Sistema de Eventos em Tempo Real utilizando Change Data Capture

Victor Tavares Moreira[1], Fabio Castro Araujo[1]

[1]Departamento de Computação
Universidade Luterana do Brasil – Palmas – TO

victor.tm135@rede.ubra.br, fabio.araujo@ulbra.br

Resumo. *Sistemas de informação frequentemente necessitam reagir a alterações em bancos de dados de forma rápida e automatizada. A abordagem tradicional para detectar essas mudanças, conhecida como polling, apresenta limitações como sobrecarga no banco de dados, latência elevada e possibilidade de perda de alterações intermediárias. Técnicas de Change Data Capture (CDC) surgem como alternativa, permitindo capturar mudanças diretamente dos logs de transação e transmiti-las a outros sistemas sem a necessidade de consultas constantes. Este trabalho apresenta a implementação e avaliação de uma prova de conceito de pipeline de CDC baseado em PostgreSQL, Debezium e Apache Kafka, capaz de capturar operações de inserção, atualização e exclusão em tabelas de um banco de dados e disponibilizá-las, com baixa latência, a um consumidor desenvolvido em Python. Os resultados demonstram a viabilidade técnica dessa arquitetura orientada a eventos para cenários que demandam integração de dados e reação rápida a mudanças.*

1. Introdução

Sistemas de informação modernos frequentemente precisam reagir a mudanças nos dados de forma rápida e automatizada. Em arquiteturas tradicionais, quando uma aplicação necessita saber se houve alterações em um banco de dados, a abordagem mais comum é realizar consultas periódicas às tabelas, técnica conhecida como *polling*. Essa estratégia, embora simples, apresenta limitações significativas: gera sobrecarga no banco de dados, pode não capturar todas as alterações e introduz latência entre o momento da mudança e sua detecção (BARBOSA, 2021).

Uma alternativa a essa abordagem é o uso de técnicas de *Change Data Capture* (CDC). O CDC consiste em um conjunto de padrões e ferramentas que permitem identificar, capturar e transmitir alterações realizadas em um banco de dados para outros sistemas, sem a necessidade de consultas constantes às tabelas. Em vez de perguntar repetidamente ao banco "algo mudou?", o CDC monitora os registros internos de transações e emite eventos sempre que uma operação de inserção, atualização ou exclusão é confirmada (SOARES, 2023).

Bancos de dados relacionais como o *PostgreSQL* utilizam mecanismos internos de log de transações para garantir a durabilidade e a recuperação dos dados. No caso do *PostgreSQL*, esse mecanismo é chamado de *Write-Ahead Log* (WAL). O WAL registra

todas as operações realizadas no banco antes que elas sejam efetivamente aplicadas às tabelas, permitindo recuperação em caso de falhas (POSTGRESQL, 2025). Esse mesmo log pode ser utilizado como fonte para ferramentas de CDC, que leem as alterações registradas e as transformam em eventos estruturados.

O *Debezium* é uma plataforma de código aberto que implementa CDC para diversos sistemas gerenciadores de banco de dados, incluindo *PostgreSQL*, *MySQL* e *SQL Server*. Executando como um conector do *Apache Kafka Connect*, o *Debezium* lê as alterações do log de transações e as publica em tópicos do *Apache Kafka*, uma plataforma distribuída de *streaming* de eventos. Essa combinação permite construir arquiteturas orientadas a eventos, nas quais diferentes serviços podem reagir às mudanças de forma assíncrona e desacoplada (DEBEZIUM, 2025).

É importante destacar o significado do termo "tempo real" no contexto deste trabalho. Em sistemas computacionais, "tempo real" pode ter interpretações distintas, desde sistemas com garantias determinísticas de resposta até aplicações que simplesmente oferecem baixa latência. Neste trabalho, os eventos são disponibilizados com latência na ordem de milissegundos a poucos segundos após a confirmação da operação no banco de dados, sem garantias determinísticas. Essa característica é adequada para aplicações que necessitam de respostas rápidas, como sistemas de notificação, painéis analíticos ou módulos de integração entre serviços.

Diante desse contexto, o presente trabalho tem como objetivo implementar e avaliar uma prova de conceito de *pipeline* de *Change Data Capture* utilizando *PostgreSQL*, *Debezium* e *Apache Kafka*. O *pipeline* proposto captura operações de inserção, atualização e exclusão em tabelas de um banco de dados e as disponibiliza a um consumidor desenvolvido em *Python*. O foco do trabalho está na demonstração da viabilidade técnica dessa arquitetura e na documentação dos passos necessários para sua implementação.

Este artigo está organizado da seguinte forma: a Seção 2 apresenta o referencial teórico sobre *Change Data Capture*, logs de transação e arquiteturas orientadas a eventos; a Seção 3 descreve os materiais e métodos adotados na implementação do *pipeline*; a Seção 4 apresenta os resultados obtidos; e a Seção 5 traz as considerações finais, limitações e perspectivas para trabalhos futuros.

2. Referencial teórico

2.1. Change Data Capture (CDC)

Em sistemas de informação, é comum que diferentes aplicações precisem ser notificadas quando dados são alterados em um banco de dados. A abordagem tradicional para detectar essas mudanças é o *polling*, que consiste em realizar consultas periódicas às tabelas para verificar se houve alterações. Embora seja uma técnica simples de implementar, o *polling* apresenta limitações significativas.

Em relação ao método de detecção, o *polling* depende de consultas periódicas às tabelas, enquanto o CDC realiza a leitura direta do log de transações. Quanto ao impacto no banco de dados, o *polling* gera alta sobrecarga devido às múltiplas consultas executadas, ao passo que o CDC causa impacto mínimo, pois apenas lê logs que já existem. No que se refere à latência, o *polling* depende do intervalo configurado entre as consultas, podendo variar de segundos a minutos, enquanto o CDC oferece latência na ordem de milissegundos a poucos segundos. Em termos de captura de alterações, o *polling* pode perder alterações intermediárias que ocorram entre duas consultas

consecutivas, enquanto o CDC captura todas as operações confirmadas no banco de dados. Por fim, a detecção de exclusões é difícil com *polling*, exigindo marcação lógica nas tabelas, enquanto o CDC detecta exclusões de forma nativa (BARBOSA, 2021).

Change Data Capture (CDC) é um conjunto de técnicas e padrões de projeto que permitem identificar, capturar e transmitir alterações realizadas em um banco de dados para outros sistemas. Em vez de consultar repetidamente as tabelas, o CDC monitora os mecanismos internos do banco de dados, como logs de transação, e emite eventos sempre que uma operação de inserção, atualização ou exclusão é confirmada (BARBOSA, 2021).

Existem diferentes abordagens para implementar CDC, cada uma com suas características. O CDC baseado em log lê diretamente o log de transações do banco de dados, sendo a abordagem com menor impacto no desempenho e maior confiabilidade, pois captura todas as operações confirmadas, embora requeira configuração específica no banco de dados para habilitar a leitura dos logs. O CDC baseado em *triggers* utiliza gatilhos no banco de dados que são disparados a cada operação, e embora seja simples de implementar, pode impactar o desempenho das transações. Já o CDC baseado em *timestamps* depende de colunas de data e hora de modificação nas tabelas, sendo mais limitado, pois não detecta exclusões e requer alteração no esquema das tabelas (SOARES, 2023).

Diversas ferramentas implementam CDC, cada uma com características específicas. O *Debezium* é uma plataforma de código aberto que oferece conectores para diversos bancos de dados, incluindo *PostgreSQL* , *MySQL* , *MongoDB* e *SQL Server* . Outras ferramentas incluem o *Maxwell* , voltado especificamente para *MySQL* , o *Oracle GoldenGate* , uma solução comercial da Oracle, e o *AWS Database Migration Service* (DMS), um serviço gerenciado da Amazon Web Services (DEBEZIUM, 2025). Este trabalho utiliza o *Debezium* por ser uma solução de código aberto, amplamente documentada e com suporte nativo ao *PostgreSQL* por meio de CDC baseado em log.

2.2. Logs de Transação e Write-Ahead Log

Bancos de dados relacionais utilizam mecanismos de log de transações para garantir a durabilidade e a consistência dos dados, mesmo em casos de falhas. Esses logs registram todas as operações realizadas no banco de dados, permitindo a recuperação do estado consistente após interrupções inesperadas (ELMASRI; NAVATHE, 2019).

No *PostgreSQL* , esse mecanismo é chamado de *Write-Ahead Log* (WAL). O princípio fundamental do WAL é que nenhuma modificação nos dados é gravada nas tabelas antes que o registro correspondente seja gravado no log. Essa estratégia garante que, em caso de falha, o banco de dados possa reconstruir seu estado a partir dos registros do WAL (POSTGRESQL, 2025).

O WAL é organizado como uma sequência de arquivos que contêm registros de todas as operações de inserção, atualização e exclusão. Cada registro inclui informações suficientes para refazer ou desfazer a operação correspondente. Além de sua função primária de recuperação, o WAL serve como base para mecanismos de replicação. A replicação física copia os bytes do WAL para outro servidor, criando uma réplica idêntica do banco de dados, sendo utilizada para alta disponibilidade e balanceamento de carga de leitura. A replicação lógica, por sua vez, decodifica o conteúdo do WAL em operações lógicas (INSERT, UPDATE, DELETE) que podem ser interpretadas e

consumidas por outras aplicações, sendo este o mecanismo utilizado pelo *Debezium* para implementar CDC.

Para que a replicação lógica funcione, o *PostgreSQL* deve ser configurado com o parâmetro `wal_level=logical`, que instrui o banco de dados a incluir informações adicionais no WAL, suficientes para identificar as tabelas, colunas e valores envolvidos em cada operação. Sem essa configuração, o WAL contém apenas informações de baixo nível, como posições de páginas e bytes, inadequadas para CDC (POSTGRESQL, 2025).

2.3. Arquitetura Orientada a Eventos

Arquitetura orientada a eventos é um padrão de projeto de *software* no qual o fluxo do sistema é determinado por eventos. Um evento representa uma mudança significativa no estado do sistema, como a inserção de um registro, a confirmação de um pagamento ou a atualização de um cadastro (HOHPE; WOOLF, 2003).

Nesse modelo arquitetural, os componentes do sistema são classificados em três categorias principais. Os produtores são componentes que detectam mudanças e emitem eventos, e no contexto deste trabalho, o *Debezium* atua como produtor, emitindo eventos a partir das alterações capturadas no banco de dados. O barramento de eventos é a infraestrutura responsável por receber, armazenar e distribuir eventos, sendo o *Apache Kafka* uma plataforma amplamente utilizada para essa finalidade. Os consumidores são componentes que se inscrevem em determinados tipos de eventos e reagem quando esses eventos ocorrem, podendo existir múltiplos consumidores reagindo ao mesmo evento de formas distintas.

A principal vantagem dessa arquitetura é o desacoplamento entre os componentes. O produtor não precisa conhecer os consumidores, e novos consumidores podem ser adicionados sem modificar os produtores existentes. Além disso, o processamento é assíncrono, de modo que o produtor emite o evento e continua sua execução, sem aguardar que os consumidores o processem.

O *Apache Kafka* é uma plataforma distribuída de *streaming* de eventos que implementa o padrão *publish-subscribe* . Os eventos são organizados em tópicos, que funcionam como canais de comunicação. Produtores publicam eventos em tópicos específicos, e consumidores se inscrevem nos tópicos de seu interesse. O *Kafka* faz com que os eventos persistam em disco, permitindo que sejam lidos múltiplas vezes e por diferentes consumidores. Essa característica o diferencia de sistemas de mensageria tradicionais e o torna adequado para cenários que exigem alta vazão e baixa latência (APACHE KAFKA, 2025).

A combinação de CDC com arquiteturas orientadas a eventos permite que alterações em um banco de dados sejam propagadas para outros sistemas de forma eficiente. O *Debezium* executa como um conector do *Kafka Connect* , um *framework* do ecossistema *Kafka* para integração de dados. Ao detectar uma alteração no banco de dados, o *Debezium* a transforma em um evento estruturado e o publica em um tópico do *Kafka* . A partir daí, qualquer consumidor inscrito no tópico pode processar o evento.

2.4. Trabalhos relacionados

Barbosa (2021), em artigo técnico intitulado *Change Data Capture* (CDC): conceito e prática, apresentou os fundamentos do CDC e demonstrou uma implementação utilizando *Debezium* , *MySQL* e *Apache Kafka* . O autor descreveu o CDC como uma técnica para monitorar e capturar alterações em bancos de dados, permitindo que outras aplicações reajam a essas mudanças. O trabalho apresentou uma arquitetura na qual o *Debezium* lê os logs de transação do banco de dados, transforma as operações em eventos e os envia para tópicos do *Kafka* , onde são consumidos por uma aplicação em *Python* . O estudo demonstrou a viabilidade técnica da abordagem em um cenário genérico de integração de dados.

Soares (2023), no artigo *Change Data Capture* com *Debezium* , detalhou a configuração e operação do *Debezium* para captura de dados em *PostgreSQL* . O autor explicou o funcionamento da replicação lógica e os formatos de mensagem gerados pelo *Debezium* . O trabalho destacou aspectos práticos como a normalização de tipos de dados e o tratamento de eventos de diferentes operações, incluindo inserção, atualização e exclusão.

Kleppmann (2017), no livro *Designing Data-Intensive Applications* , dedicou um capítulo ao conceito de CDC e sua importância em arquiteturas de dados modernas. O autor argumenta que o log de transações do banco de dados representa a fonte mais confiável de verdade sobre as mudanças nos dados e que o CDC permite derivar outras representações a partir dessa fonte. O trabalho contextualiza o CDC dentro de uma discussão mais ampla sobre *streaming* de dados e arquiteturas orientadas a eventos.

O presente trabalho se diferencia dos estudos anteriores ao focar na implementação completa de um *pipeline* de CDC como prova de conceito, documentando detalhadamente os passos de configuração e implementação. Enquanto Barbosa (2021) utilizou *MySQL* como banco de dados, este trabalho emprega *PostgreSQL* e explora suas características específicas de replicação lógica. Além disso, este estudo inclui a implementação de um consumidor em *Python* que normaliza os dados capturados, transformando formatos técnicos do *Debezium* em representações legíveis.

3. Materiais e Métodos

3.1. Materiais

Para o desenvolvimento deste trabalho, foram utilizados recursos de software e infraestrutura específicos, organizados em diferentes camadas da arquitetura. O ambiente foi configurado em um computador com sistema operacional *Windows*. O *Docker Desktop* com *backend* WSL 2 permitiu a execução dos serviços em contêineres. O *Visual Studio* Code foi adotado como ambiente de desenvolvimento. O *Docker Compose* orquestrou todos os serviços necessários, permitindo inicializar e encerrar o conjunto completo de contêineres de forma padronizada por meio de comandos únicos. Essa abordagem eliminou a necessidade de instalação manual de componentes na máquina hospedeira.

Na camada de banco de dados, foi utilizado o *PostgreSQL*, executado em contêiner a partir da imagem oficial da versão 17. Os parâmetros do WAL foram ajustados para suportar replicação lógica, conforme descrito na Seção 2.2. No banco de dados foram criadas tabelas de exemplo para demonstrar o funcionamento do *pipeline*.

O *DBeaver* foi utilizado como cliente gráfico para criação das tabelas e execução dos comandos SQL durante os testes.

Na camada de mensageria, foram utilizados *Apache Kafka* e *Zookeeper*, ambos executados em contêineres. O *Kafka Connect* recebeu a configuração do conector *Debezium* para *PostgreSQL*. A ferramenta *Kafka UI* disponibilizou um painel *web* para inspeção visual dos tópicos e mensagens.

Na camada de consumo, foi utilizado *Python 3* com ambiente virtual dedicado. A biblioteca *kafka-python* realizou a integração com o *broker*. O consumidor foi implementado como um *script* de linha de comando que assina os tópicos gerados pelo *Debezium*, desserializa as mensagens JSON e exibe as operações detectadas no terminal.

O Quadro 1 apresenta uma visão consolidada dos materiais utilizados, organizados por camada da arquitetura, incluindo os componentes e suas respectivas versões ou especificações.

Quadro 1 - Materiais utilizados no desenvolvimento

"Componentes do Ambiente (CDC)"

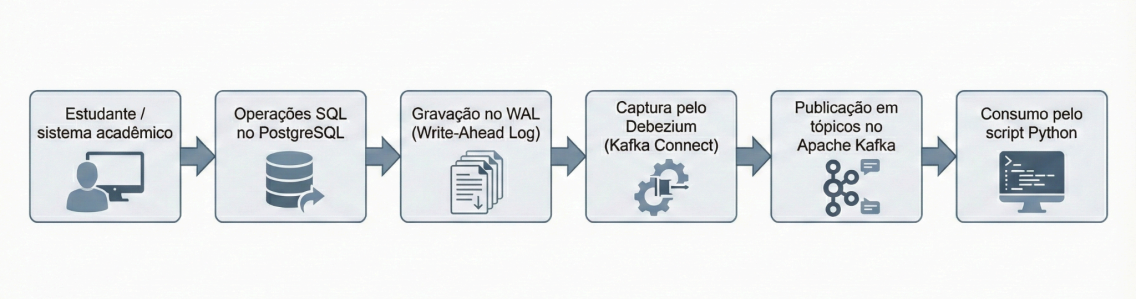
Camada	Componente	Versão/Especificação
Banco de dados	PostgreSQL	17 (imagem Docker oficial)
Banco de dados	DBeaver	Cliente SQL
CDC	Debezium	Conector para PostgreSQL
Mensageria	Apache Kafka	Imagem Confluent
Mensageria	Zookeeper	Imagem Confluent
Mensageria	Kafka Connect	Framework de integração
Visualização	Kafka UI	Interface web
Consumidor	Python 3	Com biblioteca kafka-python
Orquestração	Docker Compose	Gerenciamento de contêineres

3.2. Métodos

O desenvolvimento deste trabalho seguiu uma abordagem exploratória e incremental, característica de provas de conceito técnicas. A metodologia adotada consistiu em configurar cada componente da arquitetura de forma isolada, validar seu funcionamento e então integrá-lo aos demais componentes. Essa abordagem permitiu identificar e corrigir problemas em cada etapa antes de prosseguir para a seguinte.

Para ilustrar o fluxo completo do *pipeline* implementado, a Figura 1 apresenta as etapas pelas quais os dados transitam, desde a operação original no banco de dados até o consumo pelo *script Python*. Conforme apresentado na figura, o fluxo inicia quando uma operação SQL é executada no *PostgreSQL*. Essa operação é então registrada no WAL antes de ser aplicada às tabelas. O *Debezium*, executando como conector do *Kafka Connect*, lê continuamente o WAL por meio de replicação lógica. Ao identificar uma alteração nas tabelas monitoradas, o *Debezium* transforma a operação em um evento estruturado e o publica em um tópico do *Apache Kafka*. Por fim, o consumidor *Python* lê os eventos do tópico e os processa, exibindo os dados normalizados no terminal.

Figura 1. Fluxo do *pipeline* de CDC implementado



O desenvolvimento do *pipeline* foi organizado em quatro fases principais, cada uma correspondendo a uma subseção a seguir: configuração do banco de dados e do WAL, configuração do *pipeline* de CDC, implementação do consumidor e procedimento de testes.

3.2.1 Configuração do banco de dados e do WAL

O banco de dados *PostgreSQL* foi configurado para suportar a captura de mudanças por replicação lógica. O serviço foi executado em contêiner *Docker* a partir da imagem oficial da versão 17. Os parâmetros de inicialização foram definidos no arquivo *docker-compose.yml* para atender aos requisitos do CDC. O parâmetro "wal_level" foi configurado como "logical" para habilitar a replicação lógica. O parâmetro ''max_wal_senders'' foi definido como 4, especificando o número máximo de processos que podem enviar dados do WAL simultaneamente. O parâmetro "max_replication_slots" também foi definido como 4, determinando o número máximo de *slots* de replicação disponíveis.

No banco de dados, denominado ''gamificacao'', foram criadas duas tabelas para demonstrar o funcionamento do *pipeline*. A tabela "public.entregas_de_trabalhos'' contém os campos: "id" (chave primária), "id_aluno" (identificador do aluno), "nota'' (valor decimal) e "data_entrega'' (timestamp). A tabela "public.frequencia" contém os campos: `id` (chave primária), `id_aluno` (identificador do aluno), "data_aula" (data) e "presente" (valor booleano).

Para permitir que o *Debezium* acessasse o fluxo de mudanças, foi criada uma *role* específica no *PostgreSQL* com permissões de *login*, replicação e leitura nas tabelas de interesse. Em seguida, foi criada uma *publication* denominada "dbz_publication", incluindo as tabelas que seriam monitoradas. Essa *publication* define quais tabelas participam da replicação lógica e, consequentemente, quais operações são expostas para captura pelo CDC.

3.2.2 Configuração do pipeline de CDC

O *pipeline* de CDC foi configurado para transportar os eventos do banco de dados até a camada de mensageria. A *stack* de serviços foi definida em um arquivo *docker-compose.yml* que especifica os contêineres do *PostgreSQL*, *Zookeeper*, *Apache Kafka*, *Kafka Connect* e *Kafka UI*. A imagem do *Kafka Connect* utilizada já inclui o

conector *Debezium* para *PostgreSQL*. Todos os serviços foram inicializados por meio do comando “docker-compose up -d”, que executa os contêineres em segundo plano.

Com os serviços em execução, foi criado o arquivo “connector.json” contendo a configuração do conector *Debezium*. Os principais parâmetros configurados foram: “connector.class”, especificando a classe do conector *PostgreSQL*; “database.hostname”, “database.port”, “database.user” e “database.password”, definindo as credenciais de acesso ao banco; “database.dbname”, indicando o banco de dados monitorado; “topic.prefix”, estabelecendo o prefixo dos tópicos *Kafka* criados; “table.include.list”, listando as tabelas a serem monitoradas no formato “schema.tabela”; “plugin.name”, especificando o *plugin* de decodificação lógica (“pgoutput”); “publication.name”, referenciando a *publication* criada no *PostgreSQL*; e “slot.name”, definindo o nome do *slot* de replicação.

O parâmetro “tombstones.on.delete” foi configurado como “false”. Esse parâmetro controla a emissão de *tombstone messages*, que são mensagens com valor nulo publicadas após eventos de exclusão. Essas mensagens são utilizadas por alguns consumidores *Kafka* para compactação de logs, mas foram desabilitadas neste trabalho por não serem necessárias para a demonstração.

O conector foi registrado por meio de uma requisição HTTP POST à API REST do *Kafka Connect*, enviando o conteúdo do arquivo “connector.json”. Após o registro, o status do conector foi verificado pela mesma API para confirmar que a *task* estava em execução. O *Debezium* cria automaticamente os tópicos correspondentes às tabelas monitoradas, seguindo o padrão “{topic.prefix}.{schema}.{tabela}”. Neste trabalho, foram criados os tópicos “server1.public.entregas_de_trabalhos” e “server1.public.frequencia”.

3.2.3 Implementação do consumidor

O consumidor responsável por ler os eventos publicados pelo *Debezium* foi implementado em *Python*. Um ambiente virtual dedicado foi criado para isolar as dependências do projeto, e a biblioteca *kafka-python* foi instalada nesse ambiente.

O *script* “consumer_cli.py” foi desenvolvido para conectar-se ao *broker Kafka* e processar os eventos recebidos. O consumidor foi configurado com os seguintes parâmetros: endereço do *broker* (“localhost:29092”), identificador do grupo de consumidores, política de *offset* inicial (“latest”, para ler apenas novas mensagens) e funções de desserialização para converter as mensagens JSON.

A lógica de processamento considera a estrutura de *envelope* adotada pelo *Debezium*. Cada mensagem contém um campo “payload” com informações sobre a operação realizada. O campo “op” indica o tipo de operação: “c” para inserção (*create*), “u” para atualização (*update*), “d” para exclusão (*delete*) e “r” para leitura de *snapshot*. Os campos “before” e “after” contêm, respectivamente, o estado da linha antes e depois da operação.

O consumidor implementa funções de normalização para converter os tipos de dados específicos do *Debezium* em formatos legíveis. Datas representadas como número de dias desde 1970-01-01 (tipo “io.debezium.time.Date”) são convertidas para o formato ISO. *Timestamps* em microssegundos (tipo “io.debezium.time.MicroTimestamp”) são convertidos para o formato “YYYY-MM-DD HH:MM:SS”. Valores decimais codificados em *base64* (tipo

”org.apache.kafka.connect.data.Decimal”) são decodificados e formatados com a escala apropriada.

3.2.4 Procedimento de testes

Os testes foram conduzidos com o objetivo de validar o funcionamento do *pipeline* de ponta a ponta. Com todos os serviços em execução e o consumidor *Python* iniciado, comandos SQL de inserção, atualização e exclusão foram executados nas tabelas monitoradas por meio do *DBeaver*. Cada transação foi confirmada explicitamente (*commit*) para garantir o registro da operação no WAL.

O procedimento de testes consistiu em executar operações nas tabelas e observar os eventos correspondentes em dois pontos: no *Kafka UI*, verificando a publicação das mensagens nos tópicos, e no terminal do consumidor *Python*, verificando a exibição dos eventos normalizados. Esse procedimento permitiu validar cada etapa do fluxo apresentado na Figura 1.

É importante destacar os fatores que influenciam a latência do *pipeline*. O tempo entre a execução de uma operação no banco de dados e a disponibilização do evento para consumo depende de diversos elementos: o intervalo de *polling* do *Debezium* ao ler o WAL, a latência de rede entre os contêineres, o tempo de processamento do *Kafka* para persistir e disponibilizar a mensagem, e o intervalo de *polling* do consumidor *Python*. Em ambiente local com todos os serviços executando na mesma máquina, a latência observada foi da ordem de centenas de milissegundos a poucos segundos, caracterizando o comportamento de baixa latência mencionado na introdução. Em ambientes de produção com componentes distribuídos em diferentes servidores, essa latência pode variar conforme a infraestrutura de rede e a carga do sistema.

4. Resultados

Nesta seção são apresentados os resultados obtidos com a implementação do *pipeline* de CDC. São descritos o funcionamento do fluxo de eventos, a captura e publicação das mensagens no *Apache Kafka*, o consumo e a normalização dos dados pelo *script Python*, e as limitações observadas durante o experimento.

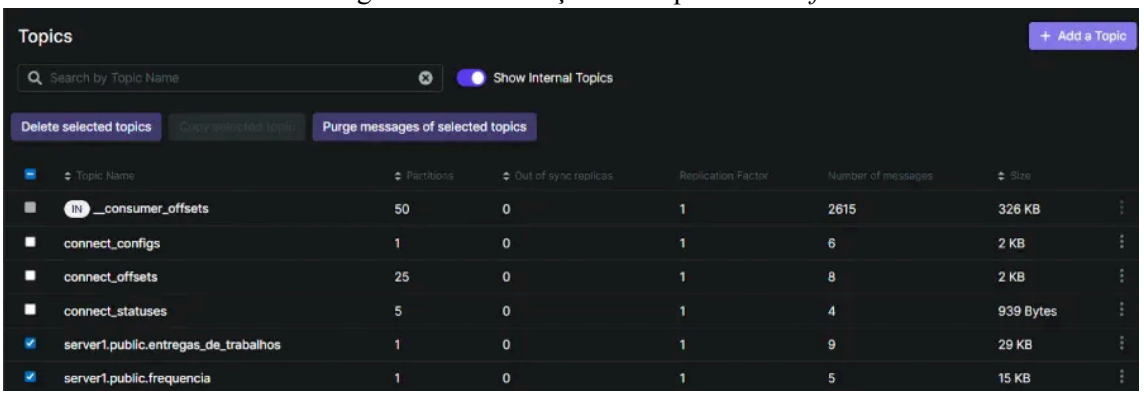
4.1 Funcionamento do *Pipeline*

Após a configuração do ambiente descrita na Seção 3, o *pipeline* de CDC passou a operar de forma contínua, monitorando as tabelas “public.frequencia” e “public.entregas_de_trabalhos” no banco de dados “gamificacao”. O conector *Debezium* decodifica os registros do WAL sempre que uma operação de inserção, atualização ou exclusão é confirmada nessas tabelas, publicando os eventos correspondentes em tópicos específicos no A*pache Kafka*.

A integração entre *PostgreSQL*, *Debezium* e *Kafka* foi confirmada pela criação automática dos tópicos pelo conector. A Figura 2 apresenta a interface do *Kafka UI* exibindo os tópicos criados durante o experimento. Conforme pode ser observado na figura, os tópicos “server1.public.entregas_de_trabalhos” e “server1.public.frequencia” foram criados automaticamente pelo *Debezium*. A coluna "Number of messages" indica

a quantidade de eventos publicados em cada tópico, confirmando que o conector estava ativo e capturando as operações realizadas no banco de dados. Os demais tópicos visíveis na interface (“__consumer_offsets”, “connect_configs”, “connect_offsets” e “connect_statuses”) são tópicos internos utilizados pelo *Kafka* e pelo *Kafka Connect* para gerenciamento.

Figura 2. Visualização dos tópicos no *Kafka UI*.

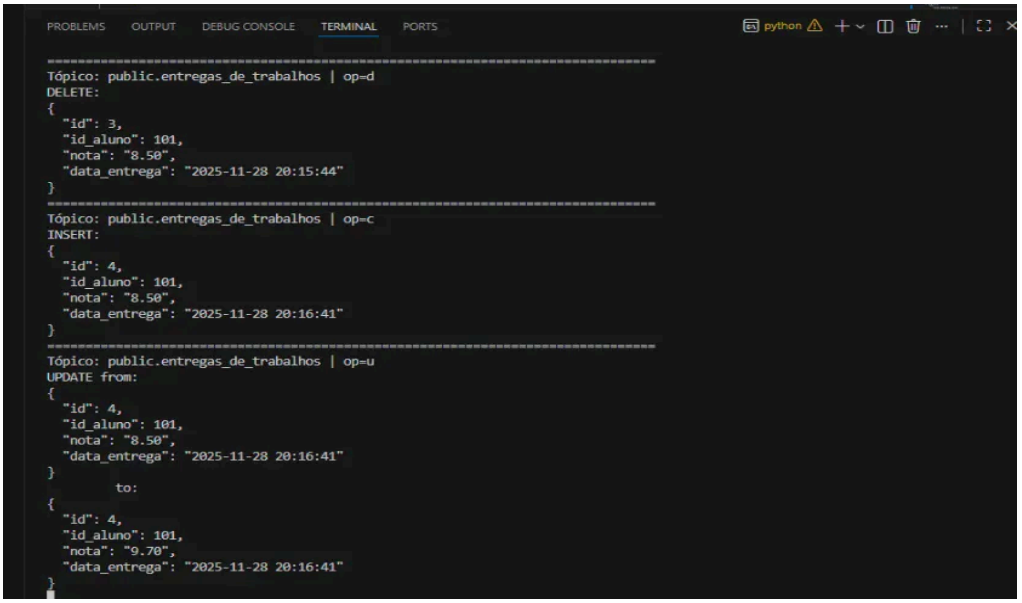


4.2 Captura e Exibição dos Eventos

O consumidor implementado em *Python* foi configurado para assinar os tópicos criados pelo *Debezium*. Para cada mensagem recebida, o *script* acessa o campo “payload”, identifica o tipo de operação por meio do campo “op” e extrai os estados anterior (“before”) e posterior (“after”) da linha modificada.

A Figura 3 apresenta a saída do consumidor no terminal, exibindo exemplos de eventos capturados na tabela “public.entregas_de_trabalhos”. Na parte superior da figura, é possível observar um trecho do código do *script* “consumer_cli.py”, mostrando a configuração do “KafkaConsumer”. Na parte inferior, o terminal exibe três eventos distintos: uma operação de exclusão (“op=d”), uma operação de inserção (“op=c”) e uma operação de atualização (“op=u”). Para a operação de exclusão, são exibidos os dados da linha removida. Para a inserção, são exibidos os dados da nova linha. Para a atualização, são exibidos os estados anterior e posterior, permitindo identificar que o campo “nota” foi alterado de “8.50” para “9.70”.

Figura 3. Eventos capturados pelo consumidor *Python*.

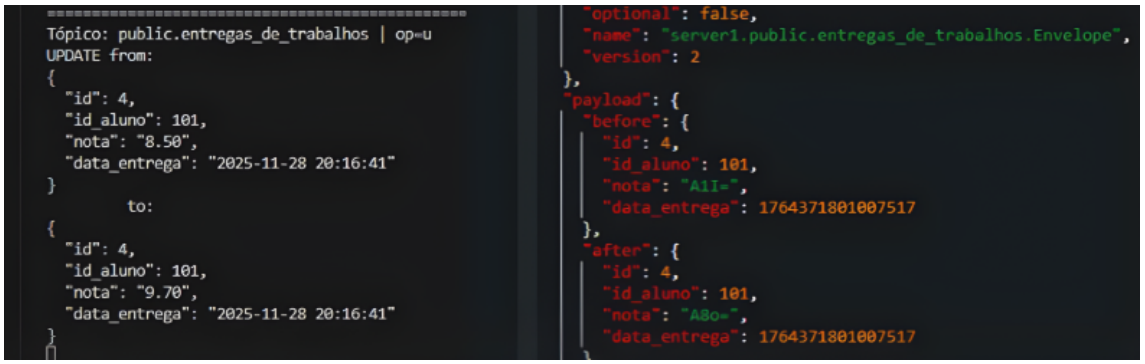


4.3 Normalização dos Dados

Os eventos emitidos pelo *Debezium* utilizam formatos específicos para representar determinados tipos de dados. Datas são representadas como número inteiro de dias desde 1970-01-01. *Timestamps* são representados em microssegundos desde a mesma data de referência. Valores decimais são codificados em *base64*. Esses formatos, embora precisos, dificultam a leitura e o uso direto dos dados por outras aplicações.

O consumidor implementado inclui rotinas de normalização que convertem esses formatos em representações legíveis. A Figura 4 apresenta uma comparação entre o evento no formato bruto emitido pelo *Debezium* (lado direito) e o mesmo evento após a normalização pelo consumidor *Python* (lado esquerdo). No formato bruto, o campo "nota" aparece como "AII=" e "A8o=" (valores em *base64*), e o campo "data_entrega" aparece como `1764371801007517` (microssegundos). Após a normalização, o campo "nota" é exibido como "8.50" e "9.70", e o campo "data_entrega" é exibido como "2025-11-28 20:16:41".

Figura 4. Comparação entre evento bruto e normalizado.



Essa normalização é importante porque permite que os dados capturados sejam consumidos diretamente por outras aplicações sem necessidade de tratamento adicional. Sistemas de notificação, painéis analíticos ou módulos de integração podem utilizar os eventos normalizados de forma imediata.

4.4 Limitações

O experimento apresenta limitações que devem ser consideradas na interpretação dos resultados.

Em relação ao ambiente de testes, o *pipeline* foi avaliado em um ambiente controlado, com todos os componentes executando na mesma máquina por meio de contêineres *Docker*. Apenas duas tabelas foram monitoradas, e o volume de dados foi reduzido. Todas as operações foram executadas de forma manual por meio do *DBeaver*. Esse cenário não representa as condições de um ambiente de produção, no qual múltiplas aplicações podem gerar operações simultâneas em grande volume.

Em relação às medições, não foram realizados testes sistemáticos de desempenho. A latência do *pipeline* foi observada de forma qualitativa, constatando-se que os eventos eram disponibilizados rapidamente após a confirmação das operações no banco de dados. No entanto, medições precisas de latência, vazão máxima ou comportamento sob carga elevada não foram conduzidas.

Em relação ao consumidor, o *script Python* implementado é um protótipo que exibe os eventos em linha de comando. Ele não está integrado a sistemas externos, não persiste os eventos processados e não implementa tratamento robusto de erros ou reconexão automática em caso de falhas.

Essas limitações indicam oportunidades para trabalhos futuros, como a realização de testes de carga, a ampliação do conjunto de tabelas monitoradas, a evolução do consumidor para um serviço de *backend* com API e a avaliação do *pipeline* em ambientes mais próximos de produção.

5. Considerações finais

Este trabalho apresentou a implementação de um *pipeline* de *Change Data Capture* (CDC) utilizando PostgreSQL, Debezium e Apache Kafka, com o objetivo de demonstrar a viabilidade técnica de uma arquitetura capaz de capturar operações de inserção, atualização e exclusão em um banco de dados e disponibilizá-las como eventos estruturados para consumo por outras aplicações.

A arquitetura implementada combina o banco de dados *PostgreSQL* com replicação lógica habilitada, o conector *Debezium* executando sobre o *Kafka Connect*, o Apache Kafka atuando como barramento de eventos e um consumidor desenvolvido em Python. Essa combinação permitiu que alterações confirmadas no banco de dados fossem capturadas e publicadas em tópicos *Kafka* com latência na ordem de milissegundos a poucos segundos, eliminando a necessidade de consultas periódicas às tabelas (*polling*).

O desenvolvimento foi organizado em quatro fases: configuração do banco de dados e dos parâmetros do WAL, configuração do *pipeline* de CDC, implementação do consumidor em *Python* e execução dos testes de validação. O banco de dados “gamificacao” foi utilizado com as tabelas “public.frequencia” e “public.entregas_de_trabalhos” como exemplos, gerando eventos correspondentes nos tópicos *Kafka* a cada operação confirmada. Os resultados demonstraram que o *pipeline* capturou corretamente os três tipos de operação (INSERT, UPDATE e DELETE), disponibilizando os eventos para consumo de forma imediata.

A normalização dos dados no consumidor Python mostrou-se um aspecto relevante da solução. O *script* implementado interpreta os tipos lógicos específicos do *Debezium*, convertendo datas, *timestamps* e valores decimais codificados em *base64* para formatos legíveis. Essa camada de normalização faz a ponte entre o formato técnico do log de mudanças e o formato esperado por aplicações como sistemas de notificação, painéis analíticos ou módulos de integração.

Conforme apresentado na Seção 4.4, o experimento possui limitações que devem ser consideradas. O ambiente controlado, o volume reduzido de dados, a execução manual das operações e a ausência de testes sistemáticos de desempenho restringem os resultados ao contexto de uma prova de conceito. Apesar dessas limitações, o trabalho cumpriu seu objetivo de demonstrar a viabilidade técnica da abordagem proposta.

Como trabalhos futuros, destacam-se: a realização de testes de carga para avaliar a latência e a vazão máxima do *pipeline* sob diferentes volumes de operações; a ampliação do conjunto de tabelas monitoradas para cenários mais complexos; a evolução do consumidor *Python* para um serviço de *backend* com API REST, permitindo integração com interfaces *web* ou aplicações móveis; e a avaliação do comportamento do *pipeline* em ambientes distribuídos, com os componentes executando em servidores distintos.

Em síntese, a arquitetura de CDC baseada em *Debezium* e *Apache Kafka* demonstrou ser uma alternativa eficiente ao *polling* tradicional para cenários que demandam reação rápida a mudanças em bancos de dados. A documentação detalhada dos passos de configuração e implementação apresentada neste trabalho contribui como referência técnica para a adoção dessa abordagem em projetos que necessitem de integração de dados com baixa latência.

6. Referências

APACHE SOFTWARE FOUNDATION. Apache Kafka Documentation. 2025. Disponível em: https://kafka.apache.org/documentation/.

BARBOSA, A. Change Data Capture (CDC): conceito e prática. Medium, 1 mar. 2021. Disponível em: https://armandobs14.medium.com/change-data-capture-cdc-8e52c260baa7.

DEBEZIUM. Debezium Documentation. 2025. Disponível em: https://debezium.io/documentation/.

ELMASRI, R.; NAVATHE, S. B. Sistemas de banco de dados. 7. ed. São Paulo: Pearson, 2019. 1122 p. ISBN: 978-8543025001.

HOHPE, G.; WOOLF, B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston: Addison-Wesley, 2003. 736 p. ISBN: 978-0321200686.

KLEPPMANN, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2017. 616 p. ISBN: 978-1449373320.

POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL Documentation: Write-Ahead Logging (WAL). 2025. Disponível em: https://www.postgresql.org/docs/current/wal-intro.html.

SOARES, J. W. Change Data Capture com Debezium. The Data Engineer, 12 maio 2023. Disponível em: https://thedataengineer.com.br/2023/05/12/change-data-capture-com-debezium/.