential privacy. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 27.

Klabnik, S. and Nichols, C. (2023). *The Rust Programming Language*. No Starch Press, San Francisco, CA, USA.

Krawczyk, H., Bellare, M., and Canetti, R. (1997). Hmac: Keyed-hashing for message authentication. RFC 2104.

Kroese, D. P. and Rubinstein, R. Y. (2012). Monte carlo methods. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(1):48–58.

Mironov, I. (2012). On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 650–661. ACM.

Mironov, I., Pandey, O., Reingold, O., and Vadhan, S. (2009). Computational differential privacy. In *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 126–142. Springer.

Oliveira, V. H. G. et al. (2024). Anonimização de dados em conformidade com a lgpd: Uma abordagem para bases de dados relacionais. In *Anais do WASHES - CSBC 2024*, pages 11–20. SBC.

Pomès, P. (2026). myanon: Fast mysql dump anonymizer. `https://github.com/ppomes/myanon`. GitHub. Acesso em: 03 mar. 2026.

Ritchie, D. M. (1984). The unix time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(8):1577–1593.